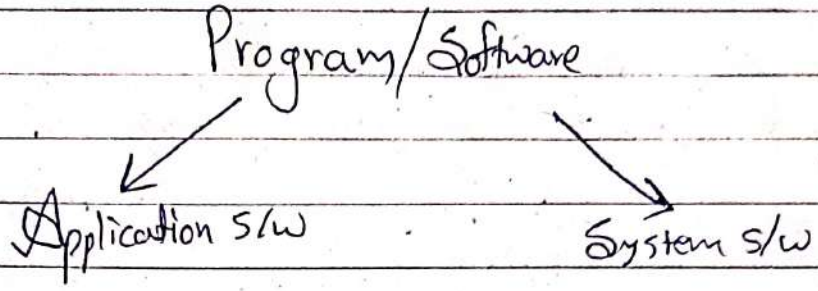
- Os & It's functions
- Process Scheduling
- Process Synchronization
- Deadlock
- Memory Mgmt
- Hard Disk Architecture
- File System in OS
- Protection & Security

Contents

Operating System :

It's a sys s/w that acts as an interface b/w user & hardware.

S/w is nothing but a tested prog + documentation.



- **Functions :**
  1. Resource Management
  2. Processor Mgmt (Scheduling)
  3. Memory Mgmt
  4. I/O Device Mgmt
  5. Storage Mgmt
  6. Security & Protection

- **Goals :**
  1. Convenience
  2. Efficiency

- **Interface :**
  1. GUI
  2. CUI (Character UI)

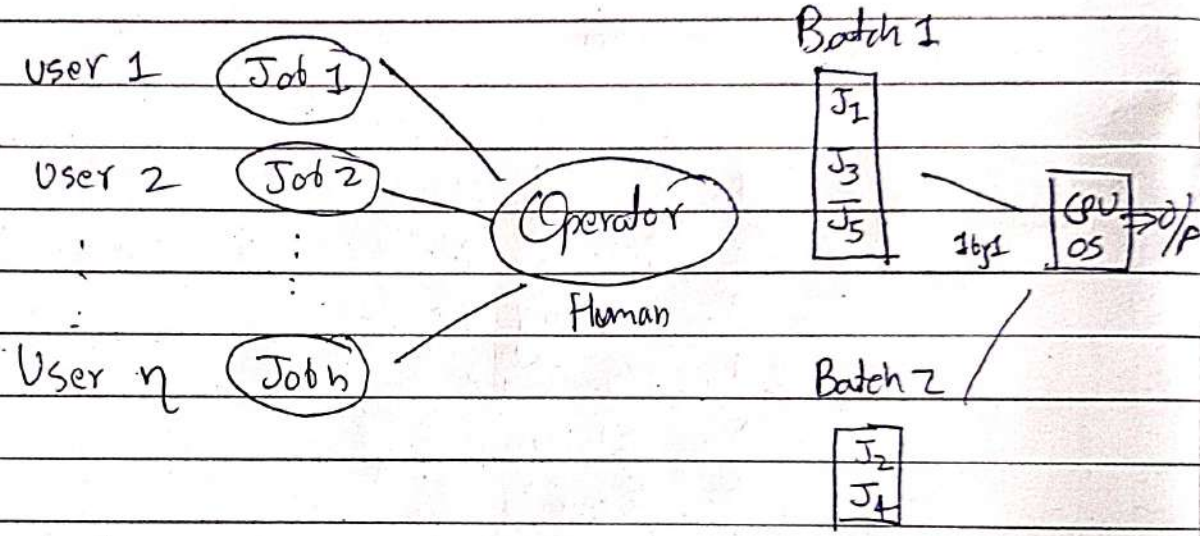
- **Types :**
  1. Batch OS
  2. Multiprogramming OS (Non-Preemptive)
  3. Multitasking / Time Sharing / Multiprog^{m} w RoundRobin
  4. Multiprocessing / Parallel System OS
  5. Real time OS
  6. Embedded OS
  7. Clustered OS
  8. Distributed OS

(1) Batch OS :

Program I/P Data Control Instruction

← A Job (task to perform)

User prepare jobs in punch cards & magnetic tapes & submit it to operator for execution



Disadvantages :

1) Non-interactive, i.e Human has to pass batches manually

2) Idle CPU, during loading & unloading of batches CPU is idle

eg: Fortran (IBM), IBSYS709X (1960)

(2) **Multiprogramming OS** : Processes are in queue but do not execute simultaneously but 1 by 1. It's non-preemptive.

Ready Queue | P4 P3 P2 P1 | CPU

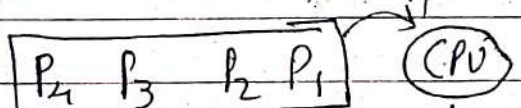
Preamptive : Process cannot halt (pause) in-between the execution, i.e it can only go out iff its fully completed.

Advantage : CPU does not remain Idle

Disadvantage : Smaller process might've to wait for unfixed (indefinite) amount of time (i.e Starvation)

(3) Multitasking / Time-Sharing / Multiprogramming with Round Robin OS :

Every process is executed 1by 1 for a fixed time (time slice or time quantum) repeadedly giving illusion that many tasks are executing @ once

eg: 

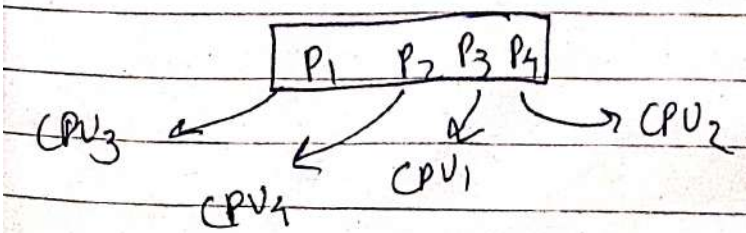
Quantum = 2 sec

Proc	Burst Time	Ran for
P_1	5	2 2 1
P_2	10	2 2 2 2 2
P_3	3	2 1
P_4	4	2 2

Pass1 Pass2

(4) Multiprocessing / Parallel System OS :

Processor has multiple core & able to do ||^{el} processing



BUT WE STUDY ONLY FOR UNIPROCESSOR

(5) Real time OS : They're time constant OS (bounded by time)

eg: Missible launchiers

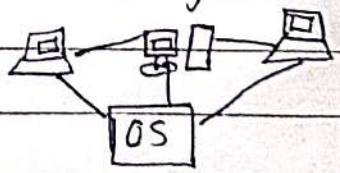
- Response should be within specified time constraint
- It's specific strictly deadly system i-e no delay while executing any program

Hard Real Time System	Soft RTS
1) Can never mis its deadline not even minor delay accepted	1) Can miss its deadline with some acceptable low probability (small delay allowed)
eg: Satellite & Missile System	eg: Banking & Telephone systems

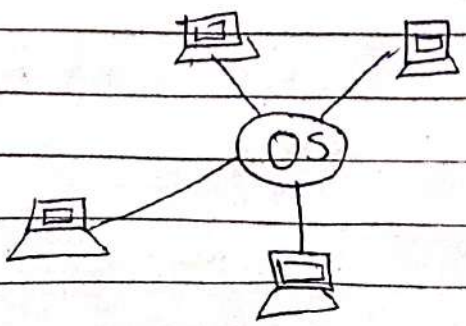
(6) Embedded OS : They're specifically designed for embedded systems. eg: Smart watches dashbord in car.

(7) Clustered OS : Multiple computers are merged & work together.

It is to paralled System OS.



(8) Distributed OS : Task (OS) is distributed among various systems.



Process State Diagram :

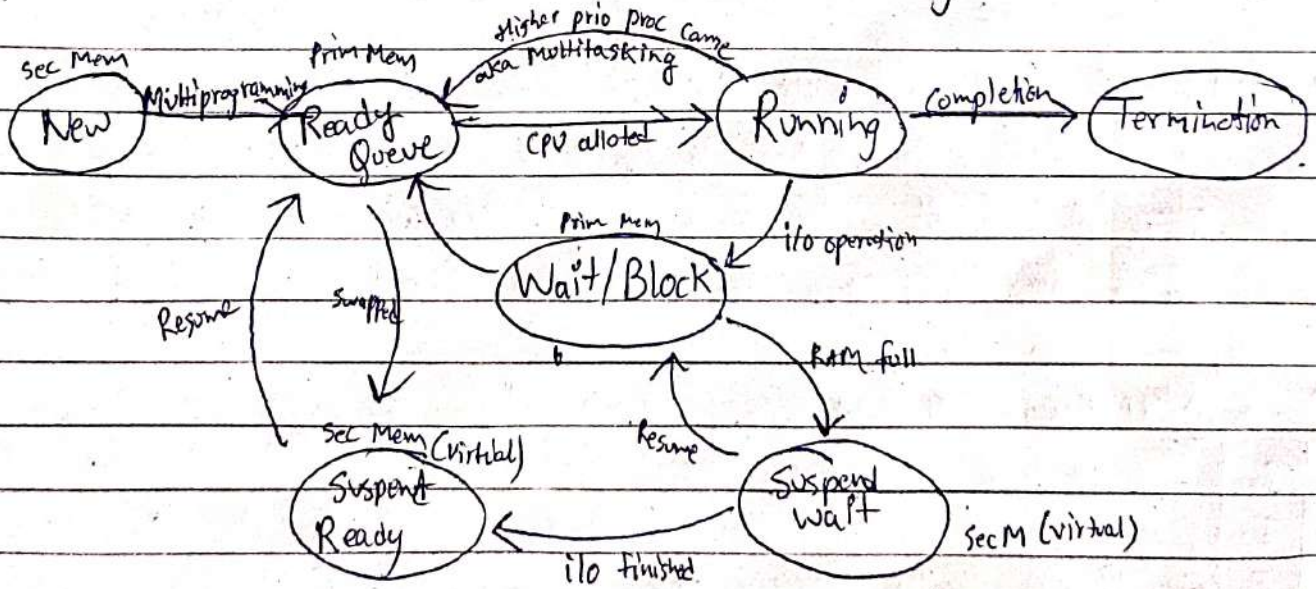
A process has to go through various states during it's execution. Order is -

1. In New State program is created & stored in 2nd ry memory (HDD or SSD)

2. In Ready State. program comes from 2nd ry memory to main memory & waiting to get CPU

3. In Running state programs (various others) are waiting in queue & @ a time only 1 process can run. So it runs here when its turn comes

4. In termination state the process completes its task & deallocates all the assigned resources



Since RAM is limited & if there are a lot of wait & block then we use 2 other states.

Scheduling: Bringing processes to CPU is K/a Scheduling. it's done by Schedulers. ie Bringing program from Sec Mem (New) to Prim Mem (Ready Queue).

slow 1) **Long Term Scheduler:** Moves from SecM to PrimM

in b/w 2) **Middle Term Scheduler:** for swapping b/w states

fast 3) **Short Term Scheduler:** Moves from PrimM (Ready Que) to Running state (CPU assigned).

Note: Amount of processes which can reside in RAM is K/a degree of multiprogramming. & Hence determined by LTS.

- STS is also K/a Dispatcher because it transfer program from Ready Q to Running.
- MTS's job is to transf proc to wait & block & if (W&B) is full then transfer it to suspend wait. & say if (Ready Q) is full MTS will transf proc to suspend Ready.
- **Interruption of process when Higher priority comes**

I_1
I_2
I_3 ← Program Counter (in PCB)
I_4

Say CPU completed only till I_2 & its now replaced by higher priority. So after higher priority process when this will run again it'll start from I_3 (given by Prog^m Counter)

Process means program under execution

Page: 7
Date:

Process Control Block (PCB): Each process has it.

It resides in MM & occupies CPU to execute instruction

- **Attributes / Context of a process :**

1) Process ID: Unique id of a process assigned by OS @ creation of process (ReadyQ)

2) Process State: It contains current state info of a process, where it's residing.

3) Program Counter: Locates the address of next instruction that CPU is supposed to execute.

4) Priority: It's a parameter assigned by OS @ a time of a process creation. eg 2,6,5

5) CPU Registers: The current info of registers must be saved so process can continue even after interruption eg: addition, multiplication, etc.

6) CPU Scheduling Information: It contains algorithm that determines how processes will be allocated to CPU.

7) Memory Management Info: It includes value of base, limit registers, page table segment tables, etc.

8) Accounting Information: It includes amount of CPU & real time used, i.e all %age in Task Manager.

9) I/O Status Info: It includes list of I/O devices allocated to the process.

- PCB: It's a repository for any information that may vary from process to process. Each process in OS is represented by PCB aka Task Control Block. It has various info related to a process aka Attributes.

Process ID
Process State
Program Counter
Registers
Memory Mgmt
Priority

PCB Diagram

Each Process in Ready Q has own PCB & each is independent of other & is deleted when process is terminated.

- PCB of a process is stored in Main Memory
- PCB " implemented using Doubly linked list.

- Multiprograming

Non-Preamptive

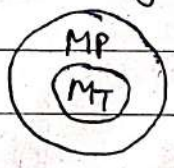
Preamptive (Multitasking) (aka Time sharing or MP with Round Robin)

Operations that can be performed on a process

1) Creation
2) Scheduling (Assigning CPU)
3) Dispatch (Move from 1 state to another)
4) Execution
5) Termination (Delete PCB)
6) Suspend
7) Resume

Important Points :

1) All Multitasking process are multiprogramming but vice versa is not true



2) When process is in Ready, Running & wait state it resides in main memory (RAM) (Unicore)

3) If resources are not sufficient to manage the processes in Ready, they're dispatched to suspended state (Suspend Ready)

4) When process is in Suspend state it resides in 2ndry memory (virtual)

5) Each & Every time a process is dispatched its context (PCB) will changes.

Q: A system has n CPUs find min & max no. of processes that can reside in Ready, Running & wait states

Soln

States	Min	Max	
Ready	0	depends	depends on size of RAM
Running	0	n	
Wait	0	depends	

Q: Repeat for system having 2 CPUs & n as degree of multiprogramming.

Soln

States	Min	Max
Ready	0	n
Running	0	2
Wait	0	n

- Processes

CPU Bound

Process which require more amount of CPU time spend more time in Running state

I/O Bound

Process that spend more time in Block & wait state & require more I/o time.

- **Schedulers :**

1) Long Term Schedulers (Job Sch) : It's responsible for creating & bringing new processes to MMem.

2) Short Term Scheduler (CPU sch) : It's responsible for selection one of the processes in Ready state for scheduling to Running state (assigns CPU)

3) Middle Term Scheduler (Swapping Job) : It's responsible for suspending & resoming the processes.

- **Dispatcher (aka STS)** : It's responsible for performing context switching & is also responsible for loading selected job onto CPU.

- **Context switching** : Saving context of one process & loading the context (or PCB) of other process is K/a Context switching.

. It's considered as **burden/Overhead** for system

CPU Scheduling : It's basis of any multiprogramming OS

There of 2 types a) Preamptive Scheduling
b) Non- "

a) **Preamptive** : Running process can be halted in middle of execution to serve higher priority process or when the running process wants to perform I/O.

8) Non-Preamptive: Running process cannot be halted & can go out only if finished

Scheduling Criteria:

1) CPU Utilization: We want to keep CPU as busy as possible
Real time sys utilizes 40-90% CPU constantly

2) Throughput: # of processes that are completed per unit time.

$$Th = \frac{\#\text{ of processes}}{\max(CT) - \min(AT)}$$

3) Turn Around Time (TAT): Total time taken by process i.e from Arrival to Completion
it involves waiting & execution everything.

4) Waiting Time (WT): It's time spend waiting in Ready Q

5) Response Time (RT): It's time @ which process is alloted CPU

6) Arrival Time (AT): It's time @ which process arrives @ Ready Q

7) Burst Time (BT): AKa Execution, it's fixed time required by a process to execute fully.

CPU Scheduling Algorithms

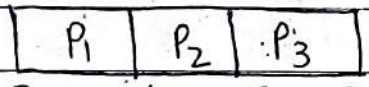
① FCFS : Criteria : AT
Mode : Non-Preemptive

Q1 :

Proc	AT	BT
P_1	0	24
P_2	1	4
P_3	2	4

Find ⟨TAT⟩ & ⟨WT⟩

Sol^M



Gantt Chart

Proc	CT	TAT	WT
P_1	24	24	0
P_2	28	27	23
P_3	32	30	26

⟨TAT⟩ = 24+27+30 / 3 = 27.

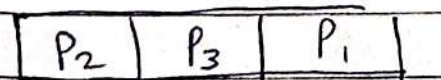
⟨WT⟩ = 0 +23+26 / 3 = 16.$\bar{3}$

Q2 :

Proc	AT	BT
P_2	0	4
P_3	1	4
P_1	2	24

Repeat :

Sol^n



Proc	TAT	WT
P_2	4	0
P_3	7	3
P_1	30	6

⟨TAT⟩ = 4+7+30 /3 = 13.6

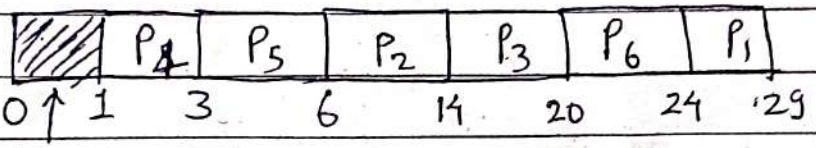
⟨WT⟩ = 0+3+6 /3 = 3

Q_3:

Proc	AT	BT
P_1	6	5
P_2	3	8
P_3	4	6
P_4	1	2
P_5	2	3
P_6	5	4

Repeat

Soln



Proc	TAT	WT
P_1	23	18
P_2	11	3
P_3	16	10
P_4	2	0
P_5	4	1
P_6	19	15

$\langle TAT \rangle = 12.5$

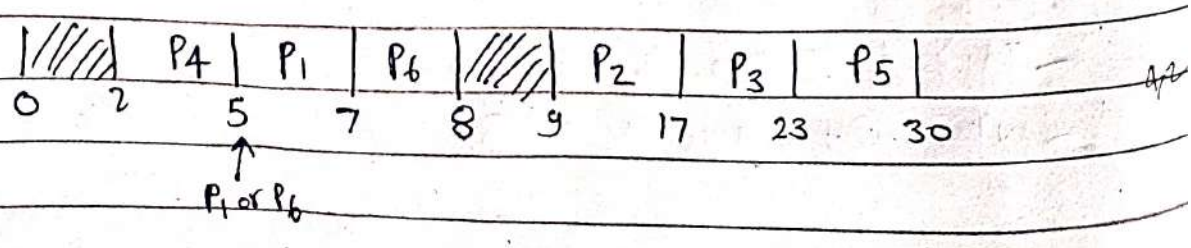
$\langle WT \rangle = 7.83\overline{3}$

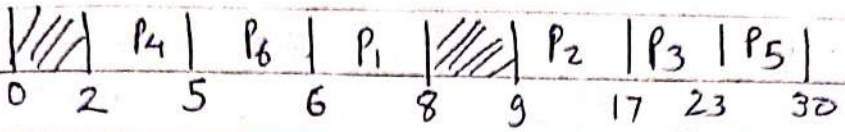
Q_4

Proc	AT	BT
P_1	3	2
P_2	9	8
P_3	12	6
P_4	2	3
P_5	15	7
P_6	3	1

Repeat

Soln



OR 

2, 3

Answer will Not be same. pick one & stick to it

Proc	TAT	WT
P_1	4 or 5	2 or 3
P_2	8	0
P_3	11	5
P_4	3	0
P_5	15	8
P_6	5 or 3	4 or 2

$\langle TAT \rangle = 7.\bar{6}$ or 7.5

$\langle WT \rangle = 3.1\bar{6}$ or 3

Note: We try to minimize WT.

Problems in FCFS:

Convoy Effect: All process wait for one big process to get off the CPU. Its different from starvation as we know the fixed and defined time on how much we'll wait unlike undefined time in starvation.

(2) Shortest Job First: This algo is also K/a Shortest next CPU Burst algorithm.

Criteria = BT

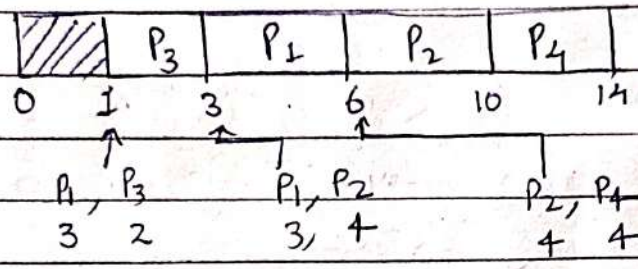
Mode: Non-preemptive.

Q1:

Proc	AT	BT
P_1	1	3
P_2	2	4
P_3	1	2
P_4	4	4

Use SJF algo to find <TAT> & <WT>

Soln



(pick any, ans will be same)

	TAT	WT
P_1	5	2
P_2	8	4
P_3	2	0
P_4	10	6

<TAT> = 6.25

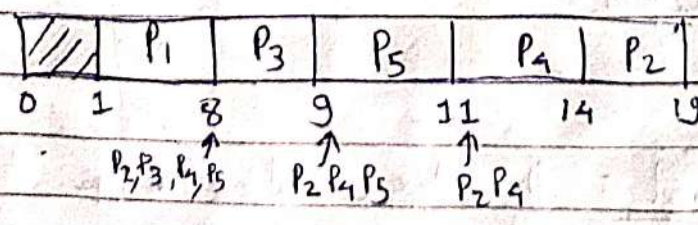
<WT> = 3

Q2:

Proc	AT	BT
P_1	1	7
P_2	2	5
P_3	3	1
P_4	4	3
P_5	5	2

Repeat

Soln



	TAT	WT
P_1	7	0
P_2	17	12
P_3	6	5
P_4	10	7
P_5	6	4

<TAT> = 9.2

<WT> = 5.6

Limitations:

1) Scheduler can't know length of next CPU burst time unless it arrives.

