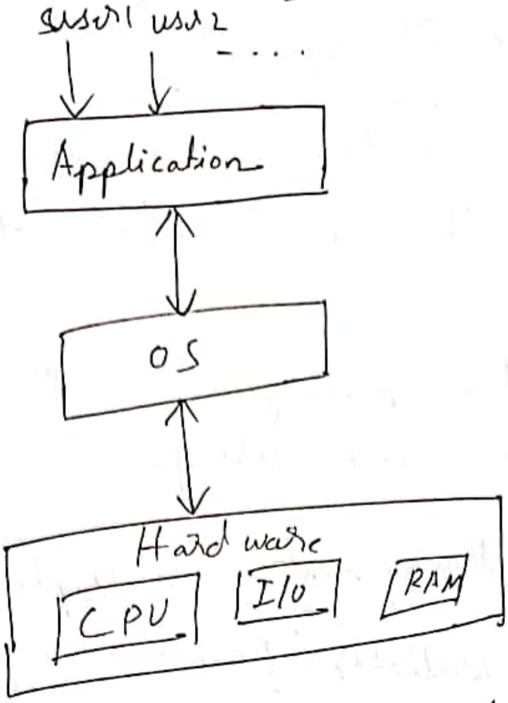
Operating System and its function :=>



* OS is a system software which works as an interface between user & Hardware

Why? → If there is no OS then user have to write Program for each operation. (We want to print something instead of just clicking. We have to call printer with Program)

- Suppose a user is directly accessing a specific hardware then if any other user want to access it then this is not possible, there is no authority of Resource management. OS solve this problem.

Primary goal of OS is to provide Convenience (easiest way to access)

Throughput : Number of tasks Performed in One unit time. (Linux throughput is high)

Functions:

(i) Resource Management: When many user try to access a hardware / Particular system (Server l.. OS provide the resources for specific amount of time.

In task manager We can See CPU howmuch Hardware how much using.

(ii) Process Management: Multiple Processes are executing Parrallally (Paint, ward, music etc.):
With the help of CPU Scheduling Process are managed.

(iii) Storage Management: ⇒ By the help of file system OS manage storage (Secondary Memory (harddisk)

(iv) Memory Management: ⇒ RAM is limited.
Allocation & deallocation of Ram to a process.

(v) Security: ⇒ To keep the password in secured file Provide security between Processes.

Types of OS

(i) Batch
(ii) Multiprogrammed
(iii) Multitasking
(iv) Real time OS
(v) Distributed
(vi) clustered
(vii) Embedded

(i) Batch: Program is loaded in Punch card (Paper tape / Magnetic tape. Then it is given to operator (in the company). Operator make batches of similar kind of works.

B_1 B_2 B_3

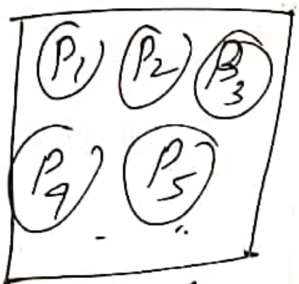
CPU → I/O → CPU

After completing first batch it goes to next batch

- There is a lot of idle time.

Later IBM introduced Fortran (Monitor enabled) for eliminating operator.

(ii) Multiprogrammed: ⇒



RAM

- non Preemptive ⇒ P_1 goes to CPU Then when P_1 goes to other device then CPU takes P_2 and so on.

But if P_1 needs some other operation then only

CPU goes to P_2. If there is no operation then it first complete P_1 then P_2

- CPU is not idle anytime

(iii) Multitasking / Time sharing :=> Realtime

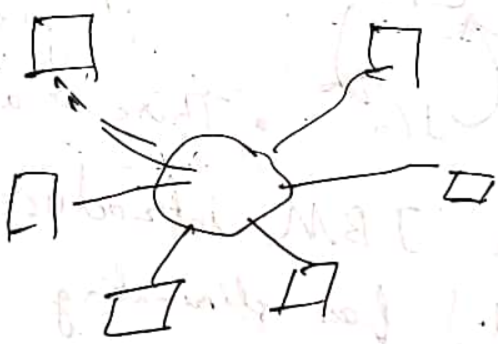
- Preemptive : After a specific time CPU automatically goes from P_1 to P_2, and so on
- Responsiveness : Every Process is executed Partially at a time.

(iv) Real time OS :- Windows CE
RT Linux

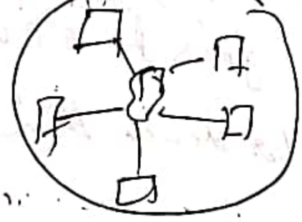
(a) Hard - Missile launch

(b) Soft → Gaming, youtube live Session

(v) Distributed - Geographically separated computer connected by network



(vi) Clustered :=> Multiple devices connected by LAN.



- Load balancing
- Availability

(vi) Embeded ⇒ It works on a fixed function
Microwave, AS Washing Machine can do only its own work. We can't chang its function But in computer we can change.

Process State: ⇒

Process is a program in execution

- New state: Creating a new process in secondary Memory.
- Ready state: The process came in RAM

⊛ Long term scheduler (LTS) takes Process from New state to Ready State.

- Running state: One Process is dispatched to Running state (executed by CPU) (In uni Processor System (1 CPU) only can run a process at a time)

- Terminated state: After finishing execution of process it is terminated (Deallocation of RAM as it is limited)

⊛ When any process is running but if higher Priority process came / the process was given specific time, then the running process came to Ready State.

(*) Short time Scheduler (STS) takes Process from Ready to Running State.

Non Preemptive :=> Doesnot care about Priority or time quantum. Only care I/O request

Preemptive :=> care about Priority / time quantum.

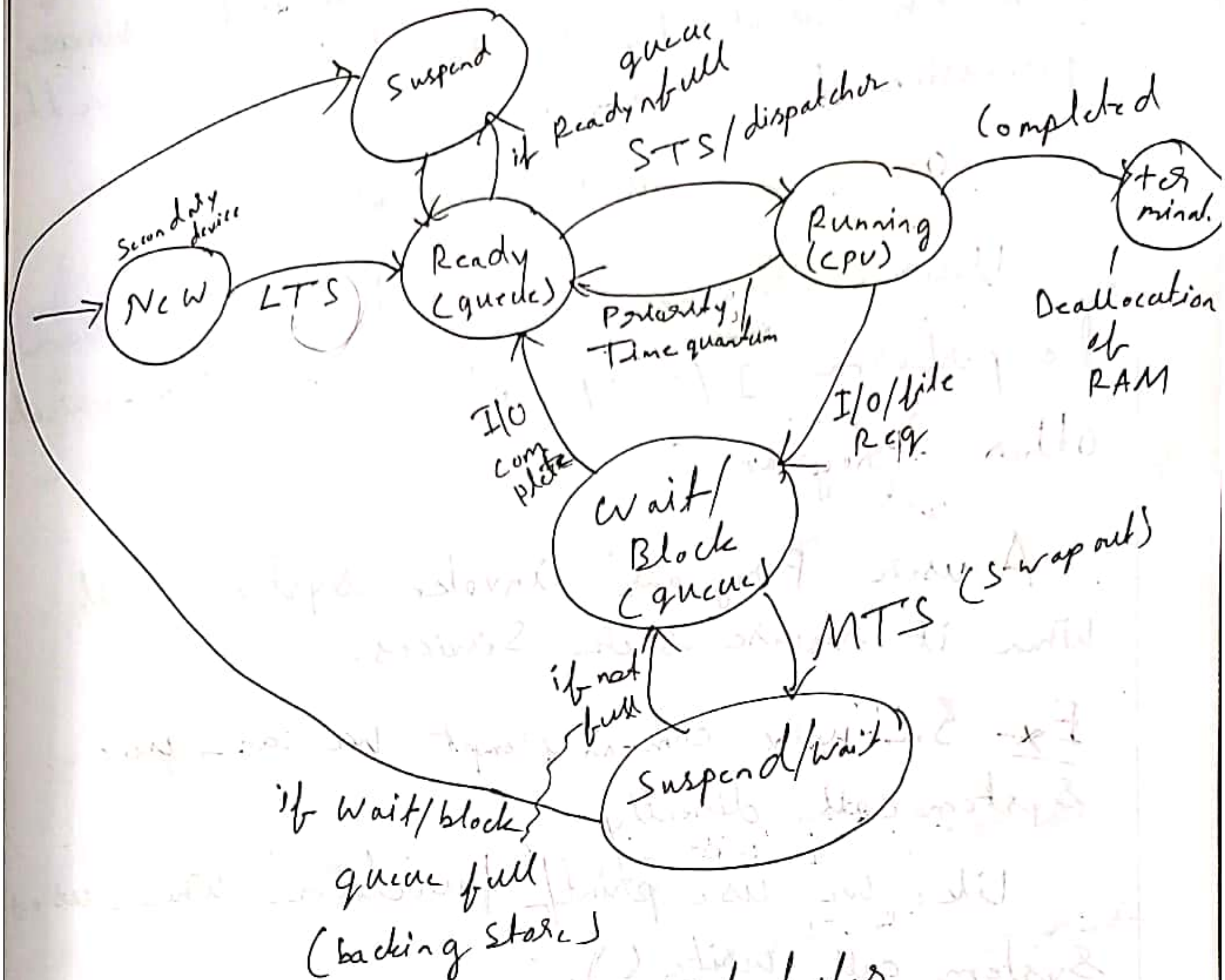
Wait / block state :=> if any process need I/O Request then it goesto wait / block state.

After I/O complete it goes to Ready state.

Suspend State :=> If a lot of Process need I/O operation and RAM is filled then We take some process to suspend state. When queue is filled.

Some Process are taken to Secondary Memory for I/O operation. (Swap out)

This is done by Mid time Scheduler (MTS)



LTS - Long term Scheduler
STS - short term scheduler
MTS - Mid term scheduler

System Call - It is a Mechanism using Which a user program can request a service from the kernal for which it does not have the permission to perform.

OS governs all the resource

User Program do not have Permission to perform I/O operation or communicate other Program.

A user Program invoke system call When it require such Services.

Ex- In unix comman prompt we can use System call directly

Like we use printf function which uses System call write ().

System call

- File Related: open(), Read (), Write() close() etc
- Device Related => Read, write, Reposition
- Information - getPid, attributes, get System time and data
- Process Control - Load, execute abort, Fork, Wait, Signal
- Communication - Pipe(), create/delete

Metadata :⇒ Data about data is called meta data.

if Image is itself a data and its attributes like size, pixels, time taken, picture format is meta data.

Fork() System call :⇒

Fork() System call is used to create clone Process (child process).

child process id is different from Parent Process.

Fork → 0 child Process
Fork → +1 Parent Process
Fork → −1 child Process not created

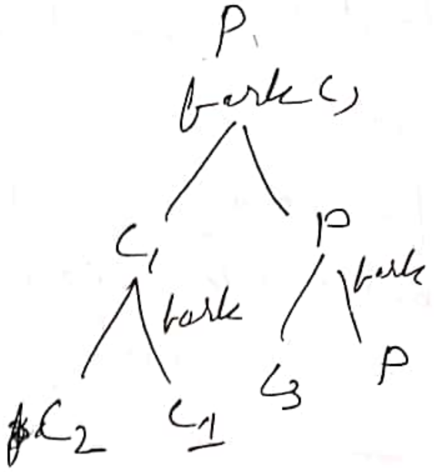
```
main ( )
{
  fork ( ) ;
  printf ("hello\n");
}
```

→ hello
hello

P
fork()
C_1 P

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

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



(*) child Process id returned 0
Parent Process id returned 1

Question –

```
main ( )
{ int a ;
    for (a=1 ; a<5 ; a++)
        fork()
  printf ("I") ;
}
```

How many times it will print I in output?

(a) 15 (b) 16 (c) 31 (d) 32

⇒ fork will be called 4 times

$2^4 = 16$ times I will be printed.

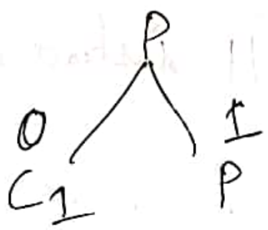
a)
```
main ()
{ if ( fork () && fork ())
        fork ();
    printf ("Hello");
    return 0;
}
```

How many times it will print hello in o/p

(a) 2 (b) 3 (c) 4 (d) 5

=> (*) After fork () the Program divided into Parent & child. Each Program execute the subsequent instruction after fork.

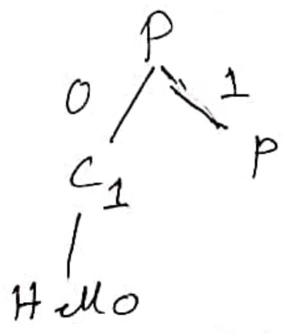
In this case fork () create C_1 & P



0 && 0 = 0

0 && 1 = 0

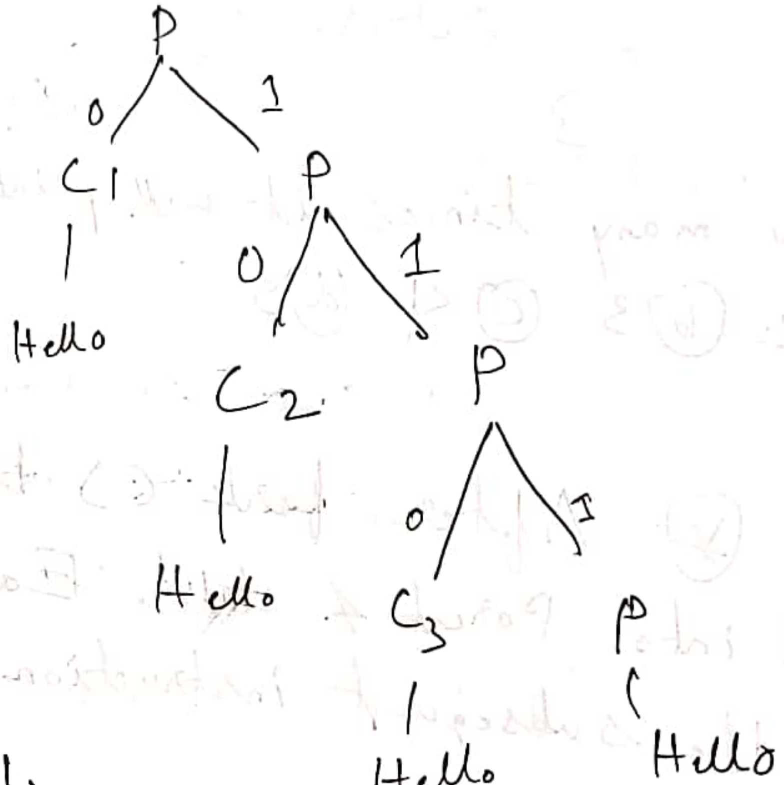
So the second fork() not checked as it is false (if (0)) and control goes to ~~fork () of if block~~ Printf ().



now case 2 if (1 && fork)
fork()

1 && 0 — 0
1 && 1 — 1

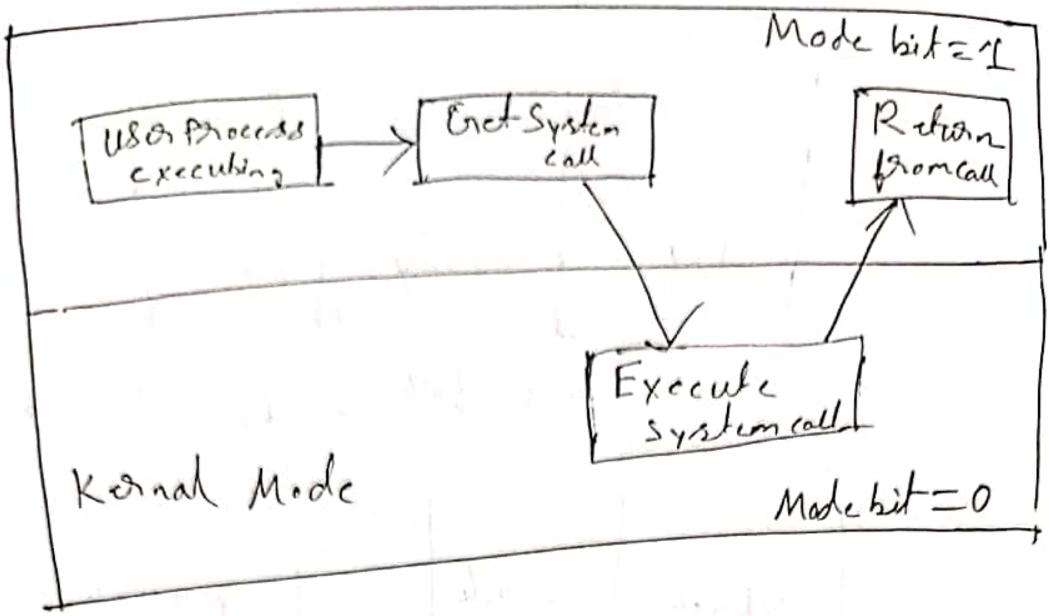
So next fork() in if ( ) will checked



→ (c) 4 times

(*) if there was || instead of && then it will be 5

User Mode & Kernal Mode :=>



In any C Program we have to read a file then we have to invoke system call (Read()) then it will get into Kernal Mode then after executing it will go to user Mode.

