6/7/19
Saturday

Lecture 1

→ More Problem oriented in Gate.

[Concept + Problems]

- 3 weekends.

1st weekends

Syllabus

1. Process
2. CPU Scheduling Algorithms. — I week (2M)
3. Inter Process Communication.
4. Concurrency
5. Process Synchronisation — II Week (2M)
6. Deadlocks.
7. Threads.
8. Memory Management.
9. Virtual Memory Management
10. File Management. — III Week. (3M)

3 Week Plan

OS → 7M (avg.)

- CPU Schedul. Algo → 2M (guaranteed)
- Process Sync., Concurrency → 2M (Guar.)

Mercury Mess / Deadlocks ... 1+2M.

PROCESS

OS

- must in PC / cell.
- without it, no apps can run.
  (support AP, SP)
- interface H/W user & h/W.

OS loaded here.

Several Partitions (with Processes)

Program → (exist) → Process (gets loaded in MM) → Program under execution

e.g. google chrome.exe → executable file (program)

double click → process (loaded in MM) (PS)

⇒ P2 → word doc process

- Task Manager → helps to look what process are running.

  itself is a Process.

  TM.exe.

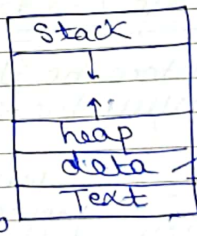
⇒ for (i=0; i<10; i++)
  s = s + i * 2;

Prog: 1 mul opern.
Process: 10 mul opern.

Process → Active Entity
Program → Passive Entity.

V.I

* Contents of Process



divided into 4 Sections.

initialized data

uninit. data.

Text: contains code

data: contains global data. (GV)

→ both grow at run time

- Heap : dynamically allocated memory (relative pointers)
- Stack → funct^n & its correspond. info is stored in Stack (paramet, return value, local V, control links, access links)

* int a = 10, b ; → ~~text~~ data (Global data). [b → UI]

```
void main ()
{
  int c=5 ;      → Stack secti^n (∵ main() → func)
}
```

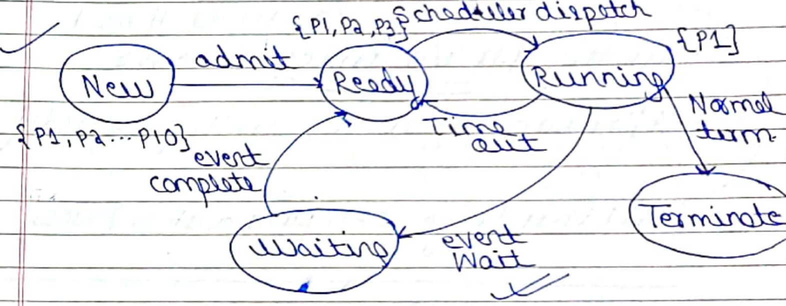
** if static int c=5; → in data section [Static V → data]

(LV → Stack). [Automatic Stor. Class] ∵ reinit. with every funct^n call. → AR pushed & popped. [as per function call].

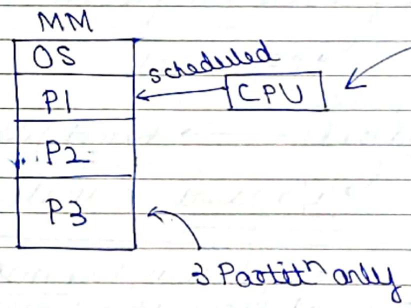
(SV → Data) → [Static Storage Class Variables]

* Process States

⇒ 5 State Process Model → Preemptive Scheduling



- if Process init. created → new State
  [P1, P2, … P10] newly created Process. (New State).



∴ P1 → running state [CPU assigned]

- {P1, P2, P3} → ready State [depending on Partitions].
- if P1 terminated → terminated S.

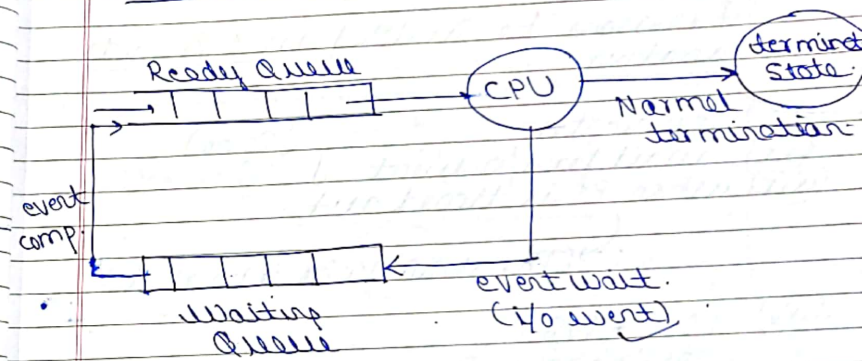
→ If P1 wants to perform I/O event.
↳ Waiting State
[CPU event / I/O event] → 2 types of event
· waiting for I/O device → WS
→ After comp of event, waiting → ready.
till this time, CPU picks another Process (CPU Sch. Algo).

Types of event

eg: large file —into→ array [store]
I/O event : read file.
eg: after reading, need computation.
↳ CPU event.
eg: array (updated) → target file
↳ I/O event : write.

* Time out : every Process has max amt of time it can spend with CPU.
· must have a TO interrupt.
· Running → ready (after TO)

(Give chance to other processes)
∴ Threshold time → TO
· Process inside ready / waiting State use Queue data structure.



· CPU event → running of CPU (execution)

* Non-preemptive Scheduling → (No forced Stop)
P1 ← CPU
P2 ← (switch).
(In which scenarios it will switch to P2).
(i) When P1 terminates.
(ii) When P1 is waiting for I/O event to happen.

→ (Process have no interest to switch).
1. terminated.
2. wait for i/o.

* Preemptive Scheduling

4 reasons to switch from 1 process to another:

(i) Terminates. } (Same)
(ii) wait for i/o want }
(iii) when P1 is timed out.
↳ [Threshold time is finished].

(iv) P1 → WS.
CPU chooses P2.

After P1 finish want → Ready S.

Now, CPU picks P1. (leaves P2).
∴ [P1 → adhe finish thaq]

eg: light off, read book

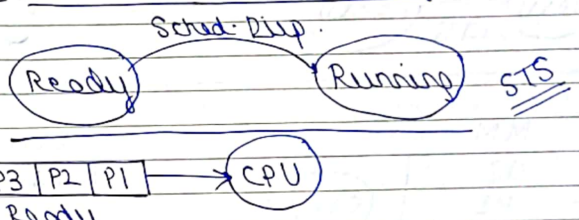
VI

Early Windows → Non-preemptive.
[Preemptive → CPU idle time ↓
Aga] performance ↑.
CPU utilizn ↑.

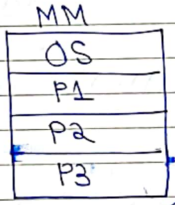
→ where you are applying Scheduling:

(i) long term Scheduling
(ii) Medium term Scheduling] → swapping concept.
(iii) Short term Scheduling.

(iii) Short term Scheduling

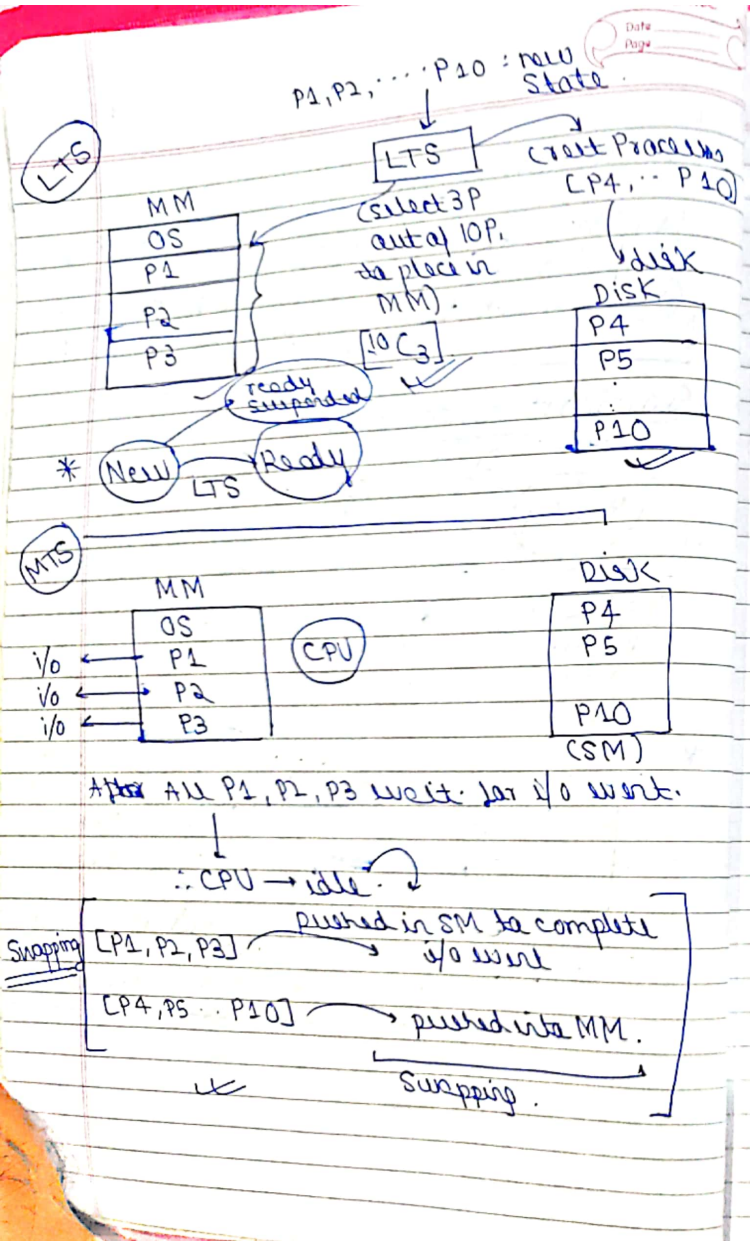


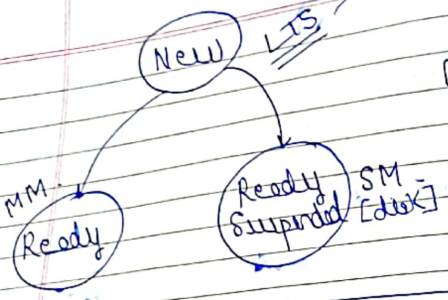
MM



CPU

[Which process to be assigned to CPU → Short term Scheduler].
(3/ more)
out of 3/ more.
[STS picks 1 order of execution].





→ will be placed in MM / SM (decision)

[ready queue / disk]

* Process Control Block (PCB)

→ actually stored in MM.

Process States
PC
CPU registers
CPU scheduling info
Memory mang. info.
Accounting info.
Device info & list of open Files

(7)

- stores complete info abt the process

(i) Process States

stores the state.
(Lists of States, it has traversed)

(ii) Prog Counter.

(iii) List of registers used by this Process
[CPU registers] → Specialised.
General Purp, Accumul.

[Obj code → Assembly code] VI

(iv) CPU sch. info: WT, AT, ET, BT, RT, TUT of Process.

(v) Mem Mang info:

If System uses Paging technique,

LA → Phys. A.
↳ requires [Page Table] (Stored in Mem. M. info).

(vi) Accounting info

[Process id, amt of real time used].

(vii) Device info

- List of resources which are held / released / requested by the Process
(& list of open files)

```
struct PCB
{
    struct mem *mm;
};
```

mem. manag. info is itself a struct (like pointer)

- every block itself is a struct.
- format stored in OS.

**

ready queue. STS.

P3 | P2 | P1 → CPU

all PCB loaded in MM.

PCB of P1 : PCB of P2 PCB of P3

front rear

(linked list implem of queue DS)

Funct^n call Scheduler helps decides where, the Process will be exactly inserted.

- if P loaded, corr. PCB must be loaded inside MM.

* Process Switching

- based on concept of context Switching.

(Context Switch, dispatcher, dispatch latency)

*

P1 P2 Process Switch.

(internally uses CS).

P1 → PCB1 stores PCB1

[CS].

P2 loades PCB2

[PCB2]

VI

[Saving the current context & loading the contexts of new Process]

PCB

context Switch.

50% CPU → then I/O → CPU.
[context needs to be saved again & again].
- or else, Process want to completed.

(ii) dispatcher

- dispatcher & scheduler work closely

Ready (STS) → Scheduler dispatch → running

Scheduler: tells, which process next to be executed.
Dispatcher: move control of CPU to that process.

(iii) Dispatcher Latency

Time taken to exec Dispatcher Code/ time taken to switch from 1 Process to another Proc.
→ move control of CPU to respective process.

* Opⁿ on Process → Proc creation
→ Proc. termination.
→ next.

* OPERATIONS ON PROCESS

(i) Process Creation.

1 Process can create another Process.
Parent → child.

P (creator) → C (newly created)

(Also known as Process Spanning)

→ fork() ; Unix. (System call)
[child P gets created].

returns → -ve value (-1) : failure.
(due to lack of mem, something, can't create child P].

Successful case. [in diff Process]
→ 0 to child Process.
→ child id to Parent

* void main()
{
pid_t pid; ← pid_t → datatype capable of storing pid.
pid = fork(); ← upto here 1 P. process
if (pid == 0) ← creates P → C
{
// child process (exec by)
}
}

Date Page 18

else
{ wait();
// executed by Parent Process
}

pid = 12345 → P
pid = 0 → C

[Text code copy to both process]

both blocks exec. by (1 per each process).

(ii) Order of exec of Parent & child:

1) Parent → child
2) child → parent
3) concurrent / parallel

but result will differ (on some system, diff order)

* To make a process wait(), till other P executed & terminated,

VI wait()
P. → Parent wait, till child gets terminated.

Date Page 19

design choices.

(iii) Impact to resource usage

→ Sharing of resources b/w P & C (all) (common resources)
→ Copy
→ Sharing few resources (not every)

Linux: fork()
main()
eg: { int a=5; → resource

if(fork()==0)
{
a=a+5; → child Process.
}
else
{
a=a-10; → Parent.
}
printf("%d",a);
printf("%d",&a);
}

P
C

C	P
a=10	a=-5

C → P
o/p 10 -5
or
-5 10
P → C
based on order.

(pid → reports copy, but stored virtually, same)

address of a → same value.

VI) &a → Virtual/logical address same, but physically a diff mem locations

internal implement.

o/p

10 -5 3538 3538
-5 10 3538 3538

V) void main()
{
fork();
printf("Hello\n");
}

P → C

copy sent to both P & C.

o/p → Hello, Hello } 2 times.

P —fork()→ C

(both will execute printf(Hello)).

I) void main()
{
fork();
fork();
printf("Hello");
}

P → C, x; C → y

∴ 4 Pro.

∴ 4 times ✓

II) void main()
{
fork();
fork();
fork();
printf("Hello");
}

8 times

P → C, x; C → y

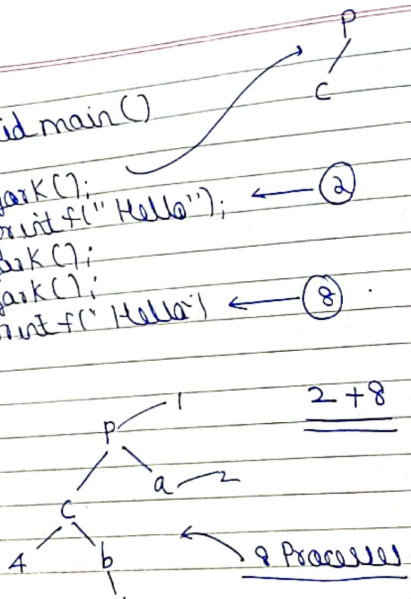
1, 2, 3, 4

→ 8 Process.

∴ If no. of fork() → n.
2^n times Hello will be printed.

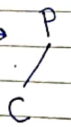
VI

Q. void main()
{
fork();
printf("Hello"); ← ②
fork();
fork();
printf("Hello"); ← ⑧
}



∴ In total → 10 times. ✓
[Hello Printed]

Q. void main()
{
int i=1;
pid_t pid;
pid = fork();
if(pid==0)
{
i++;
printf("%d\t", i);
}
printf("%d\t", i);
}



② 2 2 1

Soln

Parent	Child
	i++ ⇒ i=2
	o/p → 2 2 ∵ C terminates.
o/p → 1	

∴ o/p → 2 2 1
(if child exec)
[Parent waits until child gets terminated]
(design)

pid = fork()

Parent	Child
o/p → 1	
	c/p → 2 2

∴ o/p → 1 2 2 (if Parent exec first)
[Child waits]

Linux System

Q. wait(), inside child block, C waits, P executes. → {1 2 2}

Q) void main()

Code:
```
{
  int i=1;
  for(i=0; i<10; i++)
  {
    if(i%2==0);
    fork();
  }
}
```

Parent → P, Child → C

How many child Process?

⇒ 0, 2, 4, 6, 8

⇒ 25 = 32 − 1 = 31

0 → P, C

Soln

{i = 0, 2, 4, 6, 8} → if cond" will be true (fork() executes)

∴ 5 fork() calls

$2^5 = 32$

[Total Process = 1 + Child Process]

∴ No of child Process = 31

VI

Q)
```
void main()
{
  int i=10;
  pid_t pid;
  pid = fork();
  if(pid==0)
  {
    i=i+5;
    printf("%d\t", i);
  }
  else
  {
    i=i-5;
    printf("%d\t", i);
  }
  printf("%d\t", i+5);
}
```

O/P:

15 20 0

C/P

15 20 5 10 ✓

P → C

5 10 15 20 ✓

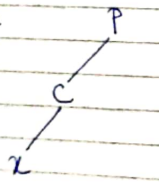
```
void main()
{
  pid_t pid1, pid2, pid3;
  pid1 = fork();
  if(pid1 == 0)
  {
    pid2 = fork();
    if(pid2 == 0)
    {
      printf("Hello");
    }
  }
  else
  {
    printf("World\t");
  }
  pid3 = fork();
  printf("Hi\t");
}
```

P → C

[X prints hello]

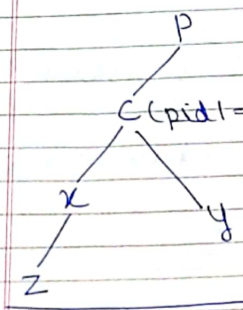
Hello.

Try:



X Hello Hi World Hi /

World Hi Hello Hi.



→ total 5 Processes.

* Process Termination

- execution of process comes to end.
  - → Normal termination / abortion
    - ↳ aborting normally
    - normal completion.
  - → using exit() System call.
  - → cascaded termination.
    [parent & child getting terminated together]
  - → Parent Process terminates the child Process.
    (if observ., C is of no use)
    - [no longer needed by Parent]
  - → trying to exceed resource usage.

P1 → m1 P2 → m2

⇒ P2 wants to use m1 / kernel resource,
∴ Terminate.

[trying to access mem. of other process / OS memory]
∴ Process terminated.

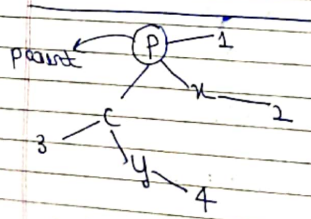
* Same Gate Qs:

Q.1 A process executes the code:

```
fork();
fork();
fork();
```

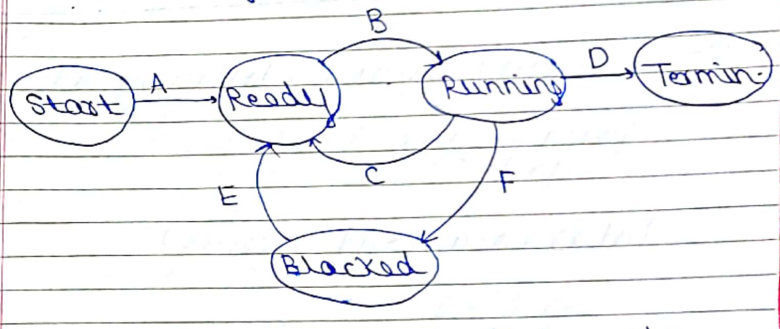
The total no. of child processes created is: 7

∵ $2^3 - 1 = 7$. Total P = 8



but all are child.

Q2. In the following Process State transition diagram, for a uniprocessor system, assume that there are always some processes in the ready state.



Now, consider the following statements:

S1: If a Process makes a transition D, it would result in another Process making transition A immediately.

S2: A Process P2 in Blocked State can make transition E while another process P1 is in running state.

S3: The OS uses preemptive Scheduling.

S4: The OS uses Non-preemptive Scheduling.

(a) S1 & S2
(b) S1 & S3
(c) S2 & S3
(d) S2 & S4.

Q3. A Process executes the following code:

```
for(i=0; i<n; i++)
    fork();
```

Total no. of child Processes?

⇒ $2^n - 1$.

Q4 Consider the following code fragment

```
if(fork()==0)
{
  a=a+5;
  printf("%d %d %d\n", a, &a);
}
else
{
  a=a-5;
  printf("%d %d %d\n", a, &a);
}
```

(&a → x, &a → y)

(a → u, &a → v)

Let u, v be the values printed by the Parent process & x, y be the values printed by the child process.
Which one of the following is true?

(a) u = x+10 & v = y
(b) u = x+10 & v ≠ y
(c) u+10 = x & v = y
(d) u+10 = x & v ≠ y

Soln:

Suppose a = 10

CP ⟶ a = 15 (x) ∴ x = 15
P ⟶ a = 5 (u) u = v.

u+10 = x, (Address is same)
⇒ print Virtual add.
(but Physically diff)

4/12/22 Sunday

Lecture 2

Scheduling Criteria: (CPU scheduling criteria.)

1. Comparing algorithms criteria:

1. CPU utilisation. (max)
2. throughput (max)
3. turn around time (min)
4. Waiting time (min)
5. response time: (min)

1. CPU utilisation

- % of CPU time utilised properly.
  if idle all time → 0% CU.

CU should be max ↑ performance ↑.

2. Throughput

No. of process executed per unit time.

Throughput → max

SRTF.