2) We cannot stop a process in-between (Non-P)

(3) Shortest Remaining Time First:

Criteria: BT

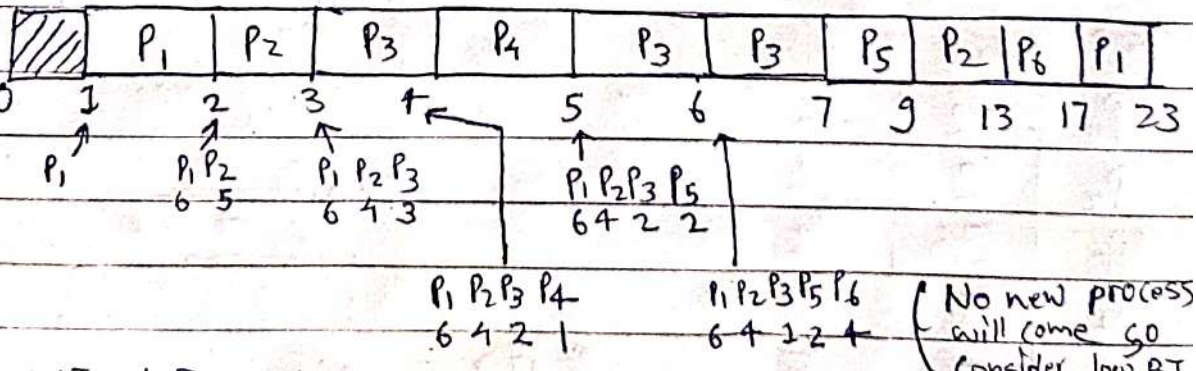
Mode: Preamptive (overcomes drawback of SJF)

Q1:

Proc	AT	BT
P_1	1	7
P_2	2	5
P_3	3	3
P_4	4	1
P_5	5	2
P_6	6	4

Use SRTF & find <TAT> & <WT>

Soln



	TAT	WT
P_1	22	15
P_2	11	6
P_3	4	1
P_4	1	0
P_5	4	2
P_6	11	7

$\langle TAT \rangle = 8.8\overline{3}$

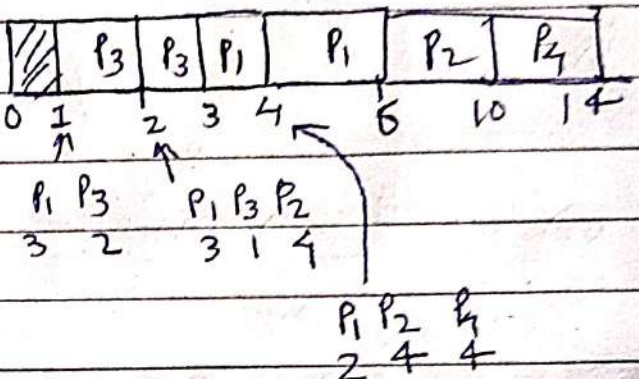
$\langle WT \rangle = 5.1\overline{6}$

Q2:

Proc	AT	BT
P1	1	3
P2	2	4
P3	1	2
P4	4	4

Repeat

Soln



	TAT	WT
P1	5	2
P2	8	4
P3	2	0
P4	10	6

<TAT> = 6.25

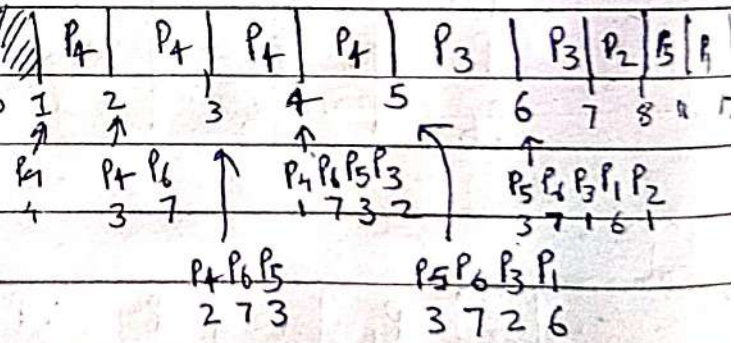
<WT> = 3

Q3:

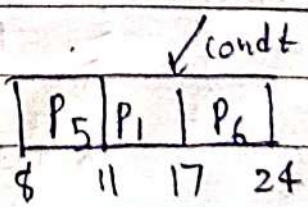
Proc	AT	BT
P1	5	6
P2	6	1
P3	4	2
P4	1	4
P5	3	3
P6	2	7

Repeat

Soln



	TAT	WT
P1	12	6
P2	2	1
P3	3	1
P4	4	0
P5	8	5
P6	22	15

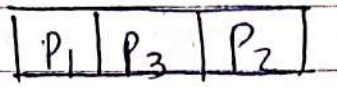


<TAT> = 8.5

<WT> = 4.67

Q_4 : Consider P_1 P_2 & P_3 scheduled by SRTF. P_1 scheduled first & has been running for 7s, P_3 arrived & ran for 1s. then P_3 occured & completed execution in 2s. What would be min BT of P_1 & P_3?

Soln / Gantt →



- P_3 was selected over P_1 ⇒ Remaining of P_1 > ~~BT~~ of P_3
- P_2 was selected over P_3 ⇒ Remaining of P_3 > BT of P_2
  Remaining of P_3 > 2s
  i.e Remaining of P_3 = 3s

BT of P_3 = 1s + 3s = 4s

Remaining of P_1 > 4s
i.e Remaining of P_1 = 5s

BT of P_1 = 7s + 5s = 12s ⇒

	BT
P_1	≥12
P_2	2
P_3	≥4

(4) Longest Job First : Criteria : BT
Mode : Non-preamptive

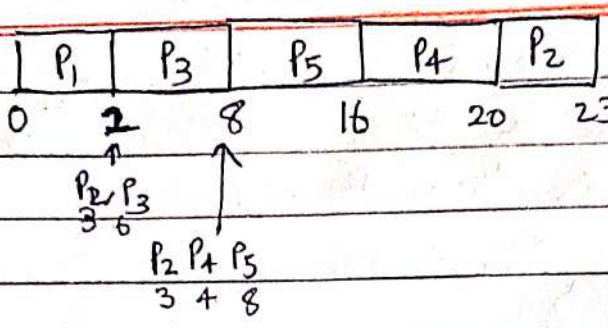
Q:

Proc	AT	BT
P_1	0	2
P_2	1	3
P_3	2	6
P_4	3	4
P_5	4	8

Find <TAT> & <WT> using LJF.

20

Soln

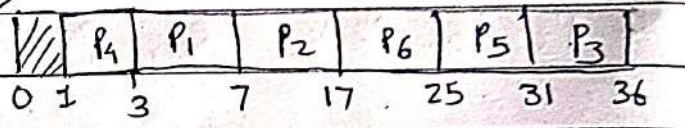


	TAT	WT
P_1	2	0
P_2	22	19
P_3	6	0
P_4	17	13
P_5	12	4
	<> = 11.8	<> = 7.2

Q_2:

Proc	AT	BT
P_1	3	4
P_2	7	10
P_3	4	5
P_4	1	2
P_5	6	6
P_6	5	8

Soln



	TAT	WT
P_1	4	0
P_2	10	0
P_3	32	27
P_4	2	0
P_5	25	19
P_6	20	12

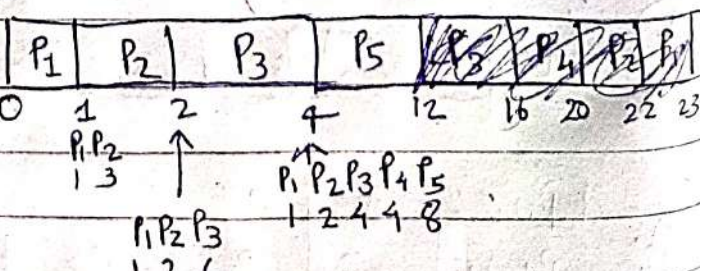
Note: If BT is same check AT

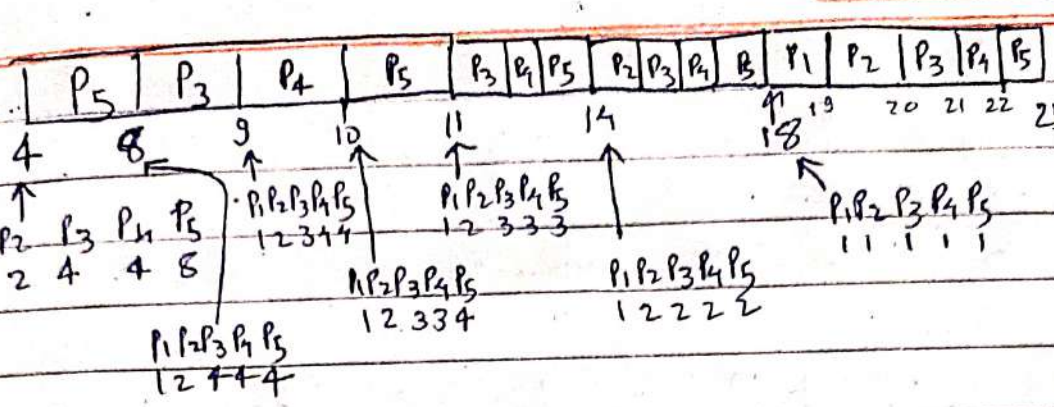
(5) Longest Remaining Time First! Criteria: BT Mode:- Preamptive

Q_1:

Proc	AT	BT
P_1	0	2
P_2	1	3
P_3	2	6
P_4	3	4
P_5	4	8

Soln





	TAT	WT
P1	19	17
P2	19	16
P3	19	13
P4	19	15
P5	19	11

$\langle TAT \rangle = 19$

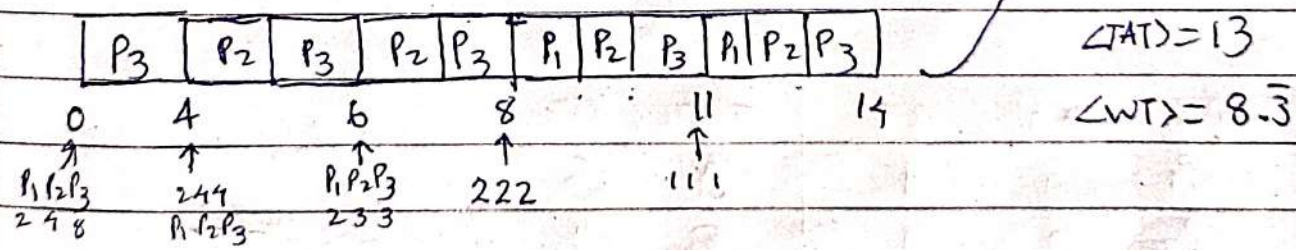
$\langle WT \rangle = 14.4$

Q2:

Proc	AT	BT
P1	0	12
P2	0	4
P3	0	8

Soln

Proc	TAT	WT
P1	12	10
P2	13	9
P3	14	6



$\langle TAT \rangle = 13$

$\langle WT \rangle = 8.\bar{3}$

(6) Round Robin Scheduling: Criteria → AT (tq) Mode → Preamptive

- Specially designed for time sharing systems.
- It is similar to FCFS but with preamptive mode (swith b/w processes)

- Time slice / Quantum is defined here, generally 10 to 100ms

- Most OS follow this algorithm. (Gives illusion of multitasking)

Q1:

Proc	AT (ms)	BT (ms)
P_1	0	24
P_2	1	3
P_3	2	3

$t_q = 4ms$

Sol^n

P_1	P_2	P_3	P_1
0	4	7	10	30

→

Proc	TAT	WT
P_1	30	6
P_2	6	3
P_3	8	5

<> = 14.6 ms <> = 4.6 ms

Q2: 4 processes P_{1-4} in Readq @ t=0, with BT 4,1,8,1. What is CT of P_1. Assume RR algo with tq 1ms.

Sol^n

Proc	AT	BT	TAT	WT
P_1	0	4	9	5
P_2	0	1	2	1
P_3	0	8	14	6
P_4	0	1	4	3

=> , CT of P_1 = 9ms

P_1	P_2	P_3	P_4	P_1	P_3	P_1	P_3	P_1	P_3	P_3
0	1	2	3	4		6		8	10	14

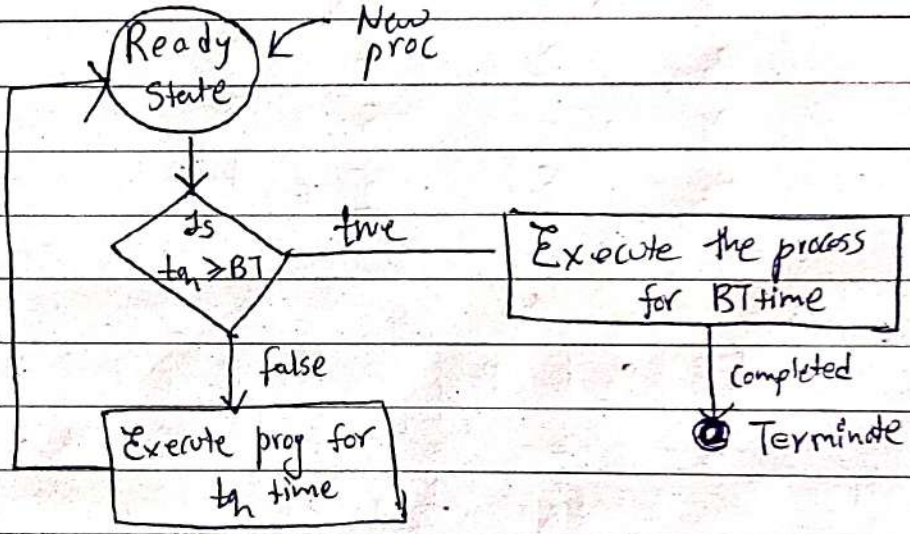
P_2 & P_4 done

Important Points

1) In RR if t_q is small, we've to do lot of context switching rapidly which is high burden but lot more responsive system.

2) If t_q is large vice versa.

3) If t_q is $\geq$ max(BT) than algo behaves like FCFS



⑦ Priority Scheduling:

Criteria → Priority
Mode → Can be both

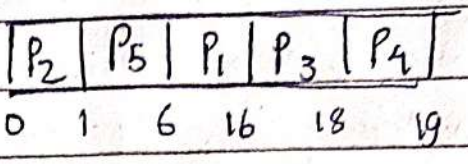
- Special case of Shortest Job first more like important job first.
- If priority is same serve like FCFS.
- If AT is not given assume all ATs to be 0.
- Lower the priority integer the more important it is.

Q1:

Proc	BT	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Sol^n

Non-P

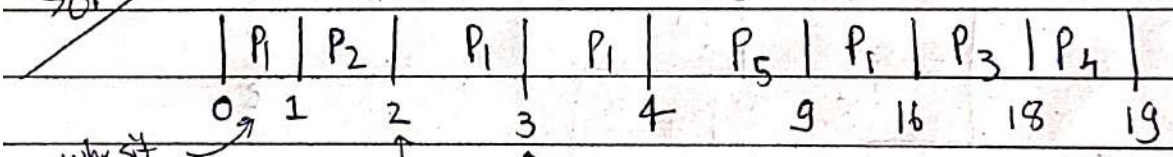


Q2:

Proc	BT	Priority	AT
P_1	10	3	0
P_2	1	1	1
P_3	2	4	2
P_4	1	5	3
P_5	5	2	4

Find <TAT> & <WT> using Preamptive priority Scheduling

Sol^n



why sit idle? Id

P_1 P_3

BT 9 2

Prio ③ ④

P_1 P_3 P_4

③ ④ ⑤

8 2 1

P_1 P_3 P_4 P_5

③ ④ ⑤ ②

9 2 1 5

(Now complete by Priority)

	TAT	WT
P_1	16	6
P_2	1	0
P_3	16	14
P_4	16	15
P_5	5	0

<TAT> = 10.8 , <WT> = 7

Disadvantage:

1) If higher priority process keep on comming, other lower already arrived priority processes will have to wait indefinitely aka Starvation.

Solution → Gradually increase the priority of processes that wait in a system for long time this will avoid starvation & this technique is k/a Aging.

(8) Multilevel Queue Scheduling:

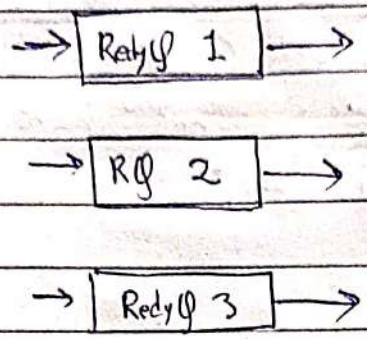
It's well suited for situations where processes can be classified into different groups so common division made is b/w foreground (interactive) & background (batch) processes. These 2 types of processes have different response time & scheduling needs.

It partitions the Ready Q into several seperate queues

eg: | 1 | 2 | 3 | 4 | : Data Mining | 16 | 17 | 18 | : Comp Network | 5 | 6 | 7 | 8 | 12 | 69 | : IIIrd processing

Batches (group) →
- System Processes → Highest Priority
- Interactive Processes
- Interactive Editing Processes
- Batch Processes
- Student Processes → Lowest

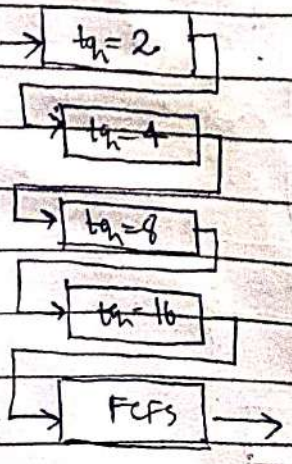
Division of Ready Q



Queue with highest priority will execute.

- Depending on the priority of processes, it is decided in which Ready Q the process will be placed.
- In highest priority process will be placed in the top level Ready Q & lowest priority @ bottom
- Only after completion of all processes in top level Ready Q the further level Ready Q process will be scheduled.
- Bottom level Ready Q will suffer from starvation
- To avoid starvation we can use: 1) Aging
  2) Round Robin
  3) 60% time to highest, 30%, 10%

(9) Multilevel Feedback Scheduling: 1 process can move b/w various Ready Q



Criteria: BT (lower BT ⇒ higher priority)

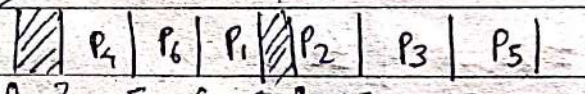
Idea is to separate proc according to their BT

Algorithms	Starvation	
FCFS	✗	
SJF	✓	
SRTF	✓	Use Aging to avoid Starvation.
LJF	✓	
LRTF	✗	
RR	✗	
Priority (NonP)	✓	
Multilvl Q	✓	
Multilvl FB	✓	

Q: Use FCFS algo to find CPU Idle time & Throughput.

Proc	AT	BT
P_1	3	2
P_2	9	6
P_3	12	8
P_4	2	3
P_5	15	9
P_6	3	1

Sol^n



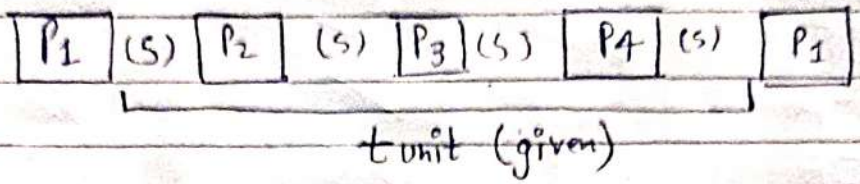
$$Th = \frac{6}{32-2} = \frac{1}{5} = 20\%$$

CPU Idle time = 3

Q: Consider n processes sharing CPU in RR algo, context switching time is s unit. What must be t_q such that # of context switches are reduced but @ same time each process is guaranteed to get its turn, after every t unit time.

a) $t_q \leq \frac{t-ns}{n+1}$ b) $t_q \geq \frac{t+ns}{n-1}$ c) $t_q \leq \frac{t-ns}{n-1}$ d) $t_q \geq \frac{t+ns}{n+1}$

Consider 4 processes, Each process ran for t_q unit



=> 4s i.e ns for n processes

=> 3 processes in between i.e $(n-1)t_q$ time consumed for n processes

$$t = (n-1)t_q + ns$$

$$\text{or } t_q = \frac{t-ns}{n-1}$$

if t_q is too large => FCFS
& FCFS => Convoy effect
Convoy effects => Avoids guarantee for each processes getting a chance

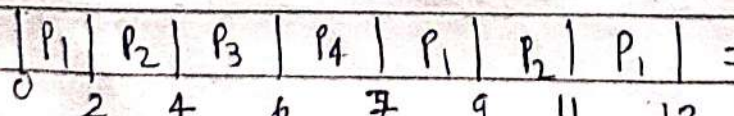
$$\Rightarrow t_q \leq \frac{t-ns}{n-1}$$ Ans

Q:

Proc	AT	BT
P_1	0	5
P_2	1	4
P_3	2	2
P_4	4	1

$t_q = 2$, using RR find <TAT>, <WT> & <RT>
Also find # of context switching

Soln



	TAT	WT	RT
P_1	12	7	0
P_2	10	6	1
P_3	4	2	2
P_4	3	2	2

<TAT> = 7.25
<WT> = 4.25
<RT> = 1.25
# of context switching = 6

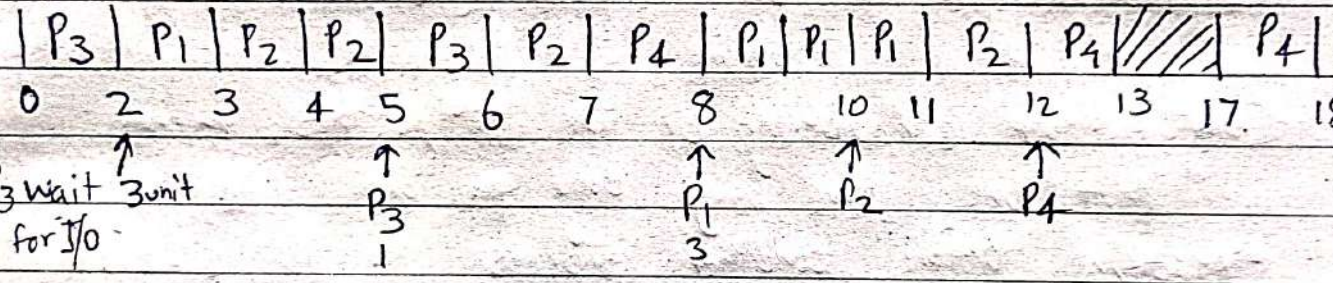
(Low = high priority)

Q:

Proc	Priority	CPU	I/O	CPU
P_1	2	1	5	3
P_2	3	3	3	1
P_3	1	2	3	1
P_4	4	2	4	1

Using preemptive priority sch to find CT, TAT of all processes

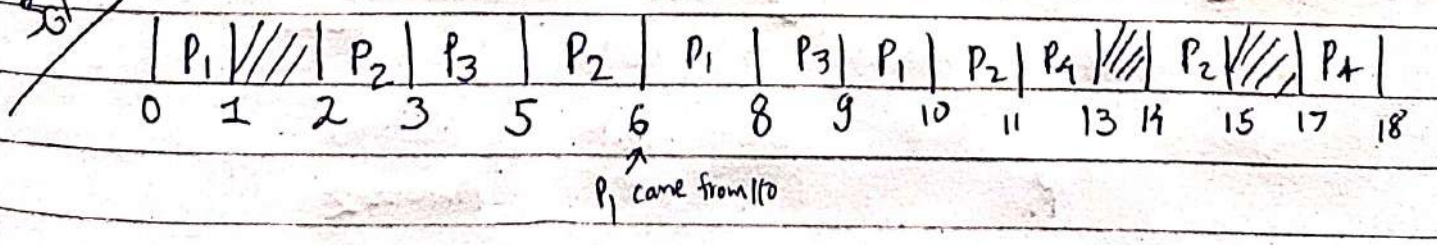
Soln All arrived @ AT=0.



Q:

Proc	AT	Prio	CPU	I/O	CPU	Repeat
P_1	0	2	1	5	3	
P_2	2	3	3	3	1	
P_3	3	1	2	3	1	
P_4	3	4	2	4	1	

Soln



Synchronization : Processes are categorized into 2 categories

1) Cooperative processes → Processes that are dependent on one another & can affect each other. (Synchronization is needed among them)

2) Independent Processes

- The execution of 1 proc affects other process.
- Cooperative process share data & code
- Can execute concurrently or in ||el, this means that one proc may only partially complete execution before another process is scheduled.
- Problem arises if cooperative proc are not synchronized.

eg : Producer Consumer problem.

They share resources → Variable, Memory, Devices, Code

P1		Process 2
int x = shared;		int y = shared;
x++;		y--;
sleep(1);	(Run ||ely)	sleep(1);
shared = x;		shared = y;

& initially int shared = 5.

