CSE 321

Introduction to Operating system

☑ What is OS?

Web browser

↓

OS is a program/ software that creates interface/module for users to use raw hardware

↓

Hardware: keyboard, monitor, sound, graphics.

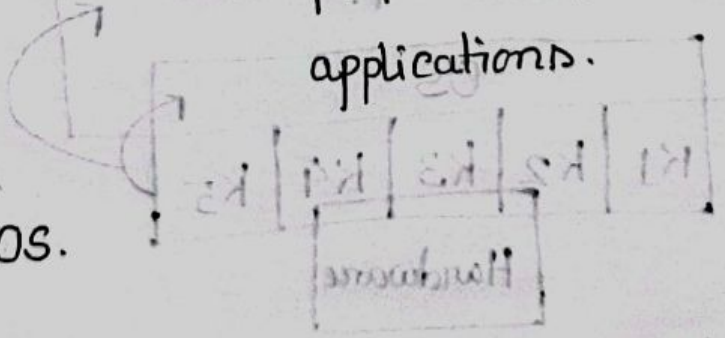
☑ system software vs application software

↓

Directly communicates with hardware.

↓

designed for end users. people who are using the applications.



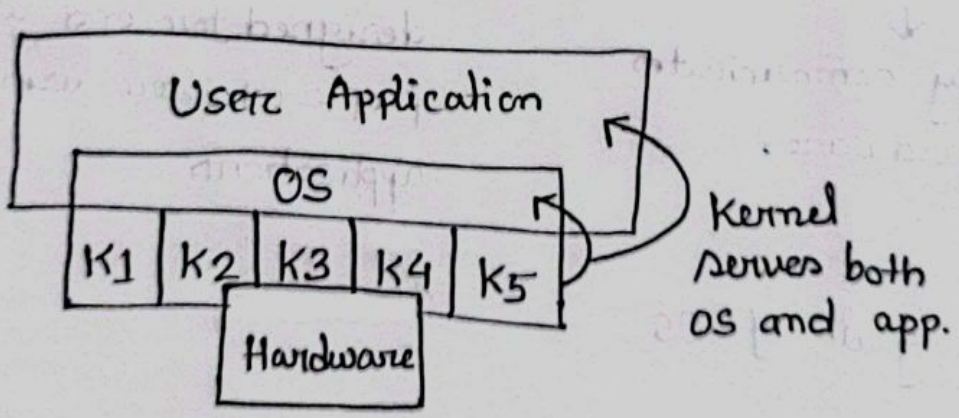
☑ Major goals of OS.

☑ Computer system organization

Users
↓
compiler, assembler, text editor, database system
system & application programs
↓
OS
↓
computers hardware.

☑ Kernel.

→ One program running at all times
→ Smaller the better
→ loads first
→ remains in main memory.



→ loaded into a protected area so that they can't be overwritten.

☑ Bootstrap program

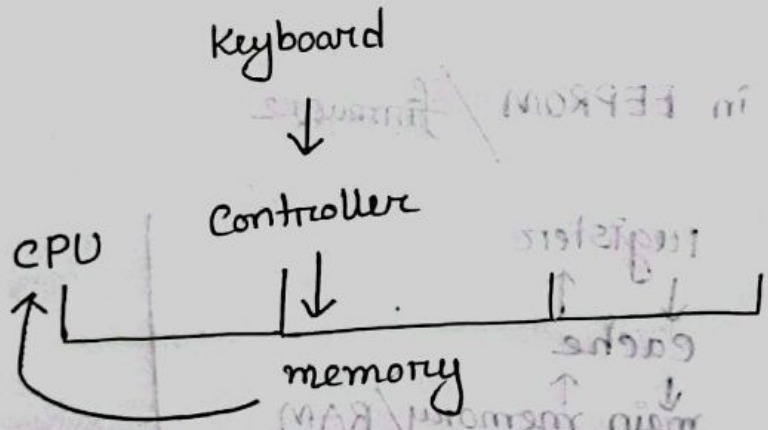
→ stored in ROM or EEPROM.

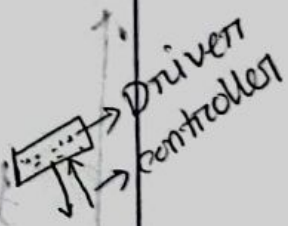
☑ Computer system organization.

→ there can be 1 or more CPU

→ data communication is through bus.

☑ Operation of a controller





☑ Device driver vs device controller.