3. Turn around time

Entire time Gap b/w process submission & process completion

TAT → min

4. Waiting time

D-1: Amount of time process waits for CPU.
D-2: Amount of time process is spending inside ready queue

WT → min

5. Response time

Time taken to produce the first response

RT → min

- Avg WT & TAT is calculated for every algo.
  - Gantt Chart used.

Burst time: execution time. / service time

Based on this: (burst time) → CPU bound, → I/O bound.

⇒ CPU bound → spend more time with CPU than I/O devices (CPU burst > I/O burst).

Burst time → CPU burst time
→ I/O burst time.

① FCFS (First Come First Serve)

↳ Non-Preemptive Algorithm.

eg:

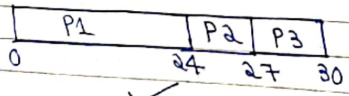
Process	Bt
P1	24
P2	3
P3	3

all Process have arrived at t=0 (AT=0)

~~Case~~ 1:

Process stored in queue : P1, P2, P3
order → [Suppose]

Gantt Chart →



* $\text{avg. WT} = \sum_{i=1}^{n} \frac{wt_i}{n}$ — n → total no. of Process.

WT of P1 = 0
WT of P2 = 24
WT of P3 = 27.

$\frac{0+24+27}{3} \Rightarrow 51/3 = 17$ ✓

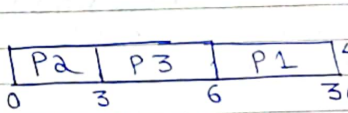
* $\text{avg. turnaround time} = \sum_{i=1}^{n} \frac{TAT_i}{n}$

TAT of P1 = 24
TAT of P2 = 27
TAT of P3 = 30.

$\frac{24+27+30}{3} = 27$ ✓

~~Case~~ 2:

P2, P3, P1 → order of arrival.



Gantt Chart.

avg WT = $\frac{0+3+6}{3} = 3$ ✓

avg TUT = $\frac{3+6+30}{3} = 13$ ✓

case 2 → min WT & TUT.

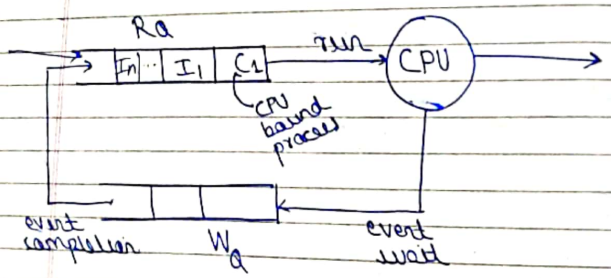
∴ Better Performance.

* execution of larger jobs before shorter jobs leads to delay in shorter jobs execution.

→ Convoy Effect

case 2 → Shorter jobs executed first.

* ~~Convoy~~ Effect → real time example



C_1 → CPU bound Process → spend most tym with CPU
not i/o bound Proces

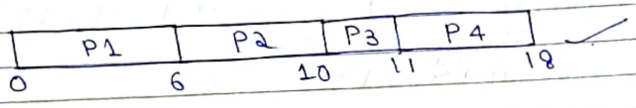
wit: CPU → C_1 = larger job
not = shorter jobs
∴ long time delay for i/o jobs.

Arrival times not Given.

eg:

Process	BT
P1	6
P2	4
P3	1
P4	7

order → P1, P2, P3, P4 (default)



- Avg WT = $\frac{0+6+10+11}{4} = \frac{27}{4} = 6.75$
- Avg TUT = $\frac{6+10+11+18}{4}$
= 11.25

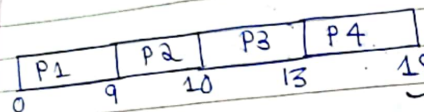
* When arrival times are Given. Model 2

Process	At	Bt.
P1	0	9
P2	2	1
P3	5	3
P4	7	5

(Order acc. to AT)

→ if same AT, accor to Pid.

- Non-preemptive → switch to new Process only when terminated.



WT of P1 = 0-0 = 0
WT of P2 = 9-2 = 7 (9-AT=7)
WT of P3 = 10-5 = 5
WT of P4 = 13-7 = 6

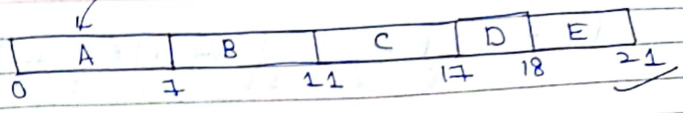
Avg WT = 18/4 = 4.5

TAT(P1) ⟶ 9-0 = 9
TAT(P2) ⟶ 10-2 = 8
TAT(P3) ⟶ 13-5 = 8
TAT(P4) ⟶ 18-7 = 11

Avg TAT = 36/4 = 9

Process	AT	BT
A	0	7
B	1	4
C	3	6
D	5	1
E	9	3

Gantt Chart



WT(P1) = 0
P2 = 6
P3 = 8
P4 = 12
P5 = 9
∴ Avg WT = 7

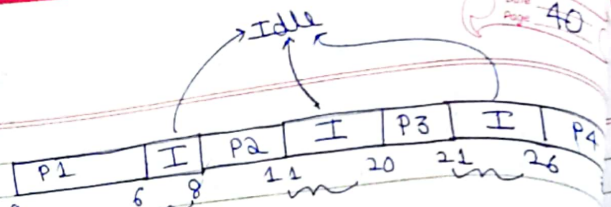
TAT of P1 = 7
P2 = 10
P3 = 14
P4 = 13
P5 = 12

∴ Avg TAT = 11.2

* Computing CPU percentage / CPU utilization.

Process	At	Bt
P1	0	6
P2	8	3
P3	20	1
P4	26	7

% of time CPU is idle



% of time = $\frac{2+9+5}{33}$

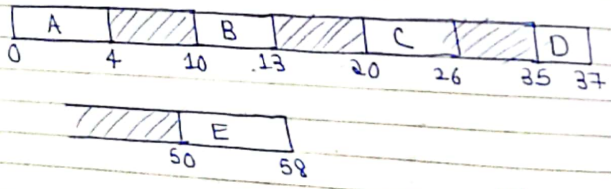
= $\frac{16}{33} \times 100$

% of time CPU busy (CPU utilisn) = $\frac{17}{33} \times 100$

Process	AT	BT
A	0	4
B	10	3
C	20	6
D	35	2
E	50	8

Gantt Chart.

⇒



% of time = $\frac{6+7+9+13}{58}$

= $\frac{35}{58} \times 100$

[CPU idle % ⇒ 60.34 %]

% CPU utilisation ⟶ 40.66 %

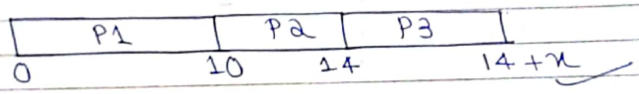
Model 3

Suppose

Process	Bt
P1	10
P2	4
P3	x

Given avg TAT = 20.

[All processes arrived at t=0; P1 P2 P3] [order]



TAT = $\frac{10+14+14+x}{3} = 20$

⇒ $38 + x = 60$

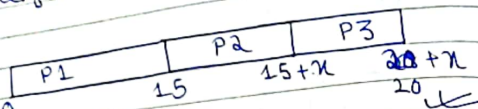
$\boxed{x = 22}$

P1, P2, P3 ⟶ order

Find x?

Process	Bt
P1	15
P2	x
P3	5

Avg WT = 20.



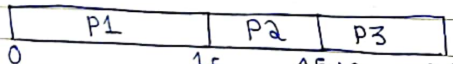
$\frac{0 + 15 + 15 + x}{3} = 20$

$x = 30$

Process	At	Bt (Arrival time Given.)
P1	0	15
P2	2	x
P3	5	5

Avg WT = 25

Soln



Avg WT = $\frac{0 + 15 + 15 + x - 5}{3} = 75$

$\Rightarrow x = 45$

P1 → 0
P2 → 15 − 2 = 13
P3 → 15 + x − 5 = 10 + x

$\frac{0 + 13 + 10 + x}{3} = 25$

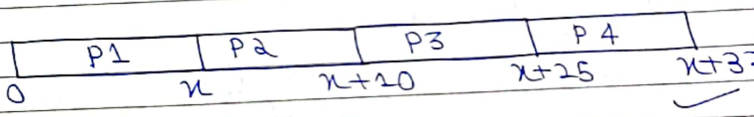
$23 + x = 75$

$x = 52$

Process	At	Bt
P1	0	x
P2	4	10
P3	7	15
P4	10	8

cond^n: $x > 10$ → VI

Avg WT → 25



P1 = 0
P2 = x − 4
P3 = x + 10 − 7 = x + 3
P4 = x + 25 − 10 = x + 15

$\frac{0 + x - 4 + x + 3 + x + 15}{4} = 25$

$3x + 14 = 100$

$3x = 86$

$x = 28.6$

Q

Process	Bt
P1	x
P2	y
P3	z

VI

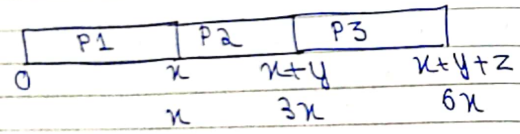
Given order of execution → P1 P2 P3

Also, $y = 2x$, $z = 3x$

Avg WT = w

Find x, y, z in terms of w

Soln



$\frac{0+x+3x}{3} = w$

$\frac{4x}{3} = w$ $\quad x = 3w/4$

$y = 3w/2$

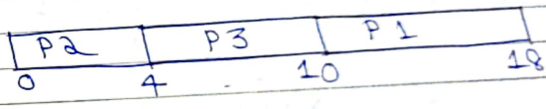
$z = 9w/4$

② SJF [Shortest Jobs First]

- shorter jobs given priority for execution. (BT)

Process	Bt
P1	8
P2	4 ← shortest job
P3	6

P2 < P3 < P1



Avg WT = $\frac{0+4+10}{3} = 14/3 = 4.6$

Avg TAT = $\frac{4+10+18}{3} = 32/3 = 10.66$

* Shortest job is selected & executed.
* No Convoy effect Here.

VI

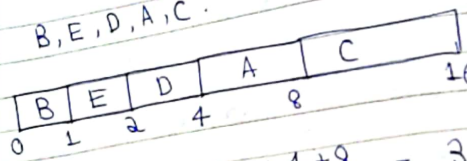
Q

Process	Bt
A	4
B	1
C	8
D	2
E	1

AM → Max

B & E → 8.

B, E, D, A, C.



Avg WT = $\frac{0+1+2+4+8}{5} = 3$

Avg TAT = $\frac{1+2+4+8+16}{5} = 6$

* Non-preemptive SJF = SJF
Preemptive SJF = (SRTF). VI

Shortest remaining time First

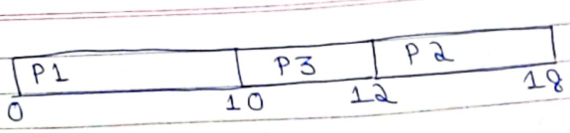
* Late coming shorter job will take control from longer time current job.

Process	AT	BT		
P1	0	10 9	10 9	10 9
P2	1	6 4	6 4	6 4
P3	3	2	2	2

I Non-Preemptive SJF

P1 0 10 9
P2 1 6 4
P3 3 2

	AT	BT
P1	0	10 9
P2	1	6 4
P3	3	2



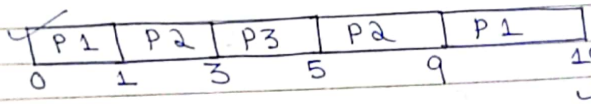
Avg WT = $\frac{0+7+11}{3} = 6$

Avg TAT = $\frac{10+9+17}{3} = 12$

* Preemptive SJF (SRTF)

	AT	BT
P1	0	10 9
P2	1	6 4
P3	3	2

- much better Results



Avg WT =

P1 → (9 − 1) − 0 = 8
P2 → (5 − 2) − 1 = 2
P3 → 3 − 3 = 0

Avg WT = 10/3 = 3.3

Avg TAT

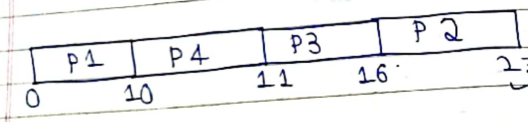
P1 → 18 − 0 = 18.
P2 → 9 − 1 = 8
P3 → 5 − 3 = 2

Avg TAT = 28/3 = 9.33

Process	AT	BT
P1	0	10 8
P2	2	6
P3	3	5 3
P4	5	1

Find avg WT & TAT in NP-SJF & P-SJF.

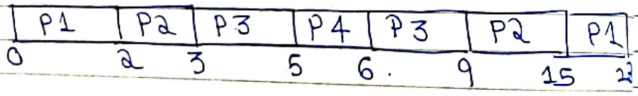
- Non-Preemp SJF



Avg WT = $\frac{0+5+8+14}{4} = 6.75$

Avg TAT = $\frac{10+6+13+21}{4} = 12.5$

- Preemptive SJF (SRTF)



Avg WT = P1 (15-2) - 0 = 13
P2 = (9-1) - 2 = 6
P3 = (6-2) - 3 = 1
P4 = 5 - 5 = 0

Avg WT = 20/4 = 5

⇒ Avg TAT

P1 = 23 - 0 = 23
P2 = 15 - 2 = 13
P3 = 9 - 3 = 6
P4 = 6 - 5 = 1

Avg TAT = 43/4 = 10.75

* SJF → 1 type of Priority Scheduling.

Priority ∝ $\frac{1}{BT}$

VI

* In SJF → Process can suffer from Starvation

longer jobs → More Wait.
(Waiting for the control of CPU)

⇒ cal. (BT) → predictive BTs
(exponential averaging technique).
BT = CPU BT + I/O BT.

VI

Predicting Next CPU Burst

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

$0 \le \alpha \le 1$

(technique of exponential averaging)

where, τ_{n+1} = predicted value of next CPU Burst

t_n = actual CPU Burst at n^{th} iteration

τ_n = predicted CPU Burst at n^{th} iteration.

- If, $\alpha = 0 \Rightarrow \tau_{n+1} = \tau_n$
- $\alpha = 1 \Rightarrow \tau_{n+1} = t_n$
- $\alpha = 0.5 \Rightarrow \tau_{n+1} = \frac{t_n + \tau_n}{2}$

$t_1 = t_2 = t_3 = t_4 = t_5 = 6$.

$\alpha = 0.5$ $\quad \tau_1 = 8$ (my prediction)

$\tau_{i+1} = 7 \quad (t_1 + \tau_1)/2$ (my next prediction)

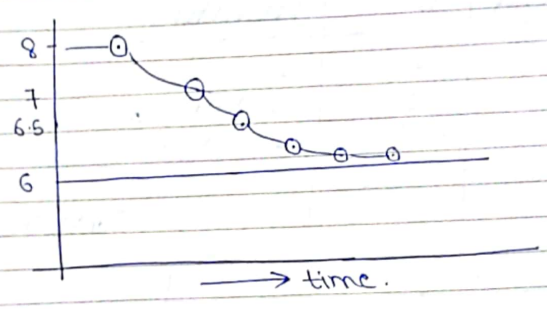
$\tau_3 = (7+6)/2 = 6.5$

[Predicted CPU Burst] (come closer & closer to actual CPU Burst)

$\tau_4 = \frac{6+6.5}{2} = 6.25$

$\tau_5 = \frac{6.25+6}{2} = 6.125$

$\tau_6 = \frac{6.125+6}{2} = 6.0625$



$P \propto \frac{1}{\tau_{n+1}}$ (Priority is inversely prop to next CPU Burst)

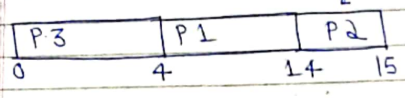
(3) PRIORITY SCHEDULING

- Process with highest Priority get executed
- in Linux: lower no → higher priority
- in General: implem. in any way

Q

Process	BT	Priority
P1	10	2
P2	1	3 (low)
P3	4	1 (high)

Gantt Chart



Avg WT = $\frac{0+4+14}{3} = 6$

Avg TOT = $\frac{4+14+15}{3} = 11$

- (i) Same Priority → FCFS)
- (i) Priority or BT → SJF)
- (I) Priority or AT → FCFS)

VI

* Priority Scheduling

- Preemptive
- Non-preemptive

⇒ Non-Preemptive Priority Scheduling

- late coming high priority Process can't take control.

VI

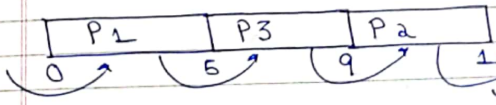
In Preemptive: late coming high priority process can take control from the process currently (with lower Priority)

Q

Process	AT	BT	priority
P1	0	~~5~~ 3	3 (low)
P2	2	~~7~~ 4	2
P3	5	4	1 (High)

~~5~~ 3
~~7~~ 4
4

Non-Preemptive Priority



{4 Context Switches}

[Don't include CS at t=0 & end]

{2 CS}

Avg WT: $P_1 = 0$
$P_2 = 9 - 2 = 7$
$P_3 = 5 - 5 = 0$

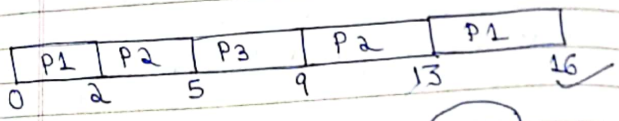
Avg WT ⇒ $7/3 = 2.3$

Avg. TAT:

P1 = 5
P2 = 14
P3 = 4

Avg TAT = 23/3 = 7.6

* Preemptive Priority Scheduling



6 CS

VI

Avg WT.

P1 = (13-2) - 0 = 11
P2 = (9-3) - 2 = 4
P3 = (5-5) = 0

Avg WT = 5

Avg TAT

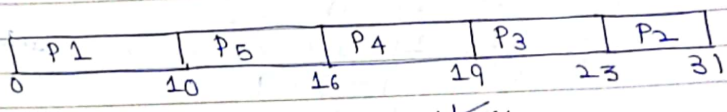
P1 = 16 - 0 = 16
P2 = 13 - 2 = 11
P3 = 9 - 5 = 4

Avg TAT = 31/3 = 10.3

Process	AT	BT	Priority	BT ↓
P1	0	10 8	5 (Low)	10 8
P2	2	8 7	4	8 7
P3	3	4 2	3	4 2
P4	5	3 2	2	3 1
P5	7	6	1 (High)	6

Soln:

Non-Preemptive



Avg WT:

P1 = (0)
P2 = 23 - 2 = 21
P3 = 19 - 3 = 16
P4 = 16 - 5 = 11
P5 = 10 - 7 = 3

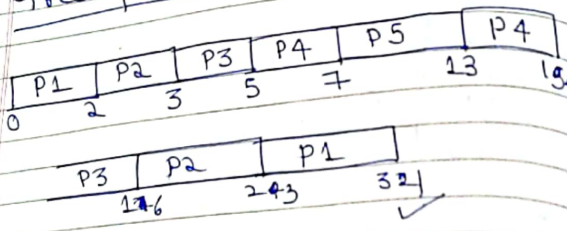
Avg WT = 51/5 = 10.2

Avg TAT

P1 = 10
P2 = (10 + 9 + 14 + 20 + 29)/5

⇒ 16.4

Preemptive



Avg WT

P1 = (23 − 2) − 0 = 21
P2 = (16 − 1) − 2 = 13
P3 = (14 − 2) − 3 = 9
P4 = (13 − 2) − 5 = 6
P5 = (7 − 7) = 0

Avg WT = $49/5 = 9.8$

Avg TAT

$$\frac{32 + 14 + 11 + 8 + 0}{5}$$

=

* Priority based on internal / ext. factors

internal Priority

ext Priority.
[If Priority of process is High]

VI

⇒ Low Priority process (in the case of SJF, SRTF) → suffer from Starvation.

[Technique to overcome Starvation]

* Prob: Starvation

Solution: Aging → improve the priority of the process (So that, that process can be exec. with other process)

* eg: In Windows OS → can change priority of Process [Task Manager]

update priority of Browser to update net Speed.

Linux → nice System call.

VI

GATE Qs

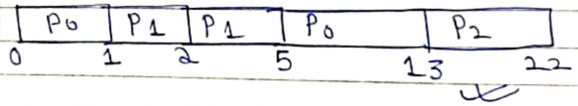
Consider the following table of Arrival time and Burst time for 3 Processes P0, P1 and P2:

Process	Arrival time	Burst time
P0	0	8 8
P1	1	4 3
P2	2	9

The preemptive SJF Scheduling Algo is used. Scheduling is carried out only at Arrival time / completion of Process.
What is the Avg WT for the 3 Processes?

(a) 5.0 ms
(b) 4.33 ms
(c) 6.33 ms
(d) 7.33 ms

SRTF



$P_0 = (5-1) - 0 = 4$
$P_1 = 0$
$P_2 = 13 - 2 = 11$

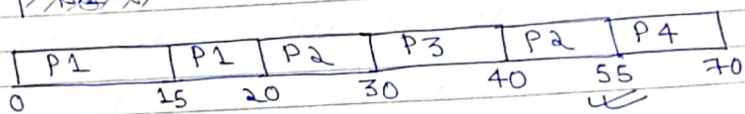
$15/3 = 5.0$ msec (a)

An OS uses SRTF process Scheduling Algo. consider the Arrival times & execution times for the following Processes.

Process	Execution time	Arrival time
P1	20 15	0
P2	25 15	15
P3	10	30
P4	15	45

BT

what is the total WT for P2?



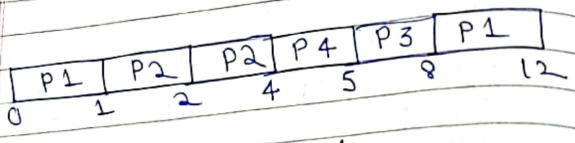
$(40-10) - 15 = 15$ (Ans)

Consider the following Set of Processes with ATs and CPU BTs (given in msec)

Process	AT	BT
P1	0	5 4
P2	1	3 2
P3	2	3
P4	4	1

5 4
3
3
1

what is the avg Turn Ar. tym for this process with the preemptive SRTF algorithm.



$$TAT = \frac{12+3+6+1}{4}$$

$$= \frac{22}{4} = 5.5$$

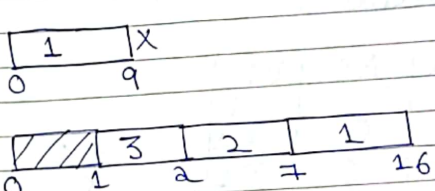
The Sequence ____ is an optimal non-preemptive Scheduling Sequence for the following Jobs which leaves the CPU idle for ____ units of time.