Constraint → Uniprocessor

- Say P_1 gets CPU first => finally shared = 4

sleep pauses that process, CPU does not remain idle & will do context switching to P_2

- In these P_1 & P_2 we were adding & subtracting 1 from 5 so answer should remain 5 but it's NOT because these 11 copperative processes are NOT synchronized

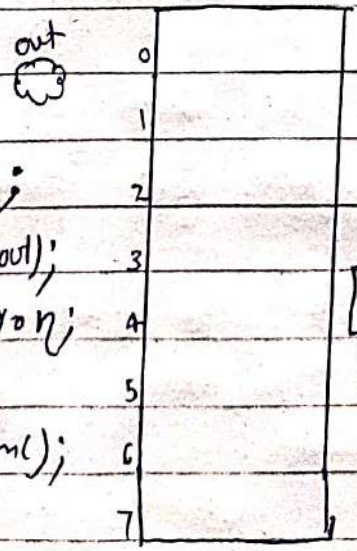
- Say P_2 gets CPU first => final shared = 6
still wrong.

This problem is called Race Condition
6 & 4 are racing (both wrong tho)
many laptops will give 4 as answer & many 6.

Producer – Consumer Problem

```
Void consumer(){
  int itemc;
  while(true){
    while(count==0);
    itemc = Buffer[out];
    out=(out+1)%n;
    count--;
    Process_item(itemc);
  }
}
```

```
n=8; count=0;
```




```
Void producer(){
  int itemp;
  while(true){
    produce_item(itemp);
    while(count==n);
    Buffer[in] = itemp;
    in=(in+1)%n;
    count++;
  }
}
```

Local

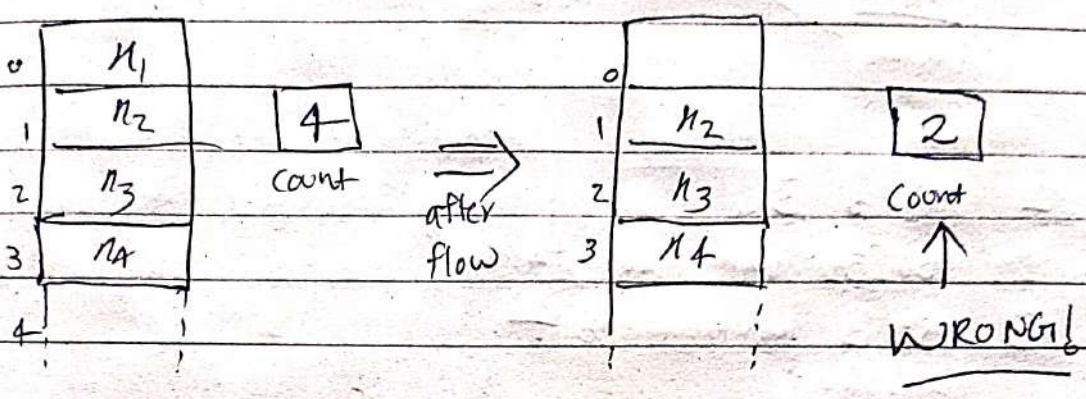
Buffer
(Goods Storage)

Producer produces goods & puts in buffer, & count++
Consumer consumes goods & takes out from buffer & count--

- Count is shared variable. also the whole Buffer
- If Buffer is full (count==n) then while will go in ∞ waiting. for Producer
- If Buffer is empty (count==0) then while will go in ∞ waiting for consumer.
- Count--; is one instn but CPU converts em in micro instruction. => load Rc, memory[count]
  DECR Rc;
  Store memory[count], Rc
- Count++; => load Rp, mem[count] Instruction I_1
  INCR Rp; I_2
  Store mem[count], Rp I_3
- Problem: When process gets preampted after Increasing / Decreasing value of register & failed to store it in memory, ie load Rp, m[count]
  INCR Rp
  stop → store m[count], Rp
  & other cooperative process got executed.
- It's possible to get interrupted any time by any H/w, S/w interrupt or higher priority process came or tq expired, etc.

Flow → Producer (I_1, I_2) (halted) → Consumer (I_1, I_2) (halted) → Producer (I_3) (Resumed & finished) → Consumer (I_3) (Resumed & finished)

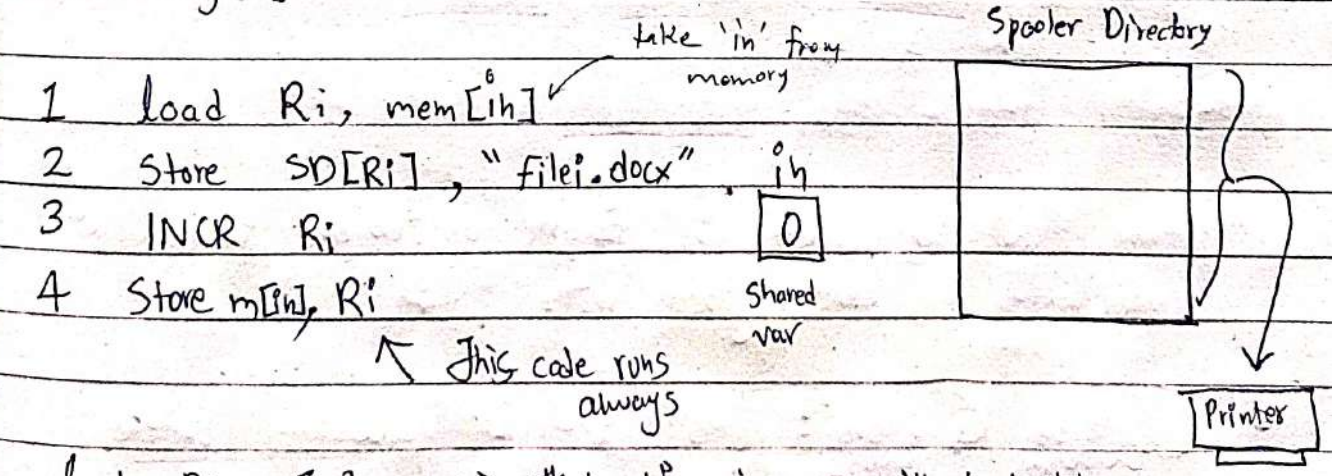
eg:



- This Race condt can be removed using Semaphores, Locks etc

Printer - Spooler Problem:

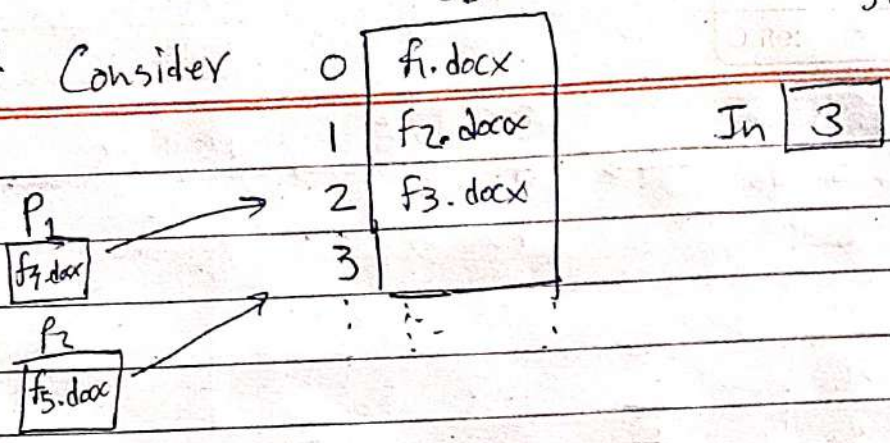
We know printer is a slow device, so when it's connected in a network & lot of PC want to print documents they go in Spooler directory (acts as buffer) & spooler picks 'em 1 by 1.



load Ri, m[in] => 0^{th} location in mem will be loaded

SD[Ri], "filei.docx" => Ri^{th} register location will go to spooler directory (s) & that'll store file1.docx

Problem: Consider



P1 1. R_1 is 3
2. SD[3] will have "f4.docx"
3. R_1 is increased to 4.
Halted!

P2 1. R_2 is 3
2. SD[3] will have "f5.docx" => f4.docx lost!
3. R_2 is increased to 4
Halted!

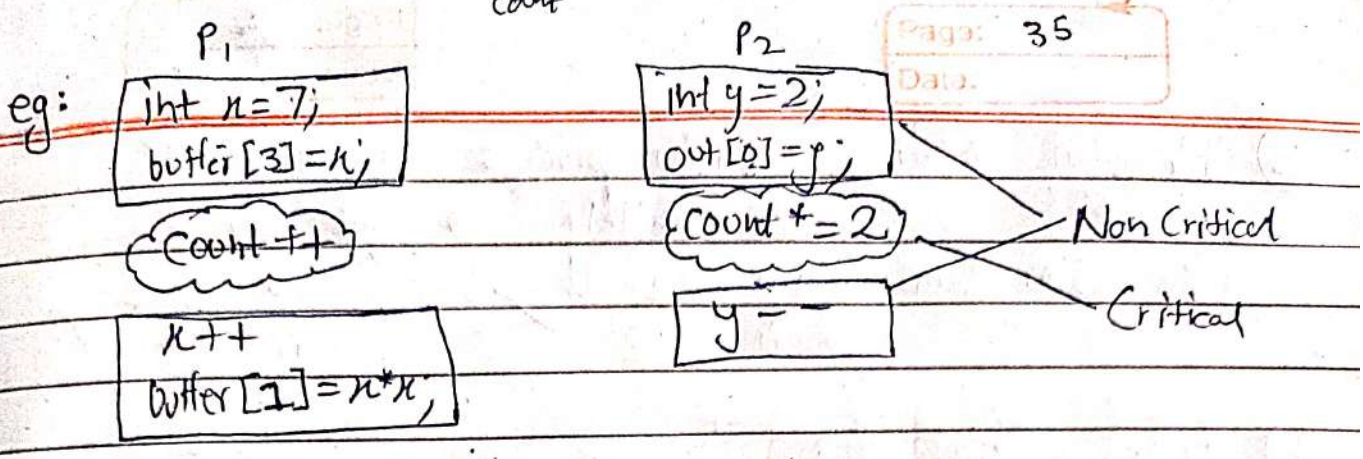
P_1 (Resume) in is 4
Terminated!

P_2 (resume) in is 4
Terminated.

Hence, due to lack of Synchronization Race Condt occured & data got lost.

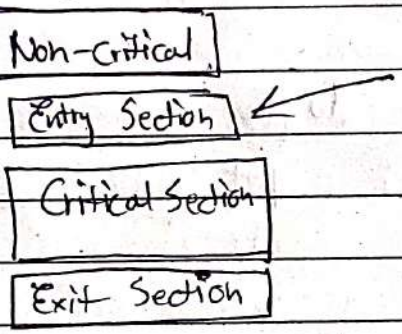
- **Critical Section (CS):** It's part of program where the shared resources are located & accessed by other cooperative processes. Other part is Kla Non-Critical section.

count



- To avoid Race condt we add Entry Section just before entring Critical section like a real life lock

i.e



can be semaphore, locks, etc.

- Conditions to achieve Synchronization :

1) Mutual Exclusion 2) Progress 3) Bounded Wait 4) Hardware independent

Mandatory ie Primary Rule

(1) Mutual exclusion : If one process is operating on CS then no other process should enter CS, this pheromenon is K/a mutual Exclusion.

(2) Progress : Making sure that any other process cannot prohibhit the interested process i.e the process that wants to enter CS.

eg: Real life society.

③ <u>Bounded</u> <u>Wait</u>: No process should wait indefinitely to get into CS like starvation. 10 (P_1) 1 (P_2) ✓ , ∞ (P_1) 0 or 1 (P_2) ✗

④ <u>Hardware</u> <u>Independent</u>: Should not be limited to a particular hardware like only works on 64-bit or ≥3 GHz processor or only Linux. It should be universal.

• <u>Lock</u> <u>Variable</u>:

```
do {
    put lock        ← Entry Code
    [CS]
    release lock    ← Exit Code
}
```

```
do {
  I1 while (lock==1);   ⇐
  I2 lock=1;
  I3 [CS]
  I4 lock=0;
}
```

Lock [0] ← Initially
Shared

It works good when processes come 1 by 1

• Say (lock=0) initially

P1	→	P2	→	P1 (resume)	
I_1		I_1, I_2, (I_3)		I_2, (I_3)	Both in CS
Halted!		Halted!			
lock=0		lock=1		lock=1	

⇒ No gurantee of Mutual Exclusion.

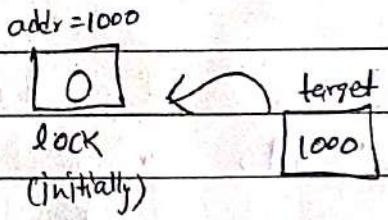
• Test_&_set :

Lock variable solution failed because in b/w checking while(lock==1) { lock=1. system can get preampted. But here we combine them into 1 ✓ instruction using hardware modification (A dedicated processor opcode), "xchg" in Intel x86 is used for it.

Software Representation →

```
bool test_and_set (bool *target) {
    bool r = *target;
    *target = 1;          ← can't be preampted here
    return r;
}
```

```
do {
    while (test_and_set (&lock));
    [CS]
    lock = false;
}
```



=> Gurantee of Mutual Exclusion & no other code it there that can stop progress
Hence solution of Race condt aka Sync².

• Turn Variable :

1. Works only on 2 processes
2. Runs @ User mode, OS independent

P_0	P_1
Entry → while(turn != 0);	while(turn != 1);
[CS]	[CS]
Exit → turn = 1;	turn = 0;

int turn
0 or 1

Say initially turn = 0.

=> P_0 can go in CS & when it's in CS P_1 cannot.
Now P_0 is done => turn = 1

=> P_1 can go in CS & when it's there P_0 cannot.

- No matter where preamption occurs both cannot go in CS
  => Mutual Exclusion ✓

- Say initially turn = 1 & P_0 wants to go in CS
  (CS is empty ofc) it's not possible
  i.e P_1 is stopping P_0 from going into CS
  => Progress ✗

- It's either P_0 then P_1 then P_0 then P_1 ...
  OR P_1 then P_0 then P_1 then P_0 hence same process cannot go again & again while other starves.
  => Bounded Wait ✓
  (That's why it's K/a Strict Alternation)

- User Mode => Hardware Independent ✓

System Calls

- User Mode → Accesing APIs, writing programs, etc.

Kernal Mode → Accessing functionality of OS.

- User Mode ⟵ System Call ⟶ Kernal Mode

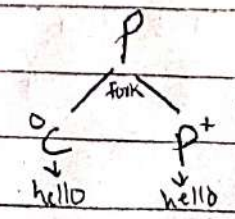
- printf() prints on monitor, file.write() modifies file on secondary memory. So we actualy write code in user mode & these fuction perform sys calls & tells OS to print msg on monitor or delete file on drive, etc.

"System Calls" (win7 ≈ 700 calls, linux ≈ 300s, some have no call ⇒ Use API)

- → File Related : open(), read(), write(), close(), ~~create~~file, etc. (lseek, rseek());
- → Device Related : Read, Write, Reposition, ioctl, fcntl
- → Information : getPID, attributes, get SysTimeAndDate
- → Process Control : Load, Execute, abort, fork, Wait, Allocate, etc.
- → Communication : Pipe, create/delete connections, shmget() (get shared memory)
- → Security : chmod +x, etc.

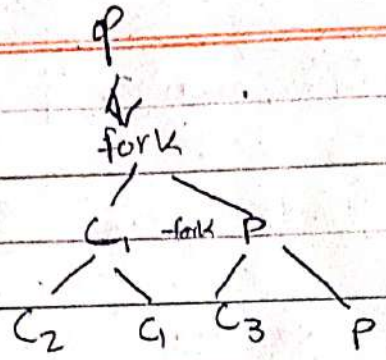
Fork System Call : Process creates its child process & both work ||ly to achieve multithreading
Fork()

- Fork() → 0 : Child process
  - 1 : Parent process
  - -1 : Child process failed to create.

eg:
```
int main(){
    Fork();
    printf("Hello");
}
```
⇒ 

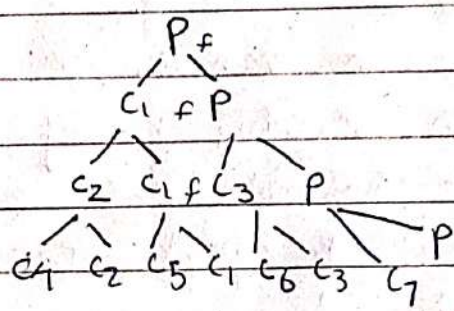
⇒ o/p
Hello
Hello

Q: main(){
fork();
fork();
Printf("Joe");
}



O/P: Joe
Joe
Joe
Joe

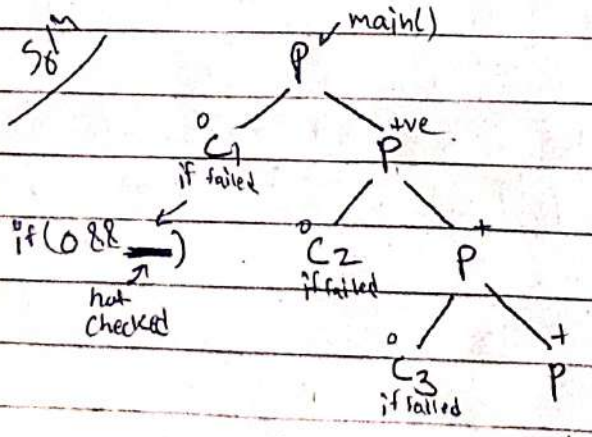
Now if 3 fork(); →



7 child, 1 parent
$2^n - 1$ 1

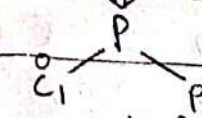
Q: #include <stdio.h>
#include <unistd.h>
int main(){
if (fork() && fork())
fork();
printf("Hello");
return 0;
}

Find O/P.



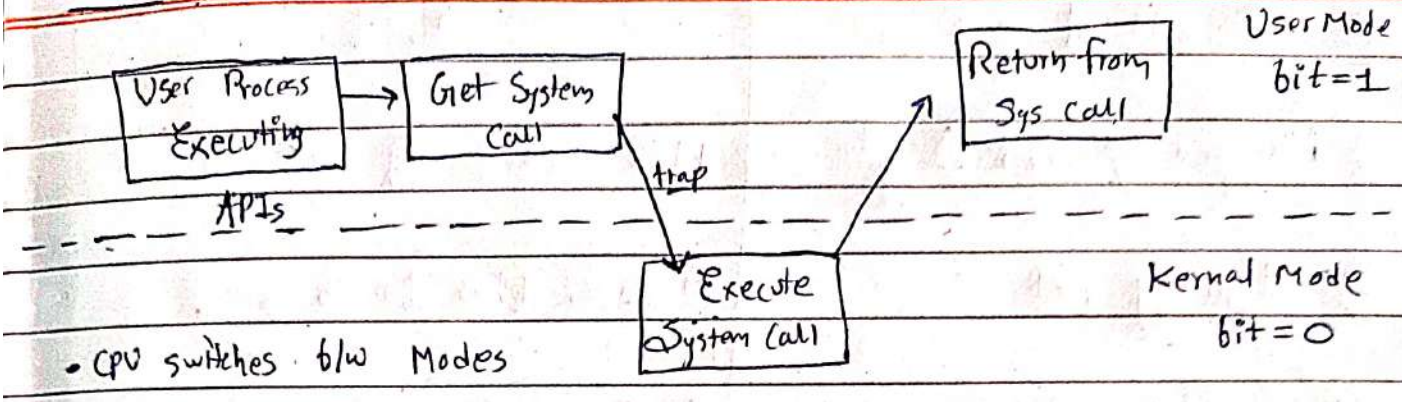
O/p → 4 Hello

• if (fork() ....)



Now this fork() can be 0 or 1
C1 P
if we put 0 if(0 && ...) will always be false no need to check other fork() too & moment we execute it P will be forked.
& so on....

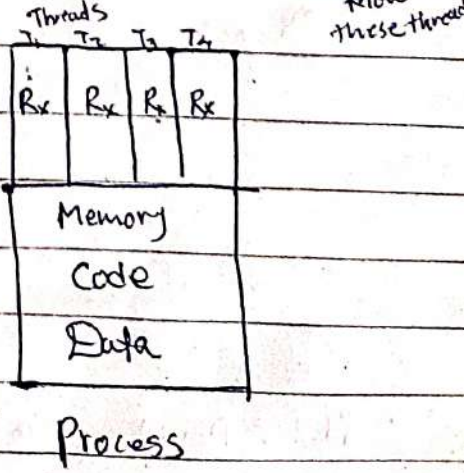
User Mode v/s Kernal Mode



- CPU switches b/w Modes

Process	Threads
1) Sys calls involved in process	1) There is no system call involved
2) OS treats diff process differently	2) All user level threads as single task
3) Diff process have diff copies of Data, files, code	3) Threads share same copy of code & data
4) Context switching is slower	4) Almost no switching (same Resources)
5) Blocking a process will not other	5) Blocking a process/thread will block the whole process (coz Kernal doesn't know about these threads)
6) Independent	6) Inter dependent.

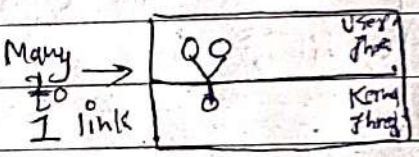
- This threading can be achieved by 2 ways
  - User level
  - Kernel level



User level Thread	Kernel level Thread
1) Managed by User (User lvl library)	1) Managed by OS (Sys calls)
2) Typically fast	2) Slower (has to do sys calling)
3) Context switching is faster	3) slower (")
4) If 1 User lvl thread performs I/O whole process gets blocked	4) If 1 kernel lvl thread blocked, No affect on others

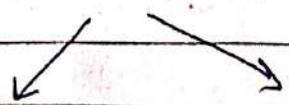
But in reality there's hybrid system. (link blw K lvl & User lvl)

- 1 to 1 link in Linux



Semaphores: Methodology to prevent Race condt & achieve sync. It's an integer variable which is used in mutual exclusive manner by various Concurrent Cooperative Processes

They're of 2 types



Counting	Binary
$(-\infty$ to $+\infty)$	[0,1]

→ Pela gye

- P(), Down, Wait ← Entry
- V(), Up, Signal, Post, Release ← Exit

↓ Victory

Entry Section

```
Down (Semaphore S){
    S.value = S.value - 1;
    if (S.value < 0){
        put process (PCB) in suspend
        sleep();
    }
    else return; //← Execute
}
```

Exit Section

```
Up (Semaphore S){
    S.value ++;
    if (S.value ≤ 0){
        Select process from suspend list
        wakeup();
    }
}
```

flow S is shared semaphore value say S=3 initially

P_1 → [Entry] → P_2 comes → [Entry] → P_3 comes → [Entry] → P_4 comes → [Entry]

P_1: S=2, Execute, [P_1] CS

P_2: S=1, Execute, [P_1 P_2] CS

P_3: S=0, Execute, [P_1 P_2 P_3] CS

P_4: S=-1, Suspended

P_5 comes → [Entry] S=-2, Suspended

Hence, initial value of S determines # of processes that can come in CS. So for mutual exclusiveness S=1.

Suspended list → [P_4 | P_5]

Q: If S=-4. What does this mean?

Soln/ If S=-4 this mean 4 processes are in suspended list

P_1 goes out → [Exit code] S=-1 → P_2 goes out → [Exit] S=0 → P_3 goes out [Exit] S=1

P_4 comes from sus list (FCFS)

(P_4 they're not directly put in CS but first wakeup put in Ready Q)

P_5 comes at sus list & can compete (try) to come in CS

Q: What does S=0 mean?

Sol^n It means 0 processes are in [redacted] & Suspended list

Q: Repeat for S=10.