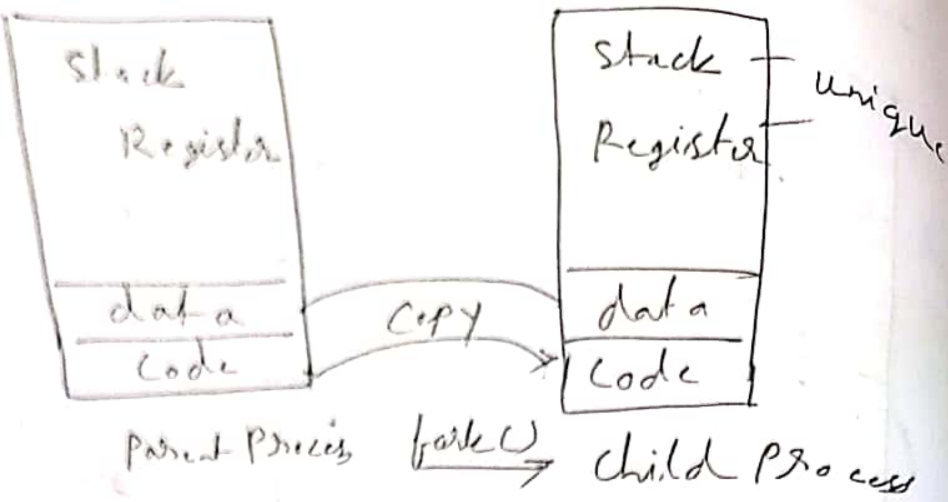
Ex- In bank you can not directly access to bank locker to take your thing, you have to approach Manager first
Withdrawing Money from bank counter.

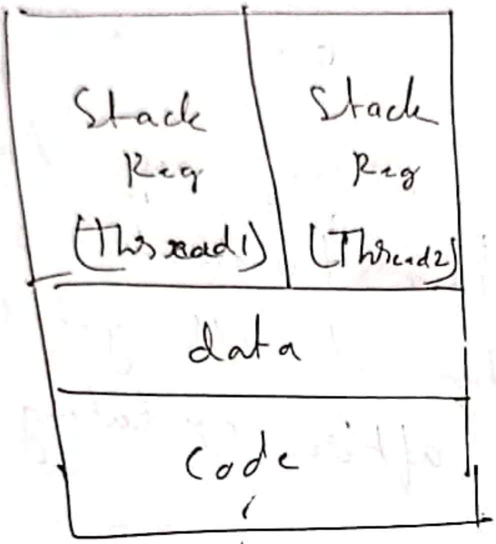
CPU Switches between user Mode & Kernal Mode.

2+2=4 — user Mode

Print 4 in monitor — Kernal Mode.

Process vs Thread :=>





moving in own room
moving to other room/house
calling mother

Process	Thread
(i) System call used	(i) usually no System call (user level Thread)
(ii) OS treat different process differently	(ii) All user level thread treated as single task for OS
(iii) Different process have different copies of code, data	(iii) Share same Memory 4 ~~data~~ code
(iv) Context switching is slower	(iv) context switching fas[ter]
(v) Blocking Process will not ...	(v) Blocking a thread will block entire Pro[cess]

(vi) independent	(vi) Interdependent

(*) Context switching of Process is slow because it have to write details on PCB (Process control Block) before switching

(*) Blocking a thread means it will go to Wait state. As kernal doesnot know how many thread available on the process and it treated as a single process So entire Process will be blocked

(*) Thread creation is user responsibility.

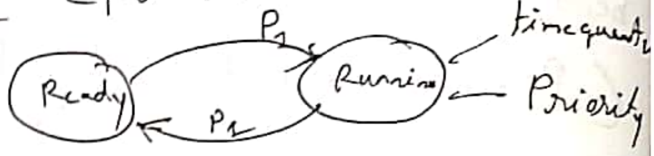
User level & Kernal Level Thread :⇒

User	Kernal
(i) managed by user program Library.	(i) managed by OS.
(ii) Faster	(ii) Slower
(iii) context switching fast	(iii) Context switching is slower
(iv) If one thread block then all blocked	(iv) If one kernal level Thread blocked, no effect on others

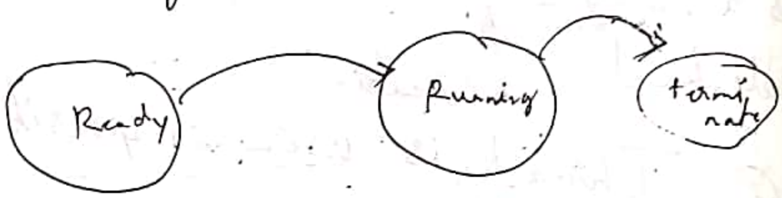
(*) as kernal know that other thread are different. Managed by OS.

Scheduling CPU :⇒

(i) Preemptive: In preemptive a process can be forced to leave CPU & switch to the ready queue.



(ii) Non Preemptive: / cooperating: a process keep the CPU untill it terminated or goes to waiting state.



Preemptive:

- (i) SRTF (Shortest Remaining time first)
- (ii) LRTF (Longest Remaining time first)
- (iii) Round Robin
- (iv) Priority

Non Preemptive:

- ✓ (i) FCFS (First come first Serve)
- ✓ (ii) SJF (Shortest job first)
- (iii) LJF (Longest job first)

(iv) HRRN (Highest Response Ratio next)

(v) Multilevel queue.

(vi) Priority

Analogy – coming to college then return home

(*) Arrival time: The time at which process enter the Ready queue or state. (point of time) - 8 am

Burst time: Time required by a process to get execute on CPU. (Duration) not included wait time. ~~1 hour~~

Completion time: The time at which process complete its execution.

Turnaround time: time spend in diff queue + time in CPU + time in diff o/p devices (Duration)

Turnaround time = Completion time − Arrival time

Waiting time: How much time it wait

Waiting time = Turn around time − Burst time. (Duration).

Response time: ⇒ (Duration)

Response time = The time at which Process get CPU first time − Arrival time

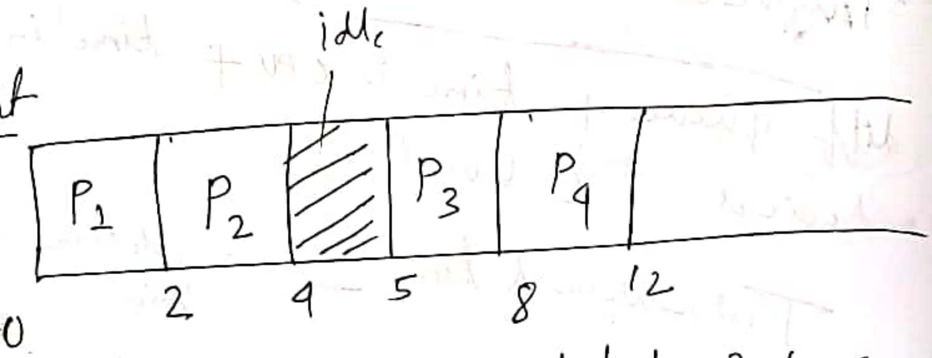
First Come first Serve (FCFS)

- non Preemptive
- high average waiting time
- Several small process may need to wait if a large Process given the CPU.

Numerical :- given

Process no	Arrival	Burst	Complete	TAT	WT	RT
P_1	0	2	2	2	0	0
P_2	1	2	4	3	1	1
P_3	5	3	8	3	0	0
P_4	6	4	12	6	2	2

Gantt chart



Suppose, at 0 baje P_1 came and it takes 2 hour and completed at 2 baje.
In this MeanWhile P_2 came at 1 baje and in ready queue. So P_2 run till 4 P.m.
But at 4 P.m. there is no process 4 CPU idle for 1 hour
At 5 P.m. P_3 came and run for 3 hour

Completation time = in which time Process executed

TAT = Completation − Arrival

WT = TAT − burst time.

$$\text{Avg TAT} = \frac{2+3+3+6}{4} = \frac{14}{4} = 3.5$$

$$\text{Avg WT} = \frac{0+1+0+2}{4} = \frac{3}{4} = 0.75$$

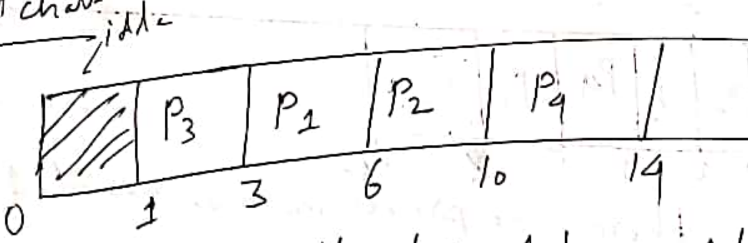
Shortest Job first (SJF) most effective

- non-Preemptive
- Process with smallest burst time will selected
- FCFS to break ties
  if AT same also then use lower Process id

Process	AT	BT	CT	TAT	WT	RT
P_1	1	3	6	5	2	2
P_2	2	4	10	8	4	4
P_3	1	2	3	2	0	0
P_4	4	4	14	10	6	6

* In non Preemptive $\boxed{WT = RT}$

Gantt chart



* first CPU is idle for 1 hour, At 1 P.m. there is 2 process (P_1/P_3) but P_3 has smallest burst time & P_3 is executed

At 3 P.m. there is 2 Process in Ready queue (P1 & P2) but P1 has smallest burst time So P1 will executed

At 6 P.m P2 & P4 both in Ready queue and same burst time.

We will break the tie with the help of FCFS. And P2 come first so it will be executed.

Shortest Remaining Time first :=>

- Preemptive
- FCFS to break

Preemptive SJF = SRTF

	AT	BT	CT	TAT	WT	RT
P_1	0	5	9	9	4	0
P_2	1	3	4	3	0	0
P_3	2	4	13	11	7	7
P_4	4	1	5	1	0	0

P_1	P_2	P_2	P_2	P_4	P_1	P_3
0–1	1–2	2–3	3–4	4–5	5–9	9–13

0 1 2 3 4 5 9 13

(*) ~~as~~ if there is no process in queue except 1 then we will start the execution of the process.

But we will constantly check every unit of time wheather any shortest burst time process came or not. If came then we will transfer the current process to ready queue and execute the new process.

P_1 runs 0 - 1 p.m. but after 1 p.m. P_2 came and in this point of time P_2 has 3 hour burst time & P_1 has $(5-1) = 4$ hour burst time so P_2 will run

P_2 runs for 1 hour and at 2 P.m there is P_1, P_2, P_3 in ready queue - $P_1 - 4$
$P_2 - 2$ - (Small)
$P_3 - 4$

Again P_2 will run for 1 hour and in 3 P.m.

$P_1 - 4$
$P_2 - 1$ - small
$P_3 - 4$

Again P_2 will run for 1 hour and P_2 completed.

At 4 P.m. there is P_1, P_3, P_4 in ready queue
4 4 1

P_4 will run

At 5 P.m. P_1 & P_3 both has 4 burst time but P_1 first came then it executed

$$\text{Avg TAT} = \frac{29}{4} = 6$$

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

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

Round Robin (RR) :⇒

- Preemptive
- a Small time quantum is defined
- each Process is allocated one time quantum

Given TQ = 2

Process	A.T.	BT	CT	TAT	WT	RT
P_1	0	5	12	12	7	0
P_2	1	4	11	10	6	1
P_3	2	2	6	4	2	2
P_4	4	1	9	5	4	4

We use Ready queue for storing the process queue

Ready queue

~~P_1~~	~~P_2~~	~~P_3~~	~~P_1~~	~~P_4~~	~~P_2~~	~~P_1~~

Running queue

P_1	P_2	P_3	P_1	P_4	P_2	P_1
0–2	2–4	4–6	6–8	8–9	9–11	11–12

At time 0 P_1 is in ready queue and it is run for 2 hour. In the time of P_1 goes to ready queue P_2 & P_3 came in ready queue. (According A.T)
(1 baje) (2 baje)

P_2 came in running state. In 4 P.m. P_4 also came in ready queue. As P_2 is not terminated and has 2 hour remaining B.T. So P_2 goes to ready queue

P_3 came in running state and 6 p.m. it terminated. P_1 again run for 2 hour and go to ready queue.
At 8 p.m. P_4 run for 1 hour and terminated

(*) In this 6 times context switches.

Priority Scheduling :⇒

- Preemptive

High no = High Priority

Priority	Process	AT	BT	CT	TAT	WT	RT
10	P_1	0	5	12	12	7	
20	P_2	1	4	8	7	3	
30	P_3	2	2	4	2	0	
40.	P_4	4	1	5	1	0	

P_1	P_2	P_3	P_3	P_4	P_2	P_1
0–1	1–2	2–3	3–4	4–5	5–8	8–12

if non-Preemptive

P_1	P_4	P_3	P_2
0–5	5–6	6–8	8–12

Mixed burst time / I/O :→

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

lowest number = highest priority.

Mode – Preemptive

P_1		P_2	P_3	P_2	P_1	P_3	P_1	P_2	P_4		P_2		P_4
0–1	1–2	2–3	3–5	5–6	6–8	8–9	9–10	10–11	11–13	13–14	14–15	15–17	17–18

I/O (1–6), I/O (2–6), I/O (11–14), I/O (13–15)

CT of $P_4 = 18$

$P_2 = 15$

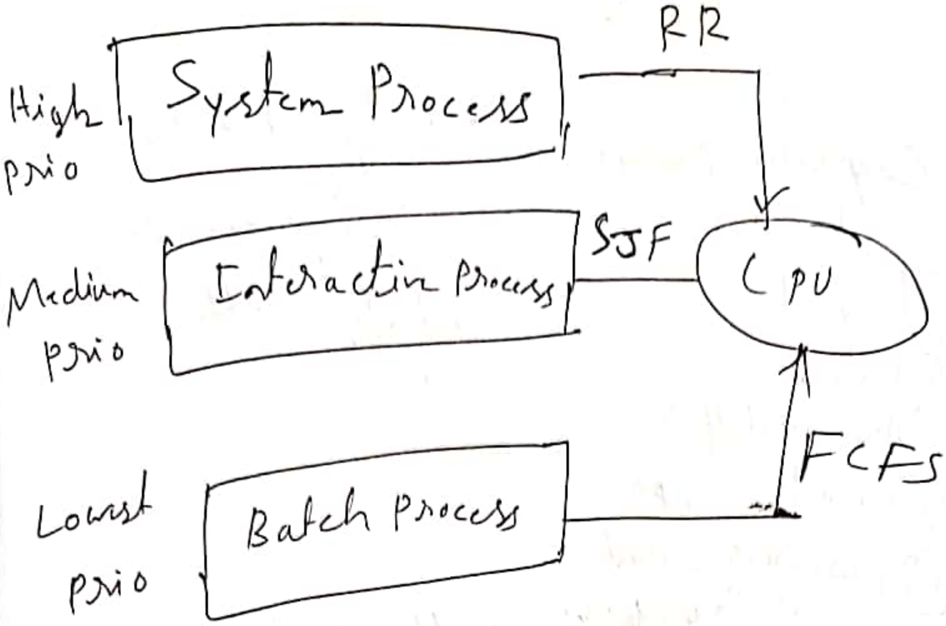
$P_1 = 10$

$P_3 = 9$

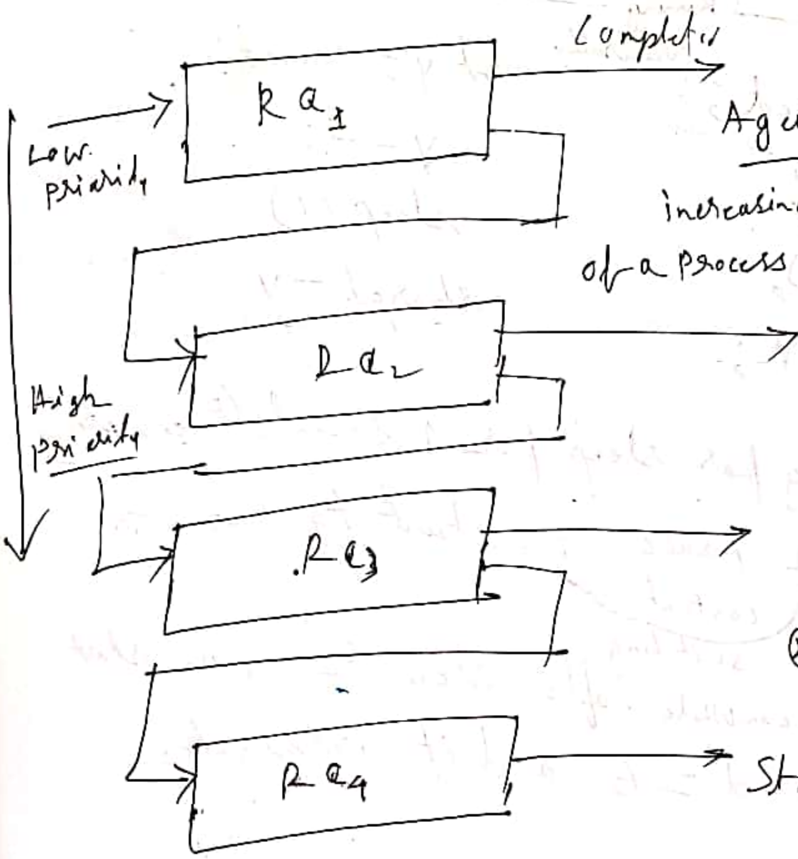
ratio of idleness $= \frac{\cancel{4}\,2}{\cancel{18}\,9} = \frac{2}{9}$

usage of CPU $= \frac{\cancel{14}\,7}{\cancel{18}\,9} = \frac{7}{9}$

Multilevel queue Scheduling



if there is lot of System process then Batch & interactive process may not get chance.
This can cause starvation
So we use Multilevel Feedback queue.