Job	AT	BT
1	0.0	9
2	0.6	5
3	1.0	1

(a) {3,2,1}, 1
(b) {2,1,3}, 0.
(c) {3,2,1}, 0
(d) {1,2,3}, 5.

VI

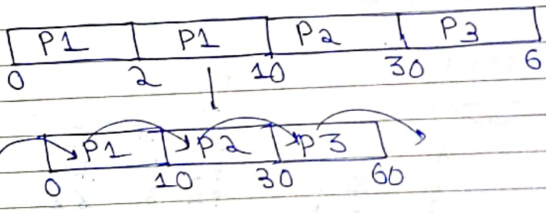
SJF → optimal. (better result than FCFS)



(a) ✓ [CPU idle in the 1st TU]

Consider 3 CPU intensive Process which require 10, 20 & 30 time units & arrive at times 0, 2, 6 respectively. How many CS switches are needed if the OS implements SRTF. (dont count the CS at t=0 & end).

	AT	BT	
P1	0	~~10~~ 8	10 8
P2	2	20	20
P3	6	30	30

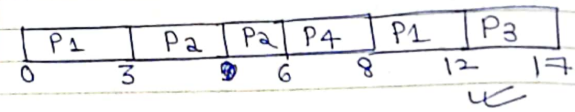


(2)

Q) 2017 Consider the following CPU processes with ATs & length of the CPU Burst (in msec) as given below:

Process	AT	BT
P1	0	7 4
P2	3	3 1
P3	5	5
P4	6	2

If preemptive SRTF algo is used, to Schedule the Process, then avg WT across all process are ____.



$P_1 = (8-3) = 5$
$P_2 = 3-3 = 0$
$P_3 = 12-5 = 7$
$P_4 = 0$

(3) msec 12/4.

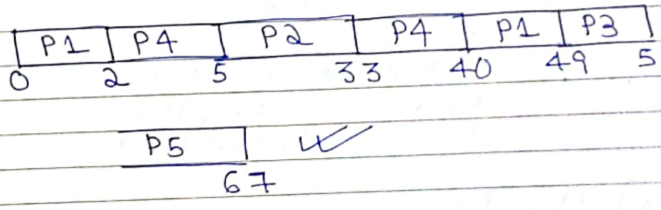
AT	BT
0	7 4
3	3 1
5	5
6	2

P	AT	BT
P1	0	7 4
P2	3	3 1
P3	5	5
P4	6	2

Q) 2017 Consider the set of Process with arrival time in (msec), CPU Burst (in msec) and priority [0 is the highest Priority] shown below:
None of these process have I/O Burst time.

Process	AT	BT	Priority	
P1	0	11 9	2	11 9
P2	5	2 8	0	2 8
P3	12	2	3	2
P4	2	10 7	1	10 7
P5	9	16	4	16

The avg WT (msec) of all the Process using preemptive priority SA.



$P_1 = (40-2) - 0 = 38$
$P_2 = 5-5 = 0$
$P_3 = 49-12 = 37$
$P_4 = (33-3) - 2 = 28$
$P_5 = 51-9 = 42$

$\Rightarrow$ 29 msec.

(4) Round Robin Scheduling Algorithm

- CPU time is equally divided among all PS.
  [time Sharing System].

(ii) Is applicable in time Shared Systems.

→ time is q shared amongst Process

"TQ/TS" ← time quantum.
= q ms → shared
{P1, P2, P3 ··· Pn}

(iii) BT(P1) > q msec : exec till q & Switch.

(iii) q time units — round 1,
then q — round 2.

* Also called as preemptive FCFS.

→ all processes exec in FCFS.
(but each P gets q time)
· then immediately switch

(iii) I : P1, P2, P3 ····· Pn.

TQ = q msec.

P1, P2 ···· Pn
↑
q

come back after ≤ (n-1)q

∵ Some Processes can be finished. So can come earlier too.

(iv) Performance of Round Robin depends heavily on Time Quantum / Time Slice.

(v) VI

Process	BT
P1	~~24~~ 20
P2	3
P3	3

all arrived at t=0

TQ = 4 msec

P1	P2	P3	P1	P1	P1	P1	P1
0	4	7	10	14	18	22	26	30

Avg WT =>

P1 = 10 - 4 = 6
P2 = 4
P3 = 7

⇒ 17/3 = 5.6 ms

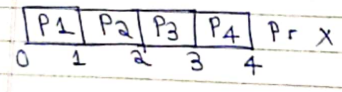
Avg. TA time

P1 → 30
P2 → 7
P3 → 10

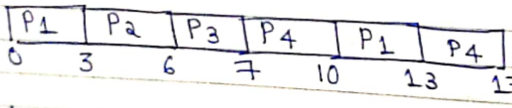
47/3 ⇒ 15.6 msec ✓

Process	BT		
P1	5	3	2
P2	2		
P3	1		
P4	6	4	3

TQ	Avg TAT
1	11
2	11.5
3	10.75
4	11.25



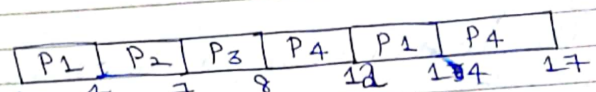
4
3



Avg. TAT = $\frac{13+6+7+17}{4}$ = 10.75 ✓

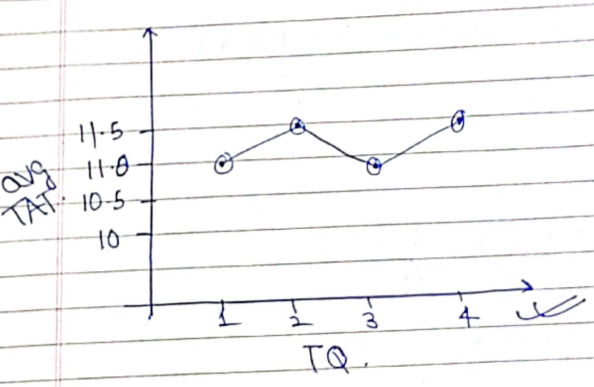
P1 6 2
P2 3
P3 1
P4 7 4 3

QT=4



Avg. TAT = $\frac{4+7+8+17}{4}$

= 11.25 ✓



(d) varies irregularly.

→ RRA produces excellent response time.

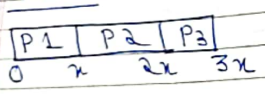
VI

Process	BT
P1	x
P2	x
P3	x

TQ = q msec.
Let $q << x$.

Technique (We mostly choose q as 20% of BT)

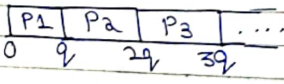
(ii) FCFS



avg Response Time = $\frac{0+x+2x}{3}$

$\Rightarrow x$

(iii) RR



avg. Response time = $\frac{0+q+2q}{3} \Rightarrow q$

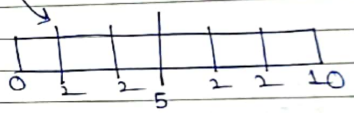
$\because q << x$.

$\therefore$ Minim. RT $\longrightarrow$ RR

* Diff b/w response time & WT.
  - no need.
  - need to subtract.

* When TQ is large, what happens to RR algorithm?
  - every process will be finished in round 1 itself.
  - $\therefore$ it behaves like FCFS

* When TQ = small, smaller than BT of each.
  - TQ ↓ small → generate more no. of context switches
  - [response time ↓].



* Round Robin Algo, when diff Arrival Times.

Q

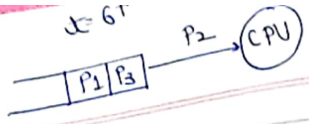
Process	AT	BT
P1	0	6 4 2
P2	1	4 2
P3	5	3 1

TQ = 2 msec

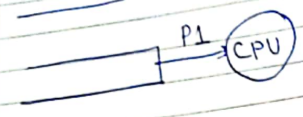
6 4 2
4 2.
3 1

P1 P2 P1

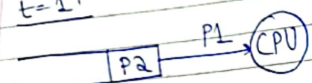
P1 P3 P2 P1 P2 P1



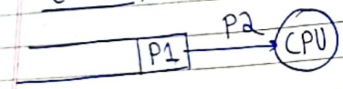
t=0+



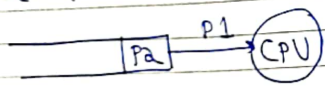
t=1+



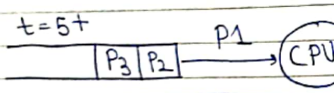
t=2+, t=3+



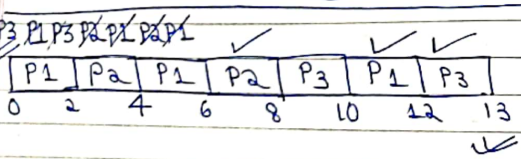
t= 4+



t=5+



P3 P1 P3 P2 P1 P2 P1



Order of completion → P2, P1, P3

WT(P1) = (10-2-2) = 6
P2 = (6-1) - 1 = 3
P3 = (12-2) - 5 = 5

Avg WT = 14/3 ⇒ 4.6

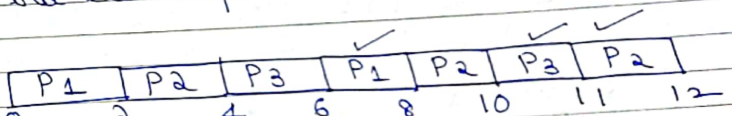
Avg TAT:

$\frac{12+8+7}{3}$

$\Rightarrow \frac{27}{3} = 9$ msec

Process	AT	BT	TQ = 2
P1	0	4 2	4 2
P2	1	5 3 1	5 3 1
P3	3	3 1	3 1

Find the completion order, AT, TAT.



WT: P1 → (6-2) - 0 = 4
P2 → (11-2-2) - 1 = 6
P3 → (10-2) - 3 = 5

Avg = 5

TAT: P1 → 8
P2 → 11
P3 → 8

⇒ Avg = 9

{P1 → P3 → P2} → completion order.

t=0+ P1 → CPU

t=1+ P2 P1 → CPU

t=2+ P1 P2 → CPU

t=3+ P3 P1 P2 → CPU

P2 P3 P2 P1 P2 P1

P1	P2	P1	P3	P2	P3	P2
0	2	4	6	8	10	11	12

avg WT = $\frac{2+6+5}{3}$

avg TAT = $\frac{6+11+8}{3}$

* Throughput
  - no. of Process completed per unit time

Throughput is poor in RR

But SJF → its good.
- optimal [Thp, Avg WT]

LRTF [Longest Remaining Time First]

5	P1	7	5	3	2	1
3	P2	8	6	4		

(in tie smaller suffix greater priority)

(P1 → P2)
[order of comp]
→ More no. of CS.

Process	BT	Priority
P1	8	H
P2	2	M
P3	3	L

Consider LRTF & when there is a tie, then value based on Prior.

Compute avg TAT.

{P1, P2, P3} → order of completion.

P1 , P2 , P3
< 8 , 2 , 3 >
5
< 3 , 2 , 3 >
1
< 2 , 2 , 3 >
1 1
< 2 , 2 , 2 >
< 1 , 2 , 2 >
< 1 , 1 , 2 >
< 1 , 1 , 1 >
< 0 , 1 , 1 >
< 0 , 0 , 1 >
< 0 , 0 , 0 >

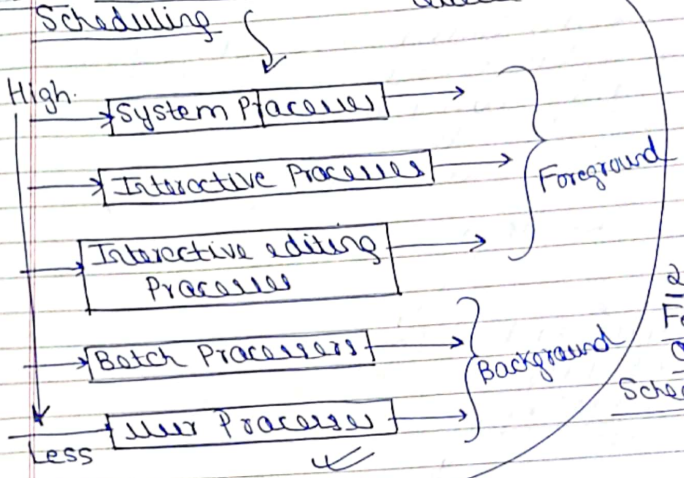
P1	P3	P1	P2	P3	P1	P2	P3
0	6	7	8	9	10	11	12

Avg TAT = $\frac{11+12+13}{3} \Rightarrow 12$ (Trick)

(6) Multilevel Feedback Queue Scheduling

Preemptive algo.

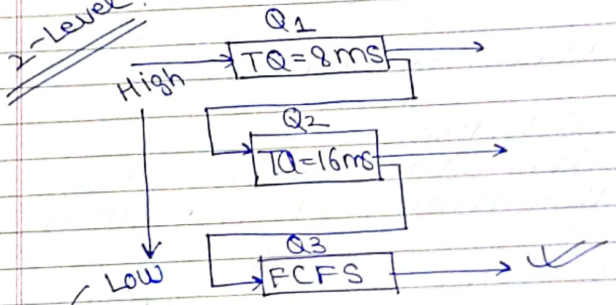
Multi-level Queue Scheduling



Multi-level Feedback Queue Scheduling

2 Level Feedback Queue Scheduling

2-Level



Process inside Q3 will be exec only when Q1, Q2 empty.

→ Max. Performance.

* Multi-level Q

- Process divided based on type of process & kept in associated queue and Priority of System P ↑ than user Processes

Process
- Foreground (served using RR scheduling Algo) eg: Interactive editing → Preemptive Algo.
- background (served using FCFS) eg: Batch Process

* Multilevel Feedback Q

- Level feedback.
- Windows till date using multi level feedback queue scheduling algorithms.

P1 = 32 ms (BT).

→ 2 Level Feedback Queue.

24 msec

8 msec

Here P1 will be exec. based on FCFS.

* Excellent Response Time

VI

- No Process generally Waiting.

(7) * HRRN [Highest Response Ratio Next]

- For every Process, compute response ratio.

$$R = \frac{W+S}{S}$$

W → Waiting time
S → Service time

- How much tym waiting for CPU?
- Burst Tym.

Non-preemptive algo

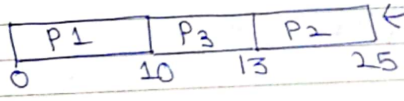
Q

Process	AT	BT
P1	0	10
P2	2	12
P3	6	3

∴ Non-preemptive.

Soln:

Non-preemptive. [Gantt Chart].



$P_2 \rightarrow R = \frac{W+S}{S} \Rightarrow \frac{8+12}{12} \Rightarrow$ @ 3/5 20/12 =

$P_3 \rightarrow R = \frac{4+3}{3} = 7/3$ ✓

$P_3 > P_2$

→ Avg WT

$P_1 \rightarrow 0$
$P_2 \rightarrow 11$
$P_3 \rightarrow 4$.

$15/3 = 5$ ✓

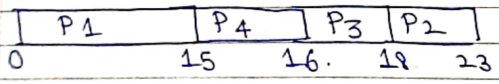
→ Avg TAT

$\frac{10+13+25}{3} \Rightarrow \frac{48}{3} = 16$ X

$\Rightarrow \frac{10+23+7}{3} = \frac{40}{3} \Rightarrow 13.3$. ✓

Process	AT	BT
P1	0	15
P2	3	5
P3	5	2
P4	8	1

VI



✓

- $R(P_2) = \frac{W+S}{S} = \frac{12+5}{5} = 17/5$ ✓

$R(P_3) = \frac{10+2}{2} = 6$ ✓

$R(P_4) = \frac{7+1}{1} = 8$ ✓

- $R(P_2) = \frac{13+5}{5} = 18/5 = 3.6$

$R(P_3) = \frac{11+2}{2} = 13/2 = 6.$ ✓

Avg WT

$\Rightarrow \frac{0+7+11+15}{4} = 8.25$ ✓

Avg TAT

$\frac{15+8+13+20}{4} = 14$ ✓

GATE Qs

Q1. Consider 3 Process (Pid 0, 1, 2 respectively) with compute time burst 2, 4, 8 with time limits. All Process arrive at t=0. Consider the longest remaining time first scheduling algorithm in LRTF, ties are broken by giving Priority to the Process with lowest Process id. Avg TAT is ——

<0,

P0, P1, P2
<2, 4, 8>
<2, 4, 4>
<2, 3, 4>
<2, 3, 3>
<2, 2, 3>
<2, 2, 2>
<1, 2, 2>
<1, 1, 2>
<1, 1, 1>
<0, 1, 1>
<0, 0, 1>
<0, 0, 0>.

no need of these many steps

P0	P1	P2	P1	P2	P0	P1	P2	P0
0	4	5	6	7	8	9	10	11	12

P1	P2	
13	14	15

when tie

Process	BT	Priority
P0	2	H
P1	4	M
P2	8	L

Avg TAT = $\frac{13+14+12}{3} = \frac{13}{}$

	BT
P0	2
P1	4 3 2
P2	8 4 3 2

2+4+8 = 14

$\frac{14+13+12}{3} = 13$

Trick

avg TAT only

Q Which one of the following Statements are true?

S-1] SRTF scheduling may cause Starvation.
S-2] Preemptive Scheduling may cause Starvation.
S-3] Round robbing is better than FCFS in terms of response time

(a) 1 only
(b) 1 & 2 only
(c) 2 & 3 only
(d) 1, 2, & 3 only

S-1] Yes, preference to short jobs.

S-2] Yes, SRTF → preemptive

S-3] Yes. ✓

Q Which of the following Scheduling algos is non-preemptive?

(a) round robin
(b) FIFO
(c) Multi level Queue Scheduling with feedback ✓

Q Consider a set of n tasks with run times $\{r_1, r_2, \ldots r_n\}$ to be run on a uni-processor machine. Which of the following processor Scheduling algo will result in max throughput?

(a) Round Robin → process completed at end.
(b) SJF
(c) HRRN
(d) FCFS

VI

→ short jobs completed fast.

SRTF > SJF ∵ preemptive.

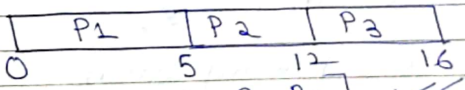
Q Consider the following table {P1, P2, P3}

Process	AT	Time units required (BT)
P1	0	~~5~~ ~~3~~ 1
P2	1	~~7~~ 5
P3	3	~~4~~ 2

The completion order of the 3 processes under the policies, FCFS & RR(2) [RR with CPU quantum → 2 units]

(a) FCFS, P1, P2, P3
(a) RR(2), P1, P2, P3
(b) FCFS P2, P1, P3, P2 ;
RR(2) P1, P3, P2
(c) FCFS P1, P2, P3
RR(2) P1, P3, P2
(d) FCFS P1, P3, P2
RR(2) P1, P2, P3

FCFS



[P1 P2 P3] ✓

$P_2 P_3 P_1 P_2$

P1 P2 P1 P3 P2 P1 P3 P2

P1	P2	P1	P3	P2	P1	P3	P2
0	2	4	6	8	10	11	13	16

P2 P3 P1 P2 P3 P1 P2 P1

t=0+ P1 → CPU

t=1+ P2 P1 → CPU

t=2+ P1 P2 → CPU

t=3+ P3 P1 P2 → CPU

P_1, P_3, P_2 (c)

Q The maximum no of processes that can be in a ready state for a computer system with n CPUs is:

(a) n
(b) n^2
(c) 2^n
(d) Independent of n.

→ depends on Size of ready queue.

Q Consider an arbitary set of CPU bound processes with unequal CPU Burst lengths submitted at the same time to a computer system. Which one of the following process scheduling algorithms would minimise the avg waiting time in the ready queue?

VI

(a) SRTF
(b) Round robbing with TQ < Shortest CPU Burst.
(c) Uniform Random
(d) Highest Priority First with priority proportional to CPU BL. → LRTF.

Min. WT → SRTF (Throughput, TAT)

Min RT → round robbing.

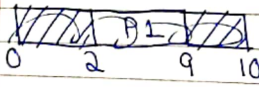
Optimal Algo.

Consider the following 3 processes; all arriving at time 0. with total execution time of {10, 20, 30 units} respectively.
Each process spends the first 20% of execution time doing I/O, the next 70% of time doing computation and the last 10% of time doing I/O again.
The OS uses a SRTF scheduling algorithm, & schedules a new process either when the running process gets blocked on I/O, or when the running process finishes its compute burst.

Assume that all I/O operations can be overlapped as much as possible.
For what percentage of CPU time does the CPU remain idle?

Soln:

	BT.	I/O	CPU	I/O
P_1	10	2	7	1
P_2	20	4	14	2
P_3	30	6	21	3



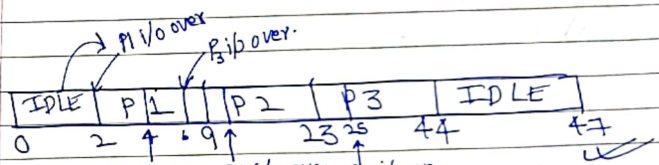
→ all have their resources.

VI

I/O.

Soln.

Process	Total exec. Time	20% [I/O]	70% [CPU]	10% [I/O]
P1	10	2	7	1
P2	20	4	14	2
P3	30	6	21	3



% of time = $\frac{5}{47} \times 100$

$\Rightarrow 10.6\%$

Consider n processes sharing the CPU in a round robin fashion.
Assuming that each process switch takes 's' seconds.
What must be the the quantum size q such that the overhead resulting from process switching is minimized but at the same time, each process is guaranteed to get its turn at the CPU atleast

every t sec?