Sol^n It mean from this point 10 more processes can come in CS

Note: A process is call successful if it was able to go inside CS.

Q: Sequentially 6P & 4V operations were performed. If S=10 & initially find final S.

Sol^n 10 → 6 + 4 = 8 Ans

Q: Repeat if S=17, 5P, 3V, 1P

Sol^n 17 − 5 + 3 − 1 = 14 Ans

Binary Semaphore

0 1

More popular & easir than Counting Semaphore

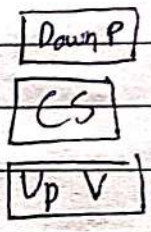
Entry

```
Down (Semaphore S){
  if (S.value == 1)
      S.value = 0;   // Executes
  else {
      Block{ put in suspend list
      sleep();
  }
}
```

- If S=1, ⇒ Process coming will be successful (i.e CS is empty)

Exit

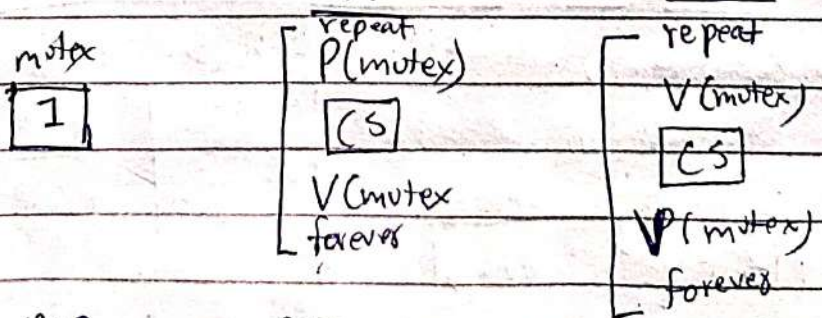
```
Up (Semaphore S){
  if (Suspend list Empty)
      S.value = 1;
  else {
      Select Process from sus list
      wake Up ();
  }
}
```



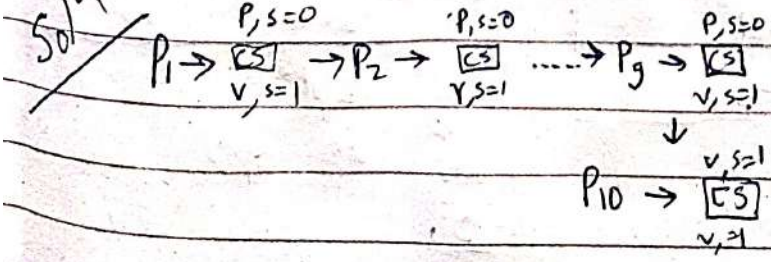
- Initially S should be 1,

Note: These 2 functions Down & Up are atomic in nature & CPU cannot preempt them in b/w.

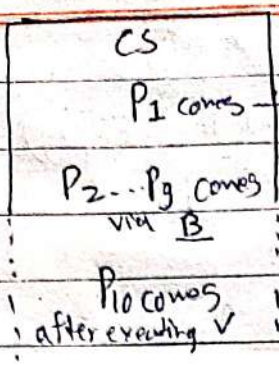
Q: $P_1 \ldots P_9$ executes A & P_{10} executes B what is max # of processes that may be present in CS @ any point of time? Assume Binary Semaphore



Soln



But this is seq verdy flow



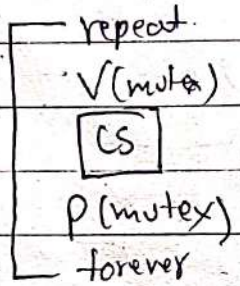
$\Rightarrow$ S=0 (P_2..P_9 all suspended) & P_{10} comes $\Rightarrow$ V

Select processes from sus list & assume from Ready if all came in CS

$\Rightarrow$ 10 Ans

P_0 is coming & going from CS $\Rightarrow$ P_2..P_9 constantly being selected.

Q: Repeat for code B



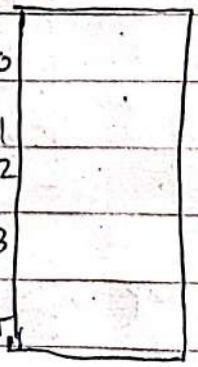
Soln: P_1 in CS (S=0) then after all P_2..P_9 suspended.
P_{10} executes (V (in B)) then P_2 will come in CS also P_{10} will come in CS & if P_{10} goes out of CS it'll be blocked.

$\Rightarrow$ 3 Ans

Solution of Producer-Consumer prob:

```
Producer (itemp){
1) Down (empty);
2) Down (S);
   Buffer[in] = itemp;
   In = (In+1) % n;
3) Up (S)
4) Up (full);
}
```

n=5



in: 0

```
Consumer (itemC){
1) Down (full);
2) Down (S)
   itemC = Buffer[out];
   out = (out+1) % n;
3) Up (S);
4) Up (empty);
}
```

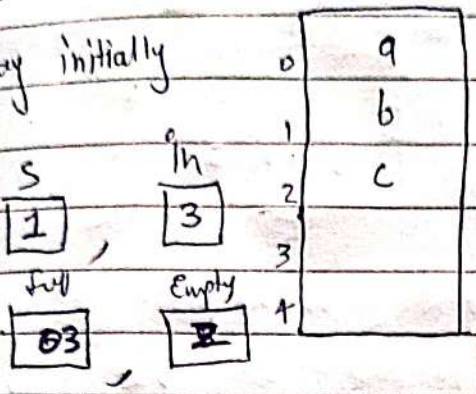
out: 0

Counting Semaphore, full = 0, Empty = n

Binary Semaphore, S = 1

Case I : Sequential flow (No preamption)

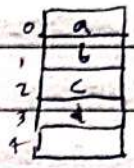
Say initially



Now we want to

Produce => 1) Down (empty)
i.e empty = 1 (one)

2) Down (S)
i.e S = 0



← CS i.e in = 4

3) Up (S) i.e S = 1

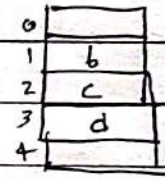
4) Up (full) i.e full = 4

Now we want to consume.

.. CS

Consumer => 1) Down (full) 2) Down (S) out = 1 3) Up (S) 4) Up (Empty)

i.e full = 3 i.e S = 0 i.e S = 1 ie Empty = 2



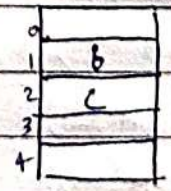
- There's Consistency ∀ full + empty = 5 (n)

Case II Preamption befor 2)

We want to produce,

Producer => 1) Down (empty) Halted!
ie Empty = 1

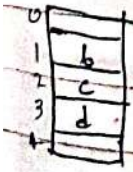
Consumer => 1) Down (full) 2) Down (S)
ie full = 2 i.e S = 0



& out = 1 CS

Producer Resumes =>

2) Down (S) i.e S = 0



& in = 4 CS

3) Up (S) i.e S = 1

4) Up (full) i.e full = 3

3) Up (S) i.e S = 1

4) Up (Empty) i.e empty = 2

Producer Resumes

Consistency Remained!

Hence, Sync^z was achieved because only 1 prog was there in CS @ a time this was guranteed by the binary semaphore S.

Solution of Reader-Writer Problem:

In databases ∃ problem of R-W, W-R & W-W problems (check DBMS). So to overcome we use Semaphore synchronization.

```
int readCount=0;
Semaphore mutex=1;          } Binary Semaphore
Semaphore db=1;
Void Reader() {
   while (1){
      down (mutex);
      readCount++;
      if (readCount==1)
            down (db);
      Up (mutex)
      Data Base
      Down (mutex)
      readCount --;
      if (readCount==0)
            up(db);
      Up (mutex)
      Process_data
   }
}
```

Entry Section

Exit Section

```
Void writer(){
  while(1){
    down(db);        { Entry Code
    Data Base
    Up(db);          { Exit Code
  }
}
```

- Run Reader() when R comes & Writer when W, you'll see W-R, W-RW & ~~R-R~~ R-W will never be able to get in data base while unlimed amout of $R_1 R_2 R_3 \ldots$ can reside in CS

1) W-R , W_1 comes => Writer (db=0 & in DB)
R_1 comes => Reader (mutex=0, rCount=1, down(db) → blocked!)

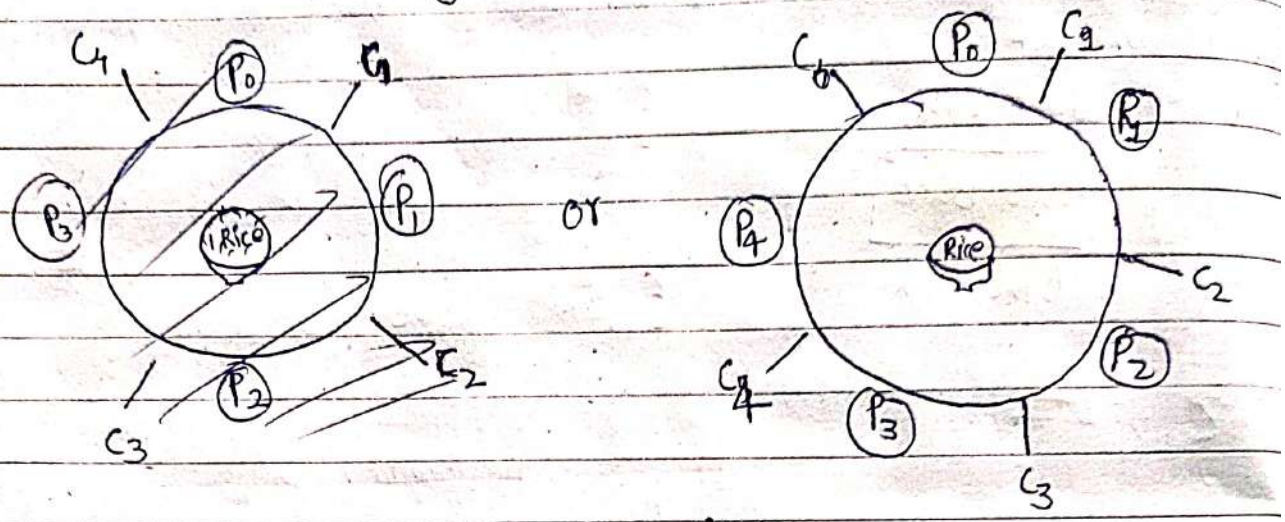
2) ~~W-W~~ W_1 comes => Writer (db=0 & in DB)
W_2 comes => writer (down(db) → blocked!)

3) ~~R-W~~ R_1 comes => Reader (mutex=0, rcount=1, db=0 & in DB)
W_1 comes => writer (down(db) → blocked!) mutex=1

4) R-R R_1 comes => Reader (mutex=0, rcount=1, db=0, mutex=1 & in DB)
R_2 comes => Reader (mutex=0, rcount=2, mutex=1 & in DB)
R_3 comes => Reader (mutex=0, rcount=3, mutex=1 & in DB)
so on

& When running Exit code in Reader() if(rcount==0) up(db)
prevents writer to come in unless no reader in DB

Solution of dining philosophers:



- Philosophers do 2 tasks → think
  → Eat with Chopsticks.
- Each guy requires 2 chopstick left & right
- Problem comes when adjacent Philosophers want to eat.

```
Void Philosopher(){                          Synchronized
    ~~Think()~~                                    ↓
    while(1){
        Think();
        take_chopstick(Ci);          |  wait (take_chop(Si));
        take_chopstick(C(i+1)%n)     |  wait (take_chop(S(i+1)%n));
        EAT();
        put_chopstick(Ci);           |  Signal (put_chop(Si));
        put_chopstick(C(i+1)%n);     |  Signal (put_chop(S(i+1)%n));
    }
}
```

- Initially all S_i, $i = 1$ to $n-1$ $= 1$.
- wait(s) => s--;
- Signal(s) => s++;

$S_0 = 0$

- So say P_0 came picked chopstick C_0 & before he could pick C_0 he got preampted

Now, P_4 came ~~but cannot pick~~ & picked chopstick C_4 & but cannot pick C_0 (coz its $S_0 = 0$) i.e P_4 is now blocked.

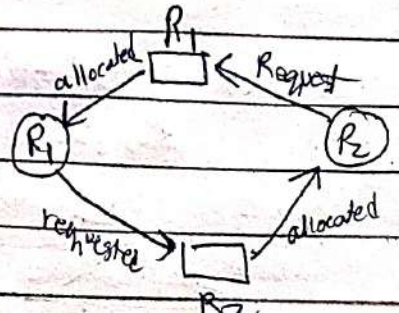
- There can be a dead lock situation, when one pick left stick & got preampted, next process picketed that unpicked stick (its left) & got preampted & so on in last, the last guy will pick one stick but will be waiting for very first guy to leave the stick i.e a deadlock.

Solution of deadlock : Last person should pick right stick first, so that'll be not possible => Block but 2nd last person will be able to eat.

Deadlock : If two or more processes are waiting on happening of some event that's never going to happen is called Deadlock.

eg: Person wants job to get experience but that job requires experience itself.

eg: Dining philosophers



Condition for Deadlock : (Necessary)

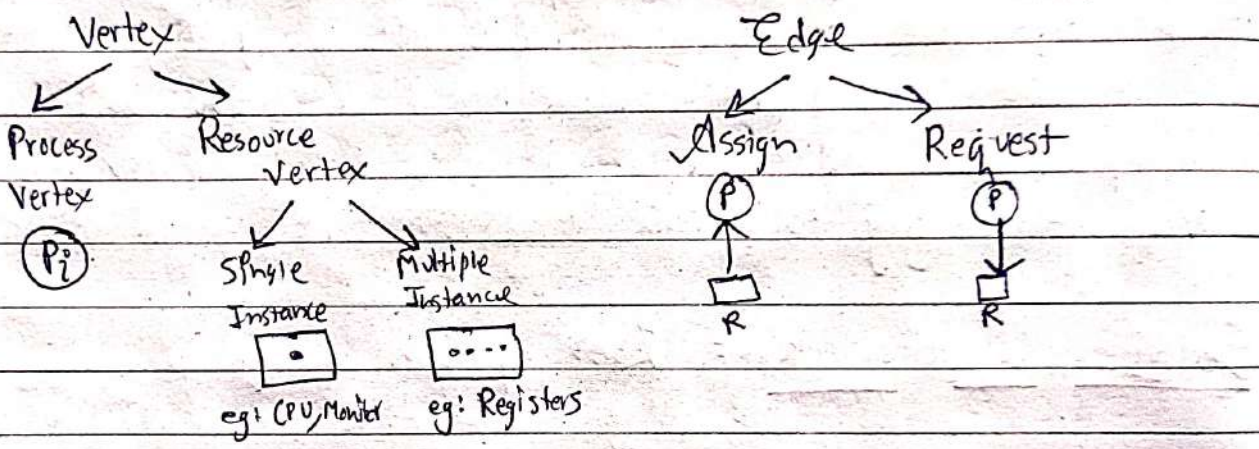
1) Mutual Exclusion → Processes should be independent of each other

2) No Preamption → No scenario of Preamption

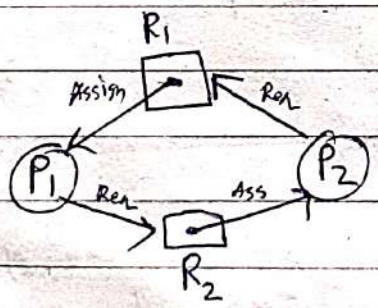
3) Hold & Wait → One process holds a Resource & waits for more

4) Circular wait → Looping in traversal from Proc & Resource

Resource Allocation Graph (RAG) : Helps to find deadlock in system

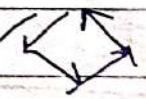


eg:



Way1

1) Mutual Exc ✓ (P_1 & P_2 indep)
2) No Preamption ✓ (always tbh)
3) Hold & wait ✓ $P_1 \leftarrow R_1$ & $P_2 \leftarrow R_2$
4) Circular wait ✓



i.e Readlock

Way2

Proc	Allocate R_1	Allocate R_2	Req R_1	Req R_2
P_1	1	0	0	1
P_2	0	1	1	0

coz R_1 is captured by P_1 & R_2 by P_2

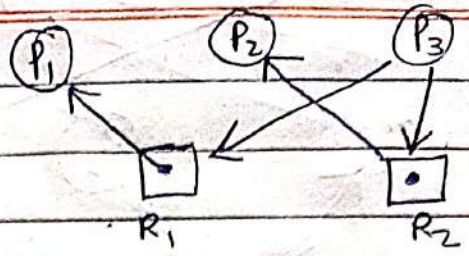
Availability of $(R_1, R_2) = (0,0)$

Request of $P_1 = (0,1)$ ✗ Cannot fulfill

$P_2 = (1,0)$ ✗ "

i.e Deadlock

Q1: Check for deadlock



Sol^n

Proc	Allocated		Requested		Availability
	R_1	R_2	R_1	R_2	A(0, 0)
P_1	1	0	0	0	✓ A(1, 0)
P_2	0	1	0	0	✓ A(1, 1)
P_3	0	0	1	1	✓ A(0, 0)

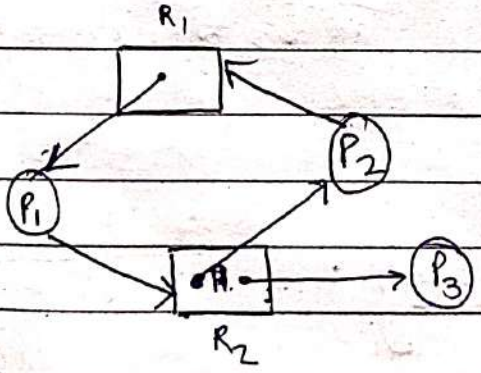
- When process gets completed (terminate) it frees up the resources

No Deadlock here.

- Also we can see there is No circular wait => No deadlock

- If there's circular wait in case of single instance ∃ Deadlock

Q2:



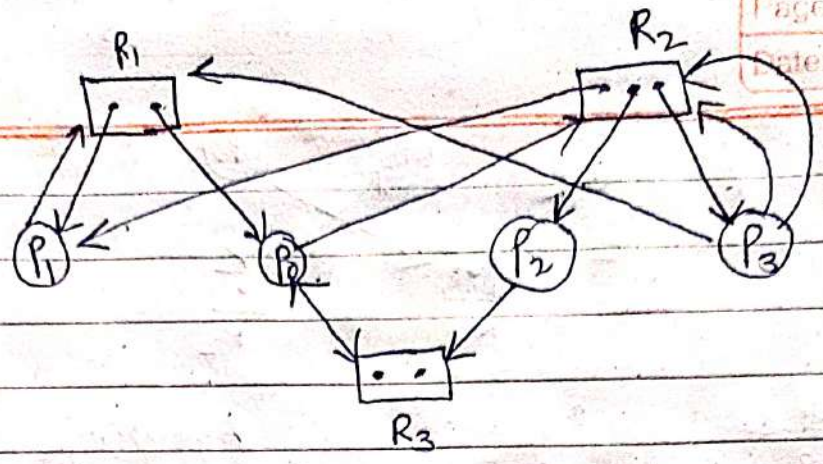
Sol^n

Proc	Allocated		Requested		Availability (R_1, R_2)
	R_1	R_2	R_1	R_2	↓
P_1	1	0	0	1	(0, 0)
P_2	0	1	1	0	
P_3	0	1	0	0	✓

Now P_3 after terminated → A(0, 1) Now P_1 can be fulfilled → A(1, 1) & P_2 can be done now!

do no gorantee of deadlock if circular path

Q:



Sol^{n}

Proc	Allocated			Requested		
	R_1	R_2	R_3	R_1	R_2	R_3
P_0	1	0	1	0	1	1
P_1	1	1	0	1	0	0
P_2	0	1	0	0	0	1
P_3	0	1	0	1	2	0

∴ Current Availibility (R_1, R_2, R_3) = (0, 0, 1)

P_2 can be fulfilled → A(0,0,1) + A(0,1,0) = A(0,1,1)

P_0 can be fulfilled → A(0,1,1) + A(1,0,1) = A(1,1,2)

P_1 can be fulfilled → A(1,1,2) + A(1,1,0) = A(2,2,2)

Now P_3 can also be full filled ⇒ A(2,3,2)

all data of resources R_1 R_2 R_3

No deadlock

Method for Deadlock Handling :

1) Deadlock Ignorance (Ostrich Method)

2) Deadlock Prevention

3) Deadlock Avoidance (Banker's Algorithm)

4) Deadlock Detection & Recovery

1) Ignorance : Deadlock is rare phenomenon & writing code for it & constantly checking for deadlock will negatively affect speed & performance, so here we chose to ignore it. It's most common.
eg: Windows, Linux.

2) Prevention : We try to make atleast 1 necessary deadlock condition false.

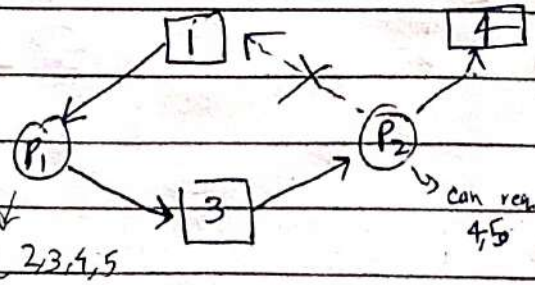
① Mutual Exclusion $\xrightarrow{\text{to make it false}}$ Make all Resources sharable (still not possible for printer, scanner, etc.)

② Non-Preamption $\xrightarrow{\text{false}}$ Make OS Preamptive, so when a process is preampted it frees up all resources.

③ Hold & wait $\xrightarrow{\text{false}}$ Previous case is good enough, or we can make sure no process can hold a resource. or provide all the resource to it (No more watting)

④ Circular wait $\xrightarrow{\text{to make it false}}$ We can remove it by numbering the Resources & new process can only request resource in increasing order

eg:



1. Printer
2. CPU
3. Scanner
4. Register
5. Webcam.

3) Avoidance: For every resource allocation we check the safe or unsafe situation. It's determined by Banker's Algorithm (aka Dijakstra Algo)

4) Detection & Recovery: We try to detect first by Resource Allocation Graph (RAG) or other method. then if detected we try to recover it. Exteremly complex task. Recovery — Kill the deadlock processes 1 by 1
— Resource Preamption

Banker's Algorithm: It's avoidance method, we tell OS the name & required resources by processes in advance. So that OS can decide what to do. It's also used for Deadlock detection.

eg:

Proc.	Allocation			Max Need			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		1	2	2
P_3	3	0	2	9	0	2		6	0	0
P_4	2	1	1	4	2	2		2	1	1
P_5	0	0	2	5	3	3		5	3	1

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

- P_2 can terminate => Av(5, 3, 2) => P_4 ✓ => Av(7, 4, 3)

=> P_5 ✓ => Av(7, 4, 5) => P_1 & P_3 cannot be satisfied by Max need but can be for Remaining need

$\Rightarrow P_1 \checkmark \Rightarrow Av(7,5,5) \Rightarrow P_3 \checkmark$

All executed !! $\Rightarrow$ No deadlock !!

Safe sequence $\Rightarrow P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_3$

- It's not possible in real life, because requirement of resources is dynamic in nature & NOT static

Q: Find if ∃ Deadlock or not if not then give safe sequence.