Ageing: Gradually increasing the priority of a process waiting for a long time.

(*) it is used for avoiding Starvation.

Process Synchronization :=>

Process

Cooperative Process → They share
(i) Variable
(ii) Memory / buffer
(iii) Code
(iv) Resources – CPU – Printer – Hardware

One process can effect other Process
Like – 1000+ customer trying to book same ticket

independent Process → One process does not effect other Process

* If cooperative Process is not synchronized then it can cause problem

P1 —— int shared = 5 (common var) —— P2

Running Parallelly

P1	P2
int x = shared;	int y = shared;
x++;	y--;
sleep(1);	sleep(1)
shared = x;	shared = y

→ going for sleep for 1 second (Preemption)

When P1 pause P2 start to run (context switching) ~~and~~

In the meanwhile after sleep P1 again start and shared = 6 and it terminates

Again P2 run and Shared = 9

This situation is called Race condition as if P1 & P2 are doing Race who will give value last. Both value is wrong.

⊗ Race condition is a situation where several process access & manipulate the same data concurrently & outcome of the execution depends on the particular order in which the accesses take place.

To avoid such situation, it must be ensured that only one process can manipulate the data at a given time.

This can be done by <u>Process Synchronization</u>.

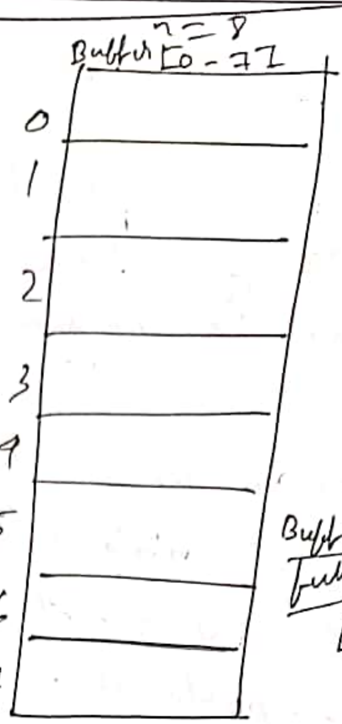
<u>Producer Consumer Process</u> :⇒ Both Process are cooperative Process. They are sharing count variable.

Producer Produce an item / items and put in the buffer & increment count & in (Buffer[in])

Consumer take the item & decrement count and increment out.

int count = 0

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




```
void producer(void)
{
    int itemp;
    while(true)
    {
        produce item(itemp);
        while(count==n);          // Buffer full
        Buffer[in] = itemp;
        in = (in+1) mod n;
        count = count+1;
    }
}
```

(*) (in+1) mod n because after 7 again starting with 0

→ infinite loop

(*) in & out value is initially 0.

(*) count & Buffer is shared by both

CPU runs a instruction of C with its own instruction

(i) count = count+1; → load R_p, m(count);
INCR R_p;
store m(count) R_p;

(ii) count = count-1; — load R_c, m(count);
DECR R_c
Store m(count) R_c

if both Process are not Synchronized then it will make some trouble

Lets think producer produce 3 item then count = 3 4 in = ~~3~~ 4 → refer next empty address.

We want to produce one more item then Buffer [3] = 4th item, then count will be incremented - In time of this after loading Rp increment Rp done but before executing last instruction, this process preempt.

In this time consumer take 1 item then count is = 3 then count will be decrement, still count is decrement is done but it is stored in Register & again preempt before executing last instruction

producer Process run last instruction & count will be 4 wrong both and count = 2

Again consumer execute

Actually count should be 3

Printer spooler Problem :- → There is only one printer in a network but many devices are using it. As we know printer is slow, so every device give document to spooler directory & then it will be printed.

Every process have to follow a instruction set to give a document to spooler

SD — Spooler Directory

1. Load R_1, m[in]
2. Store SD[R_1], "f_1.doc"
3. INCR R_1
4. Store m[in], R_1

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

← Printer

P1 — P2 Parallel Cooperative Process

f_4.doc f_5.doc

Suppose P_1 got CPU first & execute first instruction (Load 3 in R_1) then store f_4.doc in SD[3]

Increment of R_1 happen it become 4 and suddenly it Preempt (Switched)

In this mean time P_2 came & executed 1st instruction & Load 3 in R_2. Then store f_5.doc in SD[3].

P_1	P_2	P_2	P_1
$I_1 I_2 I_3$	$I_1 I_2 I_3$	I_4	I_4

The problem is f_4.doc is vanished due to non Synchronization. (Loss of data)

Critical Section Problem :=>

- Critical Section is part of the program where shared resources are accessed by various Process (like count of producer consumer Problem)
- It is kind a place / portion where common variable or code is ~~sitant~~ kept.

A critical Section Problem is defined like

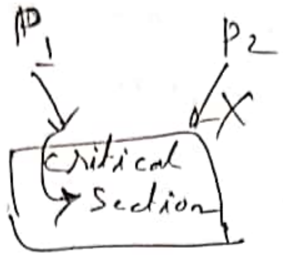
- there are n process like $P_0, P_1, P_2 \cdots P_{n-1}$
- Each process has a section of code called the critical Section in which process changes common variable & files.
- The problem is to ensure that when one process is executing in its critical section then no other process execute its critical section.
- The critical section is preceded by an entry section in which a process seeks permission from other process
- The critical Section is followed by an exit section that says its work has been done
- The remaining code is remainder Section.

Any Synchronization Mechanism should follow this 4 rules (Must follow 2)

(i) Mutual Exclusion } Primary
(ii) Progress
(iii) Bounded wait } optional
(iv) No assumption related to hardware

But we try to all 4

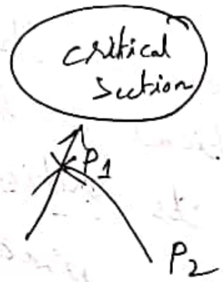
(i) **Mutual Exclusion** :⇒ If any process has granted entry in critical section then no other Process can enter in critical section



Ex- If wife is in house, gf not allowed.

(ii) **Progress** : If no process is in critical section. Some process want to enter its critical section but Somehow other process entry section can not let it happen. The selection of process can not be postpone indefinately

Ex- I will fail & also I will make my frd fail.



P_1 is not Progressing also it does not help to P_2 for progress.

(iii) **Bounded Waiting** :⇒ There is a limit of number of times other process are allowed to enter its critical section after a process made a request to entry section and before that request is granted

like- In atm a customer is using atm multiple times.

(iv) ~~Solution only Work on 32 bits O. Machine~~

Critical Section Solution using Lock :->

do {
acquire lock
Critical Section
release lock
}

Lock: 0 Vacant, 1 full

i_1 1. While(Lock==1); — Entry Code
i_2 2. Lock = 1
i_3 3. Critical Section
i_4 4. Lock=0 — Exit code

- Execute in user Mode
- Multiprocess solution

Case 1 Lock=0 starting

P1	P2
instruction 1 2 3 (Lock=1)	Lock==1 ✓ infinite loop

Case 2

P_1 execute i_1 and it false but before making lock = 1 it preempt and Then P_2 came and execute i_1 + i_2 then enter into critical section.

When P_1 again in running state then make lock=1 and enter into critical section.

So no Mutual Exclusion is guarrenty.

CS Solution using ~~lock~~ Test & Set : =>

```
While (Lock == 1);
Lock = 1;
```

$\}$ — combine & atomic. It can't Preempt using hardware.

```
While (test_and_set (&lock));
    critical section
    lock = false;

boolean lock_and_set (boolean *target)
    { boolean r = *target;
      *target = True;   // When a Process is in critical Section
      return r;
    }
```

- Mutual Exclusion achieved.
- Progress achieved

Strict Alteration Method (Turn Variable) :=>

- Execute in user Mode
- 2 Process Solution
- Mutual Exclusion guaranty
- No Progress.

Process P_0	Process P_1
Non CS	Non CS
Entry - While (turn != 0);	While (turn != 1)
CS	CS
Exit - turn = 1;	turn = 0

- if turn = 0 then P_1 does not go first so it is blocking so no progress achieved
- Bounded waiting ✓ achieved

It runs on alterly

Semaphore :=> It is an integer

variable that, apart from initialization, is accessed only through two atomic operation called wait () & signal ()

It is used to achieve Synchronization

Semaphore

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

Entry Section — P( ) Down( ) Wait( ) acquire

Exit Section - V( ) up( ) Signal( ) release

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

```
up(Semaphore S)
{
  S.value = S.value + 1
  if (S.value <= 0)
  {
     Select from suspended list & wake up
     Wake up( );
  }
}
```

CS P_1 P_2 P_3

Suppose - S = 3 (Semaphore value)

2<0 false so it will be in CS

first - S = 2 , then P_1 came in CS

S = 1 then P_2 " " "

S = 0 " P_3 " " "

S = -1 then P_4 in suspended list

If Semaphore value = n then n process can be in critical Section

(*) If Semaphore Value is -4 then 4 Process is in block stage

$S = -n$ then n value is in block stage

$S = 0$ then no process is in suspended list

Question :⇒ ① If $S = 10$ then howmany process can came in CS

⇒ 10 process can came.

② If $S = 10$ then 6P 4V operation is performed

$P \rightarrow -$ $\quad V \rightarrow +$

⇒ $S = 10 - 6 + 4 = 8$

③ $S = 17$ then 5P, 3V, 1P performed

⇒ $S = 17 - 5 + 3 - 1 = 14$

Binary Semaphore :⇒

```
Down (Semaphore S)
{ if (S.value == 1)
    { S.value = 0; }  // Success full in CS
  else
    { sleep(); // block
    }
}

up (Semaphore S)
  if (suspend list is Empty)
    { S.value = 1;
    }
  else
    {
     wake up ();
    }
}
```

S = 1 initialized (Suspend list Empty)

P1	P2
Down(S)	Down(S)
CS	CS
up(S)	up(S)

Q Each Process P_i (i = 1 to 9)

execute the following code

repeat

p(mutex)

$\boxed{CS}$

V(mutex)

forever

$\boxed{mutex = 1}$

P_{10} execute follow

V(mutex)

$\boxed{CS}$

V(mutex)

forever

What is max number of process can be present in CS

⇒ P means mutex − 1 wait

V " mutex + 1 Signal

Initially Mutex = 1

Suppose P_1 came then Mutex = 0

When P_1 in CS also P_{10} came and Mutex = 1 (executing V(mutex))

Then P_2 in CS and Mutex = 0

Then P_{10} exit Mutex = 1 again

P_3 came & Mutex = 0

P_{10} came & Repeat

$\boxed{\text{Max} = 10 \text{ Process.}}$

Producer Consumer Problem Solution :-

Semaphores -

mutex = 1 } binary

full = 0
empty = n } - counting

Producer

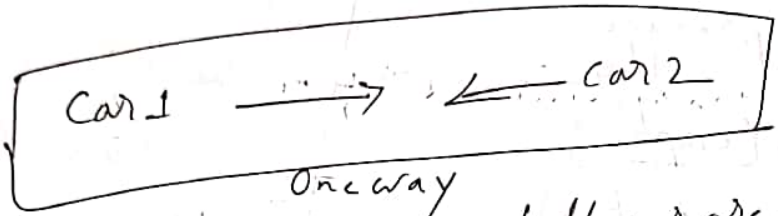
```
do { // produce item
    down(empty);
    down(mutex);
    [ add item to buffer ]
    up(mutex)
    up(full)
} while (true)
```

Consumer

```
do
{
    down(full)
    down(mutex)
    [ remove item ]
    up(mutex)
    up(empty)
```

Deadlock

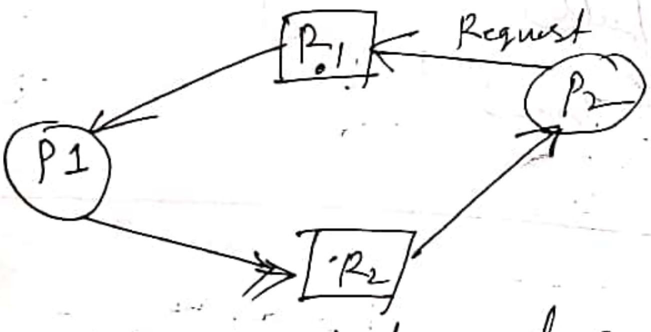
Deadlock is a situation in which two or more process are waiting indefinitely because the resource they have requested for are being held by one another.



only one car can pass both car are not ready to move back. They can never go to destination.

A process utilize its resource like.

request → use → release



Necessary condition for deadlock

(i) Mutual Exclusion — One or more non shareable resources

If any process got any resource then untill it terminated it will not release it

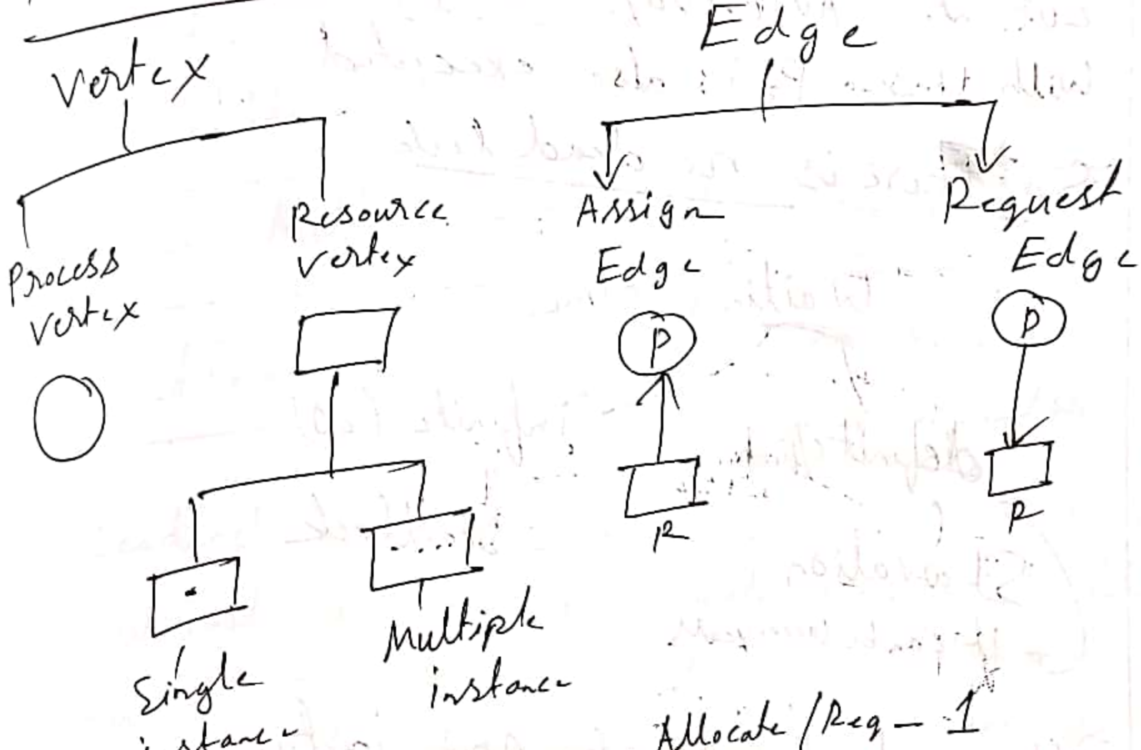
(ii) No Preemption — Resources can not be preempted

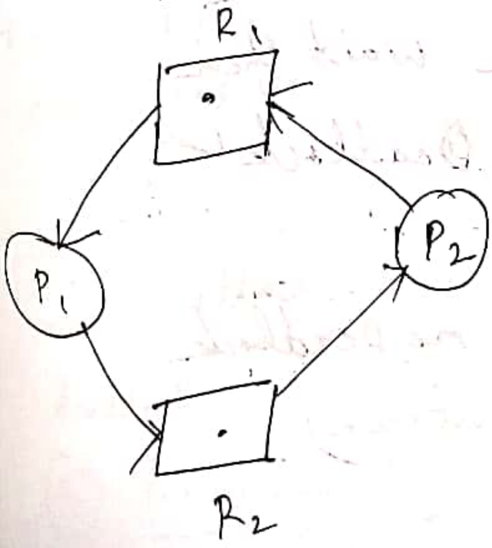
(iii) Hold Wait → A process is holding a resource and waiting for other resource.
It can't release holding resource

(iv) Circular Wait: a set $\{P_0, P_1, P_2, \ldots P_n\}$ exist such that P_0 is wait for resource hold by P_1, P1 is waiting for resource hold by P_2. ......