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

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

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

(d) $q \geq \frac{t-ns}{n+1}$

VTU.

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

$t \leq qn - q + ns$.

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

$\Rightarrow ns + (n-1)q \geq t$.

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

13/7/19 Saturday.

Lecture 3

* Inter Process Communication

Process into 2 types (wrt comm.)

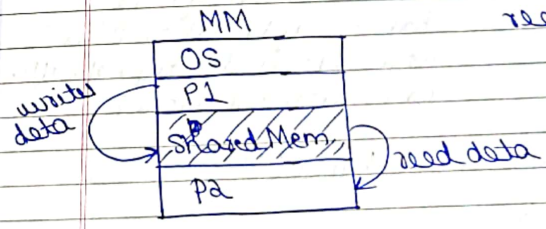
1. Cooperative: Process dependent on another proc/ relying on other for data sharing.
2. Independent: not communic. with any other Process for data Sharing

* 2 techniques for IPC: → basic for Parallel Programming, distribu. systems.

(i) Shared memory
(ii) Message Passing technique

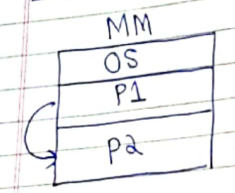
* Shared Memory

- 1 process will create the SM (P1/P2)
- Start writing data – P1
  read – P2 (from SM)



eg: blackboard
eg: Status in WA.

* Message Passing



(i) send()
(ii) receive()

- 2 proc can comm with each other by sending/receiv. msgs [Sender, receiver]

- Send(msg, Destination)
- Recv(msg, Src)

implemented in MPI (Msg passing interface).

MPI_send()

MPI_receive(...)

↳ 6 parameters

(i) data type, array name, how many loc^n it can access, receiv process, data type of buff, communicator name?

→ communication is of 2 types (MP)

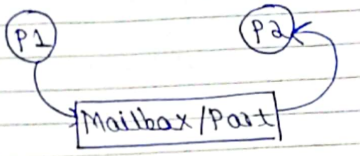
- direct comm^n
- indirect comm^n.

- Direct communic^n

send (Msg, P2)
rec, (Msg, P1)

- Indirect

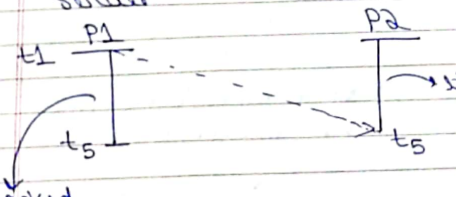
Send (Msg, Part)
Rec (Msg, Part)



* unt. blocking

1. Blocking send → sender is blocked [MPI_send]

Sender



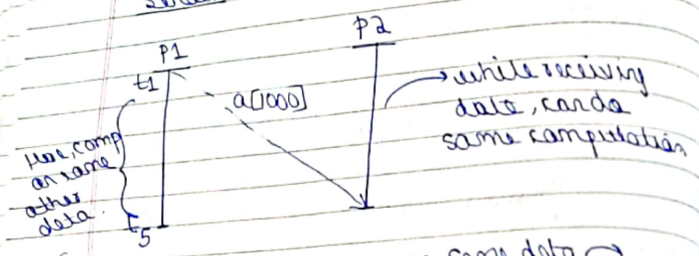
blocked
(Sender blocked until msg is received / send operation is completed).

2. Blocking received → receiver blocked till send oper^n completed.

MPI_recv()

3. Non-Blocking Send / Receive → MPI_Isend() → MPI_Irecv()

Sender — Receiving



(4 while sending, comp on some data → array data is given to P2)

→ overlap b/w communicⁿ & computation.
∴ Performance ↑

Blocking send

[overall exectⁿ time = communication time + computation time.]

⇒ 20 + 80s ⇒ 100s

⇒ Non-Blocking Send

[overall exec time = max(commⁿ time & computⁿ time)]

⇒ 80

* Buffer Capacity → msgs stored in buffer.

(i) zero capacity buffer
(ii) bounded buffer capacity.
(iii) unbounded buffer capacity.

(i) Not storing any msg.

(ii) bounded to certain length.
like Array implementn. (Static memory)
[Linear Queue / Circular Que]

(iii) no limit to capacity of buffer.
(dynamic DS)
[Queue with LL implementn]

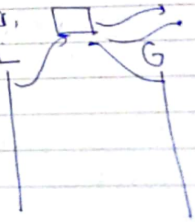
* CONCURRENCY

Serial exec, Parallel exec, Concurrent exec./ Interleaved exec

Serial: 1 caller m/c used by one by one, one after other

Parallel: 10 caller m/c (multiple CPUs, many tabs) ⇒ parallelly many will get.

Concurrent: 3 caller m/cs, Gerd & I



Single CPU concurrent exec.

CPU

S1 → S2 → S3 → S4

concurrent exec.

P1	P2
S1	S2
S3	S4

interleaved execution.

* concurrent execution

P1	P2
S1	S2
S3	S4

(CPU) → all are valid statements

⇒ order of execution:

1. S1, S2, S3, S4
2. S1, S3, S2, S4
   S2, S4, S1, S3
   S2, S1, S3, S4
   S1, S2, S4, S3
   S2, S1, S4, S3
   S1, S4, S2, S3 X (exec S2 first before S4.)
   S1, S3, S4, S2 X

(order must be preserved within process)

P1	P2	a=1, b=1
a=b+a	b=a*b	
c=a*2	c=b+2.	

Max value of c after executing both P concurrently?

6 Possibilities.

1. P1: a=b+a : a=2
   P1: c=4
   P2: b=2
   P2: c=4 → FV. ✓

2. P2: b=1
   P2: c=3
   P1: a=2
   P1: c=4 → FV. ✓

3. P1: a=2
   P2: b=2
   P2: c=4
   P1: c=4 → FV.

   (1) → 2 distinct V. of c.
   c=3, 4

4. P1: a=2
   P2: b=2
   P3: c=4
   P2: c=4 → FV.

   Max value of c = 4

5. P2: b=1.
   P1: a=2
   P1: c=4
   P2: c=3 → FV.

6. P2: b=1.
   P1: a=2
   P2: c=3
   P1: c=4 → FV. ✓

Q a=2, b=3

P1	P2
a=b+a	b=a*b
c=b*3	c=a+10

Max value of c?
How many distinct?

Check

1. a=5, c=9, b=15, c=15 ✓
2. b=6, c=12, a=8, c=18 ✓
3. a=5, b=15, c=15, c=45 ✓
4. a=5, b=15, c=45, c=15 ✓
5. b=6, c=12, a=8, c=24, c=18.
6. b=6, a=8, c=18, c=18

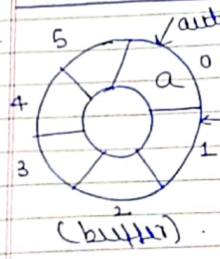
Distinct values = {45, 18, 15, 12, 9}

Q a=1, b=4, c=2

P1	P2
a=b+c	b=c*a
c=a*b	c=a+b

1. a=6, c=24, b=144, c=150
2. b=2, c=3, a=5, c=10
3. a=6, b=12, c=18, c=72
4. a=6, b=12, c=72, c=18
5. b=2, a=4, c=8, c=6
6. b=2, a=4, c=6, c=8

- Max Value → 150 ; min → 6.
- 6 Distinct Values.

* 


```
#define MAX 6
typedef struct
{
} item;
item nextProduced,
     nextConsumed;
item buffer[MAX];
int in=0, out=0;
int counter = 0;
```

where to insert — from where to delete

* Producer

```
do
{ while(counter == MAX)
    /* do Nothing */
  buffer[in] = nextProduced;
  in = (in+1) % MAX;
  counter++;
} while(TRUE);
```

circular system

* Consumer Process → not allowed to access when buffer is empty

```
do
{
while (counter == 0)
    /* do Nothing */;

nextConsumed = buffer[out];
out = (out+1) % MAX;        → out points next incoming
counter--;

}while(TRUE);
```

* Race Condition

Counter = 5

counter++	counter--
t1: R1 ← counter	R2 ← counter : t3
t2: R1 ← R1 + 1	R2 ← R2 - 1 : t4
counter ← R1	counter ← R2 : t5

Race Condn: due to improper synchronisn, it leads to incorrect results.

t1: R1 = 5
t2: R1 = 6
t3: R2 = 5
t4: R2 = 4
t5: counter = 4
t6: counter = 6

∵ updating at the same var, at the same time → RC

after 1 inc & 1 dec → counter value should same.

If counter++ b4 counter--; no race condn. But here, concurrently executed.

* To overcome this condn:

- Process must be properly synchronised.

Counter → Critical Section

(only 1P can access CS at a time).

* Critical Section/Region

It is a place where commonly shared variables, commonly accessible resources & commonly updatable tables are available.

eg: Counter variable commonly accessible by Producer & consumer, can be a part of CS.

* Solution to CS Problem

[In Gate → CODE, Progress].

[4 Non Process can't get into CS → Deadlock]

Pi (4 want to acc CS)

```
do
{
  Entry Section
  // Critical Section
  Exit Section
  // Remainder Section
}while(TRUE);
```

* Soluⁿ to CS must satisfy 3 requirements.

1. Mutual Exclusion
2. Progress
3. Bounded Waiting

Entry Section:

- Process sends req. to get entry into CS.
- req will be processed into ES.

Exit Section

- will allow some opn so as to allow other process to into into CS (while leaving).

eg:- acquire lock & then enter CS, while leaving, release lock.

(ES: acquire lock; EXS: release lock)

* Remainder Section: code not dependent on Critical Section code, in RS.
like in, out.
(Remaining code)

eg: In book my show, lock for certain time & then after movie, release lock.

[Multiple Process pay req. to entry Section]
(1 into into CS)

eg: Std Booth. (Commonly used resource → Phone)

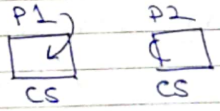
---

1. Mutual exclusion

P1 → Producer, P2 → Consumer.

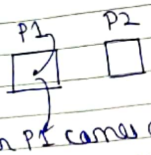
- When P1 inside CS, no other Process allowed in Corresponding CS

[At any given time, only 1 process should be inside CS].



(ii) Progress.

- CS shouldn't be empty (idle).



- When P1 comes out of CS, another process must be chosen (from ES) to enter into CS.
- Selection must be performed on the process other than RS.
  (∵ RS process work on non critical code, don't need CS)
- Process other than RS are chosen to enter into CS.

→ [Progress Satisfied, means strict alternation must be violated]

P2 → non Section work
P1 → CS
(P1 came outside)
- Now P1 can't enter again.
[CS will be empty, but satisfies Strict Alternation]
P1, P2, P1, P2 ..
∴ P1 waits. Progress X.
∴ [CS → idle]

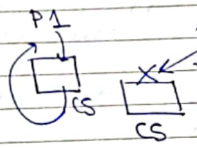
Progress.

if allows P1 again to enter CS.
if wait → SA

(iii) Bounded Waiting.

- Bound on no. of times, process enters / leaves in CS, when another process willing to enter / made request. (Req pending)

→ P1 entering again & again, not giving chance to P2.



- P1 should give chance to P2 also.

SA

⇒ flag = 1

→ Progress violated.

P1	P2
while(flag == 0) { }	while(flag == 1) { }
// CS	// CS
flag = 0;	flag = 1

Exit

now P1 can't enter again (Spinlock), will wait for P2 to modify.

P1, P2, P1, P2 ....
→ Strict alternation.
[CS empty → violating progress].

* if P1 → while(flag = -1) → DL.

* Software Solution
[Peterson's Solution]

flag[i] = true, Pi interested

Boolean Array → flag
integer → turn.

flag[i] = true, Pi is interested to enter into CS

turn = i, It is Pi's turn to enter into CS

boolean flag[2]; ← comm SV.
int turn;

Pi
```
do
{
  flag[i] = True;
  turn = j;
  while(flag[j] && turn=j);
  // CS
  flag[i] = False;
  // RS
}while(true);
```

Pj
```
do
{
  flag[j] = TRUE;
  turn = i;
  while(flag[i] && turn
        = =i);
  // CS
  flag[j] = False
  // RS
}while(True);
```

- init, flag[i & j] → false.

Pi

- Pi wants to enter CS.
- while → Pj's turn & also Pj is interested.
  ⇒ then Pi waits.
- (if any one is false) → Pi enters.
- exit time → flag[i] = false (not interested).

① Mutual exclusion

concurrent execution.

Pi : flag[i] = True;
Pj : flag[j] = True;
Pi : turn = j
Pj = turn = i
Pi = while(flag[j] && turn = = j); (flag[j]: T, turn = = j: F)
→ CS.

Pj = while(flag[i] && turn = = i); (flag[i]: T, turn = = i: T)
→ Spinlock.

if starts with Pj → Pj in CS.

② Progress

1 → Pi = flag[i] = false
(Pj will exec. while & so into CS.
while(flag[i] && turn == i)

2 → Pi will enter again & again (if Pj busy).

③ bounded wait

Pi return again. Pj → flag[j] = true.

∴ Pi not allowed, when another Pj is requesting to enter into CS.

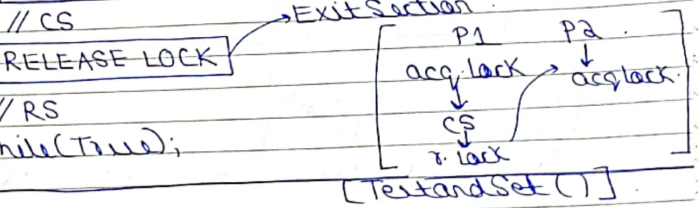
14/7/19 Sunday

Lecture 4

* HARDWARE SOLUTION.

⇒ Basic Idea:

```
do
{
    ACQUIRE LOCK        → Entry Section.
    // CS
    RELEASE LOCK        → Exit Section.
    // RS
}while(True);
```



[TestandSet()]

① * Mutual exclusion using Test() & Set()

→ atomic hit (only 1P will be testandSet at a time)

```
do
{
    while(TestandSet(&Lock));
        // do Nothing.

    // CS

    Lock = False;
    // RS
}while(True);
```

[Lock is a Shared V.]

```
boolean TestandSet (boolean *Target)
{
    boolean rv = *Target;
    *Target = TRUE;        → lock value modified to True.
    return rv;
}
```

↳ return F.

init value of Lock = False

P1 P1 Lock [F T] P2

2000

rV [F] Target [2000]

Target is a pointer V. which contains add of lock

∴ return F ⟶ P1 will enter into CS

⇒ P2

```
rV = True;
*Target = True,
return true,
```

∴ while (True); P2 ⟶ Spin Lock (busy Waiting)

∴ Mutual Exclusion

(For 1 Process, TestandSet() returns False. for rest of the processes, TS() returns True ∴ ME)

② Mutual Exclusion using Swap() → Not complete soln.

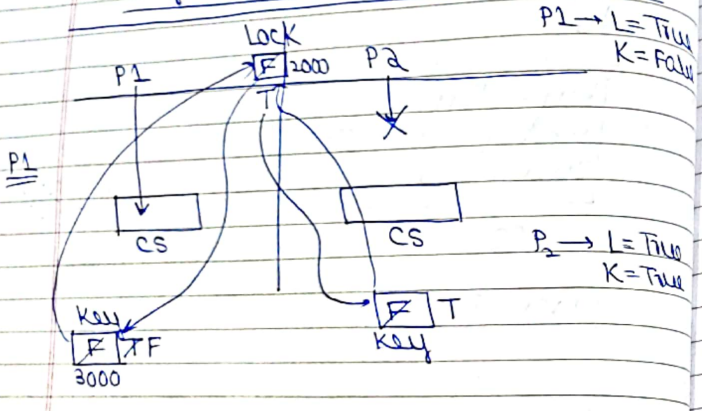
```
do
{
    Key = True;
    while (Key == True)
    swap(&Lock, &Key);

    // CS

    Lock = False;

    // remainder Section.
}while(True);
```

```
void swap(boolean *x, boolean *y)
{
    boolean temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

Key is a private V. Each private for each process. (local V)

Lock → globally Shared V.

Key: initvalue = False.



(contents of lock & Key will exchange)

(P1 inside, P2 wait).

when, P1 leaves, Lock = F.

P2 swaps → Key = F.

∴ enters into CS.

∴ [1 process into CS at given point of time].

n → Total no. of process.

* COMPLETE H/W SOLUTION

Code for 1 Process (Pi)

```
do
{
    waiting[i] = TRUE;
    Key = TRUE;

    while (waiting[i] && Key)
        Key = TestandSet (&lock);
    Waiting[i] = False;

    //CS

    j = (i+1) % n;
    while ((j != i) && !waiting[j])
        j = j+1 % n;

    if (j == i)
        lock = FALSE;
    else
        waiting[j] = False;

    //RS

} while (True);
```

Boundod Waiting.

check (test after Process)

waited for Progress

init lock & Key = False

GV. LV

P0, P1 (j) (i)

* n Process → {P0, P1 ... Pn-1}
  ✗ ✗ ... ✗
  ↓
  CS (mutual exclusion)

① Mutual exclusion

while (true & true)
  Key = False;

∴ waiting[i] = False.
  ↓ Pi
  CS

Now, Pj shouldn't be allowed.
Now, testAndSet returns true;
∴ Pj → SpinLock. ✓

② when, lock = F, (if j == i);
Pj will enter into CS
then, waiting[j] = false
↳ Pj will enter
∵ while(waiting[j] && key).
  ↓ F ↓ T
  F.

* either will enable Pj to enter

why if-else?
if only 1 Process, lock = F will allow that process to enter again (j = = i)

③ Bounded Wait

P0 P1 P2 P3 → 4 Processes. n = 4.

CS CS CS CS

due to BW, P0 after coming out, will check P1,
if P1 → RS
P2,
if P2 → RS.
check P3.
⇒ but if P1, P2, P3 wants to enter,
(P1 will enter).
if P1 not willing
(P2 ✓)

Pn-1 ... P0 P1 ... Pi+1 ... Pi

∴ i = 0 → P0 enters CS. P1, P2, P3 in SL.

j = 1 % 4 ⇒ 1

while(1 != 0) && !waiting[1].
  T F.
⇒ F.

if (0 == 1)
{ else, waiting[1] = False;
∴ P_1 enters ✓

Now, if P_1 exit waiting.

while ((1 != 0 && ! waiting[1])
T T

j = (1+1) % 4 ⇒ 2

- while (2 != 0 && ! waiting[2])
T F

∴ waiting[2] = False, ∴ (P_2 will be allowed)

⇒ BW give chance to all other Processes.

(if none will be allowed)

j = (3+1) % 4 = 0.

if (0 == 0)
wait = False; (P_0 can enter again)

import of if condition

GATE Qs

Q1 Consider Peterson's algorithm for mutual exclusion between 2 concurrent Processes i and j. The program executed by P_i is shown below:

P_i — P_j: check conditions of P_j

Repeat
flag[i] = true;
turn = j;
while (P) do no-op;

Enter Critical Section, perform actions then exit Critical Section

rem. for P_i

flag[i] = False;

Perform other non-critical Section actions until false;

For the program to guarantee mutual exclusion, the Predicate p in the while loop should be:

(a) flag[j] == true and turn = i
(b) flag[j] == true and turn = j
(c) flag[i] == true and turn = j
(d) flag[i] == true and turn = i ✓

Q2. A critical Section is a Program Segment.

(a) which should run in certain specified amount of time.
(b) which avoids deadlocks
(c) where Shared resources are accessed.
(d) which must be enclosed by a pair of Semaphore operations P & V.

Q3 2 processes P1 and P2 need to access a CS of code. Consider the following Synchronization construct used by the processes.

P1

```
while(true)
{
 wants1 = true;
while(wants2 == true)
; */do nothing */

// CS

wants1 = false
} // RS
```

P2

```
while(true)
{
 wants2 = true;
while(wants1 == true);
// do nothing.

// CS

wants2 = false;
// RS
}
```


Here, wants 1 and wants 2 are Shared variables which are initialised to False.

Which are of the following statements is true about the above construct?

(a) It doesn't ensure mutual exclusion.
(b) It doesn't ensure bounded waiting
(c) It requires that Process enter the CS in strict alternation.
(d) It doesn't prevent deadlock but ensures mutual exclusion.

Soln:

init, wants 1 = wants 2 = F.

P1 in CS, P2 can't enter (in SL).
∵ (wants 1 = true)

∴ ME ✓

if concurrent.
∴ P1 & P2 both in SL ⟶ deadlock.

both wants 1 & wants 2 = true.

(c) P1 can enter any no of times.

Q4 Consider the methods used by process P1 and P2 for accessing their Critical Section whenever needed as given below. The initial values of Shared boolean variables S1 and S2 are randomly assigned.

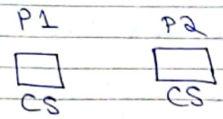
Methods used by P1	Methods used by P2
while(S1==S2);	while(S1!=S2)
Critical Section	Critical Section
S1=S2;	S2=not(S1)

⇒ which one of the following statements describes the properties achieved?

(a) Mutual exclusion but not progress
(b) Progress but not mutual exclusion.
(c) Neither mutual exclusion nor progress.
(d) Both mutual exclusion & progress.

Soln

S1	S2	
T	T	
T	F	→ P1 in CS.
F	T	→ P2 in CS
F	F	



(a)
⇒ Strict alternation ✓

P1 P2 P1 P2 ...

∴ Progress violated (P1 cant enter again, will wait for P2)

Q5 The enter_CS() and leave_CS() functions to implement critical section of a process are realized using testandSet instruction as follows:

```
void enter_CS(x)
{
  while(testandSet(x));
}
//CS.
void leave_CS(x)
{
  x=0;
}
```

x: 0 / 1

In the above solution, x is a memory location associated with the CS and is initialized to 0. Now consider the following statements:

S1: The above solution to CS problem is deadlock free.
S2: The solution is starvation free.
S3: The process enters CS in FIFO order.
S4: More than 1 process can enter CS at the same time.

Which of the above statements are true?

(a) 1 only
(b) 1 & 2
(c) 2 & 3
(d) 4 only.

S1: ✓ P1 will enter [No Starvation]
S2: ∵ proper defined order → h/w solution (turn)
→ Here, no problem of starvation ✗ in h/w.
S3: X (no order here)
S4: X.

* SEMAPHORE

→ Semaphores used for synchronisation purpose. [Synchronization tool]
→ variable upon which we perform:
→ wait() — P() (indicated)
→ signal() — V() (implemented by Dijkstra scientist)

testing oper. (test values of S)
wait → pre..
signal → Verhoeven

* wait() operation on Semaphore V. (atomic operation)

```
wait(Semaphore S)
{
while(S<=0);
S--;
}
```

test

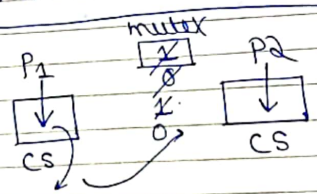
if S = +ve value, decrement by 1.