Proc	Allocation			Max			Remaining			Available
	E	F	G	E	F	G	E	F	G	(E, F, G)
P_0	1	0	1	4	3	1	3	3	0	(3,3,0)
P_1	1	1	2	2	1	4	1	0	2	
P_2	1	0	3	1	3	3	0	3	0	
P_3	2	0	0	5	4	1	3	4	1	

$\Rightarrow P_0 \checkmark (4,3,1) \rightarrow P_2 (5,3,4) \rightarrow P_1 (6,4,6) \rightarrow P_3 (8,4,6)$

No deadlock & safe seq $\Rightarrow P_0 \rightarrow P_2 \rightarrow P_1 \rightarrow P_3$

Q: A sys has 3 processes, each requires 2 units of resource R. The min no. of unit of R such that no. deadlock occurs is ?

→	Allocated	Required	Available
	A B	A B	
P_1	0 0	2 1	(x) $x = 2$
P_2	0 0	2 0	
P_3	0 0	2 1	

But there can be prob
Say P_1 got executed & freed up 2Rs but then P_2 got 1R & P_3 got 1R $\Rightarrow$ Deadlock

Ans

if 1 → P_1 P_2 P_3 Deadlock
1

if 2 → P_1 P_2 P_3 Deadlock
1 1

if 3 → P_1 P_2 P_3 Deadlock
1 1 1

if 4 → P_1 P_2 P_3 ⇒ P_1 P_2 P_3 ⇒ P_3 ✓
11 ✓ 1 1 11 ✓
1 1 11
Buffer I II

4 Ans

So we try full proof cases such that deadlock can never happen.

m → # of processes
n → # of resources

So ans ⇒ $(n-1)m+1$ Ans

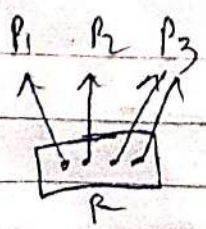
Q: Repeat if 3 processes require 3, 4 & 5 resources.

Soln P_1 P_2 P_3 all need just 1 more & each
2 3 4 will execute 1by1 & release resource

→ $2+3+4+1 = 10$ Ans i.e $\sum_{i=1}^{m}(R_i-1)+1$

Q: A sys has 3 processes that share 4 instances of same resource type. Each can req max of K instances. Largest value of 'K' that will always avoid deadlock is?

Soln



if K=1 P_1 P_2 P_3 ✓
↑ ↑ ↑↑

if K=2 P_1 P_2 P_3 → P_1 P_2 ✓
↑ ↑ ↑↑ ↑↑ ↑↑
P_1 P_2 → P_2 ✓
↑↑ ↑ ↑↑

if K=3 P_1 P_2 P_3 → P_2 P_3 ✓
↑↑↑ ↑ ↑↑↑ ↑
P_1 P_2 P_3
↑↑ ↑ ↑

if ~~P=4~~ ~~P1 P2 P3~~ Hence max K=2 for No deadlock

Way 2 Total Resources Demand $\geq (n-1)m+1$ To avoid deadlock

$4 \geq (K-1)\times 3+1$ or $3 \geq 3K-3$

or $6 \geq 3K$

or $2 \geq K$

$K_{max} = 2$ Ans

Summary: P_1 P_2 P_3 ... P_m ← Processes

r_1 r_2 r_3 ... r_m ← Need Resource (Demand)

Then ~~max~~ min ~~process~~ resource for no deadlock?

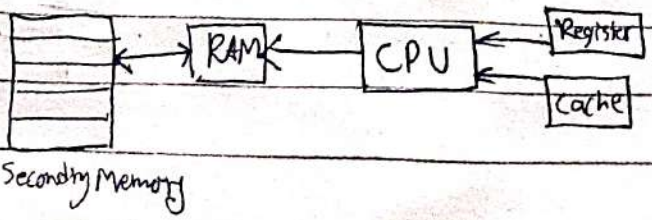
1) Waste of money => $\sum_{i=1}^{m} r_i$

2) Optimal => $\sum_{i=1}^{m}(r_i-1)+1$ i.e $\sum_{i}^{m} r_i + 1 - m$

i.e Min No Deadlock Resource = Demand − #of Processes + 1.

OR just allocate 1 less to all then it'll be deadlock +1 to remove deadlock.

Memory Management: Functionality to manage various memory (RAM, HDD, Registers, etc) but we focus on RAM only.

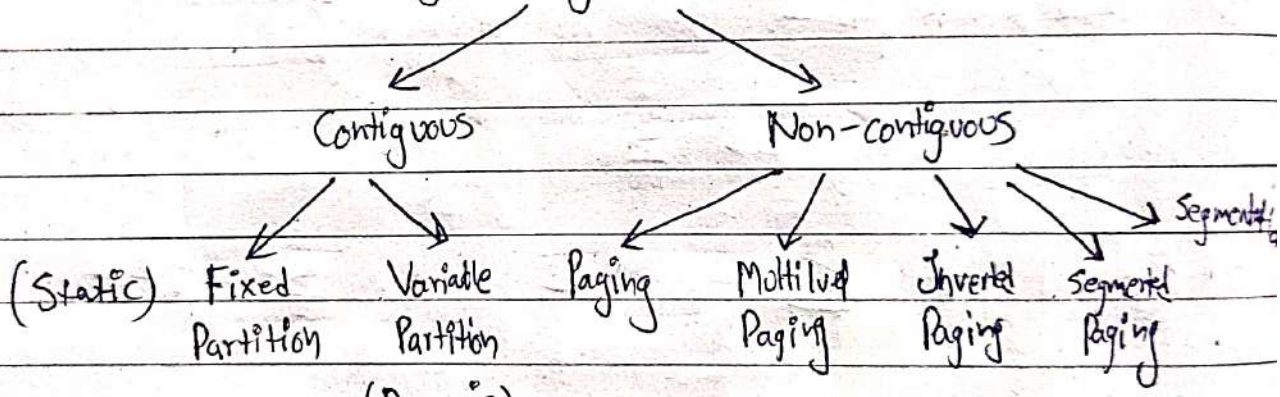


Q: System as 4GB RAM & 2GB programs in HDD & all progms require 70% time for I/O. Find max CPU utilization

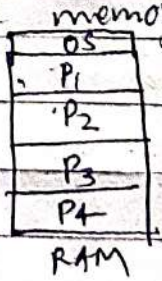
Sol^n 2 programs can come in ram i.e degree of multiprog^n
Let K be % time for I/O i.e K = 70% = 0.7

$$\text{CPU utiliz}^n = 1 - K^2$$
$$= 1 - .7^2$$
$$= 51\%$$

Memory Management Techniques

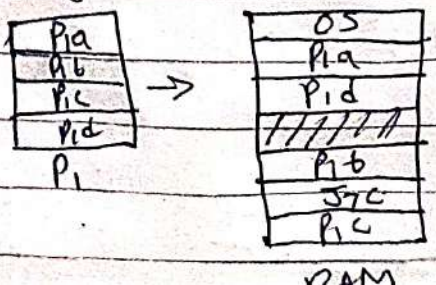


Contiguous : Processes are allocated memory in sequential order & 1 by 1 eg:



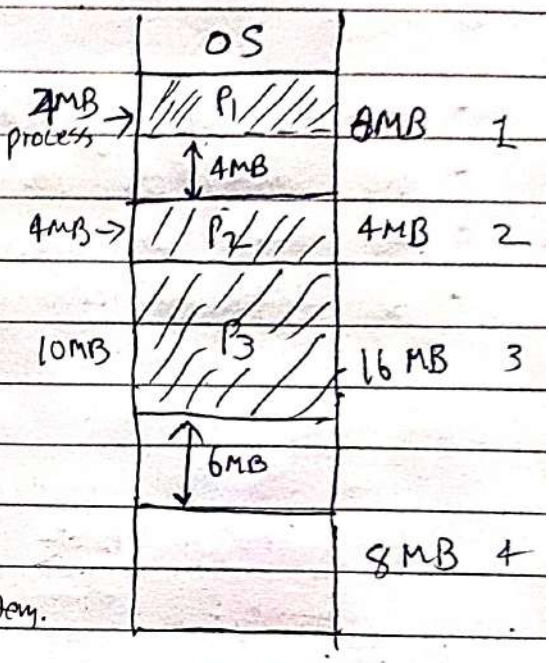
Non-Contiguous : Processes are divided into smaller parts & allocated memory in non-contiguous fashion.

eg:



Fixed Partitioning: "No. of partitions are fixed" & but size may or may not be fixed.

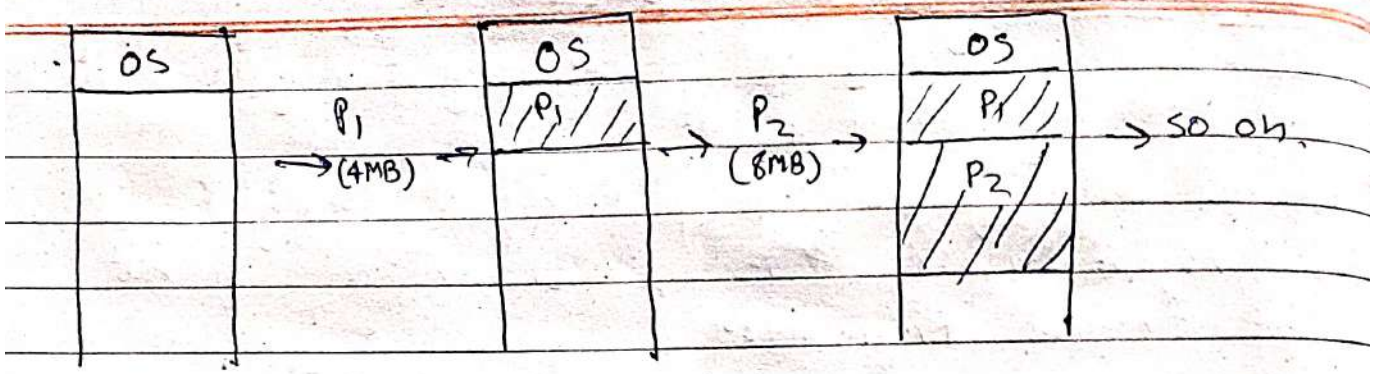
- Process will fully take up partition it's residing in no matter if there's still unused space or not i.e Internal Fragmentation.



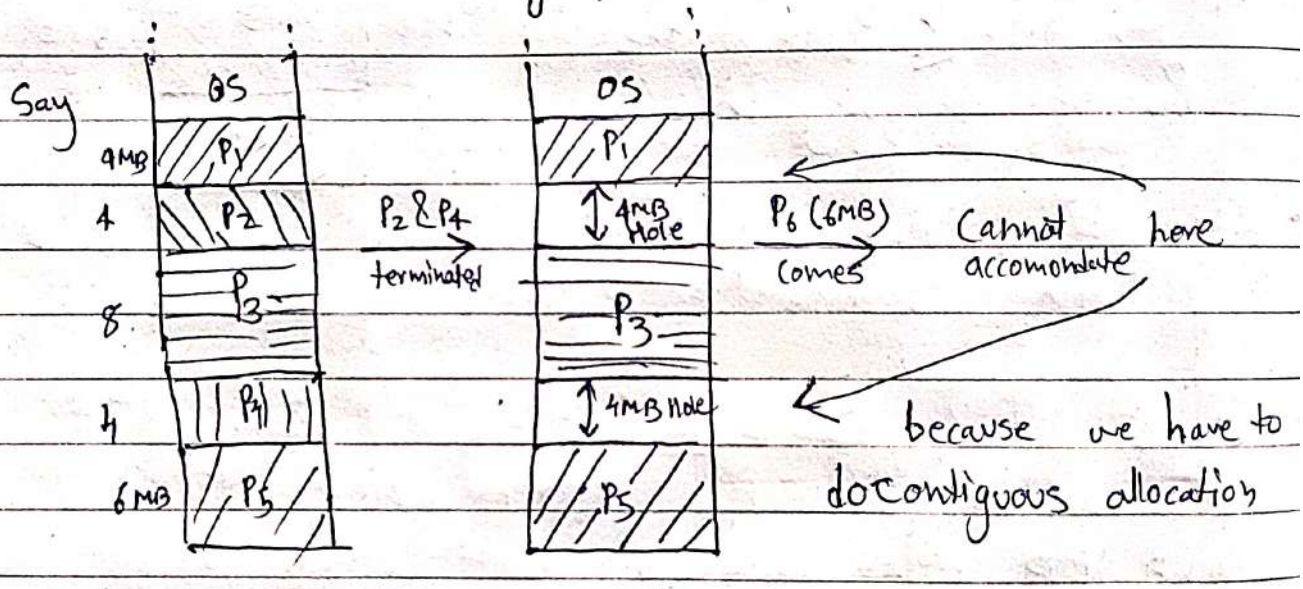
- Can only accomodate fixed no. of processes (4 here) (limited degree of multiprogramming)
- A process can only come iff its size ≤ size of the fragment it's going in.
- Sizes of partition are fixed during configuration of system.
- If size of process is ≤ free size it still cannot be allocated i-e External fragmentation
  say: 9MB process came (4MB unused + 6MB unused) but still cannot take it coz memory allocation for a process should be contiguous & fully in 1 partition.
- If ∃ internal fragmentation ⇒ ∃ External fragmentation

Variable Partitioning: "We provide allocation during runtime i.e when it arrives @ RAM"

- There is no initial partitioning.



- No Internal fragmentation
- No limit on no. of processes (degree of MP)
- No limit on process size
- Since No Internal frag ⇏ No External Frag how?



Hence ∃ External Fragmentation.

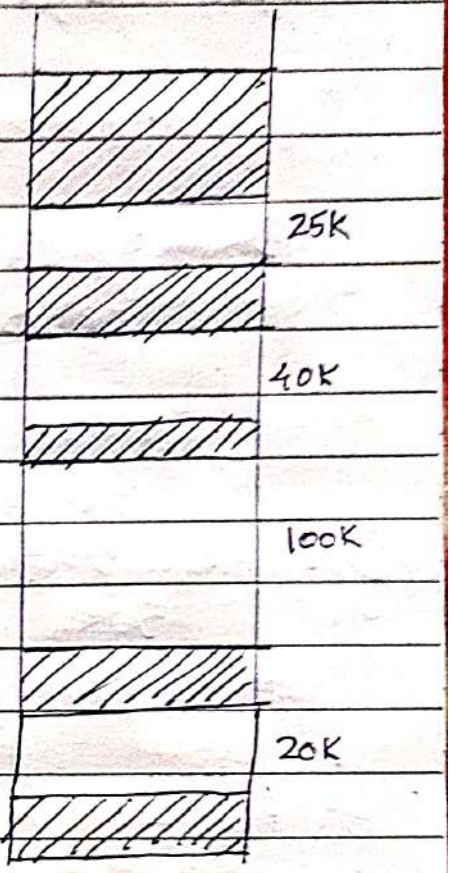
- To remove it do compaction (defragmentation like in HDD) but it's not recommended coz:
  1) We'll have to stop the processes
  2) We'll have to copy & move to other location (time consuming)

- Complex allocation & deallocation, Holes are created dynamically & so are the processes managed by bit-map & Linked List

Various allocation methods in contiguous mgmt:

There're 4 algorithms - First fit, next fit, Best fit & Worst fit.

- Assume 15K size process came



- **First-Fit:** Allocates the very first hole that's big enough for process.

Adv: V.fast

Disadv: Lot of holes can be made

→ It'll go to 25K hole

Next-fit: It's part of first fit only but we keep track of latest allocated memory & then we'll search from that pointer

Adv: Even more fast (as we don't have to start looking from top again & again)

Disadv: Same

- **Best-fit:** Allocates smallest hole big enough for the process.

Adv: V.small Internal fragmentation

Disadv: 1) Due to v.small holes that'll be created further it'll be highly unlikely that new process can go there, & in long run these small chunks can waste even more

2) Slow (has to scan whole memory

→ It'll go to 20K hole

- **Worst-fit:** Allocates the largest hole

Adv: Larger holes will be there, so further that hole will be used to accomondate new processes, like overcoming Best-fit.

Disadv: 1) Large Internal fregmentation.
2) Slower (full scanning)

→ It'll go to 100K hole

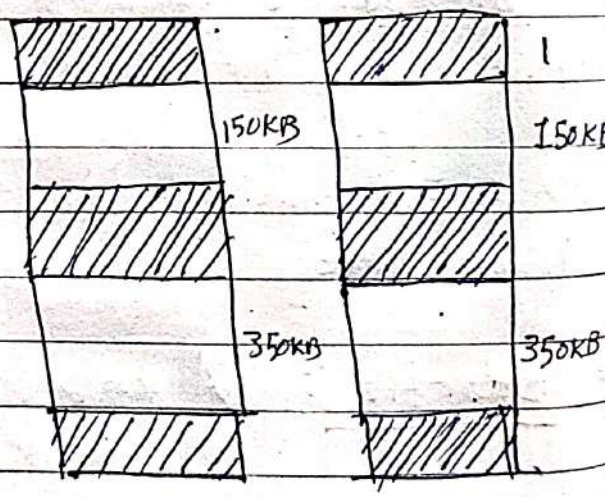
Q: Requests from various processes are 300K, 25K, 125K, 50K respectively the above req could be satisfied with?

A) Best-fit but not first fit

B) First fit but not best

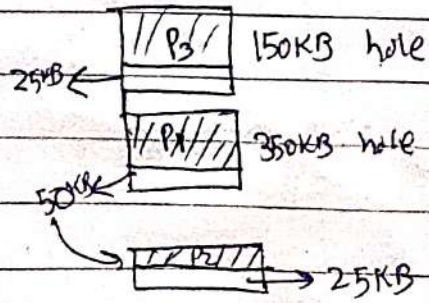
C) Both

D) None



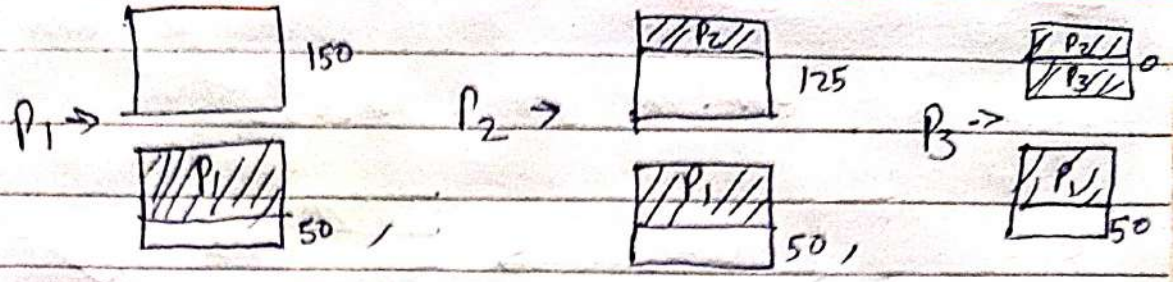
50K

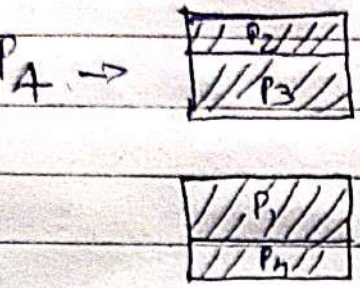
1) assume Best fit:



=> Not possible

2) First fit:





✓ Possible!

Option B) ✓

Optional → Worst fit will also work:

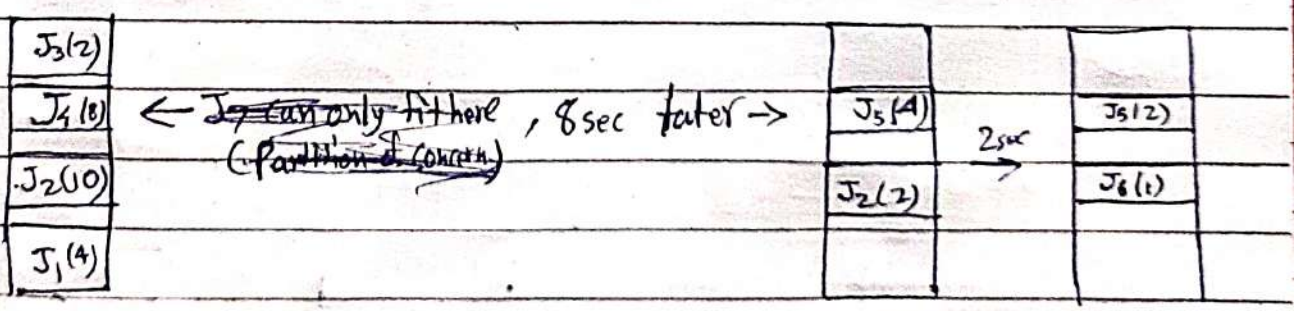
Q2 =

Req No.	J1	J2	J3	J4	J5	J6	J7	J8
Req Size	2K	14K	3K	6K	6K	10K	7K	20K
Usage time	4	10	2	8	4	1	8	6

MM

0	4K
1	8K
2	20K
3	2K

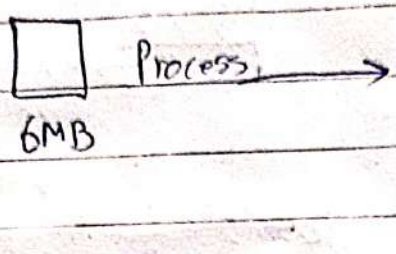
Assume Best-fit algorithm @ what time J7 will be completed?



8+2+1+8 = 19 sec

& Time @ which J7 will enter ram is 11sec

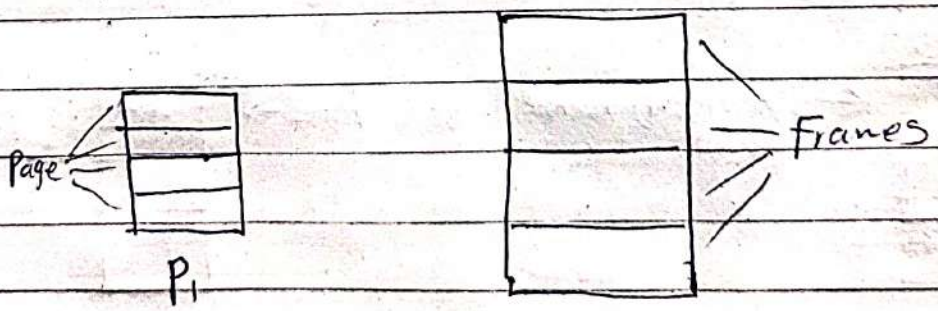
Non-Contiguous Mem Allocation: We can divide process & can put 'em in diff location.

eg: 

- External fragmentation is removed

Problem: The holes are generated dynamically & according to holes the process is divided. So scanning # of holes, size of holes & then dividing the process consumes a lot of time.

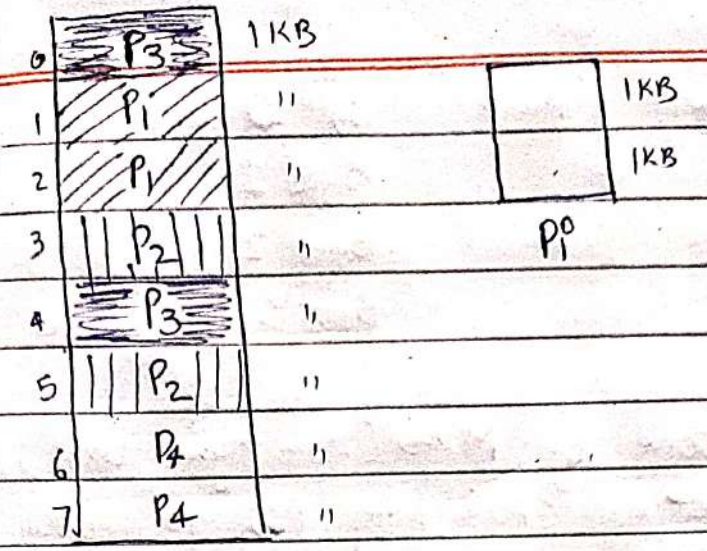
So before even coming to RAM, process is divided & each divided partion is k/a Page. And RAM is also divided k/a Frame.

i.e 

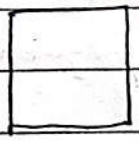
Page size = Frame size. In order to fit properly.