4 condition should be matched to determine deadlock.

Reasource Allocation Graph (RAG) :→





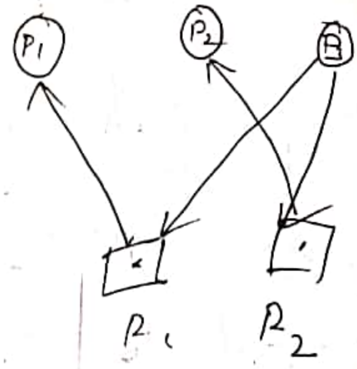
Allocate / Req — 1

	Allocated R_1	Allocated R_2	Request R_1	Request R_2	
P_1	1	0	0	1	X
P_2	0	1	1	0	X

Available (0, 0)

We can't fullfil anyone's need. Deadlock ✓

	Allocate		Req	
	R_1	R_2	R_1	R_2
P_1	1	0	0	0 ✓
P_2	0	1	0	0 ✓
P_3	0	0	1	1 ✓



Available = (0, 0)

P_2 needs nothing 4 executed so P_2 release its Resource

Current Availability $\begin{pmatrix} 0, & 0 \\ + & + \\ 0 & 1 \end{pmatrix} = (0, 1)$

P_1 also executed 4 release its resource.

current Availability (1, 1)

With this P_3 is also executed

So There is no dead lock

Waiting time

definite/finite — infinite (∞)

Starvation — Deadlock

→ It can be 10000... years

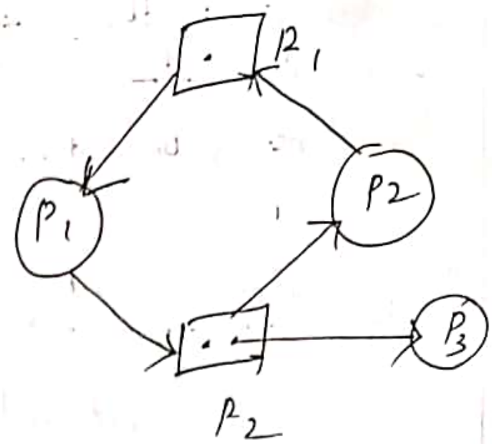
(*) If each Resource has one instance then if there is circular wait then there will always be Deadlock ✓

Vice versa also Same

if no circular Wait → no Deadlock (one instance)

Multi instance RAG :

Process	Allocated R_1	Allocated R_2	Request R_1	Request R_2
P_1	1	0	0	1
P_2	0	1	1	0
P_3	0	1	0	0



Available = (0, 0)

P_3 will execute and release is resource now $A = \{0, 1\}$ with these P_1 executed & release its resource, $A = \{1, 1\}$ with these P_2 executed & release. $A = \{1, 2\}$

no Deadlock X

Various Method for handling Deadlock :

① **Deadlock ignorance** : (ostrich Method) bird put his head on sand while sand storm

Deadlock are infrequent, may be once in a year. So we just ignore it. If we write code for it then it will effect performance. Speed is most essential.

Ex- Our Laptop / PC sometimes hang (deadlock) we restart our computer

In windows, Linux etc we use this Method

speed > correctness (share)

② **Deadlock Prevention** : => Before the deadlock we try to prevent it

ⓐ no Mutual Exclusion : Sharable resources like read only files

(printer cannot be shared)

(b) preemption: If process are preempted by time quantum then Deadlock can be Prevented

(c) no Hold & wait:

Request and get all the resources in beginning.
or
Release current resources before Requesting other

Disadv - low resourse utilization

(d) no circular wait: linear order

Arrange the resource type as R_1 R_2 R_3 R_4 .. R_n

- Request Resource in increasing order

if a process taken R_1 then it can request R_2, R_3, R_4
if it take R_3 then it can't req R_1 & R_2

Dis - low Resource utilization.

(3) Deadlock Avoidance: ⇒

Banker's Algo is used for Deadlock Avoidance.

Whenever we give resource to the process we check safe or not.

- Safe state is one in which system can allocate resources to each process up to maximum and still Avoid deadlock.

A system is in a safe state if there is a safe sequence.

4 Deadlock Detection & Recovery:

complex way

first we want to Detect then Recovery it

For recovering — (i) Process termination!

abort the processes which incur minimal cost.

- abort all deadlocked Process
- abort one process at a time untill deadlock is broken

(ii) Resource Preemption

- selecting a victim

(*) Deadlock ignorance is most used in real life.

The Banker's Algo ⇒

A new process must declare the max number of instance of each resource type that it may need.

This number should not exceed the total number of instance of that resource type in the system.

Allocated → current allocated to Process

Max need → To execute the process, maximum need

Remaining need → Max − Allocation

Available − currently Available in System.

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

Process	Allocation A	B	C	Available A	B	C	Max need A	B	C	Remaining need A	B	C
P_1	0	1	0	3	3	2	7	5	3	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
	7	2	5									

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

Available $(10-7, 5-2, 7-5)$

$A = (3, 3, 2)$

now first we will check with the available whether we can fulfil any process need

note – we should fullfil all the resource at a time of a process.

We can't fullfil P_1, but can for P_2

P_2 executed and Release its Allocated Resource.

$A = (3+2, 3, 2) = (5, 3, 2)$

now we can execute P_5 and it will release its allocation

$A = (5, 3, 2+2) = (5, 3, 4)$

P_4 executed $A = (5+2, 3+1, 4+1)$
$= (7, 4, 5)$

P_1 executed $A = (7, 4+1, 5) = (7, 5, 5)$

P_3 executed $= A = (7+3, 5, 5+2)$

$= (10, 5, 7)$

Total

---

(1) Consider a system with 3 Processes that share 4 instances of same resource type. each process can request a max of "K" instance. The largest value of k that always avoid deadlock is ______

$R = 4$

	P_1	P_2	P_3	
if $k = 1$	1	1	1	no Deadlock

	P_1	P_2	P_3	
$k = 2$	1	1	1	no Deadlock
	1			

execute & released

	P_1	P_2	P_3	
$k = 3$	1	1	1	no deadlock
	1			
	1			

execute, execut., execut.

but we have to consider each case

P_1	P_2	P_3
1	1	1
1		

So Kmax is 2

Deadlock

Shortcut –

R – Resources   n – Process

$P_1 \quad P_2 \quad P_3 \quad \dots \quad P_n$

$d_1 \quad d_2 \quad d_3 \quad \dots \quad d_n$

$(d_1 - 1) \quad (d_2 - 1) \quad (d_3 - 1) \quad (d_n - 1)$

$R \le \left(\sum_{i=1}^{n} d_i\right) - n$ – deadlock

if $R > \left(\sum_{i=1}^{n} d_i\right) - n$ – no deadlock

$\boxed{\text{Total Resource} + \text{total process} > \text{total demand}}$

In previous question

$4 + 3 > 3d$

or $3n < 7$

a $n < 2.3\dots$

$d_{max} = 2$

(4) each

2.) A system having 3 process that require 2 unit of process R. The minimum no of unit of R such that there will no deadlock

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

⇒ Total Resourse + total Process > total demand

or, $n + 3 > 6$

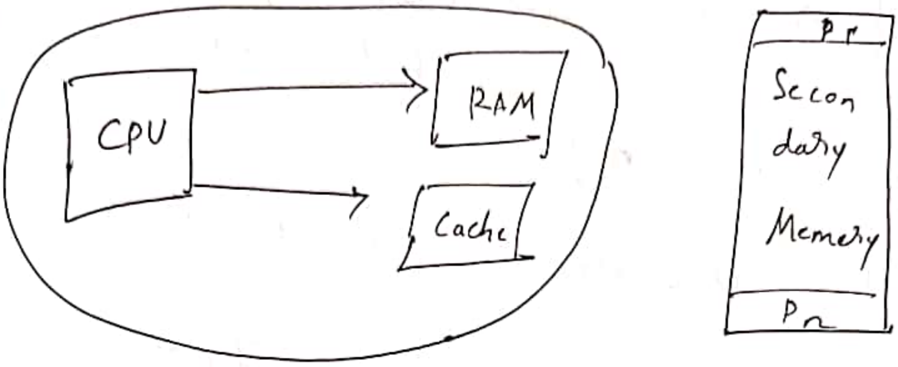
or $n > 3$

→ (d) 4 ✓

Memory Management

* Method of managing primary memory is one of the responsibility of OS.

Goal → efficient utilization of memory



all the programs are available in Secondary memory. In order to execute program first we have to transfer the program to RAM so that CPU can interact with it & execute the program.

* Degree of multiprogramming describes the maximum number of process that a single processor system can accomodate efficiently

• The primary factor is the amount of ~~the~~ memory available to be allocated to executing a program.

• OS should allocate resources to executing process in an fair & orderly fashion.

* The number of process currently in memory is also known as degree of multiprogramming.

Numerical :

If there is n independent events with success probablity $P_1, P_2 \ldots P_n$ then probablity of all events occuring successfully is $P_1 * P_2 \times \ldots P_n$

(*) If n processes has K probablity of I/O request then CPU will be ~~idle for~~ utilized for $(1-K^n)$

K^n time they are in I/O request

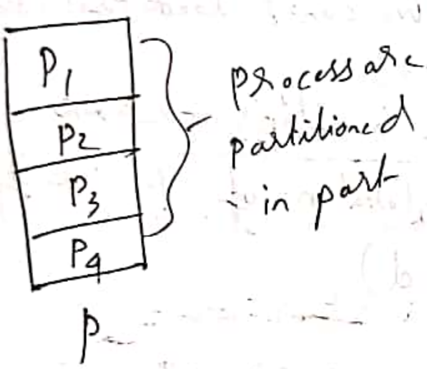
- If 1 process I/O operation is 70% probablity
  CPU utilization $= 1 - 70\% = 30\%$
- If 2 process I/O operation probablity is 70%.
  then CPU utilization $= 1 - \frac{70}{100} \times \frac{70}{100} = 51\%$.
- If 4 process I/O operation probablity 70%
  CPU utilization $= 1 - \frac{70 \times 70 \times 70 \times 70}{100 \times 100 \times 100 \times 100}$
  $\simeq 76\%$

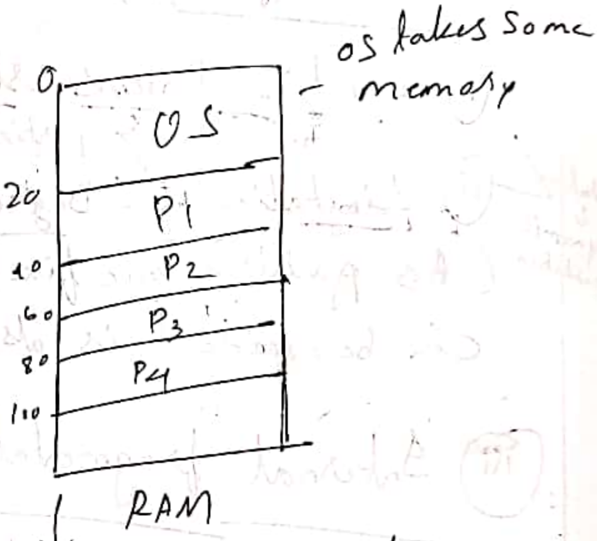
(*) RAM memory management is most important

Memory Management Technique

- Contiguous
  - Fixed partition (Static)
  - Variable partition (Dynamic)
- non contiguous
  - Paging
  - Multilevel Paging
  - Inverted paging
  - Segmentation
  - Segmented Paging

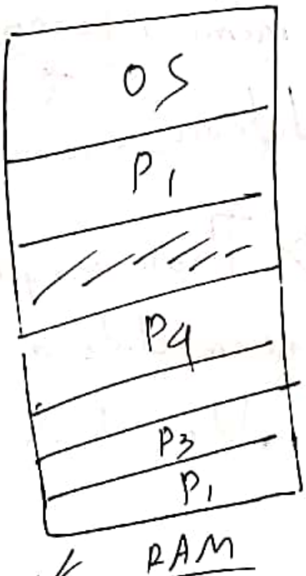
Input output operation: In time of execution some process need to access secondary memory for file need.
xyz.txt
Read (file names size)





Contiguous Allocation
Sequentially One by one allocated

no sequence/Random allocated



non contiguous

Fixed Partition (Static Partition) :=> In 1960s Mainframe

- no of partition are fixed.
- Size of each partition may or may not be same
- Spanning is not allowed (allocates max 1 block to each Process)

(*) for Suppose Process is of 8MB but max partition size is 7 MB, We cannot load the process in 2 block.

Advantages :=> It is easy to implement

Disadvantages :=>

(i) Limit in Process size (We can't load Processes whose size > partion size)

Solution ↓ dynamic partition

(ii) Limitation of Degree of multiprogramming (As partition are fixed, So total no of process that can be loaded is also fixed)

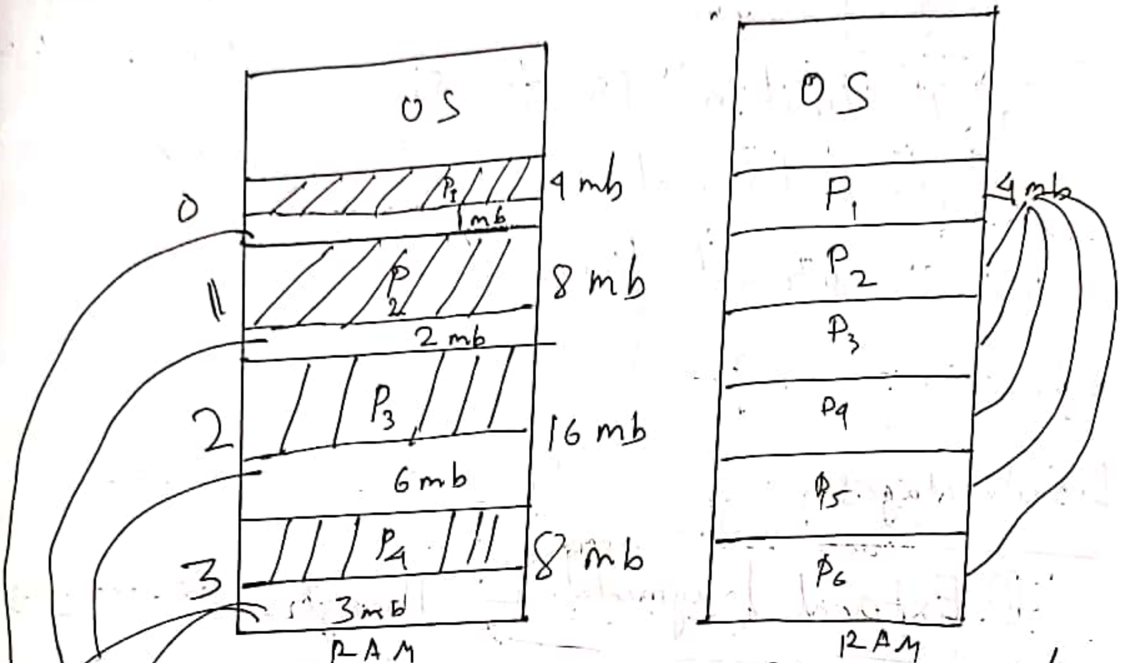
(iii) Internal fragmentation: When a process is allocated memory block but Process size < memory block, then the remaining memory become unused. This is called internal fragmentation.

(iv) External fragmentation :=> There may be enough memory left over to accomodate a new process but it is Scattered then it is called external fragmentation.

(*) Solution of internal fragmentation is dynamic Partition

(*) Solution of External fragmentation is defragmentation or compaction (very costly) → dynamic partitioning is used for memory allocation by combining all free memory into a single large block.

Also Paging is Solution



total 4 block
Size of Partition not Same

Size of each Partition same

→ internal fragmentation

→ total 12 mb space is empty this could accomodate up to 12 mb sized process

→ external fragmentation

Before the process come RAM is already partitioned

Variable Size Partition (Dynamic)

Here • no of partition is not fixed
• We allocate memory in the run time (when the process came it gets its desirable memory if available)

Advantages:

(i) No limit in process size
(ii) No limitation of degree of programming
(iii) No internal fragmentation

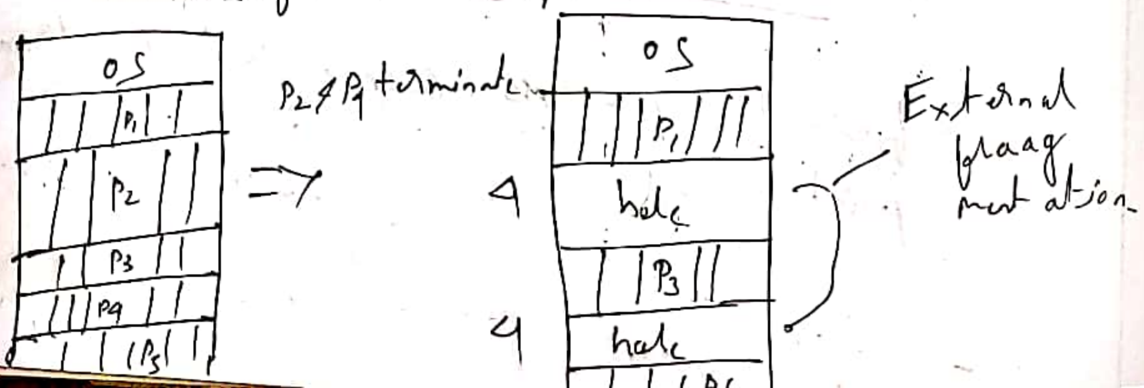
Disadvantages:

(i) External fragmentation: Though processes are allocating memory dynamically. But when a process terminate it freed the memory (Hole). In this case suppose there is 8mb process and 2 hole of 4mb is available. As spanning (divided into parts) is not allowed so we can not load the process.