```
Signal(Semaphore S)
{
S++;
}
```

* Mutual Exclusion using Semaphores

* Semaphore mutex = 1 ;

```
do
{
wait(mutex);
//CS
signal(mutex);
//RS
}while(True);
```



- P1 calls wait(mutex)
  mutex = 0
  Now, P2 wants to enter.
  calls wait(mutex).
  ∴ enters Spinlock.

when P1 comes out, exec signal(mutex)
mutex = 1.

P2 now enters by exec wait(mutex)
(P2 enters) ; mutex = 0.

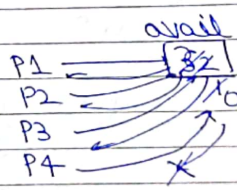
∴ Mutual exclusion ✓

* Types of Semaphores:

1. Counting Semaphore
   ↳ Value of S, is countable value.
   (no limit).
2. Binary Semaphore
   ↳ value of S = {0, 1}

* Counting S → used in resource allocation.
  ↳ avail (Proces who wants resource executes wait(avail))
  [3]
  Semaphore avail = 3

```
do
{
wait(avail);
//CS
release(avail)
//RS
}while(true)
```



⇒ {P1, P2, P3, P4}

- P1 calls wait(avail)
  avail = 2.
- Now, P2 calls wait(avail)
  avail = 1.
- P3 calls wait()
  avail = 0.

⇒ P4 calls wait(avail);
Spinlock

when P1 performs signal opern.
∴ avail = 1
Now, P4 can enter CS. (as avail that resource)

* Binary Semaphore S=0 S1, S2 → Statements

P1	CPU	P2
		wait(S);
S1;		S2; → P2 in SL.
signal(S)		

S=1

In concurrent exec: equal prob to execute S1/S2 first.
(Suppose dependency b/w S1 & S2; same apply ∴ execute S1 first then S2. [data dependency])

init S=0.

∴ only after exec S1 → S2 can be exec.

S1 → S2

if S=1 initially, fail ✗

Sa=0; Sb=0.

Q

P1	P2	P3
	wait(Sa)	wait(Sb)
S1;	S2;	S3;
signal(Sa)	signal(Sb)	

order of execution must be: S1 → S2 → S3.

⇒ Single semaphore a=0;

P1	P2	P3
	wait(a)	wait(a)
S1;	S2;	S3;
signal(a)		

Problem

S1 → S2 → S3 ✗ | S1 → S3 → S2 ✗

P1 → P2 → P3.

∴ [Use 2 Semaphore Variables]

Q

a=0

P1	P2
	wait(a)
S1;	S2;
~~signal(a)~~	signal(a)
	wait(a)
S3;	S4;

Try : S2 → ...

Page 126

Order of execution → S1 → S2 → S4 → S3

a=0, b=0, c=0 ;

P1	P2
S1	wait(a)
signal(a)	S2
wait(c)	signal(b)
S3	wait(b)
signal(c)	S4
	signal(c)

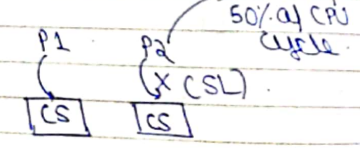
X

P1	P2
	wait(a) a=0
S1 ;	S2 ;
a=1 → signal(a)	
b=0 → wait(b)	
S3 ;	S4 ;
	signal(b) ; → b=1

∴ 2 Semaphore V [minm. of Semaphore]

Drawback :

wait(mutex) ;
// CS
signal(mutex) ;



→ when 1 process in CS, other Processes are in SL (wasting CPU cycles executing ;)

Page 127

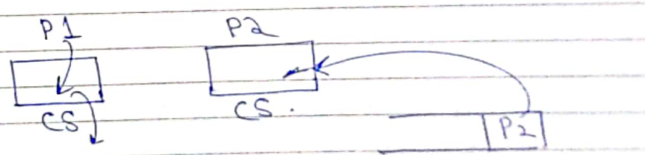
∴ Modified implementation of wait() & Signal() operation.

* Modern Implementation

Mutual exclusion can be also achieved.

```
wait(Semaphore S)
{
  S--;
  if(S<0)
  {
    Place Process in Queue;
    Block();
  }
}
Signal(Semaphore S)
{
  S++;
  if(S<=0)
  {
    Remove Process from Queue
    wakeup();
  }
}
```

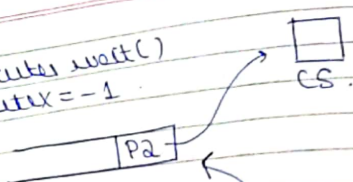
mutex = 1 → 0 → 1 → 0



P1 executes wait()
mutex=0

P2 blocked inside Queue

P2 executes wait()
mutex = -1



P1 moves out signal
mutex = 0.
∴ P2 removed from queue.

∴ No wastage of CPU cycle. (why unnecessarily exec ;)

* Deadlock

- when none of the processes can enter into CS.

Semaphore a = 1, b = 1;

CPU	
P1	P2
a=0 → P(a);	P(b); ← b=0
× P(b);	P(a); × (wait ∵ a=0)
// CS	// CS
V(a);	V(b);
V(b);	V(a);

(cont enter into CS.)

(Only 1 sequence shows deadlock → deadlock)
(No sequence → deadlock : deadlock free).

Q a = 1, b = 1;

P1	P2
a=0 ← P(a);	P(a); ×
b=1 ← P(b);	P(b); ×
// CS	// CS
V(a);	V(a);
V(b);	V(b);

(after V(a), can switch to P2 : P(a))

(P1 can enter CS)

[No deadlock]

CPU	
P1	P2
a=0 → P(a);	P(b); ← b=0
× P(a);	P(b); ×
// CS	// CS
V(a);	V(b);
V(a);	V(b);

[Obviously, causing deadlock].

eg an Semaphore.

2 students residing in same room.

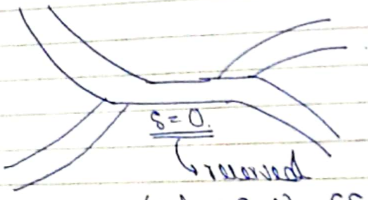
A B

turn F

start

[starvation], if both think that one will go to store & get.

[toomuch milk Prob]; both go to store at same time.



(when S=1); CS is empty
- anyone can wait(S) & enter

Q Semaphore a=1, b=1, c=1, d=1;

	P1	P2	P3
a=0	P(a);	P(b); b=0	P(a) x
x	P(b);	P(c); c=0	P(c) x
	P(c);	P(a); x	P(b)
	//CS	//CS	//CS
	V(a)	V(b)	V(a)
	V(b)	V(c)	V(c)
	V(c);	V(a)	V(b)

→ 1 combn causing DL.

∴ Deadlock.

Q Semaphore a=1, b=1, c=1, d=1;

	P1	P2	P3
	~~a=0~~		~~d~~ X
a=0	P(a);	P(b) b=0	P(a) ~~a=0~~
x	P(b);	P(c) c=0	P(d) ~~d=0~~ d=0
	P(c); x	P(d) ~~d=0~~ x	P(b) x
	//CS	//CS	//CS
	V(a)	V(b)	V(a)
	V(b)	V(c)	V(d)
	V(c)	V(d)	V(b)

[P3 → P2]

Q Semaphore a=1, b=1, c=1, d=1;

	P1	P2	P3
a=0	P(a); x	P(b); b=0	P(a)
c=0	P(c);	P(c); x	P(c) x
d=0	P(d);	P(d);	P(d)
	//CS	//CS	//CS
	V(a);	V(b)	V(a)
	V(c);	V(c)	V(c)
	V(d);	V(d)	V(d)

(Deadlock free)

[No deadlock]

1 Proc enter.

[only 1 Process can enter]

Solution to PC problem using Semaphores.

* Producer - Consumer Problem

Semaphore mutex = 1, full = 0, empty = n;

- mutex → used for mutual exc.
- full → no. of full locations inside buffer
- empty → no. of empty locations inside buffer.

Producer

```
do
{
    wait(empty);      → producer can't access when buffer is full.
    wait(mutex);      → b4 adding item
    // Adds an item
    signal(mutex);
    signal(full);     → no. of full items increase.
}while(True);
```

Consumer

```
do
{
    wait(full);       → consumer can't access when buffer is empty.
    wait(mutex);      → no. of full locations decreases.
    // takes item from buffer
    signal(mutex);
    signal(empty);
}while(True);
```


- 3 Conditions

(i) Producer isn't allowed to access buffer when buffer is full.

(ii) Consumer is not allowed to access buffer, when buffer is empty.

(iii) Both Producer & Consumer must access the buffer in a mutually exclusive manner.

*

P	C
empty = n-1	full = 0
mutex = 0	mutex = 0
(P adds item on buffer)	(Consumer item from buffer)
mutex = 1	mutex = 1
full = 1	empty = 1

n = 5

→ full = 2, empty = 3
empty = 3, full = 2
mutex = 1
mutex = 0

[Little modification in code]

Suppose, mutex = 0

wait(mutex) — wait(empty)

wait(mutex) — wait(full) ✗

full = 1, empty = n-1

Deadlock

* Reader's Writer's Problem

Condition needs to be Satisfied:

(i) Only 1 writer is allowed to perform write operation at a time.
(ii) Any no. of readers are allowed to perform read operation at a time.
(iii) When writer is performing write operation reader is not allowed and vice-versa.

⇒ Solution to RW Problem using Sephamores:

Semaphore mutex = 1, wrt = 1;
int readCount = 0;

- readCount: How many readers are performing read operation
- mutex: to achieve multiple exc. while updating readCount.
- wrt: used by Writer Process, to make only 1 writer to perform Write Operation (mex among writer Processes)

- Readers must update readCount var in a mutually exc way.
∴ Race Condition → incorrect result.

[Only 1 Process can update ReadCount at a time.]

Writer

only 1 writer can perform write operation at a time.

```
do
{
 wait(wrt);
 // write operation is performed
 Signal(wrt);
}while(True);
```

Reader

```
do
{
 wait (mutex);
 readcount++;
 if (readCount==1) then wait(wrt);
 Signal(mutex);
//Read operation is performed
 wait(mutex);
 readCount--;
if (readCount==0) then Signal(wrt);
Signal(mutex);
}while(True);
```

W1	W2
wrt=0	(cant access when wrt=0)
//write	
wrt=1	

(mutual exclusion)

R1	R2
mutex=0	mutex=0
readcount=1	readcount=2
wrt=0	mutex=1
mutex=1	//read
//read	
Signal(mutex)	
mutex=0	mutex=0
readcount=1	readcount=0
mutex=1	wrt=1
	mutex=1

→ 1st reader modifies wrt to 0.

∴ Multiple readers are allowed at same time.

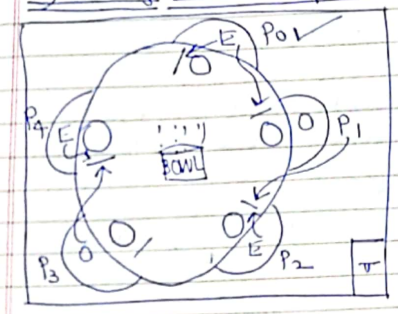
As R has no writer is allowed.

1 reader leaves now

Now, writers can enter, to update.

Q (Hiding of codes → Identify!

* Dining Philosopher's Problem



P0 P1 P2 P3 P4

spend time thinking & eating (5 Philosophers)

(need 2 chopstick to perform eating)

Semaphore chopstick[5] = {1, 1, 1, 1, 1}
Semaphore room = 4; → capacity of room.

Pi

```
do
{ wait(room);                          → take 0th chopstick
  wait(chopstick[i]);
  wait(chopstick[(i+1)%5]);            → take 1st CS
  // Eat operation
  signal(chopstick[i]);
  signal(chopstick[((i+1)%5]);
  signal(room);
} while(TRUE);
```

only for 4 Phils room>0

philosopher enters, is hungry, eats & goes.

Philos
CS → Rwaiting

* Problem: when all philosophers are hungry
(everyone will take 0 but CS)
- first wait()
- second wait() X. [Cant eat]

→ Appoint Person outside room, who will allow only 4 person in the room.

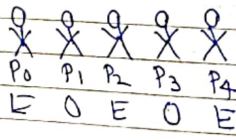
Soln 1: 4 Philos → 5 CS.
[1 Process → 2 Resources, No deadlock]

Soln 2: Perform locking of the CS, only when 2 is available

Soln 3: Asymmetric soln
(Even Philosophers, odd philosophers).

take the right CS first & then left CS.

take the left CS first & then right CS



* All these soln's are deadlock free, but not Starvation free

GATE Qs

Q.1 Suppose we want to Synchronise 2 concurrent Process P & Q using binary Semaphores S and T.
The code for the process P & Q is shown below:

Process P

```
while (1)
{
w:
    print '0';
    print '0';
x:
}
```

Process Q

```
while (1)
{
y:
    print '1';
    print '1';
z:
}
```

Synchronization Statements can be inserted only at points: w, x, y, z.

(1) Which of the following will always lead to an output starting with 001100110011

(a) P(S) at W, V(S) at X, P(T) at Y,
V(T) at Z
(S & T initially 1) → 01010011...

(b) P(S) at W, V(T) at X, P(T) at Y,
V(S) at Z
(S initially 1, T initially 0)

(c) P(S) at W, V(T) at X, P(T) at Y,
V(S) at Z
(S and T initially 1) → 01010011...

(d) P(S) at W, V(S) at X, P(T) at Y, V(T) at Z
(S initially at 1, & T at 0)
→ 000000...

(e) S=1, T=1

P(S) P(T)

V(S) V(T)

0011 X

(b) S=1, T=0 → 00110011.
(Strictly alternate Manner)

S=0
P(S)
V(T)
T=1

P(T) T=0
V(S) S=1

(one interleaving use while printing Values)

PQPQ

(2) Which of the following will ensure that the output string never contains a substring of the form: 01^n0 or 10^n1 where n is odd.

(a) P(S) at W, V(S) at X, P(T) at Y,
V(T) at Z,
(S & T initially 1) VI

(b) P(S) at W, V(T) at X, P(T) at Y,
V(S) at Z
(S & T initially 1)

(c) P(S) at W, V(S) at X, P(S) at Y,
V(S) at Z,
(S initially 1)

(d) V(S) at W, V(T) at X, P(S) at Y
P(T) at Z.
(S & T initially 1)

(a) S=0 T=0 010
P(S) P(T) 01110.

V(S) V(T).

010/ 101 → should be there.

S=0 S=1

(c) P(S) P(S)

S=1 V(S) V(S)

0011001111100...

S=1, T=1.

(d) V(S) P(S)

V(T) P(T)

if V(S); S=1. if BS = S=1

The P and V operations on counting semaphore where S is a counting semaphore are defined as follows:

P(S): S = S-1
if S<0 then wait;

V(S): S = S+1.
if S<=0, then wakeup a Process waiting on S;

Assume that P_b and V_b are the wait() & signal() operations on binary semaphores are provided.

2 binary semaphores X_b and Y_b are used to implement the semaphore operations P(S) and V(S) as follows:

$X_b = 1$

```
P(S): Pb(Xb);
      S = S - 1;
      if (S<0)
      {
        Vb(Xb);
        Pb(Yb);
      }
      else
        Vb(Xb);

V(S): Pb(Xb);
      S = S + 1;
      if (S<=0) Vb(Yb);
      Vb(Xb);
```

Binary Semaphore

wait (Y_b=0)

wakeup a proc when Proc. is sleeping (Y_b=0)

The initial values of X_b and Y_b are respectively:

(a) 0 and 0
(b) 0 and 1
(c) 1 and 0
(d) 1 and 1.

(a) $0 - X_b$ $X_b = 0$.

$X_b = 1$ → S = S-1

comp. P(S) with P(S).

Q.4 At a particular time of computation the value of the counting semaphore is 7. Then 20 P() operations and 15 V() operations were completed on this semaphore.
The resulting value of Semaphore is 2

P() wait()
V() Signal()

Soln S = 7

20 P() & 15 V() =

7 - 20 = -13

7 + 15 = 22

= 2

Q.5 A counting S was initialized to 10. Then 6 P() operations and 4 V() operations were completed on this semaphore.
The resulting value of S = 8

Q.6 Each Process P_i, i = 1, 2, 3 ... 9; is coded as follows

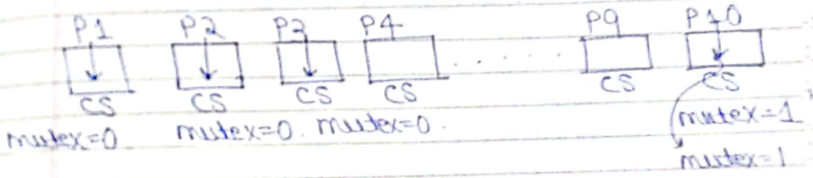
```
Repeat
  P(Mutex);
  {Critical Section}
  V(Mutex);
Forever
```


The code for P_{10} is identical except that it uses V(Mutex) in place of P(Mutex). What is the largest no. of process that can be inside the CS at any Moment?

(a) 1
(b) 2
(c) 3
(d) None of the above

• Mutex = 1 (initially)

```
P10
Repeat
  V(Mutex)
  {CS}
  V(Mutex);
Forever
```



(ALL Processes)

[P10 helps all processes to enter into CS]

if i=1,

Q.7. * Fetch and add of x, i is atomic read modify write instruction that reads the value of memory location x, increments it by the value of i and returns the old value of x. It is used in the pseudo code shown below to implement a busy wait lock.

L is an unsigned Integer shared variable initialized to 0. The value of 0 corresponds to lock being available while any non-zero value corresponds to the lock being not available.

init L=0

```
Acquire_lock(L)
{
while(Fetch_And_Add(L,1))
    L=1;
}
//CS
Release_lock(L)
{
L=0;
}
```

This implementation:

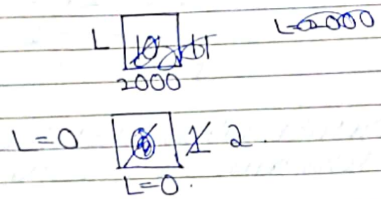
(a) fails as L can overflow.

(b) fails as L can take on a non-zero value when the lock is actually available.

(c) works correctly but may starve some process.

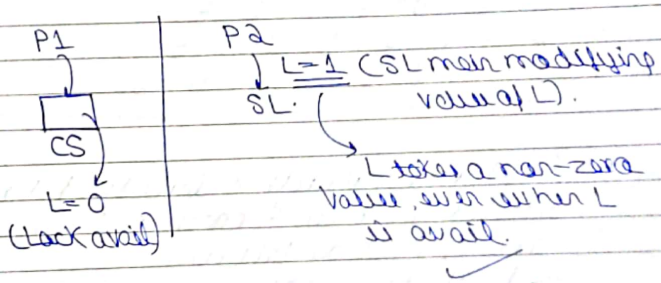
(d) works correctly without Starvation.

Sol^n:



L=1.

⇒ fetch and add → increm by 1; returning old value.



Q8. The following 2 functions P_1 and P_2 that share a variable B with an initial value of 2 execute concurrently. B=2

```
P1()                P2()
{                   {
  C = B - 1;          D = 2 * B;
  B = 2 * C;          B = D - 1;
}                   }
```

The no. of Distinct Values that B can possibly take after the execution is 3.

1. C=1, B=2, D=4, B=3
2. D=4, B=3, C=2, B=4
3. C=1, D=4, B=3, B=2
4. C=1, D=4, B=2, B=3
5. D=4, C=1, B=2, B=3
6. D=4, C=1, B=3, B=2 ✓

Q9. IIT-B A Shared Variable X initialised to 0 is operated on by 4 concurrent Process W, X, Y, Z as follows.
Each of the process W and X reads X from memory, increments by 1 and stores it to memory & then terminates.

Each of the process Y and Z reads X from memory, decrements by 2, stores it to memory & then terminates.

Each Process before reading X invokes the P() operation on a counting semaphore S and invokes the V() operation on the semaphore S, after storing X to memory.

Semaphore S is initialised to 2. What is the max. possible value of X after all Process complete execution.?

Sol:

S = 2 ; X = 0.

W	X	Y	Z
P(S)	P(S)	P(S)	P(S)
	S=1	S=0	
X = X + 1	X = X + 1	X = X - 2	X = X - 2
V(S)	V(S)	V(S)	V(S)
	S=2	X=1	X = 1 + 1 = 2
X=2	X=2		
S=2			

(make inc ops overlap dec ops)

→ Dining Philosophers

10. Let m[0] ..... m[4] be mutexes and p[0] ..... p[4] be processes. Suppose each process p[i] executes the following:

```
P[i]
    wait(m[i]);
    wait(m[(i+1) mod 4])
        // CS
    release(m[i]);
    release(m[(i+1) mod 4]);
```

This could cause:

(a) Thrashing.
(b) Deadlock.
(c) Starvation but not deadlock.
(d) None of the above.