Q: RAM of 8KB having frame size of 1KB & programs $P_1 P_2 P_3$ has 2KB size, each. Draw one of possibility of memory allocation

Soln



Actual Implementation:



-> Page size = 2B

Process P_1 4B

- 4 Byte process has each byte as (0,1,2,3)

MM

Frame No ↓		
0	0 1	2B
1	2 3	2B
2	4 5	2B
3	6 7	" 16B RAM
4	8 9	"
5	10 11	"

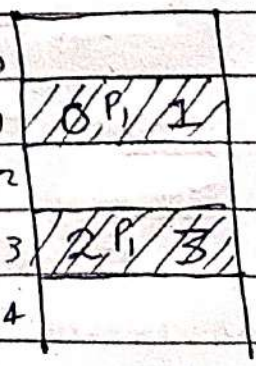
- Each page has a no. (2 pages in a process) i.e (P_g0, P_g1)

=>

Pg No		
0	0 1	Byte #
1	2 3	

Now say process P_1 goes in RAM.

- We assume memory to be Byte addressible (CPU can demand in a byte)



- Say CPU wants Byte 3 ie

0	1
2	3

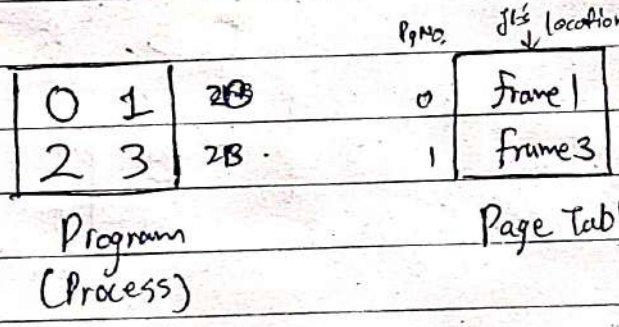
So how it Know where is the Byte 3 in RAM? (Ans is actually 7 in RAM)

It's done by mapping.

- So Mapping will map Byte 0 of Proc P1 to 2
  Byte 1 to 3
  Byte 2 to 6
  Byte 3 to 7

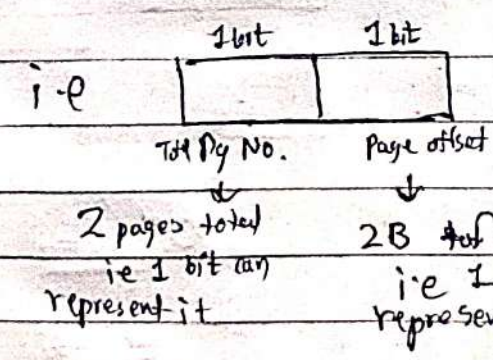
- Mapping is Done by MMU (Mem Mgmt Unit), so it convert CPU's addr (3) to absolute addr (7). Page Table is required to achieve it.

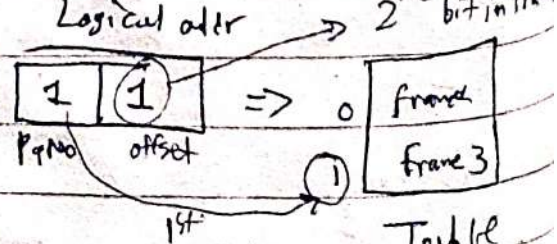
- Each process has its page Table, it contains frame no. of each page of program.



- CPU always works on logical Address

Pg No + Page offset (Page Size)

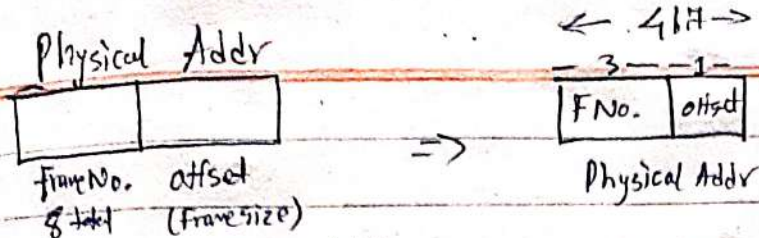


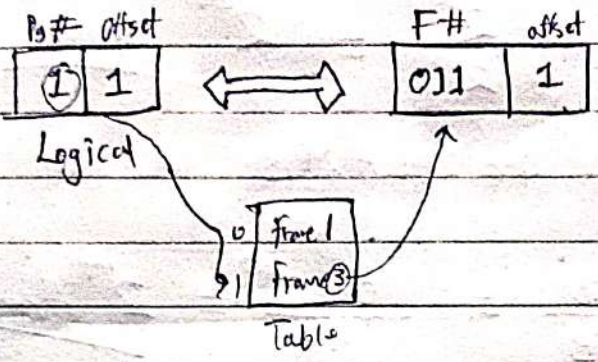


CPU needed Byte 3 => 11 =>

2nd byte in frame 3 = 7

↑ Absolute addr (Physical)

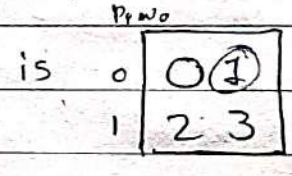




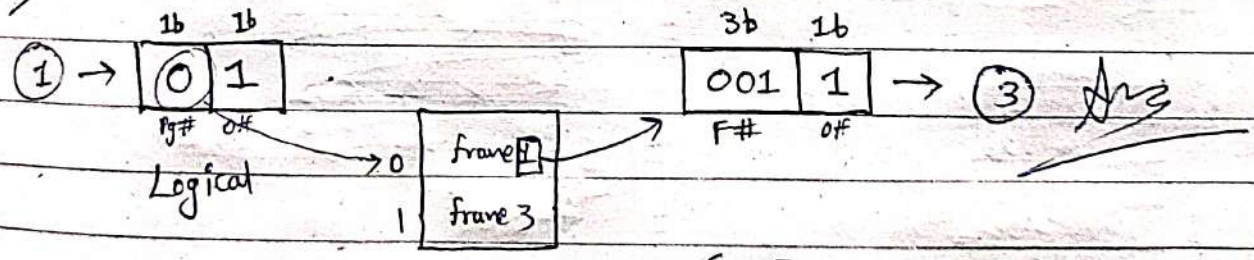
& collectively 011 means 7

- Logical Addr's Offset & Phy Addr's Offset are always same.

Q: Repeat Process if Program is & CPU wants Byte 1.



So/ afc answer will be 3 but how?



Q: Given memory is byte addressable. LAS → Logical Addr size

LAS = 4GB
PAS = 64MB
Pagesize = 4KB
# of Pages = ?, # of Frames = ?, # of entries in page Table = ?
Size of page Table = ?

Sol^n

Pagesize = frame size

(1) 4KB = frame size

- LAS represents Process size i.e 4GB

(2) # of pages = $\frac{4GB}{4KB} = \frac{2^{30}}{2^{10}} = 2^{20}$ pages

- PAS represents RAM i.e 64MB

(3) # of frames = $\frac{64MB}{4KB} = \frac{16 \times 2^{20}}{2^{10}} = 2^{14}$ frames

- No. of entries in pg table is same as no. of pages created for a program

(4) # of entries in page Table = 2^{20} entries

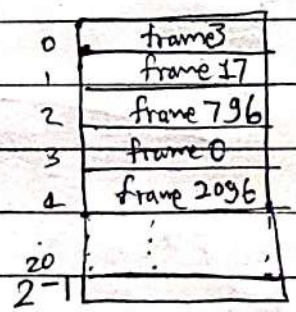
(5) Size of page table = No. of entries × size of each entry

= ~~2^{20} × size of page~~

= 2^{20} × 4 KB

= 2^{20} × 2^{2} × 2^{10} Bytes

= ~~4 GB~~



Each entry represent frame #,
Total frames are 2^{14}.

So 14 bit can represent any frame no.

= 2^{20} × 14 bits

= 2^{17} × 14 Bytes

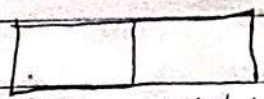
= 1.75 MB Ans (Cannot fit in 1 frame, so we'll have to divide it to)

=> LA: $2^3 \times 2^{30}$ Bytes — 32 bits (20 | 12: P. No | offset)

PA: $2^6 \times 2^{20}$ — 26 bits (14 | 12: Frame # | offset)

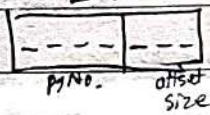
Q: System has LA = 7 bits, PA = 6 bits & page size = 8 words, find # of frames & pages

Sol: LA = 7 bits =>



page size = 8 Bytes
= ~~64 bits~~ { we can represent ~~8~~ (0-7) using 3 bits

=>



=> # of pages = 2^4 = 16

Ill[24]

=> # of frames = 2^3 = 8

~~Extra~~ We know No of entries in Table = # of pages
= 2^4 = 16

Page Table =>

0	frame 2
1	frame 7
2	
⋮	
15	frame 0

Size of Table = 2^4 x 3 bits
= 48 bits
= 6 Bytes

Page Table Entry: We've studied that each entry contains frame no.
like 1 | frame 3 | but it's highly simplified version

It's actualy

Frame No	Valid (1) Invalid (0)	Protection (RWX)	Refience (0/1)	Caching enable / Disable	Dirty (0/1)

Mandatory — Optional

- Valid/Invaild: Tells if the required page is present or not @ that frame no. i.e Pg1 [Frame3] 0/1
  It's Kla Page Fault.
  It can happen due to Virtual Mem
  - Swapping, the page we're looking for is in HDD.
- Protection: Used by OS, & data in that frame is in which permission mode i.e Read write / Execute
- Refrence: The swapping we do concerns this, If we ever swapped that frame in past it becomes 1
  Method used is LRU (least Recently used)
- Caching: We cache the frequently used data in it's cache also happens on prog end (cache of apps) but not always recommended eg: (Bank Balance)
- Dirty/Modify: It tells if data is modified or not in RAM (NOT in HDD) eg: * in title bar of sublime text.

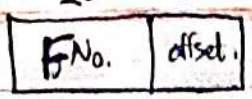
Q: PAS = 256MB
LAS = 4GB
Frame size = 4KB
Page Table Entry = 2B

Find everything else lol.

Sol^n

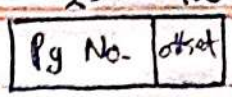
- Page size = 4KB
- # of frames = $\frac{256\,MB}{4KB} = 64 \times 2^{10} = 2^{6} \times 2^{10} = 2^{16}$
- # of pages = $\frac{4GB}{4KB} = 2^{20}$

PA => 2^{28} B



← 28 bits →

LA => 2^{32} B



← 32 →

Page Table =>

0	frame x_1
1	frame x_2
$2^{20}-1$	frame $x_{2^{20}}$

Size = $2^{20} \times 16$ bits

$= 2 \cdot 2^{20}$ B = 2MB >> 4KB (frame size)

↑ Only mandatory field.

- Page Table will have to be divided in $\frac{2MB}{4KB} = 512$ parts

Outer Pg Table =>

0	frame y_1
1	Frame y_2
⋮	
511	Frame y_{512}

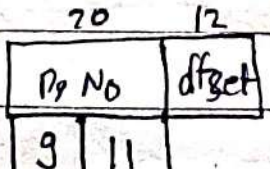
Size = 512 x 16 bits

= 1 KB ✓

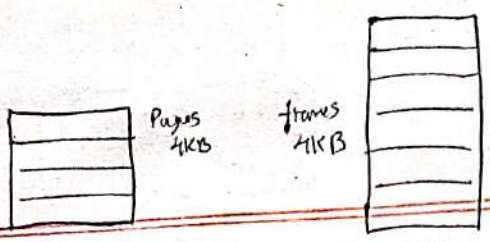
↑ location of Internal pg table (511th page) in frame of MM

- Now how will mapping be done?

LA =>



- If you cannot understand then assume the Page Table (2^{20}) as a program



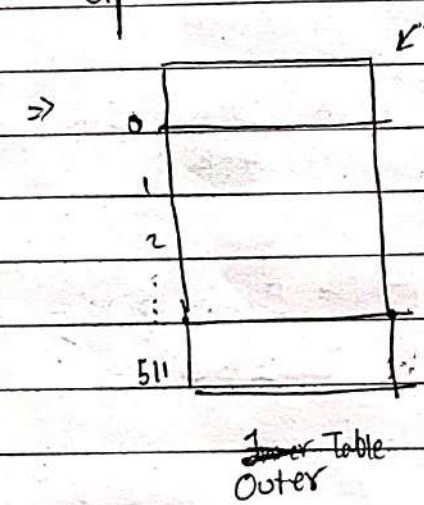
2MB Program

- Frame No. = $\frac{256MB}{4KB} = 2^{16}$

- Page No. → $\frac{2MB}{4KB} = 2^9 = 512$

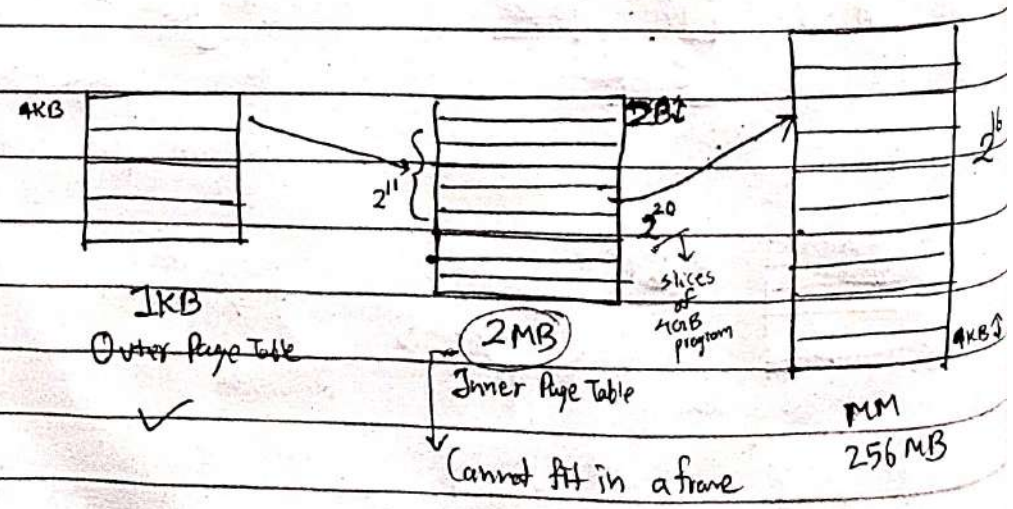
- Prog is of 2MB, to be able to access each byte we'll need 21 bits & 9 for Pg No. => 11 for offset

BUT! This is wrong, we don't fully treat it as a process.



Each page has size of 4KB & stores info of inner table whose each entry was 16 bits or 2B → In each entry of outer table ∃ 2K ie 2^{11} entries. or 2048 entries of Inner Table

- Although Table cannot contain a slice of other table it can only contain a frame no. but assuming it to contain table make it easier to understand things.



Inverted Paging:

If even 1 page of some process comes in RAM the its page Table (whole) should also come.

- Say we have 4GiB application & 8GiB RAM with 4KB frames

$\Rightarrow$ # of pages $= 2^{20}$, # of frames $= 2^{21}$

size of page table $= 2^{20} \times 21$ bits

$= 2.625$ MB

So even if ∃ only 1 page (4KB) of this program in ~~RAM~~ 2.625 MB of page table should also be there.

Upscale this & imagine multitasking scenario, hundreds of processes using only fraction of itself but huge wastage of RAM (limited resource)

- This prob is overcome by inverted paging.

Instead of all page table for all the processes, there will be 1 global page table maintained by OS.

Frame No	Pg No.	Process ID
0	P_0	P_1
1	P_1	P_2
2	P_2	P_1
3	P_1	P_3
4	P_3	P_2
5	P_2	P_3

Global PT

- Quite opposite to typical PT

PNo	Frame No
0	frame0
1	frame7
2	frame f3

- We have to perform whole scanning of GPT (slow)

- As we become more & more advance, memory is becoming cheaper & time more valuable so that's why inverted paging failed.

Q: Consider Virtual Addr Space of 32 bits & page size 4KB System has RAM 128KB. Then find ratio of page table & inverted page table size provided each entry in both is of 4B.

Sol^{n} VA Space = LAS$_{pu}$ = LA = | Page# (20) | offset (12) | (Space NOT Size)

32

PAS = | frame# (5) | offset (12) |

17

# of frames = 2^5, # of pages = 2^{20}

Global PT →		
	0	4B
	1	4B
	⋮	⋮
	2^5-1	4B

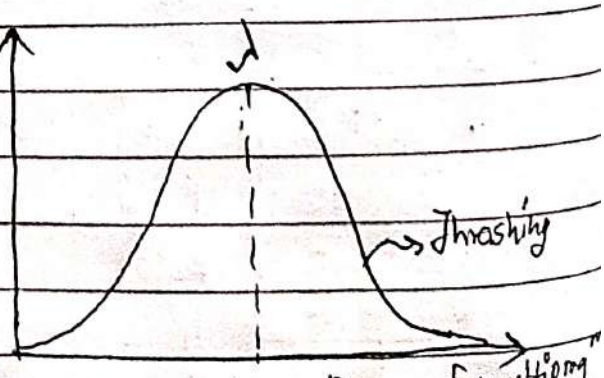
$= 2^7$B

PT →		
	0	4B
	1	4B
	⋮	⋮
	$2^{20}-1$	4B

$= 2^{22}$B

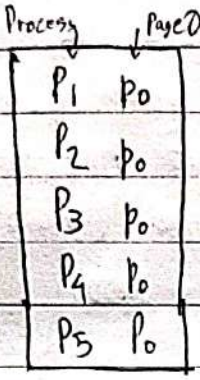
Ratio ⇒ $2^{15}:1$

Thrashing:



- If 1-2 processes are there in RAM & say both performing I/O now so Throughput & CPU utiliz^{n} will drastically go down as CPU is now idle.

- So we want large quantity of processes in RAM (degree) but RAM is also limited in size, so we bring only 1-2 page of ~~that~~ each process. So CPU will always be busy.



It's all good if CPU want p_0 of Process 2 but what if it wants p_7? or p_2, etc. There is no p_7 present in RAM i.e a page fault.

We then use Page fault Service Time (PFST) which brings the required page from Harddisk to MM. & consomes a lot of time.

- So, after a certain degree of MP, page fault occurance increases, page hit frequency goes down & more & more time is consumed in bringing pages from HDD to RAM, so CPU is idle. This is called Thrashing & CPU utiliz^n drastically goes down

- Removal :. 1) Increase RAM size (v. expensive & rarely done)

2) Long Term Scheduler (LTS) → Responsible for bringing process to RAM (degree of MP)
so, decrease its speed so degree of MP will also not be v. high.

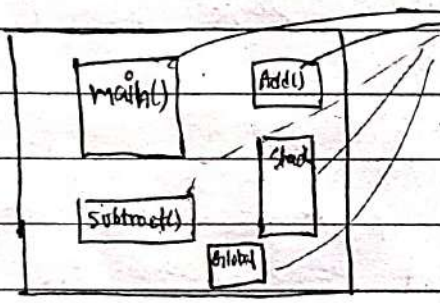
Segmentation

- In paging we blindly divided program in multiple slices of equal size

int Add(){	1B → Frame 7
int x, int y=x+y;	1B → Frame 321
return y; }	1B → frame 29
3B	

So CPU has to switch blw frames look in Table & possibility of not even Finding a frame too (fault) Coz that might be in HDD.

- In segmentation we're more concerned with user's POV & group related data



- We kla CPU works on Logical Add (Int) so to get segment addr in RAM (PA) it needs MMU (using Segment Table)

RAM

0	OS
1500	S_5
1800	Segment1
2200	S_4
2250	S_3
2700	S_2
3300	S_0
3500	

Segment No.

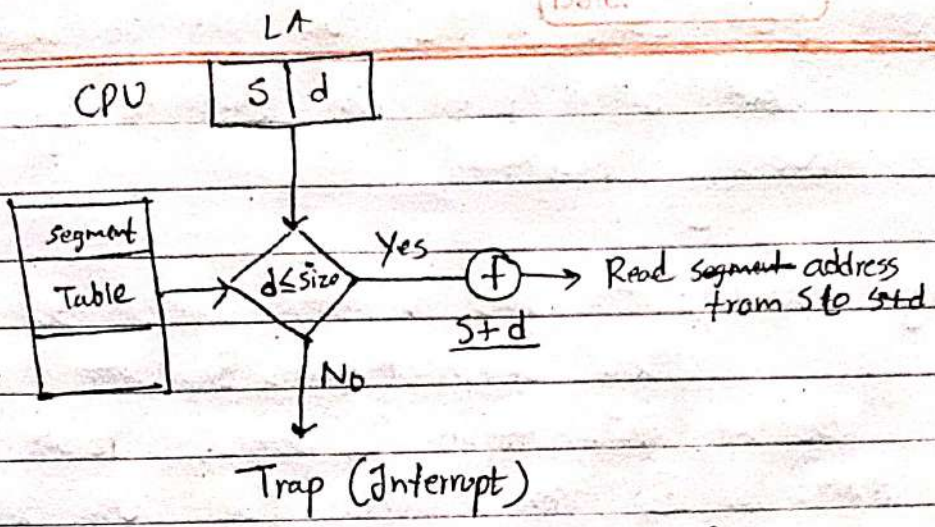
↓	Base Addr	Size
0	3300	200
1	1800	400
2	2700	600
3	2250	450
4	2200	50
5	1500	300

LAT

Segment #	offset

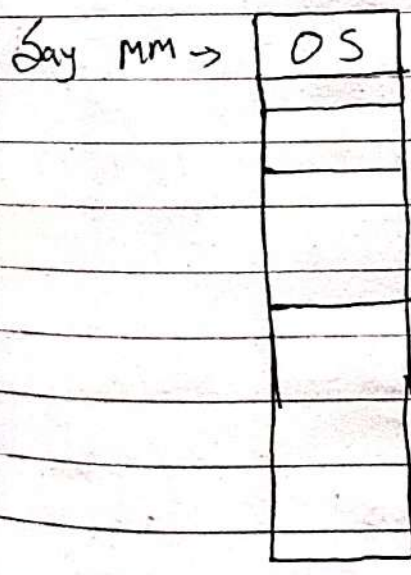
offset → Segment size

- **Working**



eg: CPU wants to read 0001 500 (500 in binary of) i.e segment 1
but size of seg 1 (from table) is NOT 500 ≤ size
Hence Error (interrupt generated.

Overlay: A method by which ~~even~~ process larger than MM can be accomondated in MM.



- OS does not divide the process User has to do it in such a way that a part of prog which User is going to use is kept in RAM & after useding it, its kept back in HDD & other parts comes in RAM.