Device driver → kernel/software

device controller → Hardware

Device controller manages the communication between device driver and OS.

device driver → device

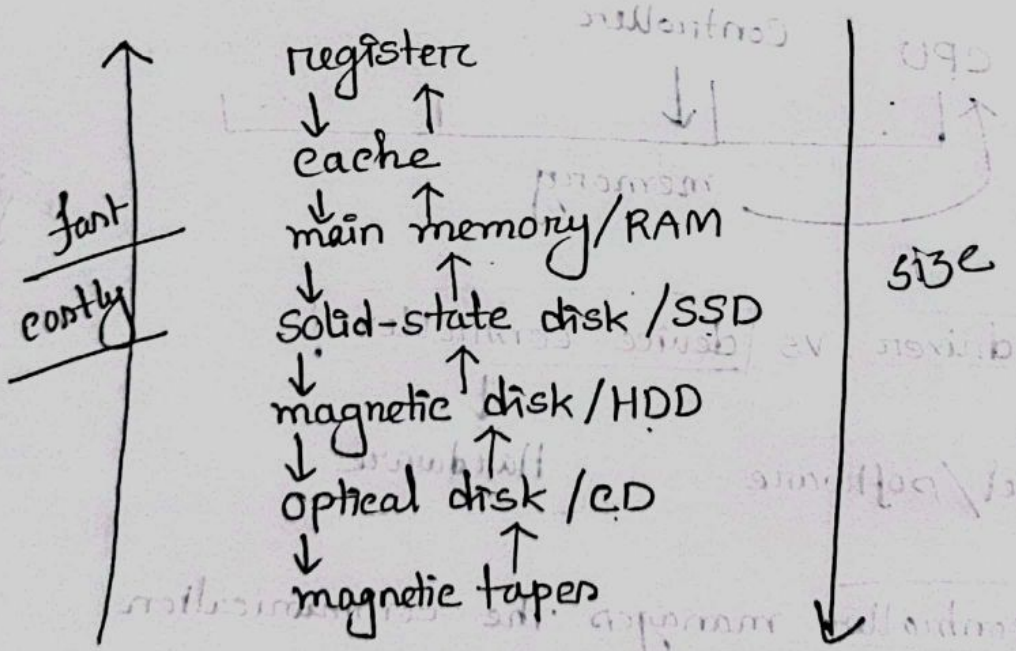
Storage Structure

I) Main memory : RAM, deletes after PC is shut-down, volatile.

II) Secondary memory : non volatile

Suppose a movie file is in HDD but our media and CPU can't access it as HDD is secondary memory. So movie is first transferred to main memory and media shows it.

→ kernel is in EEPROM / firmware



OS architecture

1] Single processor → 1 CPU + special purpose processor

2] Multipurpose processor → multiple CPU

3] Clustered system → individual systems/nodes closely linked via local area network (LAN). Fast communication. Example : InfiniBand

Multiprogramming

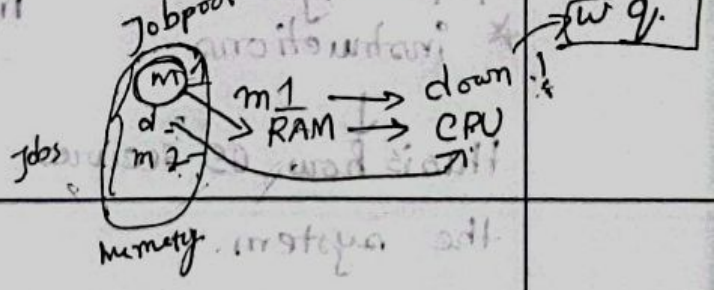
★ CPU is never idle

Job pool → OS → memory → processor

★ ensures jobs are listed in memory from pool.

Time Sharing

- interactive system
- response time is very less.



☒ Requirement of multiprogramming

1] Job scheduling

Job pool

↓ OS (Job scheduler) → decides which job will be loaded into memory.

2] CPU scheduling  memory

↓ CPU scheduler

CPU

★ Virtual memory.

☒ OS operations

interrupt driven

☒ Dual mode operation → mode change

OS code → kernel (0) → privileged instructions* → this is how OS secures the system.

user code → (1) → normal instruction

☑ System call.

OS এর যেই functionalities /module গুলো Hardware এর যে functionalities user application কে provide করার কথা যেই scope কাজে দেয়।

☑ System call Interface

যখন user application এর কোন task complete করতে হয়, তখন যে তা জানাবে System call interface কে। SCI এর কাছে table থাকে যাতে সব system call list করা থাকে। After the lines of codes are executed, value will return to SCI and then UA.