```
P[0].
    wait(m[0]).
```

→ every one having 1 CS & waiting for another.

⇒ Deadlock.

Test Paper.

9. Priority & WT. ⟶ FCFS
   TQ = 1
   ⇒ Preemptive FCFS / RR. ✓

12. P1 P2 P3 P1. (2 pairs)
    [leading 1 P. twice].

14. Y > X > 0.
    P(running) > P(ready) > P(new).
    technically non-Preemptive.
    FCFS.

11.

20/7/19 Saturday

Lecture 5

* Disk Scheduling
* Protection & Security
* Unix System calls & related OS [Commands not there] } Not in Gate.
* Spooling.

* DEADLOCKS

⇒ System Model

1) Request → Any process who wants to use the resource, may it $P_1 \xrightarrow{req} R_1$ allocated.
2) Use : P_1 uses it,
3) Release then release R_1

⇒ Types of Resources

1. Consumable
2. Reusable

Wrt to Sharability.
- → Sharable (files share)
- → Non-Sharable resources. (Printer).

resource will be consumed by the process. [like msgs, inputs, signals].

resource can be used again & again. [printer → must buy agn].

$P_1 \leftrightarrows R_1$ → started Waiting.

$P_2 \leftrightarrows R_2$

→ Deadlock.

P_2 waiting for resource, holded with P_2.

* Real time examples :

eg1] $C_1 \rightarrow$ $\leftarrow C_2$ $\rightarrow C_1$ $C_2 \leftarrow$ — Use Semaphore, to resolve Jam.

Bridge.

None of the vehicles can proceed in any direction

eg2] A, B — At one point: deadlock.

eg3] A → B, D ← C

[Sharable res —

* Proper Synchronisation can avoid deadlock.

* Deadlock Characterisations

⇒ conditions for the occurrence of deadlock:

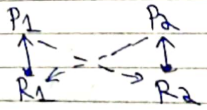
1. Mutual Exclusion
2. Hold & Wait
3. No preemption
4. Circular Wait

} [Necessary conditions]

Sufficient condition

. If CW ⟶ Deadlock.

1) Mutual Exclusion → At any time, only 1 P can operate a resource. (hold) ∵ Non-Sharable.



Here, ✓

2) Hold & wait

P1 holding R1 & waiting for R2.

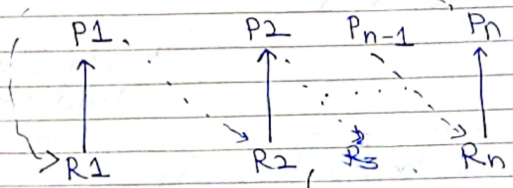
(Hold & W → ✓)

3) No-Preemption → No resource can forcebly preempted from process holding it.

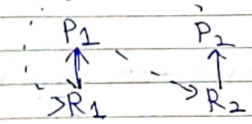
(P1 can't snatch R2 from P2)

– If preemption ⟶ No DL.

4) Circular Wait

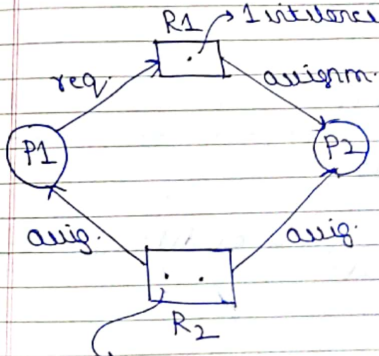


⇒ Circular Wait → sufficient condn.



* Resource Allocation Graph (RAG)

[R_1: printer]



G = (V, E)

Vertex (P & R)

edges.

→ request edge

→ assignm. edge

request assigned to resource.

P_2 holding both R_1 & R_2 (1 instance each).

* RAG: Purpose.

- After drawing RAG, if this Graph forming cycle → possibility of DL (may/ may not)
- If no cycle → No DL.

→ use cycle detection algorithm.

* Deadlock Handling Methods

- deadlock prevention/avoidance/detection/recovery
Ostrich.

* ~~Bankers~~ algorithm → "we can prevent, there is no deadlock".

[Ostrich keeps head inside Sand, pretending there is no Storm].

Ostrich Algorithm

(1) Deadlock Prevention:

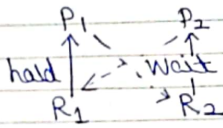
If we prevent one of the conditions of DL, DL can be prevented.

(i) Mutual Exclusion
(ii) Hold & Wait.
(iii) No preemption.
(iv) Circular Wait.

[can't Prevent]

(i) ME must hold for non-Sharable resources.

(iii)



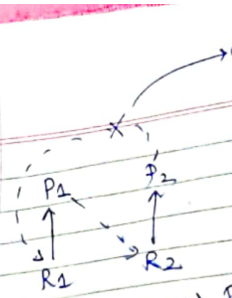
- 1 Protocol: allocated R_1, R_2 prior before start of execution.
[this with wait occur]
Here, P1 execut. R1, req R2.

- 2 Protocol: Process can request another resource, only when it has none.

(iii) No-preemption

- No resource can be forcibly preempted from P_2.
- if P1 req R_2,
R_2 is not alloc. imm to P1.
P1 needs to release all resources.

(iv) Circular Wait prevention



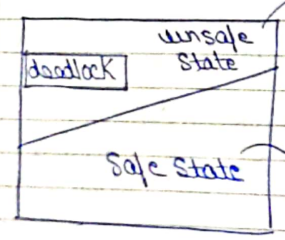
- Pi should request R (in increasing order of enumeration).

$P_1 \rightarrow R_2$ $\quad P_3 \rightarrow R_4$
$P_2 \rightarrow R_3$ $\quad \vdots$
$\vdots$
$P_n \rightarrow P_{n+1}$

② Deadlock Avoidance

1. Safe State Method.
2. Resource Allocation Graph Method.
3. Banker's Algorithm.

1. Safe State Method



→ Deadlock occurs only in unsafe State.
- DL won't occur in Safe S.

→ sequence, where process can terminate safely [Safe State Sequence]

[Max req = 6
All. req = 2
Need req = 4]

	MAX R	Alloc	Available
P1	6	2	3
P2	5	1	
P3	3	1	

AV = ~~3~~ ~~4~~ 6

Compute need? [max − Alloc]

P1	4
P2	4
P3	2

∵ AV = 3.

<P3, P1, P3>
or
<P3, P2, P1>
Safe State Seq.

∵ P_3 completed, All = 1, ∴ resource & added to avail

→ First Process to be finished: P3.

Q Find the Safe State Sequence

	MAX	Alloc.
P1	10	5
P2	4	2
P3	6	3

Total 12.

Avail = ~~2~~ ~~4~~ ~~7~~ 12
2+2 = 4+3 = 7+5 = 12

	Need
~~P1~~	5
~~P2~~	2
~~P3~~	3

<P2, P3, P1>
↳ Safe State Seq. (unique).

→ if total = 11, avail = 1 ∴ DL.

Q1

	MAX	ALLC
P1	6	2
P2	4	2
P3	7	4

Avail = 2

	Need
~~P1~~	4
~~P2~~	2
~~P3~~	3

~~2 4 6~~ 10

{P2, P1, P3}
{P2, P3, P1}
2 Safe State Seq.

Q2

	MAX	All
P1	6	1
P2	4	0
P3	7	2

Avail = x

To bring in Safe State, min Val = x.

P1	5
P2	4
P3	5

x.

∴ if x = 4, 0+4 = 4 can finish P1 & P3.

x = 5 + 1 = 6 + 0 = 6 + 2 = 8

<P1, P2, P3> ✓

(3! orders) ✓

Q

	MAX	Alloc.	Avail
P1	9	1	x = ?
P2	5	1	
P3	7	2	

Total = 9

Need

P1	8
P2	4
P3	5

Min = ~~4 5~~ 7
<P2, P3, P1>
1 rem.

Min = ~~5 7~~ 8
<P3, P2, P1>.

∴ x = 5 ✓

[Multiple resource]

3. * Bankers Algorithm → makes use of:
⇒ Safe state Algorithm

MAX → Matrix [R1, R2 ..... Rm]
res.
[P1, P2 ..... Pn] (n & m: diff)
Process

⇒ MAX[i][j] = K → matrix
↳ Pi requests max of K no. of resources of type Rj.

⇒
MAX
R1 R2 ... Rm
P1
P2
.
.
Pn

⇒ Alloc[i][j] = t → matrix.
↳ for Pi, already t instances of resource type Rj is allocated.

→
Alloc
R1 R2 R3 ... Rm
P1
P2
.
Pn

⇒ Available → Vector.
↳ R1 R2 R3 ..... Rm

Available[j] = p.
↳ For resource type Rj, p instances are currently available.

⇒ Total
↳ R1, R2 R3 ..... Rm

Total[j] = b.
↳ for resource type Rj, b instances are totally present.

⇒ need[i][j] = MAX[i][j] - Alloc[i][j]

S-1: Total resources : R1, R2 ... Rm
Total Process : P1, P2 ..... Pn.

[n Processes, m resources]

init: [finish[i] = FALSE ∀ process from i=1 to n]
↳ no process completed right now

(283 in copy.

→ copy into Work V & update Work V.

Work = Available.

S-2] Need (Max − Alloc n).

	R1	R2	R3	...	Rm
P1					
P2					
P3					
⋮					
Pn					

(i) Identify Pi based on 2 condⁿs:

- FINISH[i] = False. ← Avail.
- Need[i] ≤ Work [Avail]

[if no process is found, go to S-4]

S-3] · resources need to be reallocated & added to work[] (after Process finishes].

- Work = Work + Allocation; (like in SS)
- Finish[i] = TRUE;

S-4] Finish[i] = TRUE, for each & every Process, then System is in Safe State.

↳ all Processes are completed.

221

Q

	MAX			ALLOC		
	R1	R2	R3	R1	R2	R3
P1	6	4	3	3	2	2
P2	7	1	2	5	0	1
P3	5	3	2	3	3	1

Avail

R1	R2	R3
2	2	2

Find the Safe State Sequence.

- Work = R1 R2 R3
  2 2 2
  ~~5~~ 7 ~~3~~
  ~~10~~ 5 4 — 13 7 6.
- Need (computed).

	R1	R2	R3
P1	3	2	1
P2	2	1	1
P3	2	0	1

<P2, P3, P1>.
<P2, P1, P3>.

↳ Safe State Seq.

(it can start with P3 also.

* This application is seen in Banks.

Q. A Computer System use the Banker's Algorithm to deal with deadlocks. Its current state is shown in the table below where, P_0, P_1, P_2 are process; R_0, R_1, R_2 are resource types;

	MAX			Alloc			Avail		
	R_0	R_1	R_2	R_0	R_1	R_2	R_0	R_1	R_2
P_0	4	1	2	1	0	2	x	x	0
P_1	1	5	1	0	3	1	2	5	1
P_2	1	2	3	1	0	2			

(4, 5, 5) → 3, 5, 3?

Find the safe State Seq?

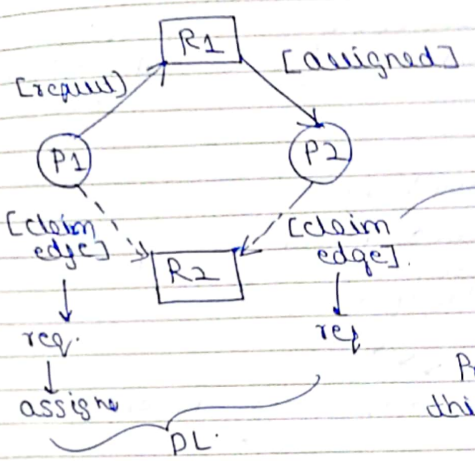
⇒ Need → Max − Alloc

	R_0	R_1	R_2
P_0	3	1	0
P_1	1	2	0
P_2	0	2	1

Safe State Seq.

$\langle P_1, P_2, P_0 \rangle$.

2. Resource Allocation Graph Method



- B4 allocating R2 to P1, it needs to check all the claim edges, ∴ (P_2 — R_2). (is it causes DL in future / not)

③ Deadlock Detection

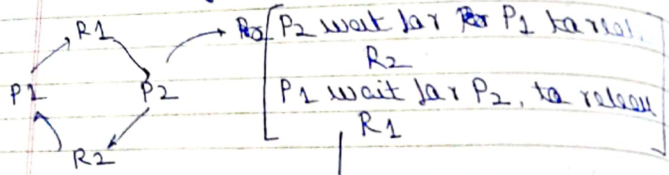
(i) Single Instance (of each resource Type)

(ii) Multiple instances of each resource type.

↓

Construct wait-for-Graph.

- in WFG, we only have Process (dependency is shown)



↓

WFG

If cycle: Deadlock

- if no cycle: No Deadlock

(ii) → Deadlock Detection Algorithm

(quite Similar to Bankers Algo).

- 4 Step: Finish[i] = False, for only 1 Process (at least)

eg:

	MAX			Alloc			Avail		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	3	3	1	1	1	0	2	2
P2	2	2	2	1	0	0			
P3	4	4	4	0	0	1			

[No process can be completed]

→ Deadlock.

④ Deadlock Recovery

1. Process Termination
2. Resource Preemption

1. Process Termination

→ 1 process will be terminated at a time.

→ Terminate all process in deadlock.

→ terminate process by process, unless DL recovers

Q Which Process to be selected for termination?

SC: · How much % of the Process is completed?
· Priority/Type of the Process?

2. Resource Preemption → Snatch resources from Process holding it.
   - Selecting a Victim (from which resources are preempted)
   - Rollback (Taking all resources from Process).
     - when we take out resources & it is repeatedly happening on same Process
       → lead to Starvation.
       How to overcome?
       * cost: No of rollBacks performed on the Process.
       (Process with the minimum cost is selected).

* GATE Qs

Q1. Which of the following is not true? of deadlock prevention & deadlock avoidance Schemes?

(a) In Deadlock Prevention, the request of resource is always Granted, if the resulting State is safe.

(b) In DL avoidance, the request for resources is always Granted, if the resulting State is safe.

(c) DL avoidance is less restrictive than DL Prevention

(d) DL avoidance requires Knowledge of resource requirements apriori.

(a) In DL P, we dont know what is safe State & unsafe State.

(c) DL Prevention → operⁿ at root level (So more restrictive)

Q.2 In A Single Processor System, has 3 resource types X, Y and Z which are shared by 3 processes. There are 5 units of each resource type (total). Consider the following scenario where the column: alloc denotes the no. of units of each resource type allocated to each process & the column request denotes the no. of units of each resource type requested by a Process in order to complete execution.

Which of these Processes will finish last?

req c need

	alloc			req			
	X	Y	Z	X	Y	Z	need
P_0	1	2	1	1	0	3	
P_1	2	0	1	0	1	2	
P_2	2	2	1	1	2	0.	

	need	total
P_0	2	[5 5 5]
P_1	0	
P_2		

avail: [0 1 2]

2 1 3

3 3 4

5 5 5.

$\langle P_1, P_0, P_2 \rangle$

safe state

Q3 Which of the following is not a valid deadlock prevention scheme.

(a) release all resources before requesting a new resource.

(b) No. the resources uniquely & never request a lower no. resource than the last one requested.

(c) Never request a resource after releasing any resource.

(d) request & be allocate all required resources before execution.

Q4 An OS contains 3 user Processes, each requiring 2 units of R.
The minimum no. of units of R such that no deadlocks will ever arise is:

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

	R_1	R_2	R_3
need			
P_1	2	2	2
P_2	2	2	2
P_3	2	2	2

	P_1	P_2	P_3
	2	2	2

Soln:

R=3

P1 1

P2 1+1=2. if R=4

P3 1

(atleast 1P will get 2 resource)

Q5. A computer system has 6 tape Drives with n Processes competing for them. Each Process may need 3 tape drives.
The max value of 'n' for which the system is guaranteed to be deadlock free is:

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

P1	2 0
P2	2 3
P3	2 3

6. → No DL.

P1	2
P2	2
P3	2

→ possib of DL.

P1	3
P2	3

→ No Deadlock [in any case].

Q.6 Two Shared resources R_1 and R_2 are used by process P_1 and P_2. Each process has a certain priority for accessing each resource.
Let T_{ij} denote the priority of P_i for accessing R_j.
The P_i can snatch a resource R_K from Process P_j if T_{ik} is greater than T_{jk}.

P_i — steal → P_j : R_K

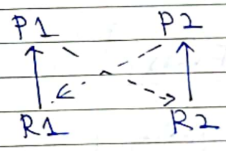
Given the following:

1. $T_{11} > T_{21}$
2. $T_{12} > T_{22}$
3. $T_{11} < T_{21}$
4. $T_{12} < T_{22}$.

Which of the following conditions ensure that P_1 and P_2 can never Deadlock?

(a) 1 & 4
(b) 2 & 3
(c) 1 & 2
(d) None. ✓

1. ✓
2. ✓
3. ✗



✓

(a) 1 and 4 ($T_{11} > T_{21}$; $T_{12} < T_{22}$.)
(both Snatching from each other)

(c) 1 & 2 — P_1 will snatch R_2
P_1 will keep hold of R_1

7. THREADS → Mult. threads in a Proc.

→ 3-4 Qs.

- Light Weight Process (LWP)
- Process contains several threads.

eg: WA → App Process.
- (Thread shows suggestions while typing).
- (loading img is another thread).
- (1 thread: accepts i/p).

* Thread contains:

(i) thread id
(ii) Prog Counter
(iii) register Set.
(iv) Stack

- Every thread: own register | Stack.

(Shares code, data & files)

→ multi-Threading.

- used in Multi-Processor architecture.

CPU_1 — T_1, CPU_2 — T_2.

- resource Sharing.

→ resource can be shared by multiple processes threads.

(i) Open an Thread > faster open an Process.

(ii) T. Switching
T. Creation
T. Termination > Fast } Process S C T.

(iii) Threads belong to same Proc (1 T can assist another T) — But, process can be independent, doesn't communicate w each other

---

- 1 T can run, other T can be blocked.
- State diagram of Proc, similar to T.

→ But, no Suspended State in T. [Process Level Concept]

∴ Suspended: P1 in SM. (all corr threads also in SM).

Lecture 6

* Types of Thread: (Vimp)

1) ULT

- User level Thread.
- Kernel doesn't have any info abt ULT [no idea, how many ULT in a Process]
- ULT
→ Advant: faster, efficient, simple representation, simple managment & cheaper; much simple.
- ∵ everything is not reported to Kernel: Fast.
- implement is cheaper
→ Disadvn: ∵ Kernel don't have idea diff (PTO)
- Non-Blocking System call must be implemen.
- (Blocking 1 ULT, will block entire Process).

2) KLT

- Kernel level Thread
- Kernel has complete info abt KLT.

→ Advant: ∵ Kernel has complete understanding & no. of KLT in each Process allocate more no of TQ to more threaded Process [Dis → ad (KLT)]
- No need for Non-Block System Call. (Blocking of 1 Thread will block only that thread, not entire Process).

→ Disadv: They are slow & inefficient as everything needs to update to Kernel. [More Kernel intervention]
- It will maintain a thread control block (additional burden)

imp

Divide: P1 (1 ULT), P2 (1000 ULT) → deserve more amt of TQ.

ULT

- Kernel allocate equal TQ to both P1 & P2.
∵ Kernel don't have any idea abt ULT.

GATE QS

Q.1 Consider the following statements about user level Threads & Kernel level threads. Which one of the following is false?

(a) Context Switch time is longer for KLT than ULT.
(b) ULT do not need any hardware support.
(c) Related KLT can be scheduled on different Processors in a multi-Processing System.
(d) Blocking 1 KLT blocks all related threads.

. Blocking 1 KLT blocks only that part-thread.

(a) true, ∵ KLT → slower & inefficient.

Q.2 Consider the following Statements with respect to User level Threads & KLT.

S-1] Context Switch is faster with KLT. F

S-2] For ULT, a system call can block the entire Process. T

S-3] KLT can be Scheduled independently. T

S-4] ULT are transparent to the Kernel. F.

Which of the above are true?

(a) 2, 3 & 4 only.
(b) 2 and 3 only.
(c) 1 and 3 only.
(d) 1 and 2 only.

⇒ S-1: CS is faster with ULT (F)
S-2: It needs to implement NBSC (T)
S-3: Kernel has complete info abt KLT.
∴ It will Schedule independently the TQ. (T) ✓
S-4: Kernel doesn't have any info (F) ✓

Q3 Which of the following is False?

(a) ULT are not Scheduled by the Kernel.
(b) Context Switching between ULT is faster than Context Switching b/w KLT.
(c) When a ULT is blocked, all other threads of its Process are blocked.
(d) KLT can't utilise multi-Process System

by splitting threads on different Processes as cases.

⇒ Main advantage of Thread: multi-Process Architecture (each thread given to each CPU/core)

for ULT → implemented by user
↳ can't split threads & assign to diff CPUs.

KLT → Yes.

---

8. MEMORY MANAGEMENT

→ William Stallings.

1. Partitioning (SP & DP)
2. Replacement Algorithms
3. Paging
4. Page Table Structures
5. Segmentation
6. Virtual Memory Mangement
7. Demand Paging
8. Page replacement algorithms.
9. Thrashing?

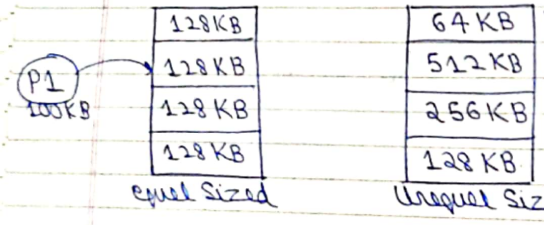
Concepts.

1. Partitioning Techniques

- 2 types of Partitioning:

(i) Fixed/Static Partitioning.

each P. diff Size.



- Partition already created in MM.
- Init: Partition is empty.
  - Process is brought to one of this P, for execution purpose. (new → ready)

(Due to this, it causes internal Fragmentation)

- P1 — newly created (Size = 100 KB)
  - wastage of memory Space: 28 KB.

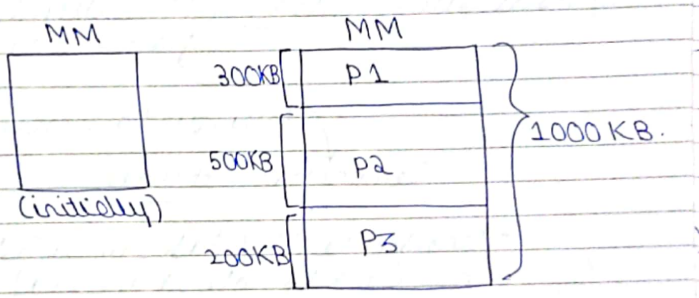
(Heap Memory)

(ii) Dynamic Partitioning

not used further.

but if 28 KB Hole, 28 KB Process arrives utilize it.

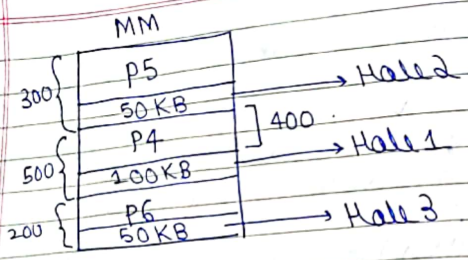
- init, there are no Partitions.