Problems: 1) Each part of prog should be independent, like part in RAM should not depend on code in other part that's in HDD

2) User can mistakenly bring wrong part

- It's NOT used in PC, as it'll be way more complicated but often used in embaded Systems like Washing machine.

eg: Washing M/c has v.limited ram. So if we select say wash (rotates 5min left & then 5min right) So we divide it in 2 parts, rotate left stays in RAM for 5 then right. (ofc user selected "Wash")

Q: Consider 2-pass assembler:

pass 1: 80 KB Symbol Table : 30 KB
pass 2: 90 KB Common Routine : 20 KB

If a time only 1 pass is used. What's min partition size if Overlay driver is 10 KB?

Sol^n The Symbol Table, Common Routine & Diver is mandatory i.e 60KB mandatory..

-> Pass1 80 + 60 -> 140KB
Pass2 90 + 60 -> 150KB

So min 150KB can be used to run either pass easily. Ans

Note: only useful when MM < 230KB
↳ 80+90+30+20+10

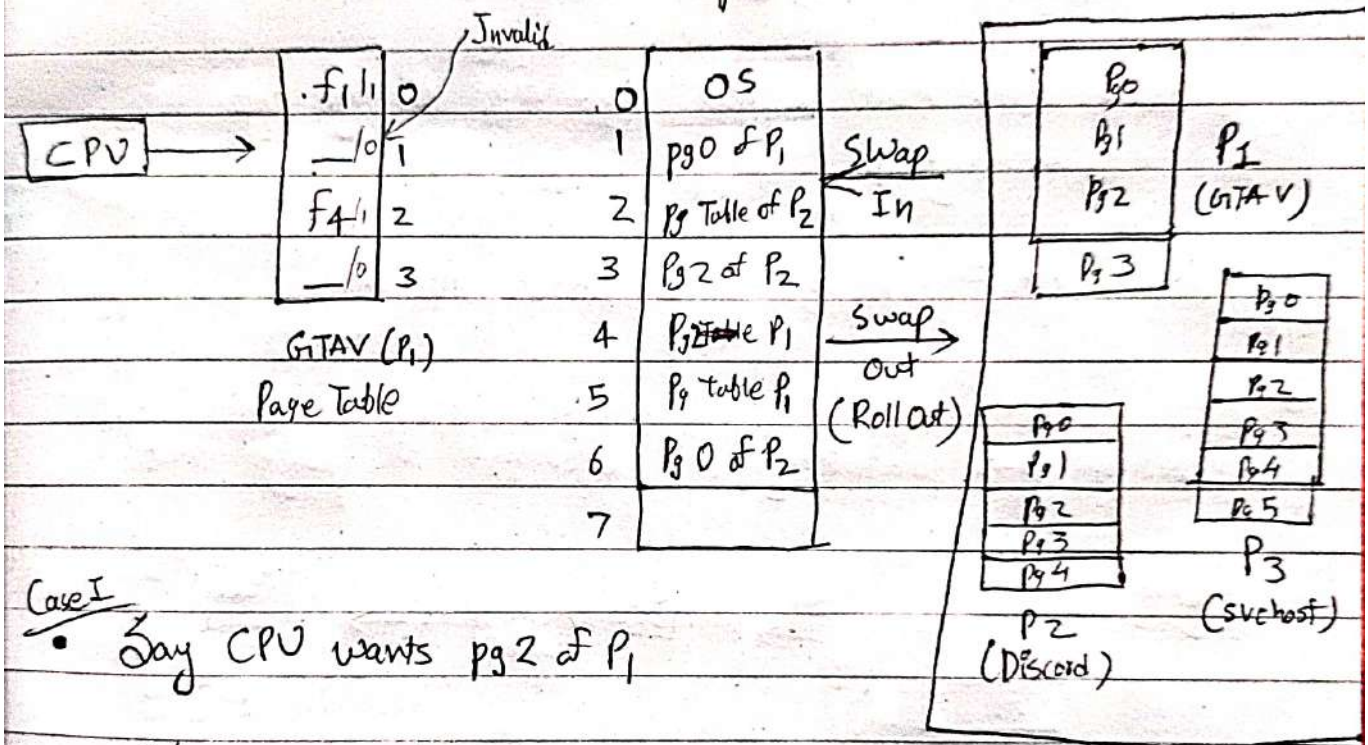
- In Real time System we use Virtual Memory concept instead of Overlay.

Virtual Memory: It provides illusion to user that programs > RAM can also be executed.

- No limitation on size of program
- No limitation on no. of programs.

} Advantages

- We only Bring necessary pages from a process & NOT the whole process itself.



Case I

- Say CPU wants pg 2 of P_1

It's already in RAM => time taken = ma

ma => time taken to access memory (~ns)

Case II

- Say CPU wants pg 1 of P_1, It's NOT in MM. (MMU told) coz Pg Table says invalid => Page Fault & Page Fault will generate Trap (interrupt) control goes from user to OS (context switching)

OS firstly check authorization of user (for security purpose) & if it's all good then OS will look for required page in LAS (HDD) here it finds pg 1 of P_1

Then it puts pg 1 of P_1 in RAM by any of memory allocation algorithm (Best, first, worst fit)

The frame in which it went is saved in P_1's pg table.

Then control is given back to User

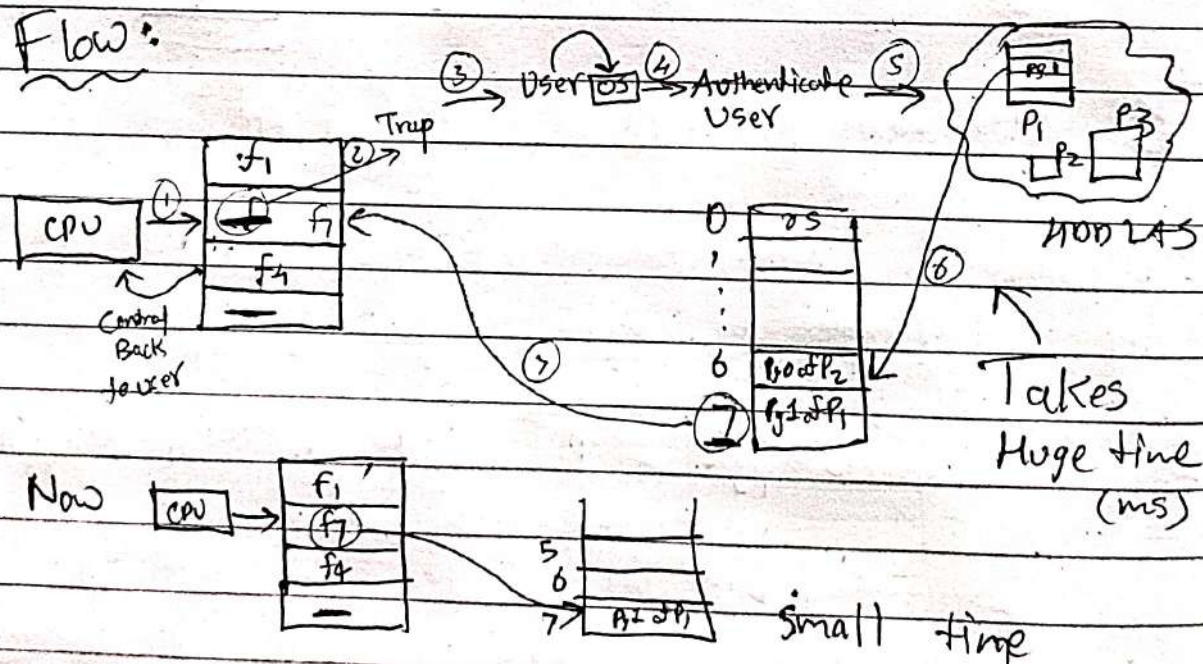
Now CPU can access pg 1 of P_1 directly coz its in memory.

User → OS → user picking from HDD & putting on MM picking from MM

Total Time = p (pg fault servire time) + ma
(~ms) (~ns)

Where p → probability of occurance of page fault.

Flow:



- Effective Mem Access Time (EMAT) = $p\left(\begin{matrix}\text{page fault}\\ \text{service time}\end{matrix}\right) + (1-p)\left(\begin{matrix}\text{Memory Access}\\ \text{Time}\end{matrix}\right)$ + ma

pg — order of ms (million times slower)

ma aka (ma) — order of ns

eg: Viva Call of Student Roll No. 10

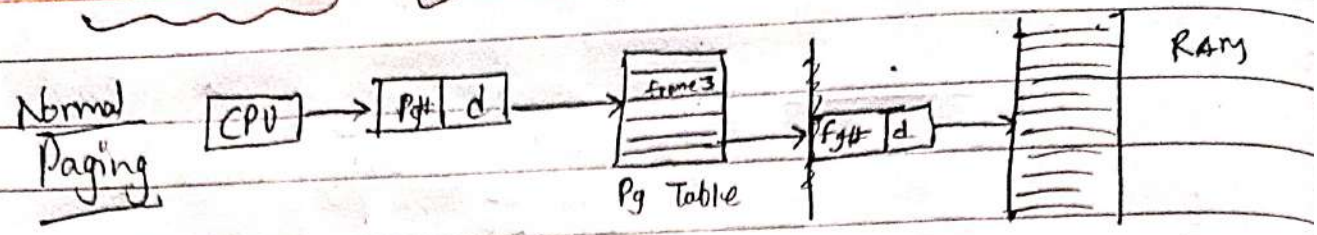
- We bring <u>necessary</u> pages to RAM only, i.e the main page of prog along with related pages. ie Kla. Locality of refrence

- No page is permanently in MM (as it's limited). So lot of pages will go out & alot of diff pages will go in RAM, Kla Page Replacement

There're various page replacement algos 1) FIFO
2) LRU
3) MRU

- Most widely used in modern day laptops & PCs
- This whole flow of CaseII is actually how Page Fault Service Time works.
- $EMAT = p\,(PFST) + (1-p)\,(mA)$
  - $p\,(PFST)$ ↑ CaseII
  - $(1-p)\,(mA)$ ↑ CaseI
- If we try do ↓p (decrease pg faults or ↑pg hits) then EMAT will work great

Translation Lookaside Buffer (TLB):

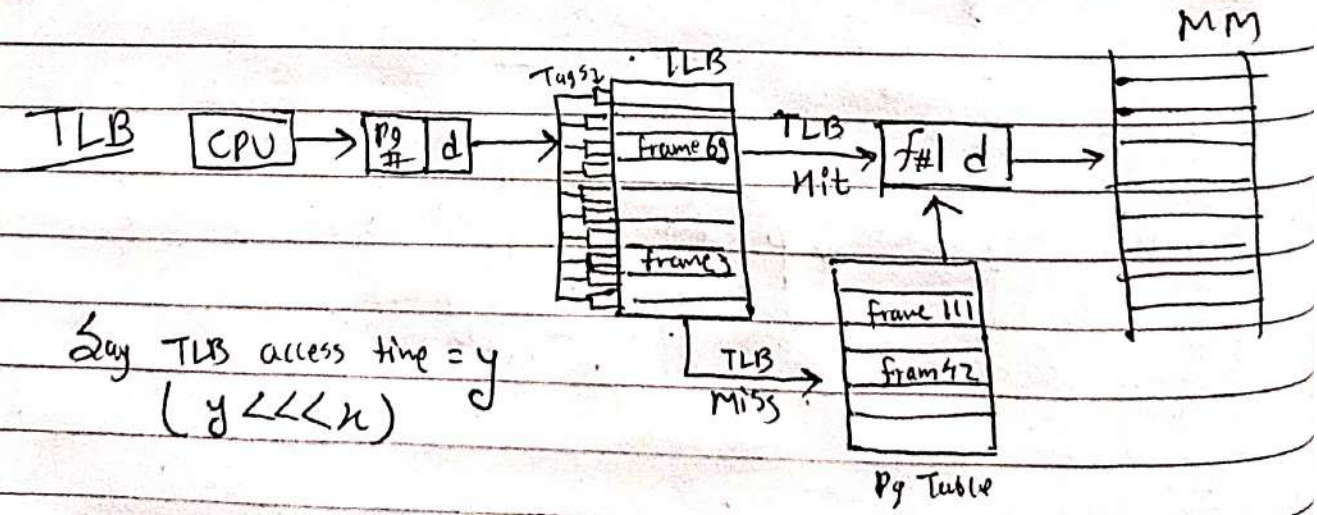


Say memory access time is n, $\Rightarrow$ Total time = $2n$ (EMAT)

- We assume no page fault (always valid page)

This $2n$ could be $3n$, $4n$... is pg Table is too big for a frame & we've to make multilevel page table

- Cache Memory (TLB) is vvfaster than RAM is used. but extremely low in size.
- We put frame # in TLB not all but as much as possible & remaining stays in pg Table.



Say TLB access time = y ($y \ll\ll n$)

Total Time a) Hit = $y + n$

b) Miss = $y + 2n$ } EMAT

Try to reduce misses & ↑ hit

- It's soldered on CPU, L1 cache is in KBs order.

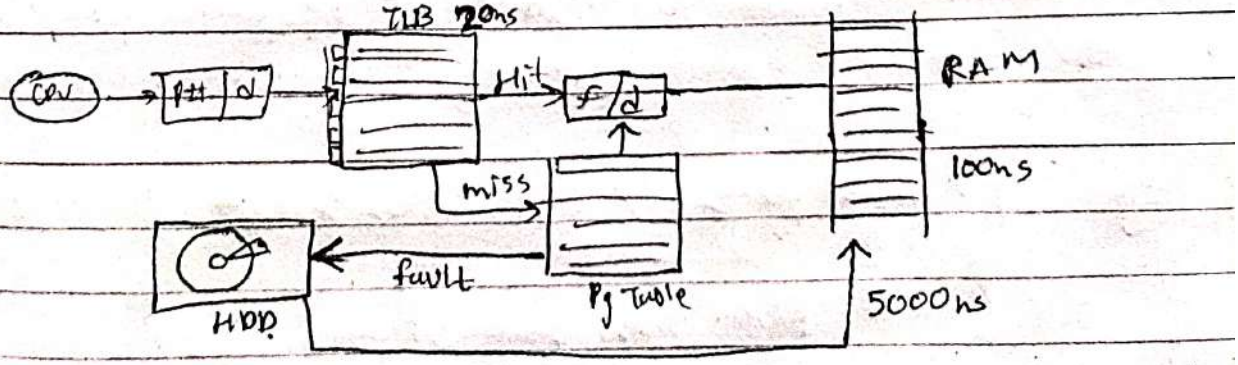
Q1: TLB access time is 10ns & MM access time is 50ns. What is EMAT (in ns) if TLB hit ratio is 90%. Assume no page fault.

Soln: EMAT = .9(50+10) + 0.1(10 + 2×50)
= 9×6 + 11
= 65 ns

Extra: if page fault occurs we'll have to add (PFST).

Q2: Paging system has 1 level Pg Table & a TLB. M/M access time is 100ns. TLB lookup time is 20ns. Pag transfer from HDD to M/M takes 5000ns. TLB Hit ratio is 95%, page fault rate is 10% & Assume that for 20% of page faults, a dirty page has to be written back to disk before required page is read in from disk. find < M/M access time > assuming TLB Update time is negligible.

Soln:



- 95% Hit, 5% miss, t = .95(120) + 0.05(.9(220) + .1(pfault))
10% pfault & got from Table
80% (PFST) & 20% Dirty page write

pfault time = 80% normal pg service & 20% Dirty write then Read

$= .8(5000+100) + .2(2\times5000+100)$

$= 8\times510 + 1000\times2$

$= 4080 + 2020 = 6100$

$t = 0.95(120) + 0.05(.9(220) + 0.1(6100))$

$= 114 + 0.05(198 + 610)$

$= 114 + 0.05\times808$

$= 114 + 40.4 = 154.4$ ns

- **Page Replacement Algo** : The swap-in & swap outs done in Virtual Memory is done by various algorithms.

1) FIFO
2) Optimal Page Replacement
3) Least Recently Used (LRU)
4) Most RU

1) **FIFO**: If all frames are full & we want to add a required page then the oldest page residing there gets replaced.

eg: CPU wants pg → 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 0

Pg Services & our RAM has only 3 frames.

Pg Services	7	0	1	2	0	3	0	4	2	3	0	3	2	0
	*	*	*	*	Hit	*	*	*	*	*	*	Hit	*	*	Hit
f_2	-	-	1	1	1	1	0	0	0	3	3	3	3	2	2
f_1	-	-	0	0	0	0	3	3	3	2	2	2	2	1	1
f_0	-	7	7	7	2	2	2	2	4	4	4	0	0	0	0

→ time

Hits = 3, Faults = 12, Total Asked = 15
20%. 80%. 100%.

Q: System has 3 frames (all empty initially). Find Hit ratio if Refrence string is 1 2 3 4 1 2 5 1 2 3 4 5

Soln
f3 1 2 3 4 5
f2 2 3 4 5
f1 1 2 3 4 5
✓ ✓ ✓ ← Hits

3 hits & ofc ⇒ 9 faults

Q: Repeat for ~~4~~ frames

Soln
f4 1 2 3 4 5
f3 1 2 3 4 5
f2 1 2 3 4 5
f1 1 2 3 4 5
Hits
✓ ✓

⇒ 2 hits & 10 faults

- Even after ↑ing frame #s our hits insted of ↑ing actually decreased it's called Belady's Anomaly in FIFO. (Hence FIFO is not v. good algo)

2) Optimal Page Replacement: Replace the page which is not going to be used in near future. or will be called v. late.

Say we've 4 frames in our system & Refrence String is following —

P.I.O

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

f_4	-	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3
f_3	-	-	-	1	1	1	1	1	4	4	4	4	4	4	1	1	1	1	1	1
f_2	-	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f_1	-	7	7	7	7	7	3	3	3	3	3	3	3	3	3	3	3	7	7	7
	*	*	*	*	Hit	↓	Hit	*	H	H	H	H	H	↑	*	H	H	H	*	H	H

3 will be swapped with either 7,0,1,2 but best will be 7 coz it's 3rd last

1 can be swapped with 4 or 3 (never called again)

Hits = 12, 60% Fault = 8, 40%

3) LRU: We replace page which is least recently used in past.

Say 4 frame sys got ref string → 7012030423032120170

f_4	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
f_3	-	1	1	1	1	4	4	4	4	4	4	1	1	1	1	1	1	1	1	1
f_2	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f_1	-	7	7	3	3	3	3	3	3	3	3	3	3	3	3	3	7	7	7	7
	4*	Hit	*	H	*	H	H	H	H	H	*	H	H	H	*	H	H			

3 wants to come

4 wants to replace

1 wants to replace

7012030 | 4

7012030 | 4

4 is farthest left | 1

Hit = 12, ~~Miss = 8~~ Fault

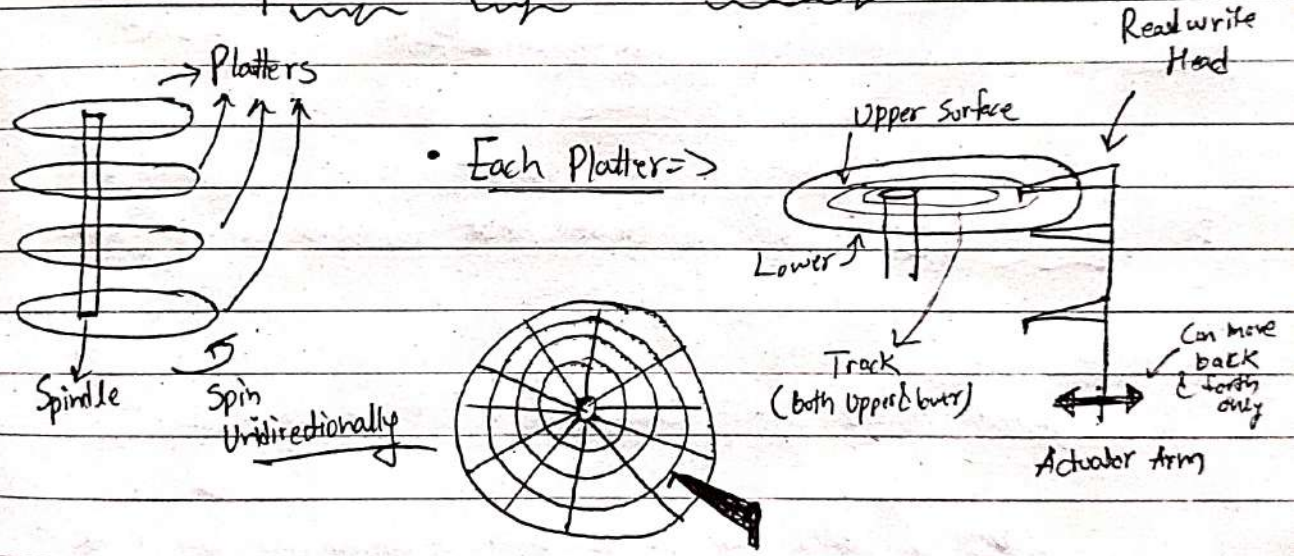
- Disadv: Slow speed than FIFO (has to always look in past) But still better work.

4) MRU: We replace page which is most recently used in past.

Assume 4 frame RAM & same refrence string as previous

		Finally	
=>	f_4	0	Hits = 8
	f_3	1	Faults = 12
	f_2	4	
	f_1	7	

Hard-Disk Architecture



- Each Platter =>
- Arms Back & forth movement helps to switch tracks.
- Each Track has fixed no. of sectors.
- Data is put on Data

- Platter → Surface → Track → Sector → Data.

Q: HDD has 8 platters, 256 Tracks on each platter & each Track has 512 sectors. Find total sectors in HDD & capacity if each sector can hold 1MB

Soln/
→ $8 \times 2 \times 256 \times 512 \times 1\,MB$
= 2TB

Disk Access Time :

go only in desired track

Extreme difference

its actually avg case $\left(\frac{0 + \text{full}}{2}\right)$

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

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

3) Rotational latency : Time taken to reach to desired sector. (half of rotation time) Avg

4) Transfer Time : Data to be transf. aka Transfer ~~Rate~~ Time
Transfer rate

- Transf Rate = Data rate = No. of heads (surface) × Capacity of 1 track × No. of Rotation in 1 sec.

5) Controller Time : It's a driver that controls whole HDD.

- Disk Access Time = ST + RT + TT + CT + QT

HDD is slower than PC so when a lot of reqs are made HDD store request in buffer as queue.

Disk Scheduling Algorithm : Goal is to reduce seek time.

1) FCFS
2) SSTF (Shortest Seek time First)
3) SCAN
4) LOOK
5) C-SCAN (Circular)
6) C-LOOK (")

① FCFS:

Q: Disk has 200 tracks (0 to 199). Request Qeue has track no. 82, 170, 43, 140, 24, 16, 190 respectively.

Current position of R/W Head = 50. Find Total # of track movements by R/W head.

Soln 50→82 (32 movements), 82→170 (88), 170→43 (127)
43→140 (97), 140→24 (116), 24→16 (8), 16→190 (174)

Total = 32+88+127+97+116+8+174
= 247 + 213 + 182 = 642 Ans

Advantage: 1) No starvation as sequence is known already

Disadvantage: 2) Performance could've been better @ one point we were @ 170 & in future we were supposed to go to 190 it would've been better if we did that from 170.

② SSTF:

Q: Repeat prev. question.