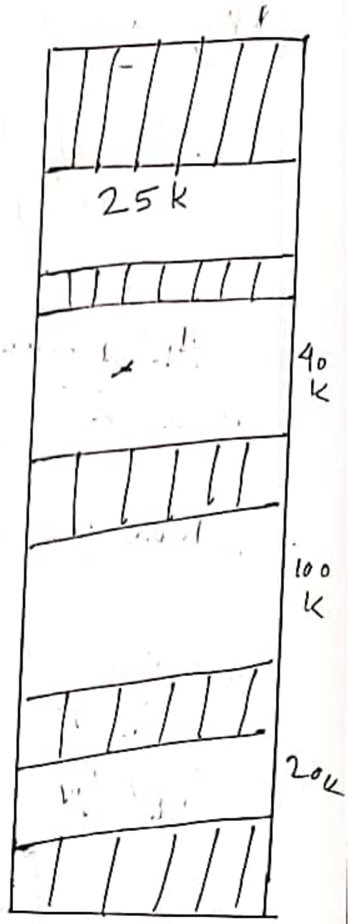
Solution - compaction, Paging

(ii) Allocation or Deallocation Complex: We have to keep track of hole.

Solution of it is bitmaps, linked list.

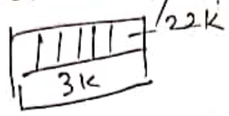


Strategis For allocation :=>



(i) **First fit** : Allocate the first hole that is big enough to acomodate the process.

Suppose process is of 22K then it will Search from the top and founds the first hole of 25K which is enough to accomodate So Process is allocated memory. 3K Will be left as hole



updated version

- **next fit** : Same like first fit but search will be started from last allocated hole.

Advantage : • It is simple & fast

Disadvantage : The remaining unused memory area left after allocation become waste if it is too smaller.

(ii) **Best fit** :=> First search entire list of holes and Allocate the smallest hole to to accomodate the process.

If process is of 12K then according to best fit it will get Partition of 20K. (picture)

Advantage : Memory utilization is much better
— Internal fragmentation will be less

Dis : Slower and may even leave tiny holes.

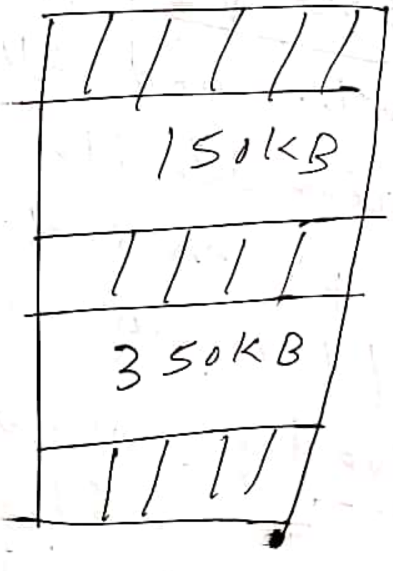
(iii) Worst fit :=> First Search largest hole and if the partion is big enough to accomodate ~~d~~ the process then store it.

Advantage – Reduce the rate of production of small gaps

Dis : - Slow
- Process requiring larger memory arrives at a later stage it will not allocated.

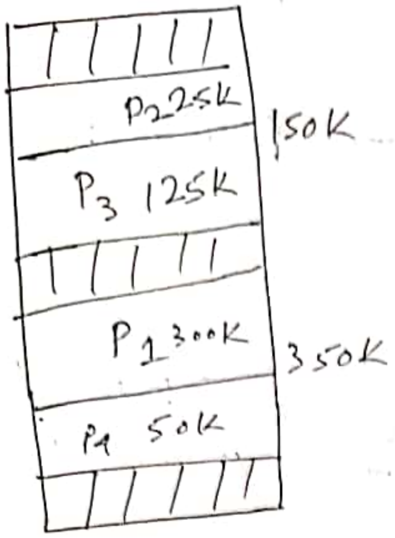
Buddy's System : Size of free block are in form of 2^n (2, 4, 8, 16 ....) up to size of memory. if k size block is requested then nearby big block will be given but if it is not available then the higher block will be spilit into parts. (like buddy)

(1) Request from Processes are 300 K, 25 K, 125 K, 50 K respectively The above request could be satisfied

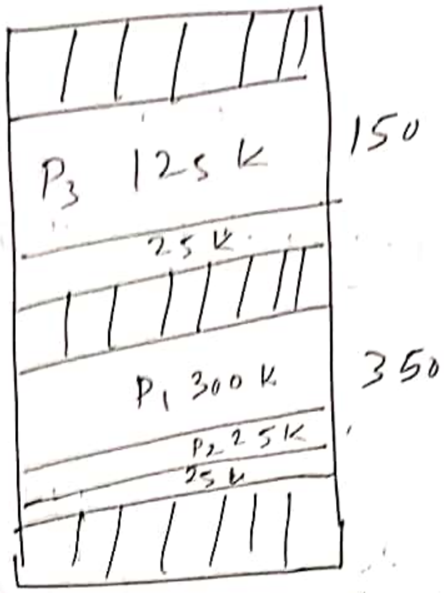


(a) Best fit not first fit
(b) First fit not best fit
(c) Both
(d) none

If we use first fit then



If we use best fit



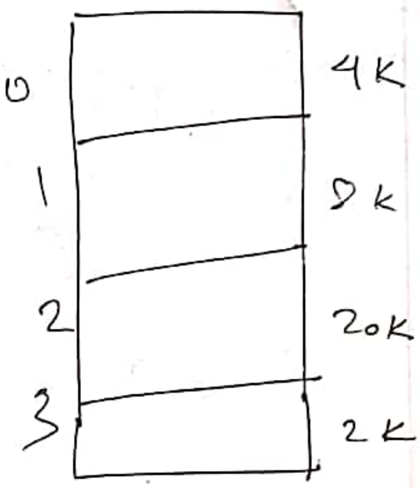
P4 can not be allocated as continous memory allocation does not allow spanning

(2) Request no - J_1 J_2 J_3 J_4 J_5 J_6 J_7 J_8

Req size - 2K 14K 3K 6K 6K 10K 7K 20K

usage time - 4 10 2 8 4 1 8 6

Follow best fit
Consider fixed partition
Calculate the time at which J_7 will be completed —

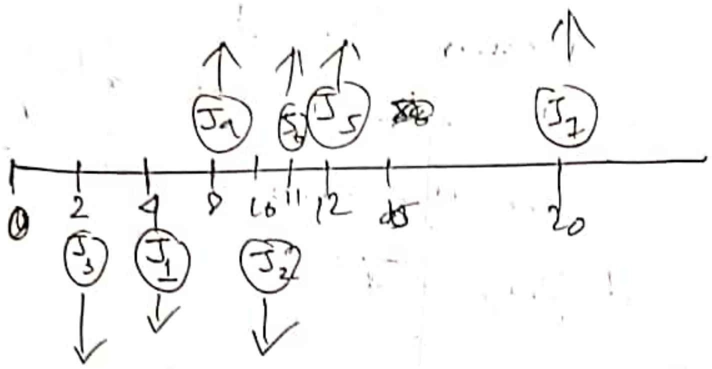


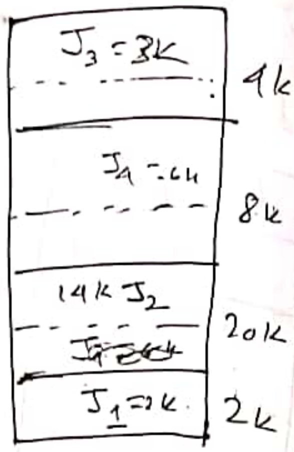
(a) 17 (c) 20

(b) 19 (d) 37

⇒ We will assume every process come at 0 second.

Only one process will allocate Space Sequentially





J_5 will get 6K When J_2 or J_4 will be released. J_4 will release first so J_5 will get the block of J_4

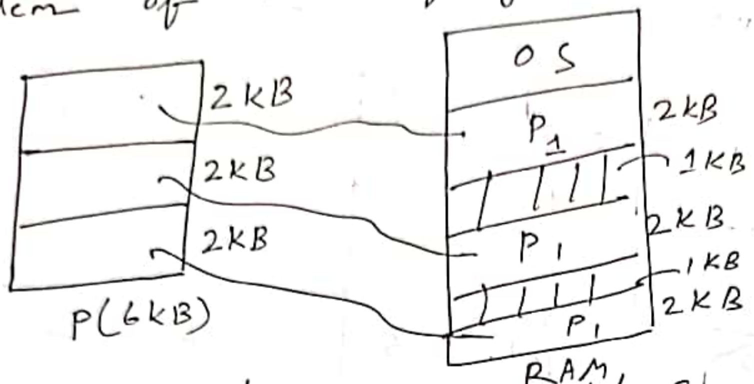
J_6 will get the block when J_2 terminate J_2 terminate and J_6 allocated till 17 sn 10S.

J_7 got the memory after J_6 release (17+8 = 19)

19 ✓ ⊛ 11 baje slot khali hua J_7 ghus gya why wait

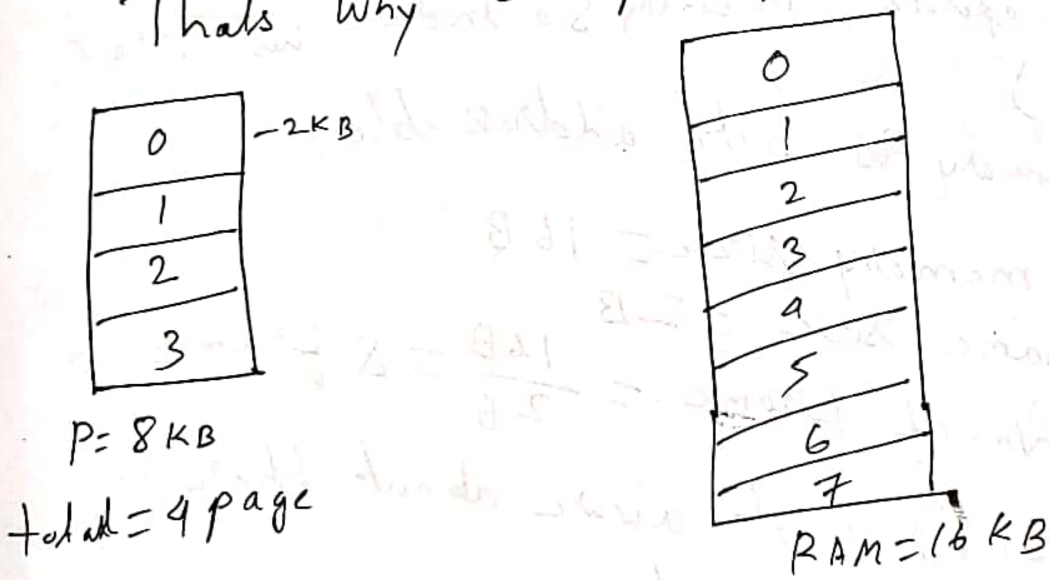
Non Contiguous Memory Allocation :⇒

In noncontiguous Memory allocation we can divide the process into smaller parts and put in different block of memory. This eliminate the problem of external fragmentation



The process first knock on the ram and know the details of available hole and sizes. According to that before entering ram it is spilited. The problem is the creation of hole is dynamic & done in run time. So everytime a process came it had to check the available hole. It is very costly & time consuming.

Thats Why Concept of paging Come.

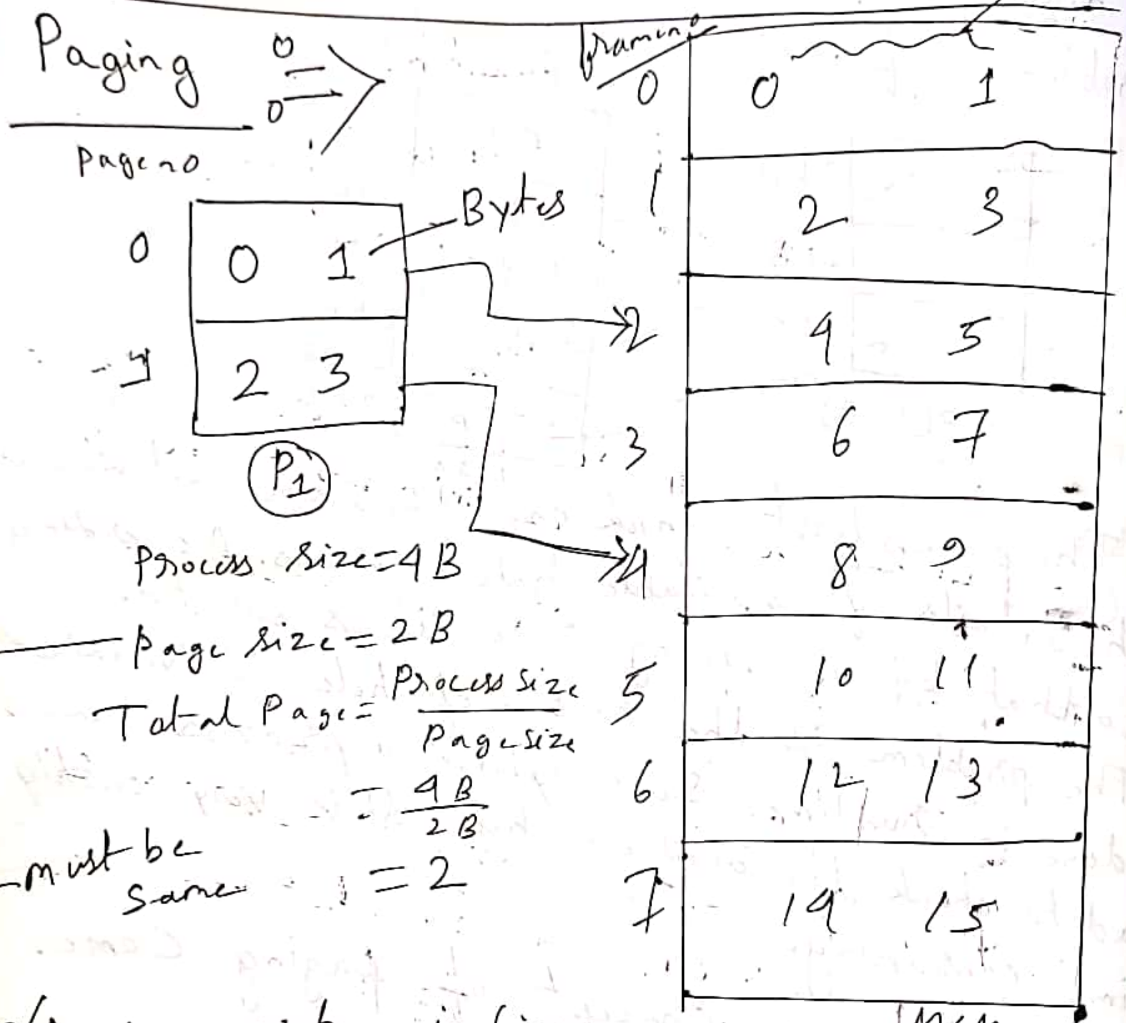


1 Page size = 1 frame size

Paging is done on Secondary memory itself so that it can fit on frame

* Non contiguous does not mean the process will be allocated scatteredly
but if there is no empty space we can allocate scattered memory
but if there is empty space we can allocate contiguously.

Paging



Process size = 4 B

page size = 2 B

$$\text{Total Page} = \frac{\text{Process size}}{\text{Page size}} = \frac{4B}{2B} = 2$$

must be same

* (As we operate in binary so index will start from 0.)

* Memory is byte addressable

Main memory size = 16 B

frame size = 2B

$$\text{No of frame} = \frac{16B}{2B} = 8 \text{ frames}$$

* CPU does not aware about there is some paging used.

CPU just say I need byte no 3

Paging - We just dividing process into pages & put them in main memory frame.