- Partition created due to placement of Process



new {
P1: 300KB (placing P1 in MM → 2 Partitions).
P2: 500KB (Placing P2 in MM → 3 Partition).
P3: 200KB
}

↓ All Process now completed.

- Partition created by Process, remain Same. {300, 500, 200}

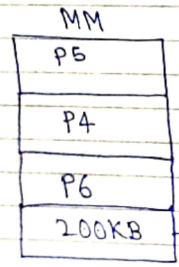


- P4: 400KB. ← Newly arrived Process
- P5: 250 KB
- P6 : 150KB

⇒ Holes getting created in MM: External fragmentation

Technique to overcome: Compaction Technique

- According to CT, (Adding up all holes & making a big hole at the end).

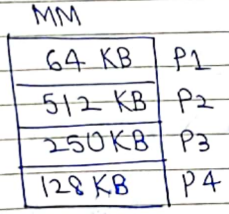


Diag

if P7 now arrives, cant be kept in MM (before CT), since not contiguous
∴ after CT ⟶ can bring P7 into hole.

- Given entire MM of 1000KB.
  Process size = 2000KB.
  ↳ more use of Paging / VM.
  - use additionally manually.

2. PLACEMENT ALGORITHMS



1 Best Fit

- I] P1 (S = 100KB) arrives.
- P1 has to be placed in best P, where, (Process Size closer to Partition Size) <=

→ placed in P4 (128 KB) [Partition 4]

2. Worst Fit

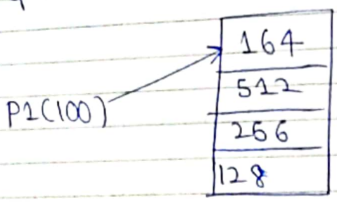
→ 512 KB.

- Process placed in Partn where PS & Part Size

is far from each other.

3. First Fit:

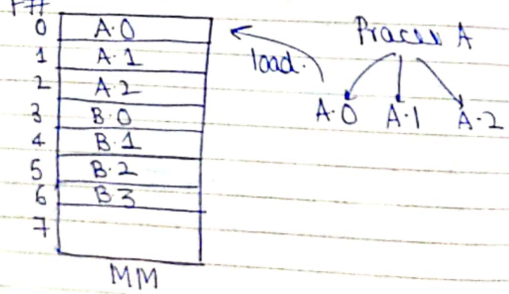
Process will be placed in the first available free block.



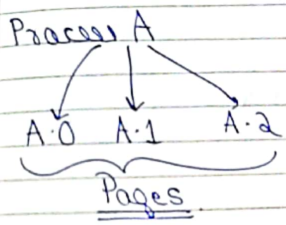
3. PAGING ⟶ Vimp.

⇒ Simple Paging Technique:

(i) MM - divided into equal size chunks called frames.



(ii) Process will be divided into equal size chunks called Pages.



(iii) For execution, pages must be loaded into frames of MM.

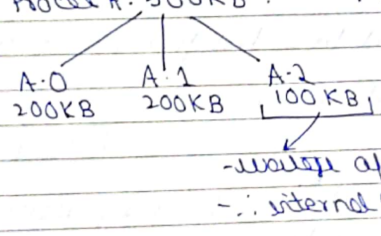
(iv) Page Size = Frame Size.

If FS = 200KB.

Process divided into pages consider Frame Size.

(v) Still can cause Internal Fragmentation.

eg: Process A: 500KB. FS = 200KB



- wastage of mem space = 100.
- ∴ internal Fragm.

(vi) Page Table will be used to store page corresponding frame information.

• Page Table of A

Pg#	F#
0	0
1	1
2	2

F# → page no.

Another Process B → B·0, B·1, B·2, B·3 } Pgs loaded into frames of MM.

• Page Table of B

Pg#	F#
0	3
1	4
2	5
3	6

Page corresponding frame information

* There will be a free frame list also.

[List of free Frames]

7
8

look into FFL & place processes into these Frames.

* ADDRESS TRANSLATION → converting LA → PA

- AT is performed by Memory Management Unit (MMU).

• CPU generated Address : Logical Address
• MM generated Address : Physical Address (refers MM with PA)

CPU → Logical Address | pg# | offset

Physical Add | FR# | offset → MM.

• offset : within a particular Page, how much displacement you can access.

eg: pg# = 3 bit ∴ 8 pgs

offset = 13 → within each page, you can access 213

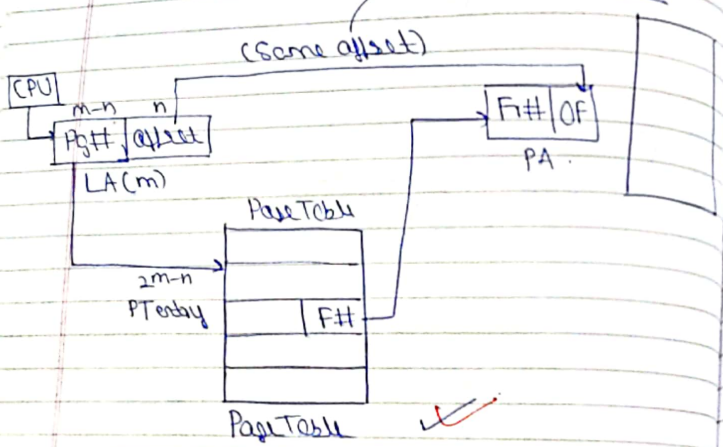
(13 bits need to access each content in a pg).

Size of pg = 213 ✓

AT → convert LA → PA
(carried by MMU)

[PageTable – DS – Pg#, F#]

∴ PgSize = FrSize



- Valid/Invalid bit. If V bit, Page is available in MM.
- Page corr. Frame info, can be identified with same offset, by looking at Page Table.
- F# obtained from PageTable.

eg: m bit LA → OF = n bit
→ Pg# = m-n bits

∴ No of entries in PageTable : 2^{m-n}

[Size of PT = Size of each entry * total entries]
→ Data Structure

Q1 Consider 32 bit logical Address and 4KB Page Size. Compute no. of Page Table entries.

Soln $\log(2^2 \times 2^{10})$

LA → 32 bit.

Pg #	offset
20 bit.	12 bit.

Pg Size = 4 KB.

2K = 4KB.

K = no. of bits (offset).

2^{12}.

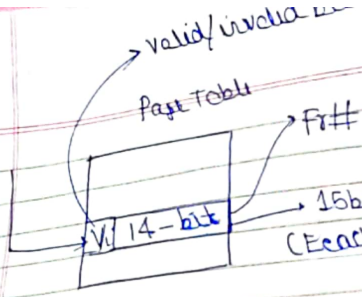
[No of Page Table Entries $= 2^{20}$ ⇒ 1 million]

Q2 Consider 26 bit Physical Address and find Size of Page Table.

Size of Pg Table = no. of PT entries × Size of each entry.

PA = 26 bit, offset = 12 bit

[Fr# = 14 bit]



(Each entry = 2 Bytes)

∴ Size of Page Table = $2^{20} \times 2$

$\Rightarrow 2^{21}$ B

$\Rightarrow$ 2MB ✓

* Size of each frame = 2^{12} (offset)

$\Rightarrow$ 4KB

* How many frames? → 214 frames in MM

* Translation LookAside Buffer (TLB)

- It is a h/w lookup Cache.
- It has limited no. of entries, Size = Small

(Generally = 1024 entries)

• Page corresponding frame info of those recently accessed Pages.

* Page table also stores same content, but Page Table contains only 1 Process, Can't page frame info.

Page Table → per Process

TLB → Whole System

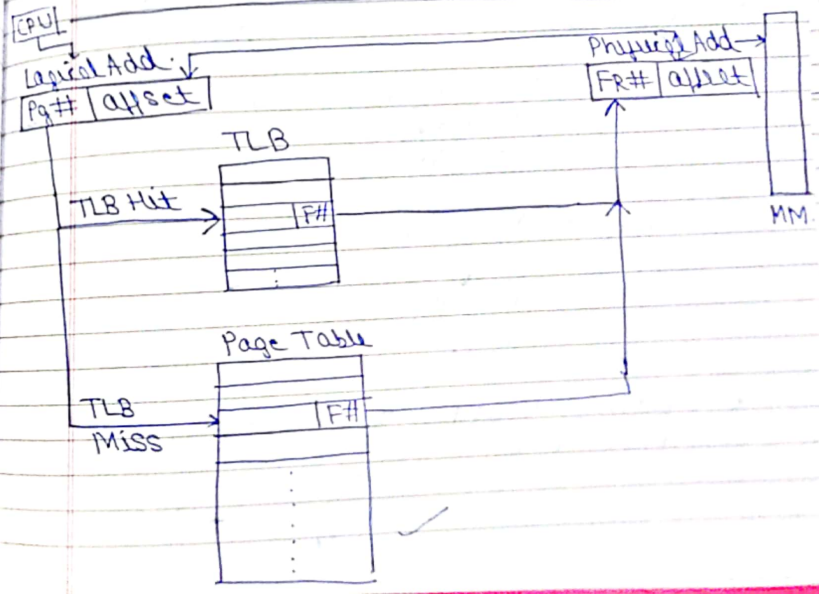
Store recently accessed page-frame info of different process.

∵ CPU Switch from Process 1 Process.

eg: Dmart → limited (few entry of each) TLB.
H&M → more (PT).

* first Search TLB, if not there, go to Page Table

easy to access.
[H/W → recent]



TLB: Page info is found in hit
(TLB Hit ratio(h)) → out of 10 references, 8 times found.

when Page corresponding Frame info. is not in Page Table also. → Page Fault (we need to replace)

* Page Table → stored in MM. 1)
reference: Memory Access 2)

* Effective Memory Access Time
$= h * (t+m) + (1-h) * (t+m+m)$

TLB Hit Ratio — TLB Access time — came to PT, only when we realise it's not in TLB

(m → time taken to access memory)
(h → TLB hit ratio)
(t → TLB access time)

$(1-h) * (t+m+m)$.

accessing PgTable in MM.

After simp.

* Effective Memory Access time $= t + m(2-h)$

Q.1 Consider a System whose TLB access time is 10nsec and memory access time is 70nsec. Compute effective memory access time.

(i) TLB hit ratio = 80%.

(ii) TLB hit ratio = 95%.

Sol^n

= EMA = $10 + 70(2-0.8) = 94$ nsec.

EMA = $10 + 70(2-0.95) = 83.5$ nsec.

$HR \propto \frac{1}{EMA}$

* In case of Paging, Process can have Shared Pages.

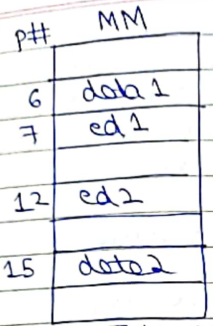
Process A

0	ed 1
1	ed 2
2	data 1

Process B

0	ed 1
1	ed 2
2	data 2

Shared Pages



Page Table (A)

0	7
1	12
2	6

Page Table (B)

0	7
1	12
2	15

shared pages (7, 12)

Frame

- ~~Same~~ Page can be shared from same frame number.

4) PAGE TABLE STRUCTURES

1] Multilevel Page Table Structure / Hierarchical Paging.

Why? necessity?

Given: 4 KB Page Size.

32 bit Logical Add:

Pg#	offset (d)
20	12 bit

* [No. of Page Table Entries $= 2^{20}$] $\approx$ 1 million.

→ If Size of each entry: 4B

PT Size = 4MB (Process corresp PT)

allocating this much Space contiguously in MM = difficult.

∴ Multilevel Page Structure.

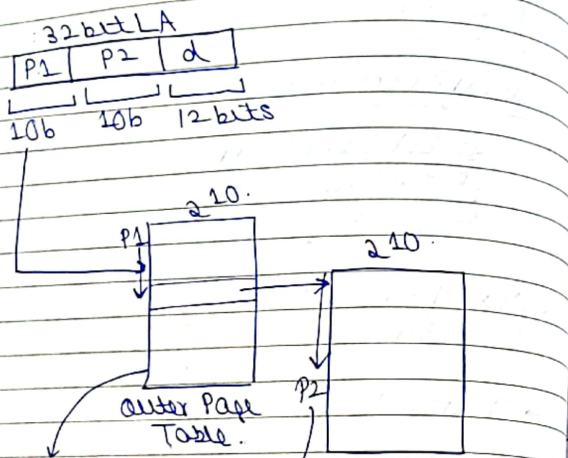
⇒ Instruction takes 1 μsec to access memory
Page fault Service time = j μsec

∴ [EMA = $1 + \frac{1}{K}(j)$]

K = Page fault occurs in every K instrn.

Page table is

* 2-level Page Table Structure



∴ Contiguous memory loc^n not req.

* Init have outer Page Table (2^{10} entries).

How P_1 & P_2 is divided → ~~10~~ 9 11
12 8 :

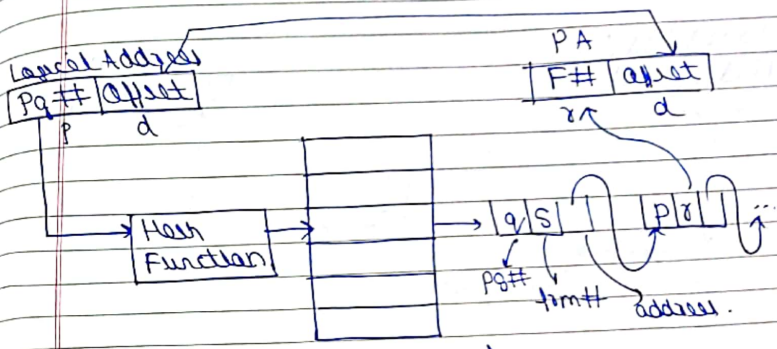
∵ 2 levels.

⇒ From Architecture to Archit. no of levels changes.

depends upon bits.

Inv Page Table: No. of entries = No. of frames in which MM is divided.

2] Hash Structured Page Table



Hash Structured Page Table.

- Pg no. is hashed & ~~stored~~ search in HSPT.
- (Hash collision may arise) [P & q produce same index value after hashing].

comes in Linked list.

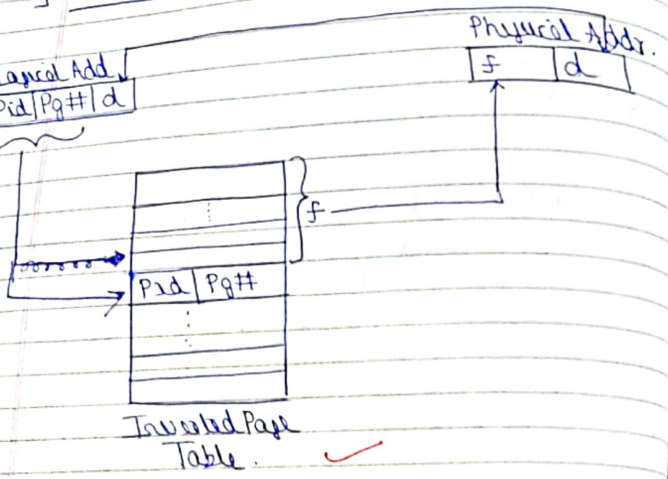
like 5 % 3 = 8 % 3 = 2 5 → 8

(placed in list, ∵ cant overwrite).

→ Page p & q got same Key Value.

→ Reverse Process

3] Inverted Page Table



→ IPT has different implementation
→ for large address Space used.

[PgT → Pg corresponding Frame info] (earlier)

- No. of entries in IPT = No of frames in which memory is provided

in which frame, which pg is stored.
index

- use need to track Pid (P_A, P_B, P_C).

(pid, Pg#) is searched in IPT.

found after f no of frames.

f → referred.

[No of entries in IPT = no. of frames]

GATE QS

Q1 A m/c has 48 bit Virtual Address and 32 bit Physical Addresses. Pages are 8 KB. How many entries are needed for the Page Table.

LA = 48 bit.

PH	d
35	13 bit

$2^3 \times 2^{10} = 13$

$\Rightarrow 2^{35}$ PT entries.

Q2 A computer with a 32 bit Address uses a 2 level Page Table. Virtual Addresses are split into 9 bit Top level Page Table field, 11 bit Second level Page Table Field & an offset. How pgs large are the pages & how many are there in the address space.

Soln.

offset = 12 bit = 2 12B $2^{10} \cdot 2^2$

P1	P2	d
9	11	12

$2^9 \times 2^{11} = 2^{20}$

32 bit

p1	p2	d
9	11	12

(i) how large is the page = $2^{12} = 4KB$ ✓

(ii) how many page = 2^{20} (in Add Space) ✓

Q3 In a System that Implements Paging, the processor uses 34 bit Virtual addresses, 40 bit Physical Addresses and a page Size of 8 KB. How many bits are needed to represent the Physical frame?

LA (34 bit)	
p	d
21	13

40 (PA)	
f	d
27	13

Pg Size = 8KB = 2^{13}

Total frames ⟶ 2^{27} ✓

Q4 Consider a computer System with 32 bit logical address & 4KB Page Size. The System supports upto 512 MB of Physical Memory.

(i) How many entries are there in a conventional single level Page Table?

(ii) How many entries are there in an inverted PT.

(i) 2^{20} ✓

(ii) 29×2^{20} $\quad 29 - 12 = 17$

2^{17} ✓

(i) 2^{20}

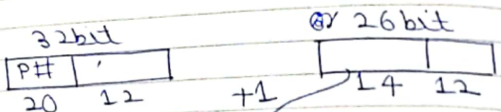
(ii) Size of PM = 512 MB = 2^{29} ✓

PA (29 bits)	
F #	d
17 bit	12 bit

No. of entries in IPT = 2^{17} ✓

$\frac{29 * 2^{20}}{2^2 * 2^{10}} \Rightarrow 2^{17}$

Q5 Consider a machine with 64 MB Physical Memory and 32 bit Virtual Address. If the Page Size is 4KB. What is the approximate Size of Page Table?



→ 2^{20} entries. = 2 Byte

→ $2 \times 2^{20} = 2$ MB ✓

106 d

Q6 In a System with 32 bit Virtual Address and 1KB Page Size, use of 1 level Page tables for Virtual to Physical Address translation is not Practical because of:

(a) The large amount of internal fragment
(b) The large amount of external fragmentation.
(c) the large memory overhead in maintaining Page Tables.
(d) The large computation overhead in the Translation Process.

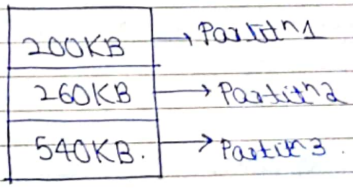
Pg# = 22 bit.

∴ No of Page Table entries = 2^{22} = 4 Million.

∴ 4M x 4B = 16MB contiguous Memory X.

Q7 A 1000 KB memory is managed using variable Partitions but no compaction. It currently has 2 Partitions of Sizes 200KB and 260 KB respectively. The Smallest Allocation request in KB, that could be denied is for:

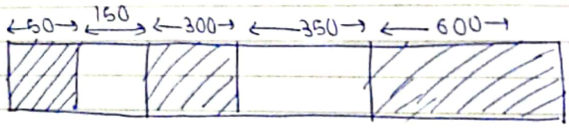
(a) 151 KB
(b) 181 KB
(c) 231 KB
(d) 541 KB.



Q8. A Multilevel Page Table is preferred in comparison to a Single level Page Table for translating Virtual Address to Physical Address because:

(a) It reduces the memory access time to read/write a memory location
(b) It helps to reduce the size of the Page table needed to implement the virtual address space of a Process.
(c) It is required by the TLB
(d) It helps to reduce the no. of Page faults in Page replacement Algorithms.

Q9. Consider the following Heap in which Blank regions are not in use and hatched regions are in use



The Sequence of requests for blocks of size 300, 25, 125, 50 can be satisfied if we use:

(a) either first fit or best fit policy.
(b) first fit but not best fit policy.

(c) best fit but not first fit policy.
(d) None

* Best fit.

dynamic partitioning

1) 300 → 350KB (50)
2) 25 (25)
3) 125 → 150 (25)
4) 50 → X (can't be placed) [contiguous].

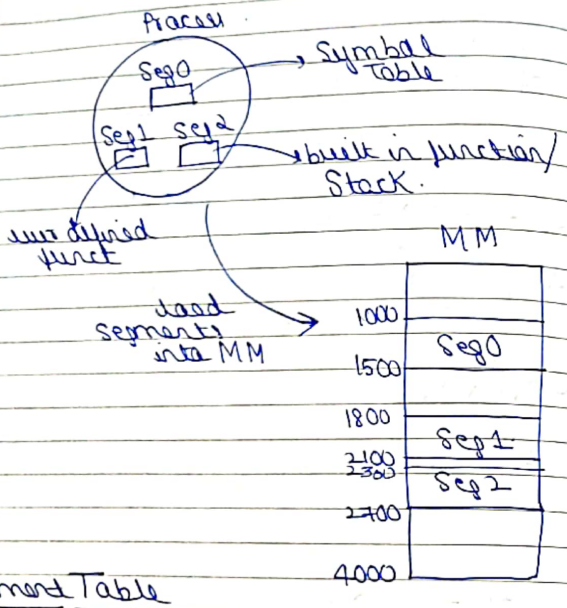
* First fit

1) 300 → 350 (50)
2) 25 → 150 (125)
3) 125 → 125.
4) 50 → 50.

(b)

5. Segmentation → Dynamic Partitioning

- Entire MM is divided into several Segments.

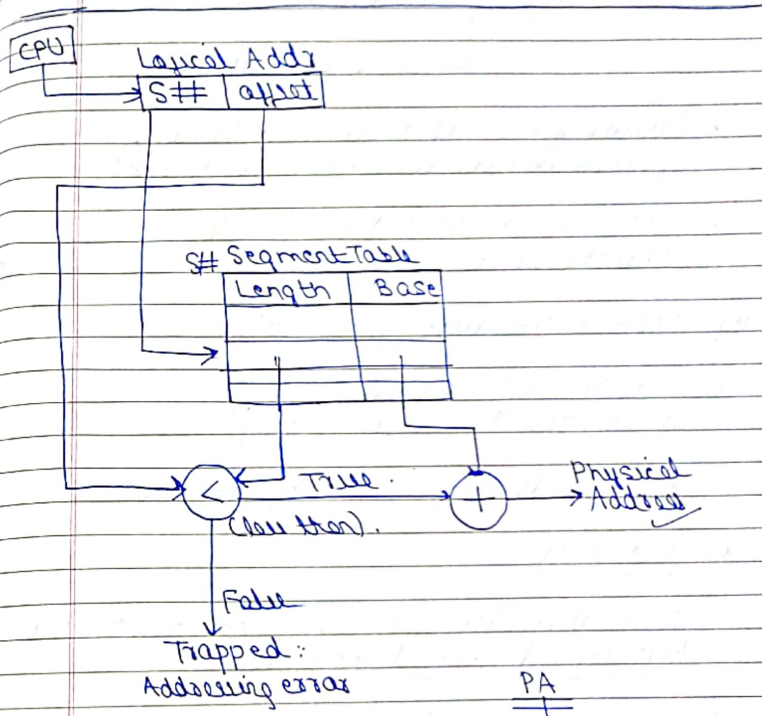


* Segment Table

S#	Length of Segment	Base of Segment
0	500	1000
1	300	1800
2	400	2300

Q Logical Address = 1,200 → go to Segm Table

PA = 2000 ← 1800+20=2000 (PA) ← (access is valid/not) 200 < 300 ✓



Q LA = (i) 0,333 — PA = 1333

(ii) 2,226 — 2226

(iii) 1,406 — X (Trap).

6. Virtual Memory Management

- Size of Virtual Memory → > Size of MM
  → < Size of SM.

* Purpose of VM: if you want to load a process whose size is larger than MM
  . MM ⟶ enhanced in form of VM.
  . Allocate some part of MM ⟶ VM.

eg: USB → 8GB, allocate 2GB → VM.

* Allocating VM acts like MM.
  if RAM size ↑, performance ↑.
  → increases degree of multiProgramming.

* Technique

Degree of multiProgramming can be increased by Demand Paging technique.

if, 100 frames ⟶ MM.
10 P ⟶ each P divided into 100 pgs.

1 Page = 1 Frame

- can load only 1 Process.

But as deg(MP) ↑, less pgs of each Process in MM
∴ Page fault ↑.

to improve degree of MP, not loading all pages of process in MM. not all.

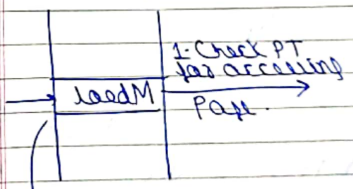
(10 P of each Process ⟶ MM). deg(MP) ↑.

* Strict DP / Pure DP

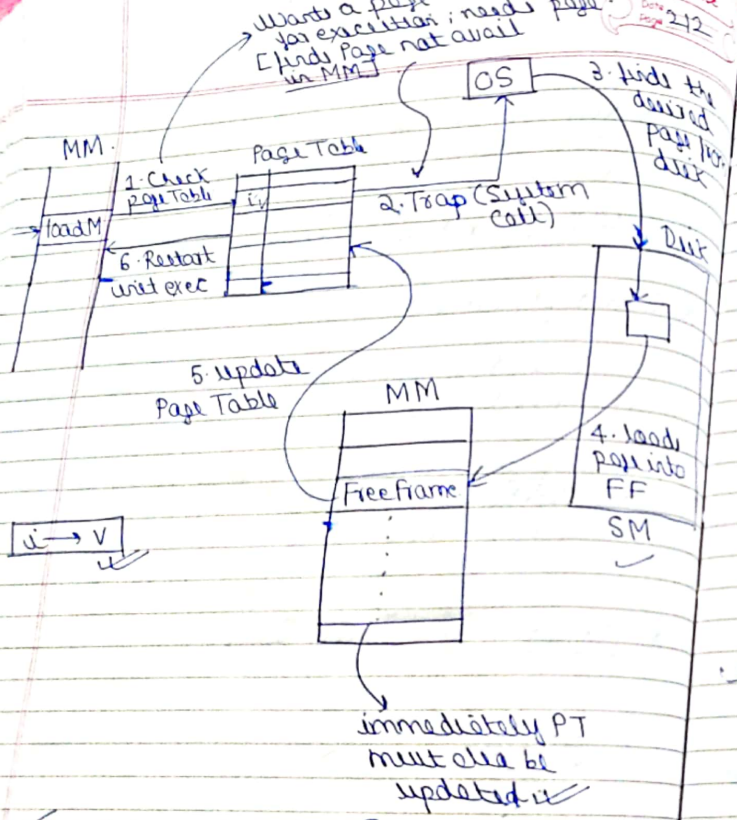
Never load a page into MM, unless it required or needed.

- When page corresponding frame info, is not available in Page table ⟶ MM;
  ∴ Page Fault

* Handling Page Fault Steps:



requires a Page, so it checks the Page Table.



- What if there is no free Frame?

- Swap page out into disk & load the page into free Frame.
  1) [Throw out some page into SM]
  2) [Bringing page into SM]

Both cases → update Page Table.

- Page replacement → when there is no free frame
  - empty frame
  - If same FF avail. → No Page replacement

* Which page must be swapped out from frames of MM?
  - Page Replacement Algorithm
    1. FIFO
    2. LRU
    3. Optimal
    } Vimp.

* Page Replacement Algorithm (Vimp)

(i) Optimal Algorithm

- used for Benchmarking Purpose (not real time)
- work on future requests.
- what kind of Page P will req in future.
- eg: Algo - 80% of Optimality.

* Pg frame size = 3 → ∵ 3 not avail, draw!

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	1	1	1	3	3	3
	2	2	2	2	2	2	2	2	2	4	4
		3	4	4	4	5	5	5	5	5	5
F	F	F	F	H	H	F	H	H	F	F	H

∴ No of Page Faults = 7

% of PFF = $\frac{7}{12} \times 100$

	Optimal	LRU	FIFO
PF=3	7	10	9
PF=4	6 ✓	8	10

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	①	1	1	1	1	~~1~~ 4	4	4
	2	2	2	2	②	2	2	2	2	2	2
		3	3	3	3	3	3	3	3	3	3
			4	4	4	5	5	5	5	5	5
F	F	F	F	H	H	F	H	H	H	F	H

(ii) LRU (Least recently Used)

- Page is not recently used → Replace.

1	2	3	4	1	2	5	1	2	3	4	5
1	1	~~1~~	4	4	~~4~~	5	5	~~5~~	3	3	3
	2	2	~~2~~	1	1	1	①	1	~~1~~	4	4
		3	3	~~3~~	2	2	2	②	2	~~2~~	5
F	F	F	F	F	F	F			F	F	F

No. of PF = 10 % of PFF = $10/12 \times 100$

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	①	1	~~1~~ 5	5	5	5	~~5~~ 4	4
	2	2	2	2	②	2	~~2~~ 1	1	1	1	~~4~~ 5
		3	3	3	3	3	3	~~3~~ 2	2	2	2
			4	4	4	4	4	4	~~4~~ 3	3	3
F	F	F	F		F	F	F	F	F	F	✓

(iii) FIFO (First in First out)

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	~~1~~ 4	4	4	~~4~~ 5	5	5	5	5	⑤
	2	2	2	~~2~~ 1	1	1	①	1	~~1~~ 3	3	3
		3	3	3	~~3~~ 2	2	2	②	2	~~2~~ 4	4

PF = 9.

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	①	1	~~1~~ 5	5	5	5	~~5~~ 4	4
	2	2	2	2	②	2	~~2~~ 1	1	1	1	~~4~~ 5
		3	3	3	3	3	3	~~3~~ 2	2	2	2
			4	4	4	4	4	4	~~4~~ 3	3	3

PF = 10

	Pg F Size = 3	PF Size = 4
Optimal	7	6
LRU	10	8
FIFO	9	10

increase PF

* Belady's Anomaly

- Generally when Page Frame Size is increased, no of page Faults will be reduced in all other algorithms except FIFO.
- In FIFO, sometimes; no of Page faults will be increased when the page frame size is increased.

* Remaining Algorithms

(iv) LIFO (Last in First out)

Pg frame Size = 3

1	2	3	4	1	2	5	
1	1	1	1	①	1	~~5~~	1
	2	2	2	2	②	2	
		3	~~3~~ 4	4	4	~~4~~ 5	

PF = 5 ✓

(v) MRU (Most Recently Used)

Pg frame Size = 3

1	2	3	4	1	2	5	.
1	1	1	1	①	1	1	
	2	2	2	2	②	~~2~~ 5	
		3	~~3~~ 4	4	4	4	

PF = 5

(vi) Counting Algorithms

(a) LFU (Least Frequently Used)

- Page which is least frequently referenced is replaced

1	2	3	4	1	2	5	
1	1	1	~~1~~ 4	4	4	~~4~~ 5	PF = 3
	2	2	2	~~2~~ 1	1	1	
		3	3	3	~~3~~ 2	2	

✓

1	2	3	1	2	4	5	.
1	1	1	①	1	1	1	
	2	2	2	②	2	2	
		3	3	3	4	5	

✓

freq (1) = 2
freq (2) = 2
freq (3) = 1

(vii)* MFU (Most frequently used)

→ page has already been reference so much, so remove it (old)

1	2	3	1	2	4	2	5
1	1	1	①	1	~~4~~	4	4
	2	2	2	②	2	②	~~5~~
		3	3	3	3	3	3

PF = 5

9. THRASHING

- Due to demand Paging, increase of Multi-Prog slowly lead to Thrashing.
- after Swapping, we ref the same page, we just throw out, (again & again).

∴ CPU spend more time in Swapping.

→ more Page faults.
→ CPU more time in Page replacement [Excessive Page i/o]

Thrashing leads to

* Working Set

Page reference String

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

Working set window 1 | Working set Window 2

pgs referred → {1,2,3,4} {4,5}

- when we refer main() → WS W1
- when lib() → WS W2
  4 5 4 5 ....

→ For same interval, 1 page can be frequently used, next → not used.
→ depending on funcn, diff pgs are refred.

∴ WSW Changes.

Here locality of Reference:

→ Spatial: → nearest memory loc again & again.
→ Temporal.
→ same memory loc again & again.

a[0] a[1] a[2]

Same mem loc will be

→ Special LOR.