Soln Diff from 50 → 32, 120, [7], 90, 26, 34, 140
(43)

Diff from 43 → 39, 127, ✓ done, 97, [19], 27, 147
(24)

Diff from 24 → 58, 146, ✓, 116, ✓, [8], 166
(16)

Diff from 16 → [66], 154, ✓, 124, ✓, ✓, 174
(82)

Queue -> 82, 170, 43, 140, 24, 16, 190

Diff from 82 -> ✓, 88, ✓, [58], ✓, ✓, 108

(140)

Diff from 140 -> ✓, [30], ✓, ✓, ✓, ✓, 50

(170)

Goto 170 to 190 (no choice else Id) -> [20]

Total movements = 7+19+8+66+58+30+20

= 34+66+108

= 108 + 100 = 208

Adv: 1) Tries to give optimal result.
(could be worse in some cases)

2) Response time is better (any process can get response doesn't matter where in q...

Disadv: 1) Starvation is possible if a request is quite far from other track.

2) Overhead exists, i-e always have to subtract from initial track over & over-again.

(3) SCAN: aka Elivator algo. We move in ~~1~~ till ~~199~~ v.last possible location.

Q: Repeat: for direction going to larger value.

Soln 50 → 82 → 140 → 170 → 190 → 199 (v.last)
but useless

199 → 43 → 16. { NOT (0)
last req

ans -> 50 to 199 (149)
199 to 16 (173)

Ans = 332

Q_2: Repeat for direction going to lower value

Soln 50 → 43 → 24 → 16 → 0 (v.last)

Now, 0 → 82 → 140 → 170 → 190 & NOT (199)

dns = 50 to 00 (50) + 0 to 190 (190)

= 240 ms

- Why goto v.last? coz dynimically it's possible that a req can come in same dir but more/less than last. So it's good practice to go till extreme

Disadv: Cannot change direction, say we scanned till 199 then changed dir. & 195 req came we'll not be able to answer it immediately.

④ LOOK: Similar to scan. We move in 1 direction but DO NOT go till v-last.

Q_1: Repeat for direction greater ~~than~~ value.

Soln 50 → ... → 190 (140)
& 190 → ... → 16 (174) => 314

Q_2: Smaller ~~th~~ value

Soln 50 → 16 (34)
& 16 → 190 (174) => 208

Adv: Better than SCAN coz we don't unnecessarily goto last

(5) C-SCAN : III^rr to SCAN but come back to other extreme. DO NOT serve request during it.

Q1: Repeat for direction to larger value.

Soln

50 → 82 → 140 → 170 → 190 → 199 (v.last) (149)
199 → 0 (199)
0 → 16 → 24 → 43 (43)

Ans = 391

Q2: Repeat for dir to smaller value

Soln

50 → 43 → 24 → 16 → 0 (v.last) (50)
0 → 199 (199)
199 → 190 → 170 → 140 → 82 ⊗ (117)

Ans = 366

(6) C-LOOK: III^er to LOOK but come back to other highest/lowest request & DO NOT serve requests during this

Q1: Repeat for direction to larger value.

Soln

50 → 82 → 140 → 170 → 190 (140)
190 → 16 (174)
16 → 24 → 43 (27)

Ans = ~~382~~ 341

Q2: Repeat for direction to lower value

Soln $50 \to 43 \to 24 \to 16$ (44)

$16 \to 190$ (174)

$190 \to 170 \to 140 \to 82 \to 50$ (140)

Ans = 358

Q: HDD has 200 Tracks & follows C-SCAN algorithm & has following reqests in queus 95, 180, 34, 119, 123, 62, 64 R/W Initially @ 50. Head moves towards smaller value. on servicing. Given seek time is 2ms. Find Total seek time assume moving ~~from~~ one end to another takes 10ms.

Soln In scan we go to v.last. (It's circular also)

$50 \to 32 \to 0$ (v.last) (50) × 2ms

$0 \to 199$ ~~(199)~~ 10ms

$199 \to 180 \to 123 \to 119 \to 95 \to 64 \to 62$ (137) × 2ms

Total movements = ~~2[illegible]~~ 87 + 1 (one end to another)

time = ~~[illegible]~~ 187 × 2 + 10 ms

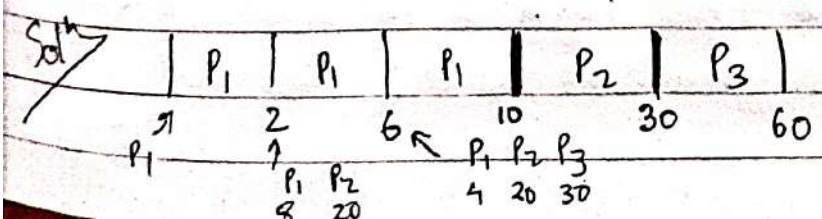
= 374 + 10 ms = 384ms

Q2: Find # of context switches if SRTF is used.

Proc	AT	BT
P_1	0	10
P_2	2	20
P_3	6	30

2 switches

Sol



File Systems

- One of major functionality of OS, It manages how data is stored & fetched

- Window → NTFS, FAT32
  DOS → FAT
  LINUX → Extended
  Bigdata → ZFS (Zettabyte file

- file system maps data (divided in block) logically in HDD's sectors

- Operations on files
  1) Creating
  2) Reading
  3) Writing
  4) Deleting (File Gone)
  5) Truncating (File Empty)
  6) Repositioning (of R/W Head)

- File Attributes (Metadata)
  1) Name
  2) Extension Type
  3) Identifier (Unique no. to file given by OS)
  4) Location
  5) Size
  6) Modified data, created date
  7) Protection / Permission
  8) Encryption, Compression

- Repositioning: a b c d e 1 2 3 f 7
  ↑ Head ↑ direct reposition to here

  in Linux done by LR seek command

3/4/1

Important Commands:

1) Unix/Linux : They're case sensitive

a) who – displays all users logged in to system currently along with terminal line, date & time

eg: davey tty2 2021-09-13 09:35 (:0)

b) pwd - print working directory

c) mKdir <folderName> - Creates newfoder with given name.

d) rmdir <folderName> - Deletes foder with given name.
(file removal) but must be an empty folder.

e) cd - changes directory., cd .. → go back, ~ → root

f) ls - lists directory's all contents

1. ls -t => Sorts by time modified
2. ls -l => long info (date, time, size, type, mode, etc)

eg: (-rw-rwxr--)
type | user permission | group | others

• permissions

mode	aka
read	4
write	2
xecute	1

• type

symbol	meaning
-	normal file
d	directory
s	socket file
l	link file

3. ls -lh => long info human readable
it just changes size to KB, MB, etc.

4. ls -a or -A => All info (even hidden files)
↳ starts with dot (.)

5. ls -lt => Long info but sorted by time modified.

Editors in Linux : Vi/vim, nano, Gedit

'i' to edit then Esc :wq Enter

g) touch <fileName.ext> → Creates empty file.

h) cp <file> <destiFolder> – copies file.

i) mv <file> <destiFolder> – moves file

j) chmod <instruction> <file.ext> – changes mode of file acc to instruction.

Instructions can be : • +x (makes file executable for all 3)
similarly +w & +r
• –x, –w, –r (removes the mode)
• +666 or 666 (r&w permission to all 3) (last 2)
• –22 (removes write permission from 2 categories)

K) Cal – Displays Calendar

l) file – Linux treats everything as a file so
file <anything> ⇒ o/p Thing : Directory or File.

m) Sort <file.ext> – Sorts contents of file by ASCII

n) grep joemama Terms.txt – Globally search a Regular Expression & Print it,
it will search "joemama" in the file & if found, it'll print the whole line.

o) man <command> – Gives a user manual for the command

p) lpr <file.ext> – Sends file to printer for printing

To open txt files use cat <file.txt>

q) passwd — for changing user's password

r) clear — clears up terminal for better look.

Q_1: which command is used to assign read only to all 3 categories?

A) chmod a-rw B) chmod go +r C) chmod ugo=r
D) chmod u+r, g+r, o-r

Ans - C

Q_2: Octal representation of chmod ugo+rw is?

A) Chmod 555 file B) chmod 666 C) chmod 333 D) chmod 444

Ans - B

Q_3: file has content 1234567890 abcdefghij & we performed lseek(n,10,SEEK_CUR); then lseek(n,5,SEEK_SET); n is file descriptor. Find current position of R/w head.

Soln lseek is not only a command but a sys call.

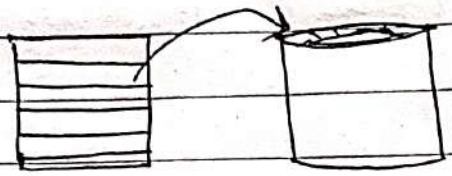
1) seek to 10th position from current.(0)
2) seek to 5th position from current (0) & SET it => curr(5)

Ans

Imagine order of commands reversed =>
1) seek to 5th position from curr(0) & set it => curr(5)
2) seek to 10th position from curr(5) i.e 15th

File Allocation Methods

- Contiguous
- Non-contiguous
  - Linked list
  - Indexed

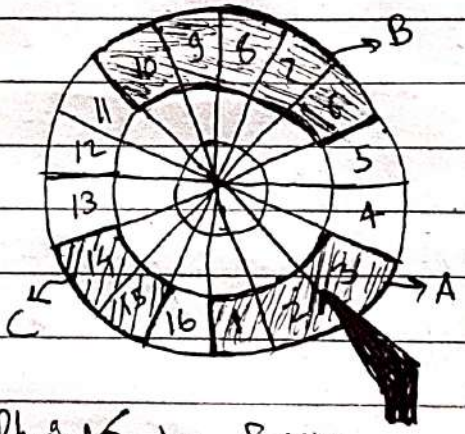


File (logically divided) HDD (physically divided in sectors)

These part of file can be allocated to Sectors in either contiguous or non-contiguous fashion

- Goals: 1) Efficient Disk utilization (avoid fragmentation)
  2) Fast Access

Contiguous:



Physical Sectors of HDD

Directory:

File	Start	length
A	2	3
B	6	5
C	14	2

- Advantages:

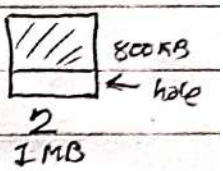
1) No overheads & easy implementation:
As there's no random allocation we don't need pointers & index file to know where the parts of a file are

2) Excellent Read performance: Once R/W Head reached the correct Track, it'll just have to spin. & evrything will be read in 1 go smoothly.

3) Direct Access: If we reach right start, we know every locat

• Disadvantage

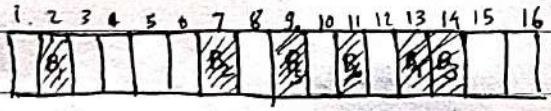
1) Fragmentation : Lot of gaps can be seen. & unless n until file of hole's size comes it'll remain a hole & unused space.

We're taking about External fragmentation, internal → 

coz internal fragⁿ is inevitable. But External is huge. here.

2) Difficult to grow file. : File A can expand 3 blocks @ most so if we try more than that it's not possible.

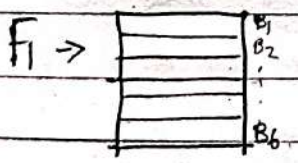
• Linked-list : Comes under non-contiguous allocation



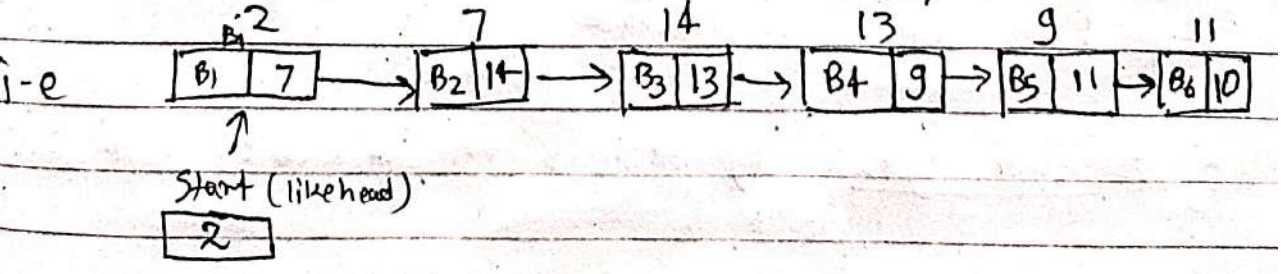
Physical Sectors of HDD

Directory

File	Start
f1	2



• We only have start so how will be find next blocks?
We use pointers that points to next

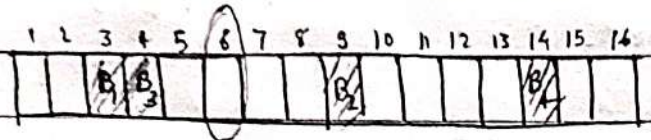


• Advantages : 1) No External Fragmentation
2) File size can increase

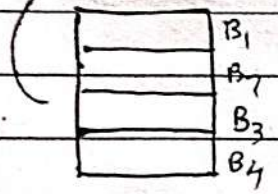
- **Disadvantage:**

1) Large seek time
2) Overhead of Pointers
3) Random / Direct Access is Difficult. (LList traversal)

- **Indexed:** Comes under non-contiguous allocation.



Directory	
File	Index-block
A	6



- 6th Sector will tell acting as an Index

B_1 is @ 3
B_2 " 9
B_3 " 4
B_4 " 14

- Each file will have index block in HDD

- **Advantage:** 1) Supports Direct access, becoz we don't have to traverse like we did in LList, every location is written in index

2) No external fragmentation
3) File size can increase

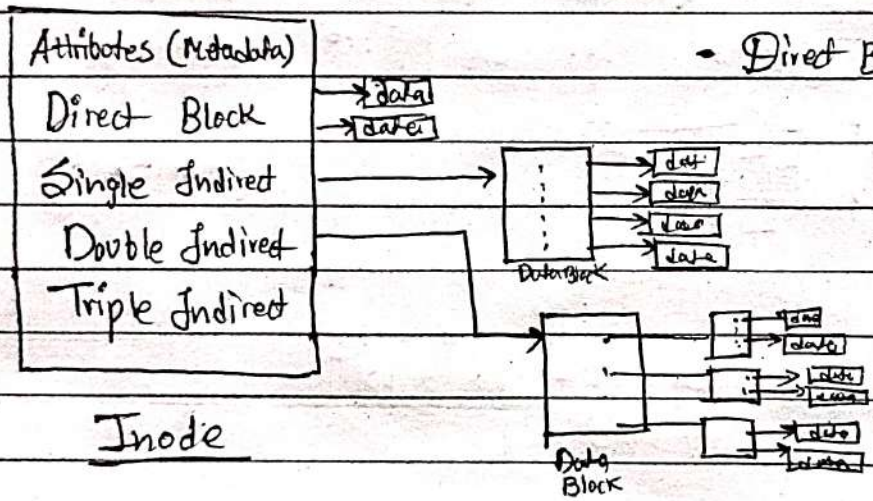
- **Disadvantages:** 1) Huge pointer overhead: start points to index & each Block have pointers pointing to their location.

2) Multilevel Indexing: If file is too large

the index will also be big and probably cannot fit in a sector. So we further divide it like a file. So extremely unconvinient because we'll have to divide it, allocate new pointers & waste large no. of sectors just for index.

- Unix uses Index node (Inode)
  ↳ block

Unix Inode Structure:



- Direct Block · directly points to data in sectors (blocks) of HDD

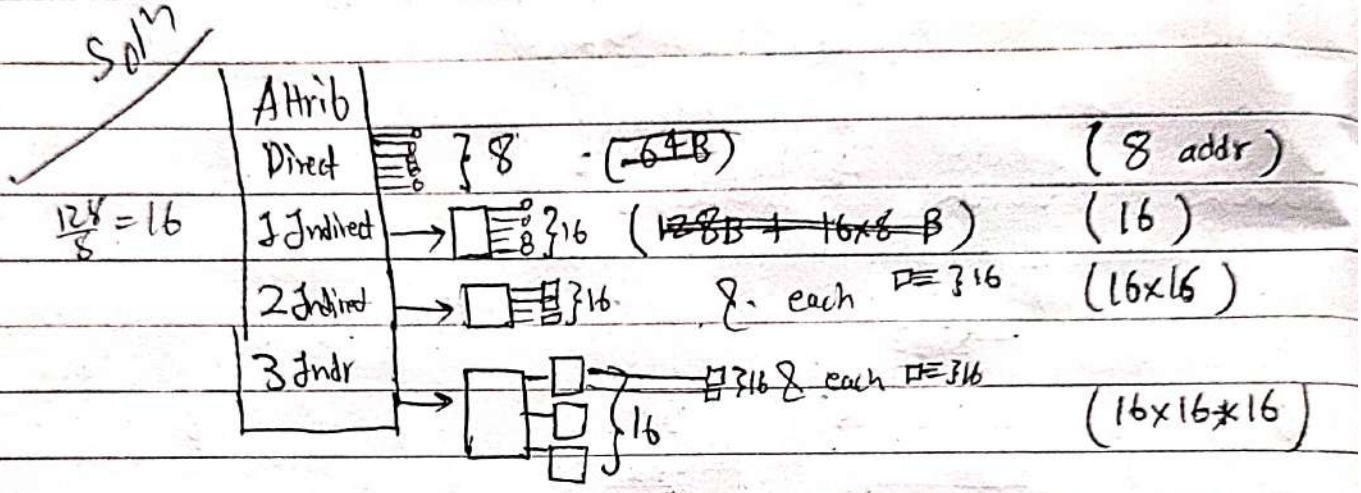
- Single Indired points to pointers, these pointers points to sectors (blocks) in HDD

- Double Indired points to pointers, these pointers points to Data block (other set of pointers) that stores address of blocks.

- Triple Indired → You should get the pattern by now, if not you're retarded.

Q: A file system uses Unix inode data structure having 8 direct block address, one (1) indirect block & 1, 1 double & triple indirect blocks. Each Block's size is 128 Bytes & size of each block addr is 8B. Find max possible file size.

Soln



Total Addresses = $8 + 16 + 16^2 + 16^3$
= 4376.

Max size = 4376 x [Data] → 128B
= 547 KB

Protection & Security

- Goals: 1) In 1 protection model, Computer consists of a collection of objects H/W or S/W

2) Each obj has unique name & can be accessed

through a well defined set of operation.

3) Protection Problem - Ensure that each obj is accessed correctly & only by those processes that are allowed to do so

- **Security Violation Categories**

It be OS or other subject, if security is topic of concern remember CIA (Confidentiality, Integrity, Availability)

1) Breach of C: Unauthorized reading of data

2) Breach of I: Unauthorized modification of data

3) Breach of A: Unauthorized deletion of data.

4) Theft of Service: Unauthz use of resources.

5) Denial of Service (DOS): Prevention of legitimate actual use by overloading buffer

- **Principles of Protection**: Use principle of least privilege

1) Programs, Users & systems should be given just enough permissions to perform their task

2) It'll limit the damage if entity has a bug or was trying to abuse.

Permissions

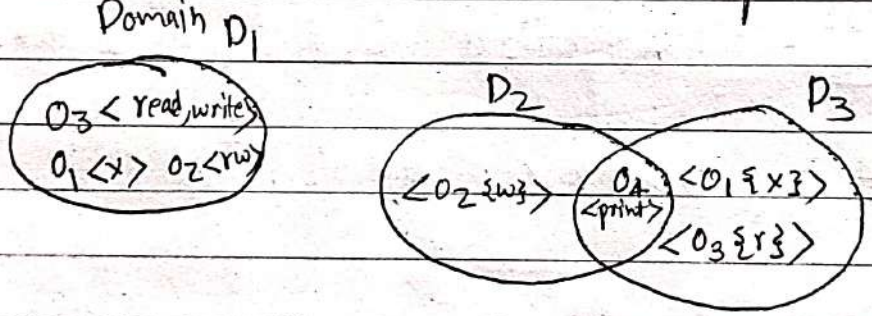
3) Can be static or dynamic (Domain switching, privilege escalation)

• Domain Structure

Domain is set of rights / permissions that a user or a program or a system has.

Access-right = <obj-name, rights-set>

rights-set: subset of all valid operations that can be performed on objects



Access Matrix : ↓ Domain → obj

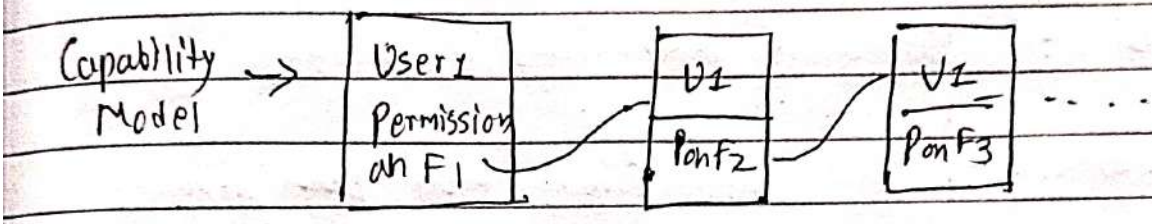
Access(i,j) means Set of operations that a process executing in D_i can invoke on Obj_j

eg:

Dmy \ obj	F1	F2	F3	Printer
D1	r		r	
D2				print
D3		r	x	
D4	rw		rw	

(files)

- If there are lot of users & each has diff-diff permission then matrix will get toobig. So we can try 1) Making ~User Obj Permission Table
  2) Capability Model



- Security Problems: A system is secure if resourced used & accessed as intended under all circumstances (Unachievable lol)
  - Intruders (Crackers) attempt to breach security
  - Threat is potential security violation, could not actually be dangerous but has done some unauthorized tasks mentioned before
  - Attack is an attempt to breach security

- Security Violation Methods:

1) Masquerading: Pretending to be an authorized user to get privileges

2) Replay Attack / Man-in-the middle attack: Intruder sits in dataflow masquerades as sender to receiver & receiver to sender &/or modifies msg.

3) Session Hijack: Intercepts session to bypass authentication (Banks Timeout)

Security Measure Levels

1) Physical : Servers, terminals, etc.

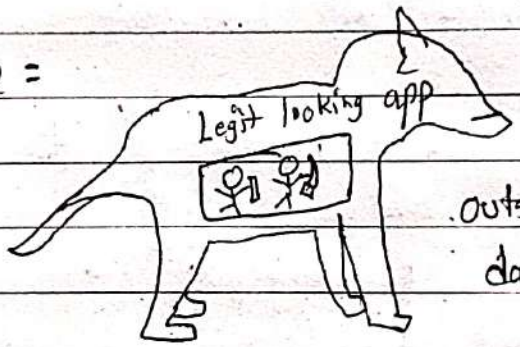
2) Human : Phishing, etc.

3) OS : Protection mechanisms & debugging.

4) Network : Intercepted Communication, DOS prevention

Program Threats :

1) Trojan Horse =



Software that looks good & legi outside but has pote dangerous software hidde

2) Malware : Any Malicious Software

3) Spyware : Spying Software

4) Virus : Tries to damage computer & spread from one PC to another, requires to be executed

They're a) Self-replicating
b) Spreading needs help from human
c) Specific to CPUs, OS & application.
d) Often spread via email or as macro

5) Worm : Sub-class of virus but can spread without help & can send

hundreds or thousands copies of itself using N/w of system.

eg: sending itself to all contacts in your e-mail.

- DOS can also be achied by forK() system calls Kla forK bomb.

Firewalls: It's placed b/w trusted & untrusted hosts.

It limits n/w access b/w these 2 domains.

- Firewle of 3 types
  1) Personal
  2) Application proxy
  3) System-call.

(1) Personal → Controls/Monitors traffic on S/w layer
(2) Appl^n proxy → Understands application protocol & controls it. (eg: SMTP)
(3) Systemcall → Monitors all imp sys calls & applies rules to them.

- We can also add our own: personal rule, like block N/w connection from/to GTAV.exe

- Windows user Kerberos internet protocol (Computer N/w authentication protocol).