We assume P_0 is stored on f_2

P_1 is stored on f_4

no - 0 byte is on 4
no - 1 byte is on 5
no 2 byte is on 8
no 9 byte is on 9

} To remember this we need a mapping. This is done by memory management unit (mmu)

mmu uses page table

- Each Process has its own Page table because each Process pageno will start from 0

Page no	
0	f_2
1	f_4

→ This is Page table

First we have to know the byte which CPU demanding in which Page so that we can get frame no from page table

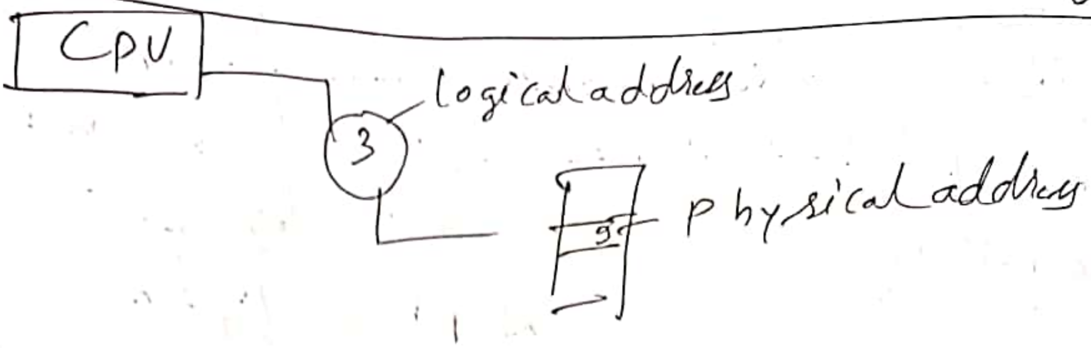
⊛ CPU always works with logical address which consist of Page no & Page offset

1	1
1	1

pageno — 1, 1 — 2nd byte, size

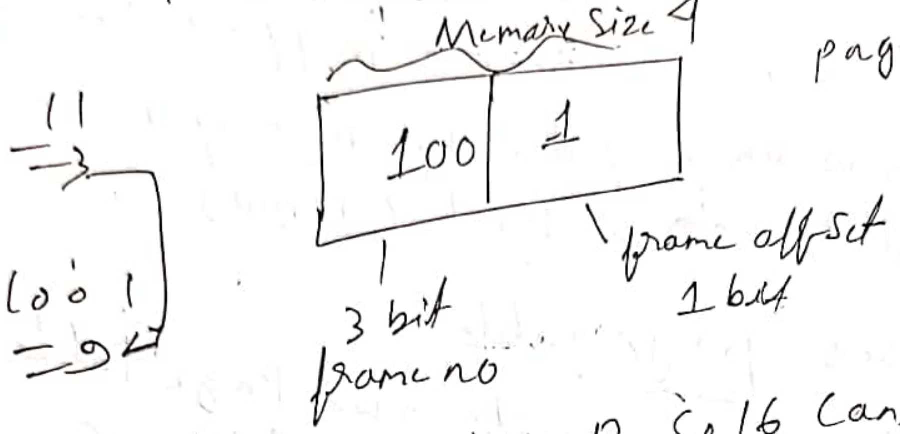
as total Page no is 2 so (0,1) can be represented by only 1 bit

as total Page size is 2 so (0,1) can be represented by only 1 bit



Physical address is Mainmemory

page offset = frame offset



As memory is of 16 B So 16 can be represented in 4 bit

frame offset is same as page offset

Remainings bits is frame no

Page no 1 points to frame no 4 according to page table

4 is represented as 100

offset is same bit sequence of frame offset as previous.

Logical Address space — ~~hardware~~ disk

Physical address - RAM

Q) Given Logical Address Space = 4 GB
Physical Address Space = 64 MB, Page size = 4 KB
No of pages = ? No of frame = ?, No of entries in page table = ?, size of page table = ?

⇒ Logical ~~Physical~~ Address Space = 4 GB
$= 2^2 \times 2^{30} = 2^{32}$ B – need 32 bits

page no	page offset
20	12

$\underbrace{\qquad\qquad}_{\text{32 bits}}$

Page size = 4 KB
$= 2^2 \times 2^{10}$ B
$= 2^{12}$ B
need 12 bit

Same

	frame offset
14	12

~~Logical~~ Physical Address Space = 64 MB
$\underbrace{\qquad\qquad}_{\text{26 bits total}}$

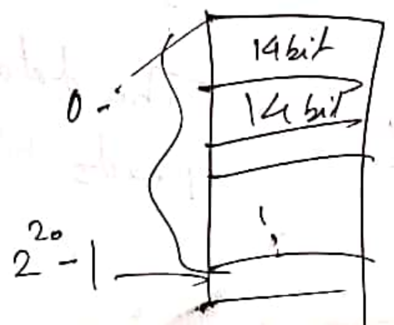
$= 2^6 \times 2^{20}$
$= 2^{26}$ B

No of Pages = 2^{20}
No of frames = 2^{14}

if bits 3
then 2^n combination possible ($2^3 = 8$)

No of entries in Page table = no of page
$= 2^{20}$

Size of Page table = $2^{20} \times 14$ bits



Q.) Consider a system which has LA = 7 bits, PA = 6 bits, Page size = 8 words then calculate no of pages and no of frames?

$\Rightarrow$

LA: $\underbrace{\boxed{4 \mid 3}}_{}$ — 7 bits

page no	page offset
4	3

PA: 6 bits

frame no	frame offset
3	3

Page size = 8 words / Byte
We can Represent in 3 bits

No of pages = 2^4

No of frames = 2^3

No of entries in page table = 2^4 (same as no of pages)

Page table Entry :- $\Rightarrow$

Frame no	valid(1) / invalid(0)	Protection (RWX)	Reference (0/1)	Caching	Dirty

Frame no — mandatory field; Protection, Reference, Caching, Dirty — optional fields; Caching — Enable/disable

Valid/invalid $\rightarrow$ It tells wheather the page is available on main memory or not.
1 - Valid 0 - Invalid.

Protection $\rightarrow$ The data protection of the Page
Read, write execute

<u>Reference</u> → Wheather the page has been in memory (main) previously or not.

1 – previously came

0 – new

<u>caching</u> → We can enable/ disable caching. Whenever cpu thinks a data is being used frequently then it stores it in nearby cache memory so that it can access it in minimum time.

if Data should be fresh (account balance) we should disable cache

if data is static then we should enable cache

<u>Dirty</u> → If a page is modified as it has a write permission

1 – modified

0 – no modified

Multilevel Paging :- ⇒

We first map the logical address and ~~logical~~ Physical address with the help of Page table. And we put page table in a frame. So that when CPU give logical address we can convert it to physical address with this.

But some times when Page table size > frame size. We can't place page table in one frame. So we use multilevel paging.

Ex – Suppose physical address space = 256 MB

Logical Address space = 4 GB

Frame size = 4 KB $= 2^2 \times 2^{10} = 2^{12}$

Page table Entry = 2B

$256 MB = 2^8 \times 2^{20} = 2^{28}$

$4 GB = 2^2 \times 2^{30} = 2^{32}$

20	12

page offset

LAS

16	12

frame offset

PAS

There is 2^{20} Row and Entry of page table = 16 bit = 2Byte

So Page table size $= 2^{20} \times 2$

$= 2MB$

2 MB can't fit in 4KB

(*) If size of page table is larger than the size of frame. Then the page table is divided into several parts and stared on main memory. Thus concept of outer page table comes.

This outer page table would contain the address of frame in which the pages of inner page table is.

Inverted Paging: => Each process has a page table and stared in harddisk

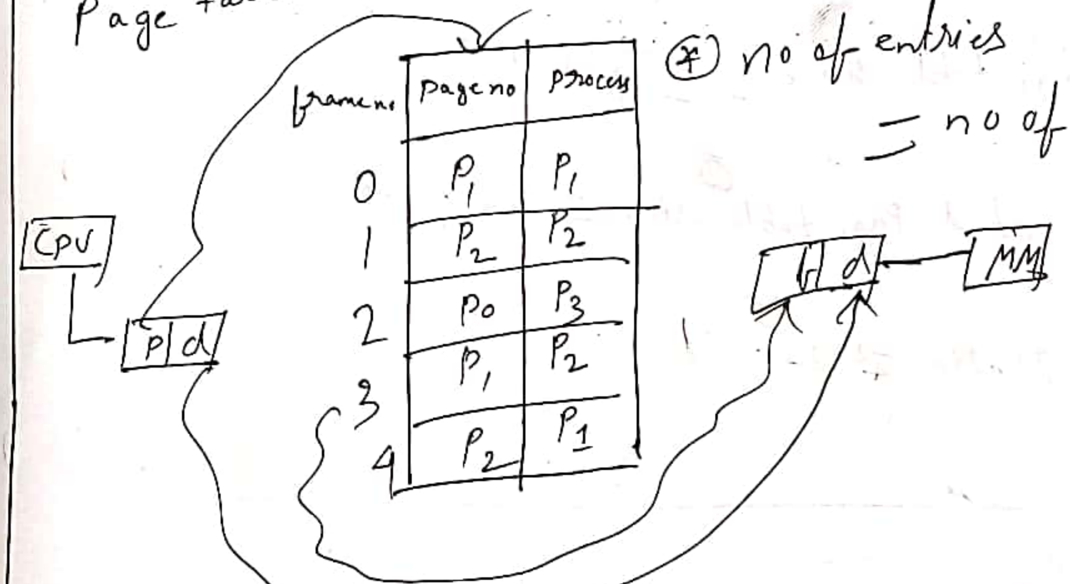
In normal paging when any page of process came to main memory, we have to take pagetable of the process also.

If there is 10 pages of 10 different process available on main memory then 10 page table also have to be in main memory. It will consume a lot of space.

In order to eliminate this we use inverted paging.

In inverted paging we use a single global page table maintained by OS

(*) no of entries = no of frame no



(*) There is multiple Page no with same names, we can differentiate with the help of subsequent process.

CPU says I need P_1 (Page no.) of P_2 (Process) so we have to do linear search on Page table (global). It is very time consuming (Disadvantage)

Q Consider a virtual address space of 32 bits and Page size of 4 KB. System is having RAM of 128 KB. Then what will ratio of page table & inverted page table size if each entry of both is 4 B ?

(a) $2^{15} : 1$ (b) $2^{20} : 1$ (c) $2^{22} : 1$ (d) $2^{10} : 1$

⇒ Virtual address = logical address

Page no	Page offset
20	12

$\underbrace{\qquad}_{32}$ logical address space

$4 KB = 2^{12} B$

$128 KB = 2^{17} B$

frame no	frame offset
5	12

$\underbrace{\qquad}_{17}$

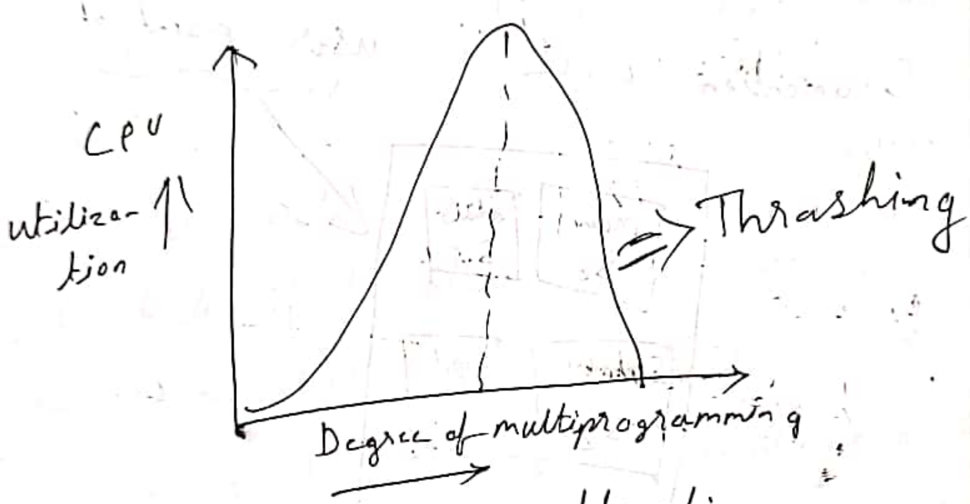
Page table size $= 2^{20} \times 4$ B

Inverted Page table size $= 2^5 \times 4$ B

∴ ratio $= 2^{15} : 1$

Thrashing :⇒ In order to utilize CPU more we increase the degree of multi programming (load more process pages in main memory). After increasing for certain time CPU utilization again gradually become lower as it faced a lot of page fault.

Page fault occurs when CPU requested a page but the page of the process is not in the main memory so page fault routine started to load the page & it takes a lot of CPU time. As a result throughput & CPU utilization become lower



To avoid Thrashing / remove thrashing

- main memory size increase
- Long term scheduler speed low.

When CPU utilization deminishes due to high paging activity is called thrashing.

⊛ Wants more money so work day & night. Health degraded —

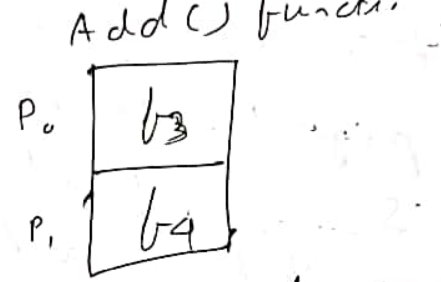
Segmentation :⇒

It is a method in which a process is divided int parts or segment & then we put them into main memory.

Its almost same paging but segments size is not same.

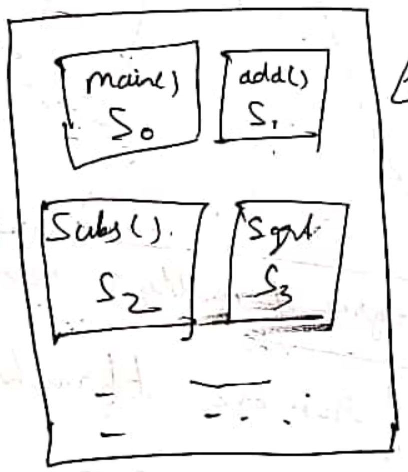
In paging without knowing its content it just divided into equal parts.

The problem is

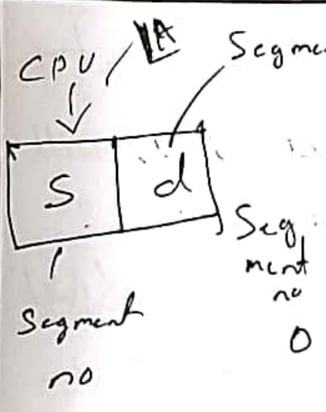


CPU executing b_3 but b_3 does not contain full add() function, Sometimes b_4 may not available in main memory.

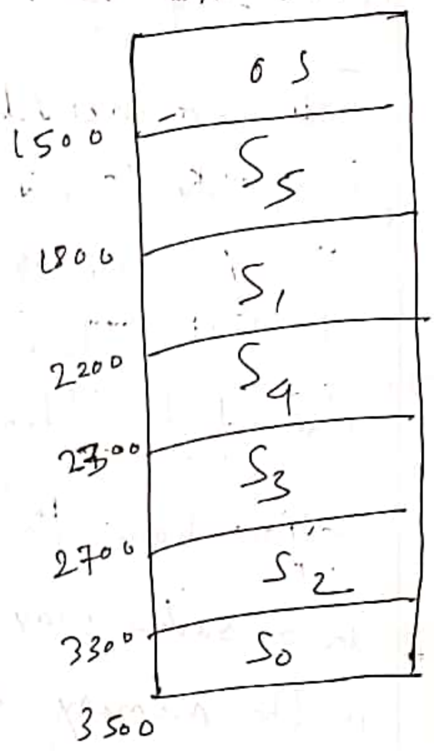
Segmention work on user point of view like



- Segmention size is different from one another
- Memory Management unit (MMU) use segment table for tracking



Segment no	BA	Size
0	3300	200
1	1800	400
2	2700	600
3	2300	400
4	2200	100
5	1500	300



BA – Base address

LA – Logical address

s tells segment no & d tells the size of which segment that CPU need

Suppose s is pointing $S1$ and d is 300

We have to check if d value $\leq$ size of segment 1

if it is true then 1800–2100 byte will go to CPU.

But if d > size of segment then it will not get access & go to trap.

Overlay :=> Whenever a process is running it will not use the complete program; Instead it will use only some part of it.

Overlay says whatever part you required just load it. Then after completation you have to unload it. Then Load the other part.

You have to divide the program into modules in a such way that not all modules need to be in the memory at the same time.

(*) It is user responsibility to take care of the partition.

Q] Consider a two pass assembler of pass 1 : 80 KB, pass2 : 90 KB, Symbol table - 30 KB, Common routine - 20 KB.
At a time only 1 pass is in use.
What is the minimum partition size required if overlay driver is of 10 KB?