```
for(i=0; i<5; i++).
    scanf("%d", &a[i]);
```

→ same mem loc i, accessed again & again

[Loop V → Temporal LOR]

* Frame Allocation

- How many frames can be allocated to a Process?

(i) Equal Allocation
(ii) Proportional Allocation
(iii) Priority Allocation

(i) if 100 Frames
5 Process (P1, P2, P3, P4, P5)

P1 → 20
P2 → 20
:
P5 → 20

equal.

- done by instruction Set Architecture.

(ii) Process which has more no of page will be given more no of frames.

100 Frames

P1	P2
160 pgs	40 pgs.
↓	↓
80F	20F.

(iii) Where Process with higher priority given more no of frames.

* Local & Global Placement (wrt. F. Allocn)

Process can't access frames allocated to other Processes.

if any frame of a Process is free, it can be used by a Process, with excess Pgf.

eg: 100 Frams

P1	P2
150 pg.	30 pg.
↓	↓
50F	50F ⇒ equal All.

rem: 20F.
→ (used by P1).

∴ efficient [Global P].

GATE Qs

Q1 Consider the Virtual Page reference String.

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

Qny demand Paged Virtual memory System running on a computer System that has MM Size of 3 PFs which are initially empty.
Let LRU, FIFO and Optimal denote the no. of Page Faults under the corresponding Page replacement Policy, then:

(a) Optimal < LRU < FIFO
(b) Optimal < FIFO < LRU
(c) Optimal = LRU
(d) Optimal = FIFO.

FIFO

	1	2	3	4	1	3	2	4	1
	1	1	1	X4	4	4	4	④	4
		2	2	2	X1	1	1	1	①
			3	3	3	③	3/2	2	2

PF=6

Optimal

	1	2	3	4	1	3	2	4	1
	1	1	1	1	①	1	1	1	①
		2	2	X4	4	4	4	④	4
			3	3	3	③	3/2	2	2

PF=5

LRU

	1	2	3	4	1	3	2	4	1
	1	1	1	X4	4	4	4	④	4
		2	2	2	X1	1	1	1	①
			3	3	3	③	3/2	2	2

PF=5 ×

Q2 A Virtual Memory System uses FIFO Page replacement Policy & allocates a fixed number of frames to a Process. Consider the following Statement:

S-P: Increasing the no of PF allocated to a process, sometimes increases the PF rate.
S-Q: Some Programs, don't exhibit locality of reference.

Which one of the following is true?
(a) both P & Q are true & Q is reason for P.
(b) both P & Q are true, but Q is not the reason for P.
(c) P is false, but Q is true.
(d) Both P and Q are false.

S-P: Belady's Anomaly (FIFO) ✓

S-Q: Accessing array → SLOR. ✓
(we can write Pro without SLOR)
↳ Hello World.

Q3 A process has been allocated 3 page frames. Assume that none of the pages of the process are available in the memory initially.
The process makes the following sequence of page referenced string

1, 2, 1, 3, 7, 4, 5, 6, 3, 1.

(i) If optimal Page replacement Policy is used, how many Page faults occur for the above reference string?

(a) 7
(b) 8
(c) 9
(d) 10

(ii) LRU Page replacement Policy is a practical approximation to optimal Page replacement for the above reference string.
How many more PFs occur with LRU than with the optimal Page Rep Policy

(a) 0
(b) 1
(c) 2
(d) 3

optimal

1	2	1	3	7	4	5	6	3	1
1	1	①	1	1	1	1	1	1	①
	2	2	2	~~2~~ 7	7	~~7~~ 5	5	~~5~~ 3	3
			③	3	~~3~~ 4	4	~~4~~ 6	6	6

X

(9)

LRU

1	2	1	3	7	4	5	6	3	1
1	1	①	1	1					
	2	2	2	2					
			3	~~3~~ 7					

✓

Q4 In a Virtual Memory System, the Address Space specified by the Address lines of the CPU must be ____ than the Physical Memory Size & ____ than the SMS.

larger / smaller ✓

Q5 The address sequence generated by tracing a particular program executing in a pure demand Paging System, with 100 records/pg.
With 1 free frame is recorded as follow:

0100, 0200, 0430, 0499, 0510, 0530, 0560, 0120, 0220, 0240, 0260, 0320, 0370

what is the no of PFs.

Soln:

(0-99) records / Page — imp

0-99	0
100-199	1
200-299	2

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

1 2 4 – 5 – – 1 2 – – 3 –

(7). ✓

Q6 A memory Page containing a heavily used Variable that was initialised very early and is in constant use, is removed when:

(a) LRU is used.
(b) FIFO is used.
(c) LFU is used.
(d) None of the above.

∴ It is constant use, ∴ it came first, leave first. ✓

Q7 The System use FIFO Policy for Page replacement. It has 4 Page frames with no Pages loaded to begin with. The System first access 100 distinct Pages in some order and then access the same 100 pages, but now in the reverse order.
How many PF?

(a) 196
(b) 192
(c) 197
(d) 195

1	1	1	1	5	5
	2	2	2	2	6
		3	3	3	3
			4	4	4

* 1, 2, 3, 4 5 6 7 · 8 · 9 · · 97, 98, 99, 100

1	1	1	1	5	5	5	5				97
	2	2	2	6	6	6					98
		3	3	3	7	7	. . . . .				99
			4	4	4	8					100

till here,

100, 99, 98, 97 · · · · · · · 3, 2, 1

97		
98		
99		
100	PH →	→ PF · · ·]

Q8 The essential content in each entry of a Page Table:
(a) Virtual Pg no.
(b) Page frame no.
(c) Both Virtual Pg no & Pg frame no.
(d) Access write info.

Q9 In which one of the following, page replacement policies, Belaady's anomaly may occur: FIFO

Q10 The min no of Page Frames that must be allocated to a running Process in a virtual memory environment is determined by:
(a) The instruction ~~set~~ architecture.
(b) Page Size
(c) Physical Memory Size
(d) No. of processes in Memory.

Q11 The optimal Page replacement algorithm will select the page that:
(a) has not been used for the longest time in the past.
(b) will not be used for the longest time in future.

(c) has been used the least no. of times.
(d) has been used more no of times.

* FILES SYSTEM

- File: collection of related data stored in records.
- Attributes of file:
  - Name, size, type, date create, auther, date modified, loc^n, path.
  - Every file has an id also (non-Human readable format)
- Operations on File
  1. Create file
  2. Open file
  3. Write file.
  4. Rename file.
  5. Close file.
  6. delete file.

* Types of Files

- File type is known based on extension
  - .exe → executable
  - .batch → batch
  - .java → java file
  - .img/.jpeg → image
  - .mp3 → audio file
- Know info it contains

* Access file

1) Sequential Access
2) Direct Access
3) Indexed Access

1) In a file:

Block → B1 B2 B3 B4 ..... B6

(After reading B1, to reach B4, it must visit all the B2, B3 blocks)
∴ Sequential order

2) Direct

B1 B2 .... B6

- for 1 block to another it can go directly.

3) Indexed Access

- Index Block denoting the blocks to be visited, & to be visited one after the other.

* How file can be accessed depends on permission

Unix → chmod UMASK.

owner group universe.

⇒ chmod 777 a.txt.

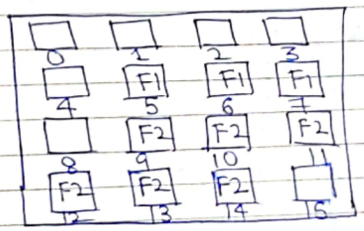
r w x
4 2 1

→ permission to all.

* Allocating memory for Files

Allocation Methods:

1. Contiguous Allocation / Continuous Allocation



. File F1

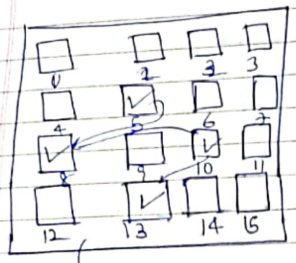
File	Start	Length
F1	5	3
F2	9	6
F3		6

∴ external fragmentation
[holes in mem]

6 blocks must be contiguous
∴ X.

: Use Compaction technique.

2. Linked Allocation



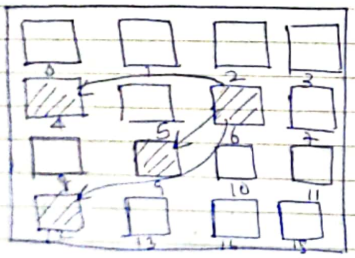
File	Start	end
F1	5	13

- Linked list Notation (Pointer)
- No external fragmentation

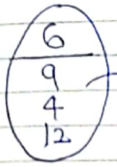
(in every block, some part of memory hold address of next Block)

Disad: When link is failed, you can't access the complete file.

3. Indexed Allocation



6 → Index Block.



→ list of blocks to be accessed

- No external Fragmentation.
- If the file size is small, it will be limited in 1 Block itself, even though, we need 1 extra block to store info.
- If file size is huge, multiple index Blocks used.

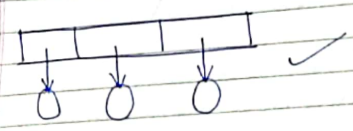
* Free Space Management

How to show, which blocks are free/allocated:

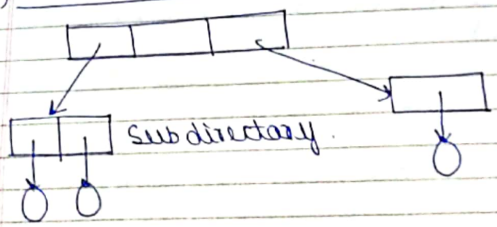
1. using bit notation → 0/1.
   - 0 → available
   - 1 → used.
2. linked notation:
   - → list of all Blocks which are free.
3. Counting technique
   - → If Block 0, 1, 2, 3 free
     it says: from Block 0: 4 are free.
4. indexed
   - → make use of index Block holding list of all free Blocks.

* Files are stored in directories:

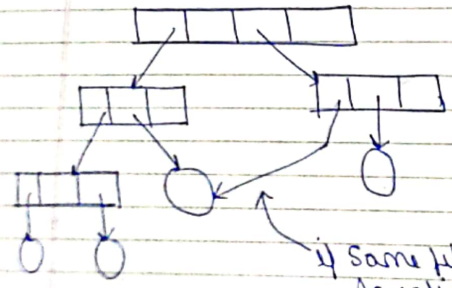
1) Single level directory Structure.



2) Two-level



3) Tree-level



↑ Same file common, Acyclic Graph. (IV).

4) Acyclic Graph.

* If any directory has files → rename problem.
  - Single level directory → avoid.

* 2 level Directory

- File name can be same across levels.
- Grouping Problem

All Java files can be grouped.
↳ Solved in Tree level.

GATE Qs.

Q1. Using a larger block size in a fixed block size file System leads to:

(a) better disk throughput but poor disk Space utilization
(b) better disk throughput & better disk Space utilization.
(c) Poorer disk throughput but better disk Space utilization.
(d) Poorer disk throughput & poorer disk Space utilization.

→ File will occupy less space of blocks → stored in 2 nodes of 30.

⇒ Time Saved.

⇒ But lot of Space wasted.

Q2 In the indexed allocation scheme of blocks to a file, the max possible size of the file depends on:

(a) The size of the blocks & the size of the address of the blocks.

(b) The no of blocks used for the index & the size of the blocks.

(c) The size of the blocks, the no. of blocks used for the index & the size of the address of blocks.

(d) None.

X

X X X