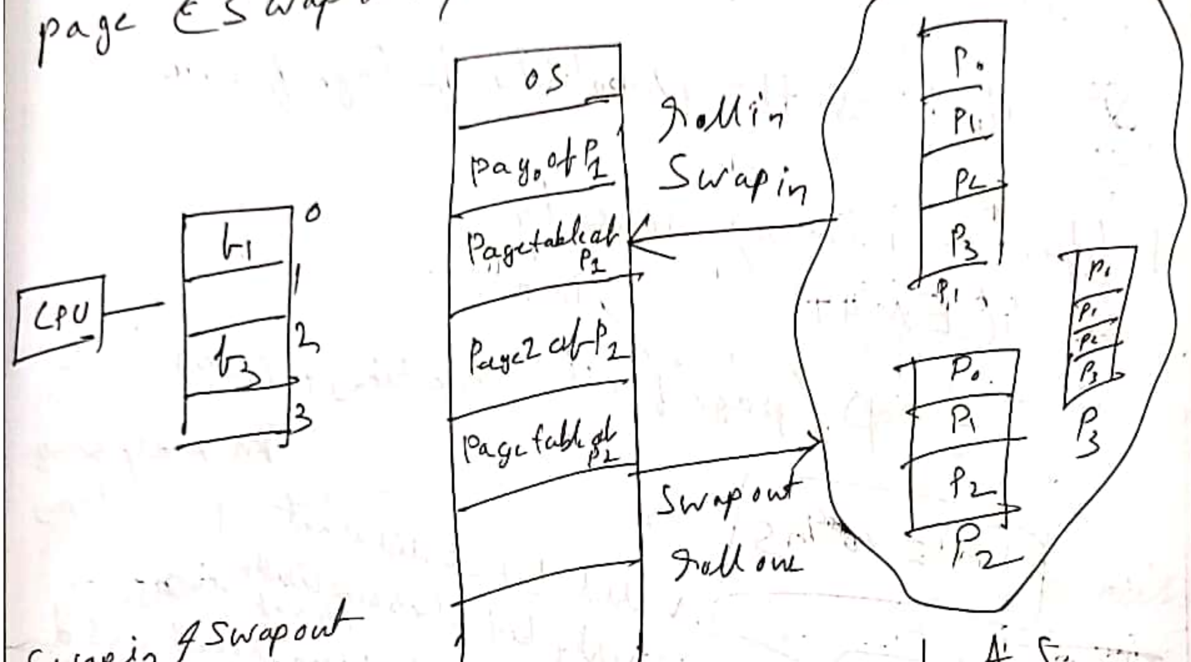
=>

80	90
30	30
20	20
10	10
140	150

We can fit 140 in 150.

Virtual Memory: ⇒ It provide a illusion to the programmer that a program larger than size of main memory can also execute. It is possible with swap in/roll in & swap out/roll out.

What we do is instead of loading whole process we just load some page of it & its page table (swap in), After some time we unload the page (swap out/roll out).



Swap in & Swap out is called page replacement

M/M

LAS (Harddisk

Page fault occurs when CPU demand for a page but the page is not in main memory. (valid/invalid bit is 0)

CPU handle the page fault with

(i) if valid/invalid bit of frame is 0 then trap generated

(ii) Control transfer user to OS (context switching)

(iii) Then OS check if the user is authentic to request the specific page.

(iv) OS takes the page from harddisk and load in main memory

(v) OS put the frame no in page table.

(vi) OS then give control back to user again.

(*) if P is the probablity of Page fault

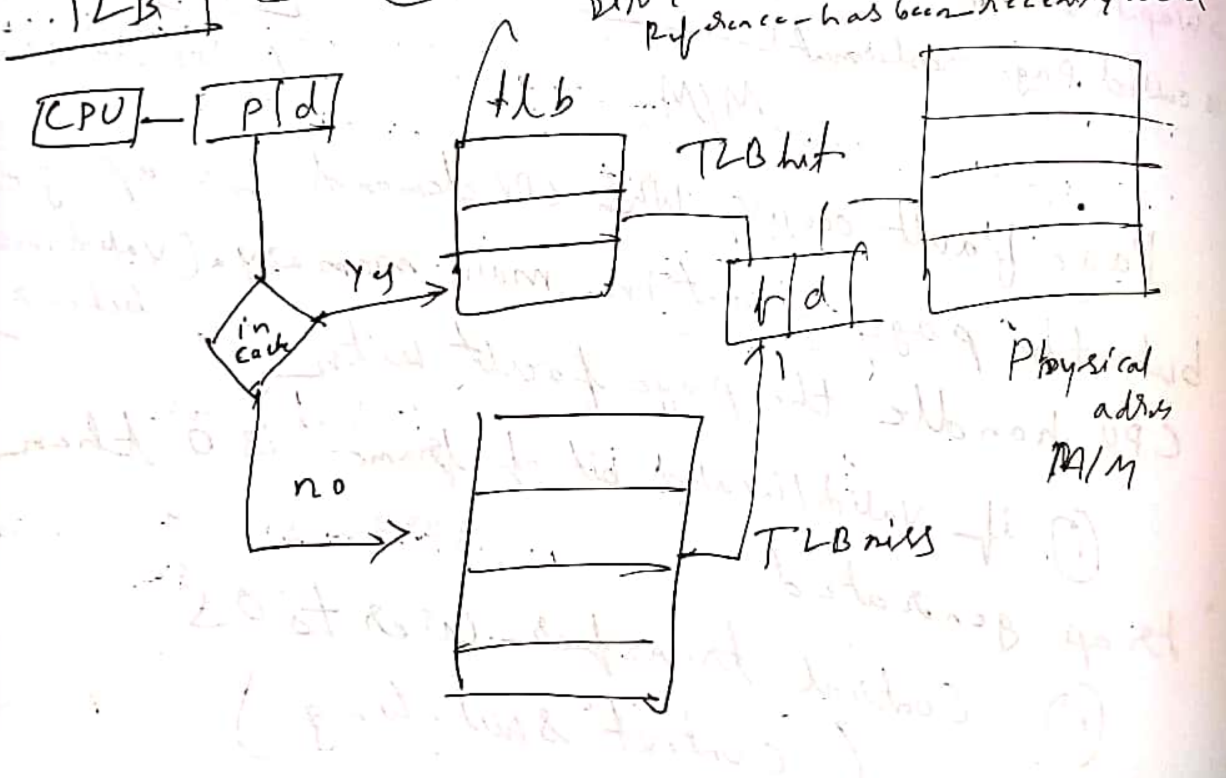
Effective memory access time (EMAT)

$= P.(\text{page fault Service time}) + (1-P)(\text{main memory access time})$ in ms

$\boxed{1\,ms = 10^6\,ns}$

Valid bit - Valid or not
Dirty bit - has been written or not ns
Reference - has been recently used

TLB



Translation lookaside Buffer :=>

each Process has a page table. Pages as well as Page table of a process has to be kept in main memory. So in order to access any page first we have to get the page table from main memory (1) then again to get the actual page we have to go to the main memory (2). Overall access time is quiet high.

Some people though to use register to place the page table but it is very low in size.

To overcome this a high speed cache is set up for page table entries called TLB. It is used to keep track of recent use page table entries.

After getting a virtual (logical) address the processor examine the TLB if it is present (TLB hit) then frame no retrieved and physical address generated.

if page table entry is not found (TLB miss) TLB check for main memory and go to the frame no to take the page from main memory.

if no page fault ocur

EMAT (Effective memory access time)

$$= h*(t+m) + (1-h)*(t+2m)$$

h – hit ratio

m – memory access time

1–h – miss

t → tlb access time

Valid bit 1 – Page available in ~~logical~~ physical address space

0 – not available

Q1 A paging scheme using TLB. TLB access time 10ns, main memory access time 50ns. What is EMAT if TLB hit ratio is 90% and there is no page fault?

$\Rightarrow$ $h = 90\%$ $t = 10ns$ $m = 50ns$

$\therefore Emat = h(t+m) + (1-h)(t+2m)$

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

$= 54 + 11 = 65$

Page Replacement Algo. $\Rightarrow$

Page replacement is the process of replacing one page by another in the memory when there is no free frame.

- Save the content of memory on disk
- Load new page
- update page table.

(i) FIFO :$\Rightarrow$

no need to note time we have to just eliminate the page of the frame which was loaded first. There is 3 frame then f_1 first eliminated then f_2 then f_3 and so on

(*) Shows belady's anomaly

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

demand of pages

f_3	[illegible]		1	1	1	~~1~~	0	0	~~0~~	3	3	3	~~3~~	2	2
f_2	0	0	0	~~0~~	~~0~~	3	3	~~3~~	2	2	2	~~2~~	1	1	1
f_1	7	7	~~7~~	2	2	2	~~2~~	4	4	~~4~~	0	0	0	0	0
	*	*	*	*	✓	*	*	*	*	*	*	✓	*	*	✓

first f_1 is loaded with Page 7 and this is page fault

* → page fault

✓ – hit

in 4 th column all frame are occupied so we have to remove the page which come first 7 is replaced with 2

total page fault = 12

total hit = 3

$$\text{Hit ratio} = \frac{\text{no of hit}}{\text{total number of reference}} \times 100\% = \frac{3}{15} \times 100\% = \frac{1}{5} = 20\%$$

$$\text{miss/fault ratio} = \frac{12}{15} \times 100\% = 80\%$$

Belady's anomly :=> sometimes (most cases)

Number of page fault increased with number of frames

like – reference string – 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

frame = 3 then Page fault = 9,

frame = 4 then page fault = 10

(ii) Optimal Page replacement algo :- =>

- Replace the pages that will not be ~~used~~ used for longest period of time
- lowest page fault
- no belady anomly
- impossible to implement as we need future knowledge of reference string.

reference-string - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

f_3			3	4	4	4	5	5	5	5	5	5
f_2		2	2	2	2	2	2	2	2	2	4	4
f_1	1	1	1	1	1	1	1	1	1	3	3	3
	*	*	*	*	✓	✓	*	✓	✓	*	*	✓

now we will check, which page (1/2/3) will be used at last (after long time)

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 will be replaced

both 1 and 2 has no demand so we can replace any of them

Hit = 5

fault = 7

iii) Least Recently used Page Replacement :=>

- replace the least recently used page
- no Belady's anomly
- implemented using counter or stack

		2	2	2	2	2					
	1	1	1	1	~~1~~	4					
	0	0	0	0	0	0					
7	7	7	~~7~~	3	3	3					
*	*	*	✓	*	✓	*					

ref. string - 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

now we are on Page 3 & will check which page has not used for long time in this case 7 is replaced now we are on Page 4 & page 1 has not been used for long time. So on

2
1
0
7

LRU using stack

- maintain a stack of page number referenced
- When page is referenced remove it from stack and put it on the top
- LRU page is at the bottom no need to search.
- Suitable for software/microcode implementation.

LRU using Counter

- Page table has a time of use field
- a logical clock which is incremented after every memory reference.
- Whenever a reference is made to a page its time of use value is set to current value.
- The LRU page has minimum time of use value. Search and replace it.

A stack algorithm does not suffer from Belady's anomaly.

Global Page Replacement: ⇒ When a process need an replacement then it can allocate it the frame from the set of all frames even it is currently allocated to other frame.

Adv: better throughput, commonly used

Dis — Page fault ratio not depend on process itself. A page depends on other page behavior.

Local Page Replacement: ⇒ A process can replace a new page in the allocated frame of itself.

adv — process can control its page fault rate.

Dis- low priority process may hinder a high priority process by not making available of its frame.

Also Frame Allocation

- Equal allocation - every process will get equal frame

  dis- processes is varying in size So no sense to give equal

- proportional allocation: every process will get according to its size.

Demand Paging — Page should only be brought into memory if the executing process demands them.

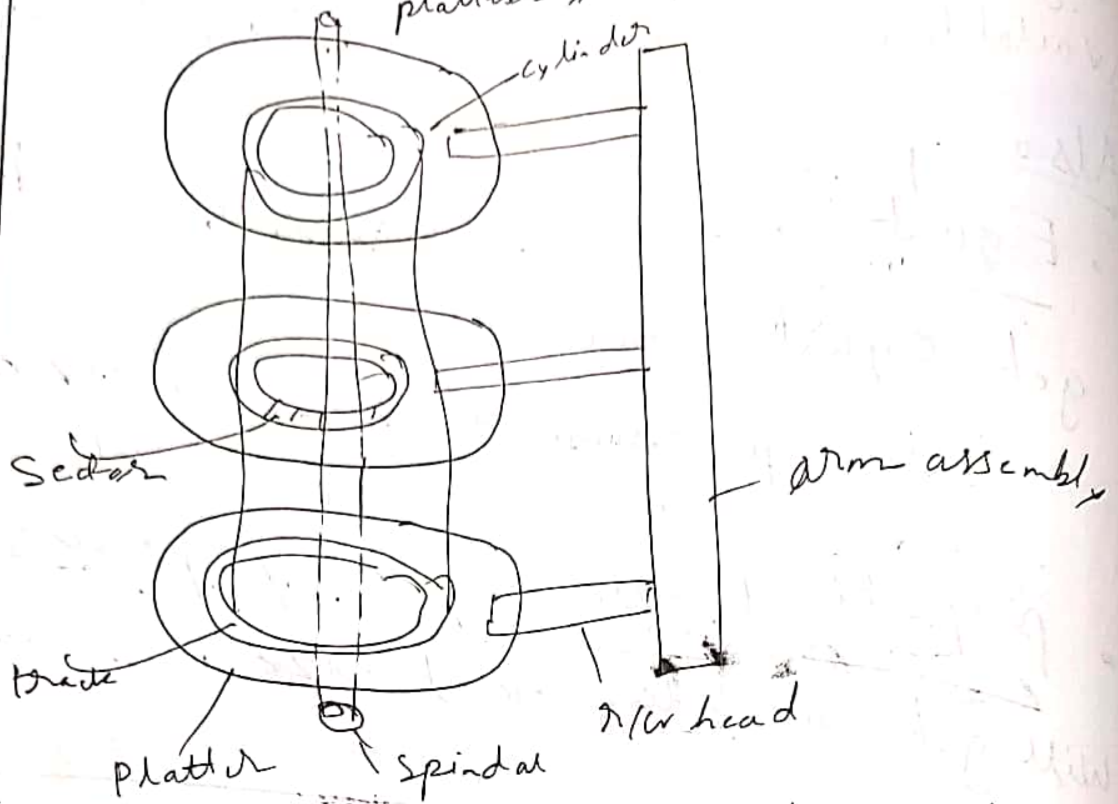
Pager: Swapper that deals with individual page is called Pager

Swapper - manipulating entire process.

⊗ Stack algorithm cant never show Belady anomaly.

Disk Management :⇒

platters → Surface → Track → Sector → data



- Each disk Platter has a flat circular shape
- Each platter has a upper & lower surface
- A read write head flies upwards ↑ or backward ↓ over platter.
- The surface of the platter divided into circle tracks
- Tracks are subdivided into Sectors
- Bits are stored magnetically on platter
- A motor rotate the disk at a high speed.
- The arm moves the head together

• The set of tracks at one arm position consist a cylinder.

(*) Seek time: time taken by R/W head to reach desired track

or

time taken to move the arm to the desired cylinder.

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

(*) Rotational latency: Time taken to reach desired Sector (average we take half of rotation time)

(*) Transfer rate: No of (surface) heads × capacity of one track × no of rotation in one second

• capacity of one track = no of sector × data of one sector

(*) transfer time = $\frac{\text{Data to be transferred}}{\text{transfer rate}}$

• Positioning time: seek time + Rotational latency

• Controller controll all of this transfer

Disk Access time = Seek time + Rotational latency + transfer time + (Controller + queue time) optional

Disk Scheduling Algorithm :⇒

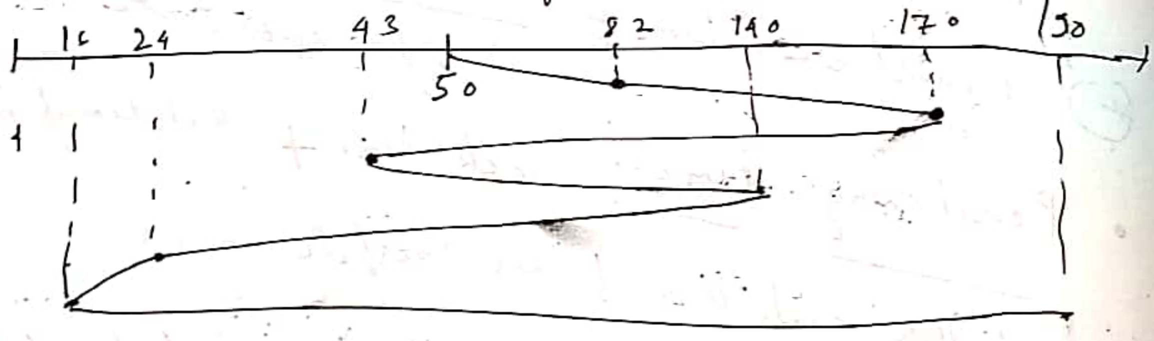
Goal: to minimize access time / Seek time
improve bandwidth

Bandwidth of disk = $\dfrac{\text{number of byte transfer}}{\text{time elapsed between first req and last service}}$

- FCFS
- Shortest Seek time first (SSTF)
- Scan (elevator) Scheduling
- C-SCAN
- Look
- C-Look

(i) FCFS :⇒

A disk contain 200 track.
Request queue is 82, 170, 43, 140, 24, 16, 190
Current position of R/W head = 50
Calculate total no of tracks movement



total no of track movement

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

$=$

- We just fallow the sequence. Which come first has been serviced.
- There is no chance of starvation

Dis:- movement is very high, Performance is low.

(ii) **SSTF** :=> Same question.

if R/W head it takes 1ns to move from one track to another then total time = ?

- We will go there where the distance between current position of R/W and that position is short.

// In this case 50-43 = 7 (lowest) so we will go to 43 first

then 43 nearest is 24, 24 nearest 16 and so on -

Adv.
- maximum time it gives lowest seek time
- Average Response time

Dis - • A request may wait for a lot of time. So starvation happen.

(iii) **Scan (elevator)** :=> Same question

- We move through same direction to the very last

// In this case we go to 82 from 50, then 140 then 170 then 190

In this way we will service till high request

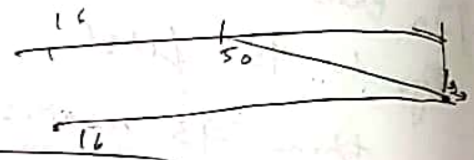
* In this case we have to go to the said direction to very last (199) but when we came back we will go to only (16).

total movement = (199 - 50) + (199 - 16)

= 332

Its work like elevator If there is people till 10 floor but it will go till 20 floor.

Dis :=> Direction change happen and then if dynamic change requenced but we can't just go that like we are backing from 199 then 195 is request but we can't go.



(iv) C Scan :=>

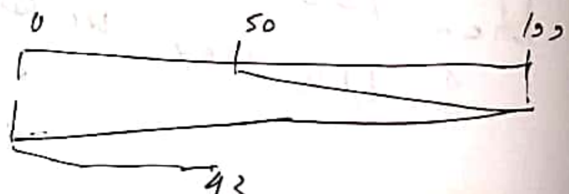
If direction is towards largest value .

- We will go to largest track value then back to 0 then again go to largest request before ~~previ~~ first position

- first we will go to 199 from 50 (though 199 is high) then came to 0 Again go to 0-43

total movement = (199 - 50) + (199 - 0) (43 - 0)

= 391



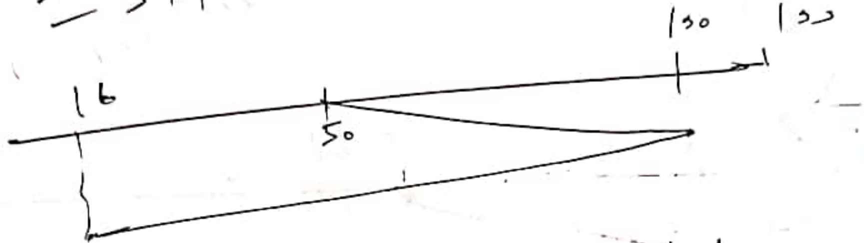
(v) Look :=> Like scan here we also should know direction

- Its same as scan but in look we will go to last largest queue not largest track value.

- we will go to 130 from 50 then back to 16 again

- $(130-50)+(130-16)$

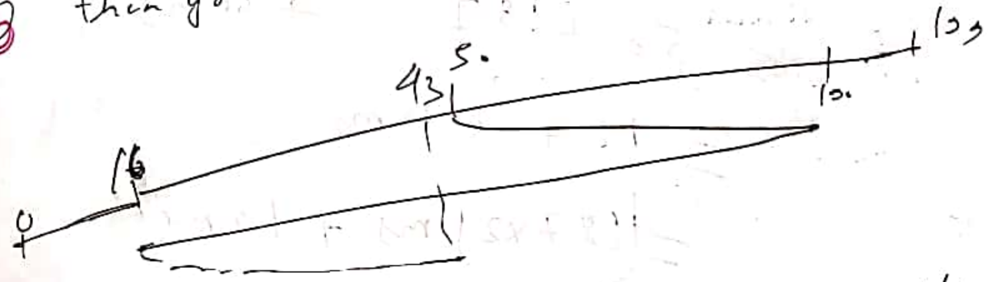
$= 314$



(vi) CLook :=> Direction is needed

- We will go from current position to the largest request value then come back to largest value before the first position then go to larget value before the first position

16



(*) Look does not cover the extra position of track

C — circular

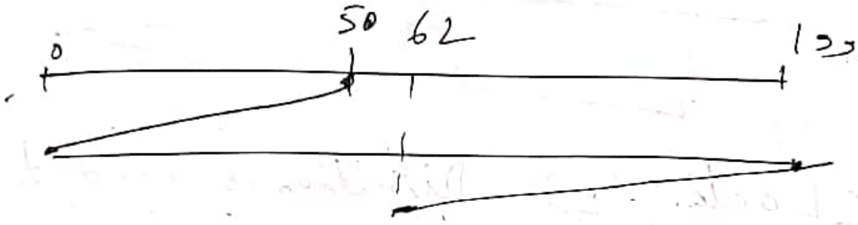
Q) Consider the following track request in disk queue

95, 180, 34, 119, 11, 123, 62, 64

Cscan algo is used. R/W head is at 50. tracks are (0–199). head is moving towards smallest value. total time need 2 msec to move one track to next track.

Total time needed is ? (time taken from one end to another end is 10 ms)

⇒



total movements

$$= (50-0) + (199-62) + \underbrace{(199-0)}_{10ms}$$

$$= 50 + 137 + 10ms$$

$$= 187 + 10ms$$

$$= (187 \times 2)\,ms + 10\,ms$$

$$= 384\,ms$$

Q)

	AT	BT
P1	0	10
P2	2	20
P3	6	36

OS use SRTF total context switch needed if you dont count at a time zero and end

(a) 1 (b) 2 (c) 3 (d) 4

P1	P2	P3

0 10 30 66

File System

Every OS has a file system which manage every file

File System is a software which manage how the data will be stored & fetched.

(*) A file is a named collection of related information that is recorded on secondary storage.

- A file is the smallest allotment of logical storage device.
- Data can not be written to secondary device unless they are within a file.
- File store both code and data

To provide efficient and convenient access to disk, the OS imposes a file system to allow data to be stored, located and retrieved easily.

File System is multilevel system

Application program
↓
Logical file system [Manage Metadata] logical block address
↓
File organization module [allocation management]
↓
Basic file system
↓
I/O control [device drivers / interrupt]
↓
Disk

Operation on files :⇒

(i) Creation (iv) Deleting — file delete along with attribute

(ii) Reading (v) truncating — file data is deleted but metadata is there file is there but empty

(iii) Writing (vi) Repositioning

File attributes: (metadata: data about data)

(i) Name

(ii) Extension (type)

(iii) Identifier

(iv) Location

(v) Size

(vi) Modified date, created date

(vii) Protection

(viii) Encryption, compression

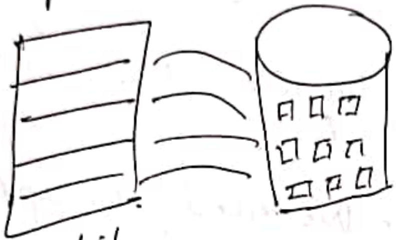
File Allocation Method :-

Contiguous

Adv
- Efficient disk allocation
- Easy access (faster)

non contiguous
- linked list allocation
- indexed allocation

In allocation file is divided in blocks logically then it is put on sectors of track of disk



file

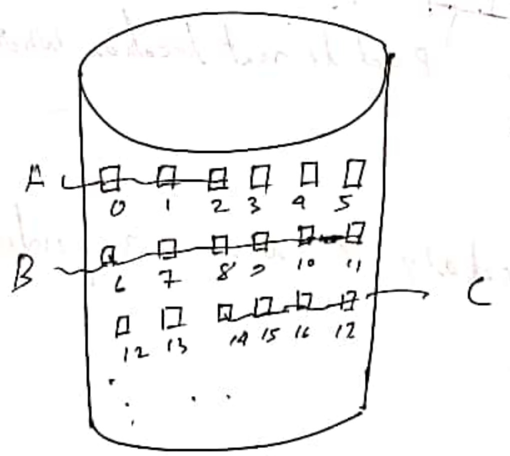
① Contiguous allocation :=>

free space

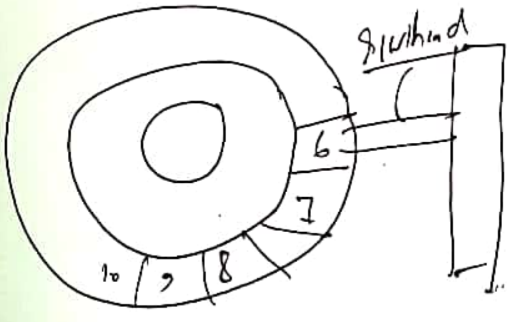
Adv - easy implement
- Excellent Read Performance

Dis: ① Disk will become fragmented. (ii) Difficult to grow file

file	Start	Length
A	0	3
B	6	5
C	14	9



file A



We have to put R/w head on same track. So we have to just move disk. no need to change track. So it is fast

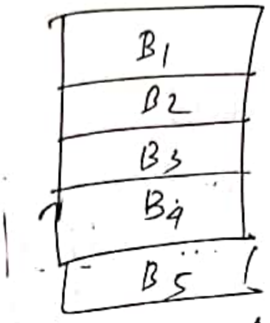
(ii) Linked list Allocation :-

Adv
- No external fragmentation
- File size can increase

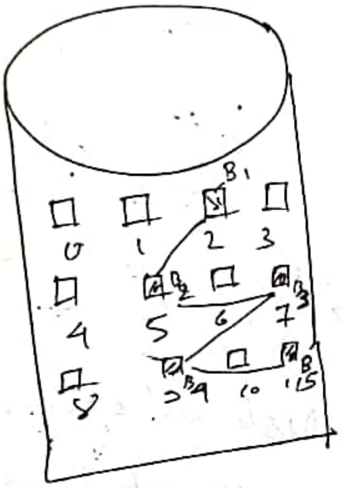
Dis
- Large Seek time
- Random access difficult
- Overhead of pointer

We can't direct access to 4th block.
with data we also keep pointer (it takes a part of memory.)

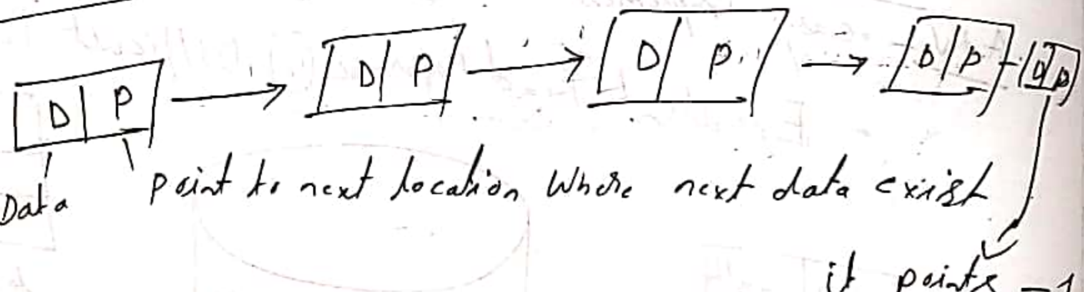
file	Start
A	2



File A



- We are giving location non continuous way.
- We need a mapping to retrieve data
- We use linked list



Directory contain a pointer to first & last block

(iii) Indexed Allocation :⇒

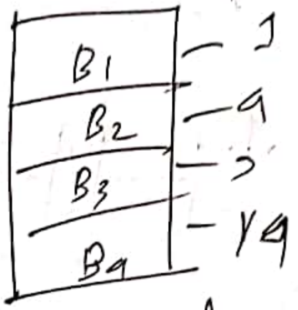
Adv

- Support direct access
- No external fragmentation

Dis

- pointer overhead
- multilevel index

file	Index Block
A	6

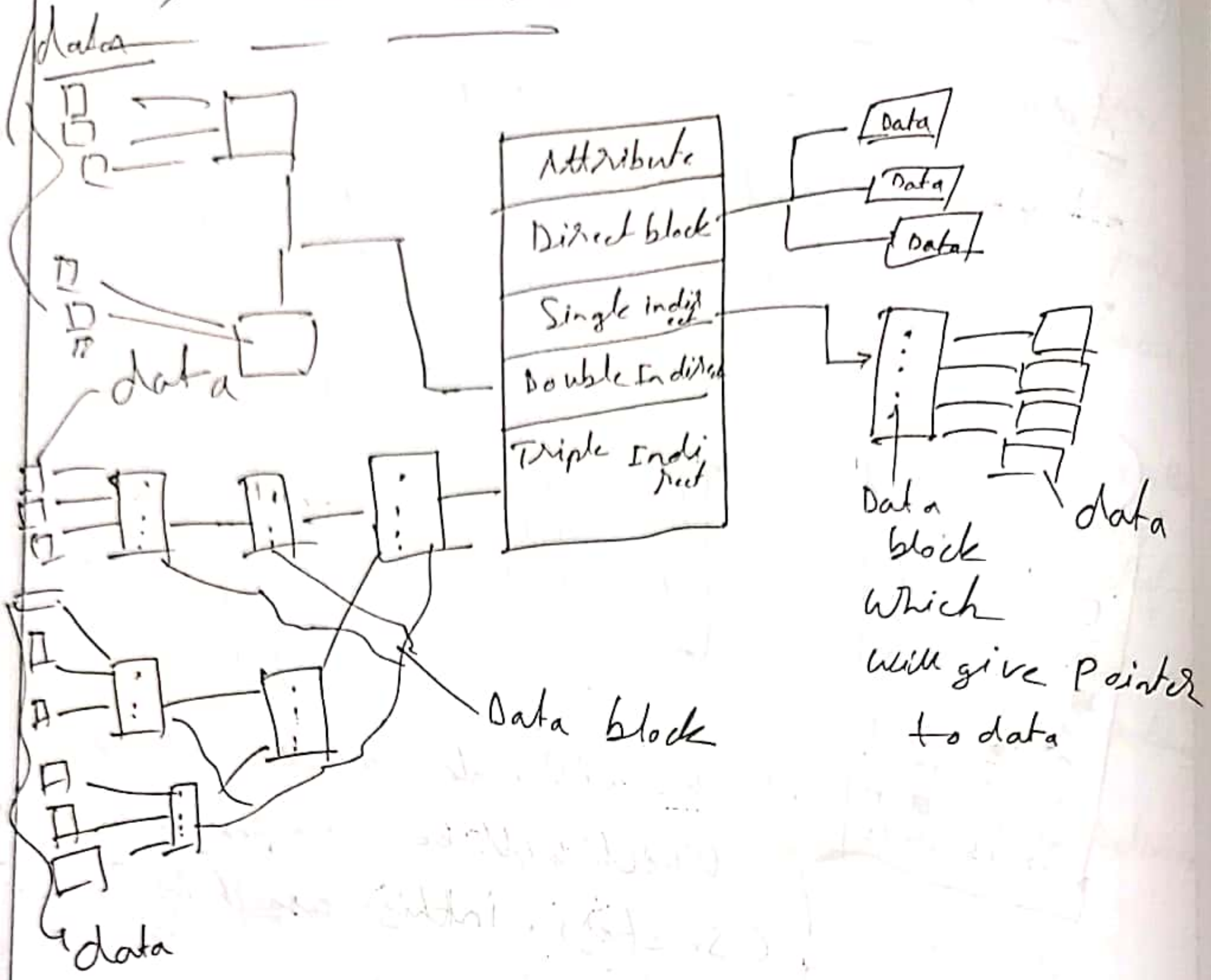


files

We will make a index of this which will be stored in a block (sector). in this case it is 6.

If file is big then index will be big so we will use multilevel index.

Unix inode Structure



- A file system uses unix inode data structure, which contain 8 direct block address, One indirect block. One double and one triple ~~block~~ indirect block. The size of each disk block is 128 B and size of each block address is 8B, Max file size.

one indirect $= \frac{128}{8} = 16$

$$\Rightarrow (8 + 16 + 16^2 + 16^3) \times 128\,B$$

$$= 547\,KB$$

Wait for graph :=>

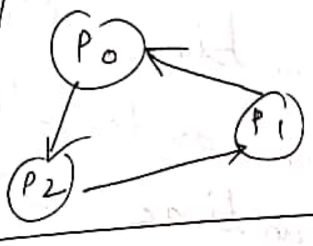
Wait for graph is used for detecting deadlock.

$P_0 \longrightarrow P_1$

P_0 is waiting for P_1 to release resource that P_0 need

if wait for graph has no cycle then no deadlock

. It only works on single instance



circular wait

cycle = deadlock

Deadlock Recovery :=>

Process to be aborted is choosed

(a) priority

. long it takes to compute

. process interactive or batch

Cost factor is low then this process aborted first

in terms of time for CPU

(*) if we preempt resource from processes and process can not continue its normal execution then it must be roll backed

atomic – Uninterruptible unit
- execute without any interrupt
- test & set in Synchronization.

Roll out → higher priority came so lower priority swap out

Roll in → higher completed so lower came.

• Compile time binding: translation of logical – Physical address in compile time

Load time :– at time of loading

Run time → in run time. Now Physical address is shifting from one to other location. This is taken care in run time binding.

• Page hit means we get the ~~physical address~~ ~~We access the~~ page from main memory (RAM) ~~the page table~~

miss means ~~we access it from Secondary (Hard disk) memory then again go to main memory to get the actual page.~~

• Page hit ratio (like $\frac{35}{100}$, $\frac{95}{100}$) × main memory access time + miss ratio × (Secondary access + main access)

$\boxed{\text{hit} + \text{miss} = 1}$

Page hit – Page Present in RAM or primary memory

page miss – we access page ~~too~~ (swapped from secondary memory

in this case first we go to main memory then if not get the page then we go to secondary memory.

~~LRU~~ MFU is useas as less used page has more chance to be used again

LFU → actively used page have a large reference count (pointer, object, memory)

Test & Set instruction : Hardware solution

Semaphore : software solution