Protection → User and kernel mode

↓ ↓

UA SC execution

SCI knows everyone and works for everyone.

fork() → create process()

☑ System Program

☑ System boot

execute bootstrap program

☑ Process

Batch system → jobs

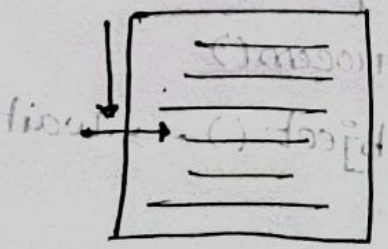
Time sharing system → tasks / user programs

Modern OS → process → a program that is in execution

- Example: Farewell program
  - place
  - Food
  - Furniture
  - Decor

process has:

- program counter



- Stack : temporary data (function param, local var, return address)
- Data section : global var
- Heap : dynamically allocated memory during runtime.
- text : programing code

program (passive)

↓ execution / load program into memory

Process (active)

{ 1 program can have multiple process.
Google chrome having multiple windows.

States of a process : 5 states

- New → HDD → RAM memory
- Running → CPU
- Waiting → RAM memory — if CPU is busy, wait in RAM memory or other waiting
- Ready → process in ready to go from RAM memory to CPU
- Terminated

☑ Process State Diagram

☑ Representation of process in OS

process control block (PCB)

☑ Process Scheduling

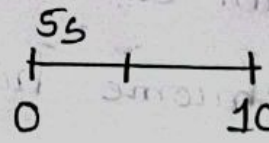
→ Scheduling queue

Job queue → ready queue → CPU → Device queue (Device 1)
→ Device queue (Device 2)

(Secondary memory) (Main memory) (processor)

→ Queuing Diagram

time slice expired

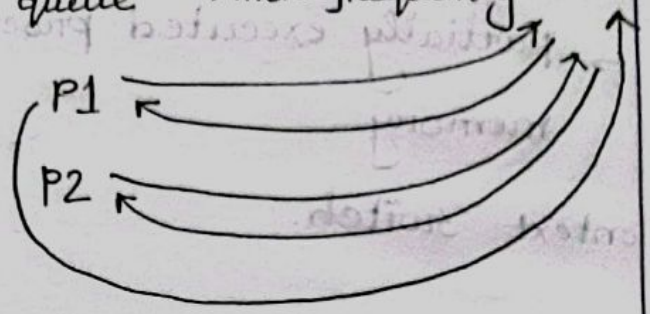


Except for ready queue, if a process is in any other queue, it is in waiting state.

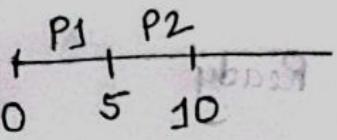
Scheduler

Job queue → long term scheduler → Ready queue → short term scheduler → CPU

- long term scheduler: → less frequently; → controls degree of multi programming
- short term scheduler: → more frequently



	P1	10 ms	5 ms
more frequently executed.	P2	15 ms	10 ms

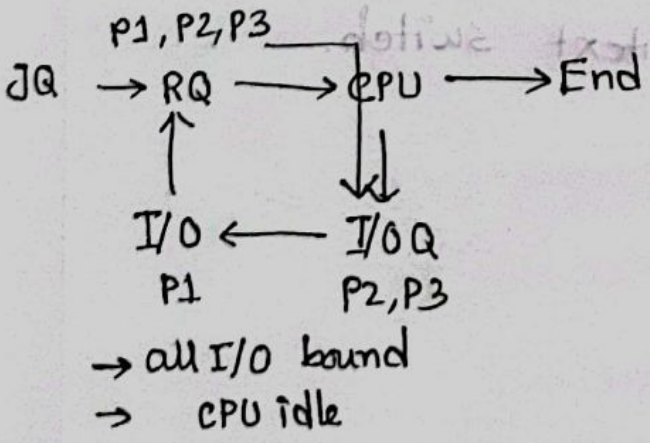


CPU bound vs I/O bound process

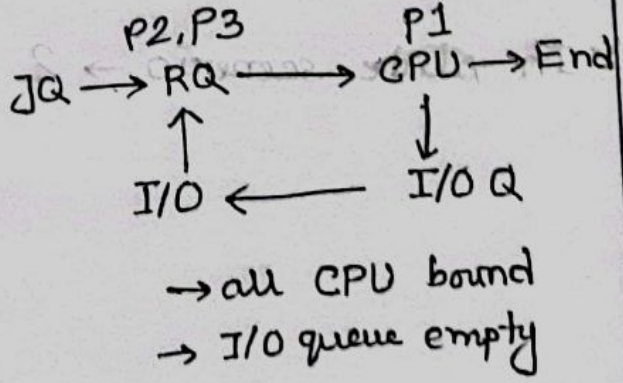
CPU bound → find prime #

I/O bound → calculator input

Long term scheduler must select wisely. They can't all be CPU bound or I/O bound only.



→ all I/O bound
→ CPU idle
→ RQ empty because we can't bring new process.



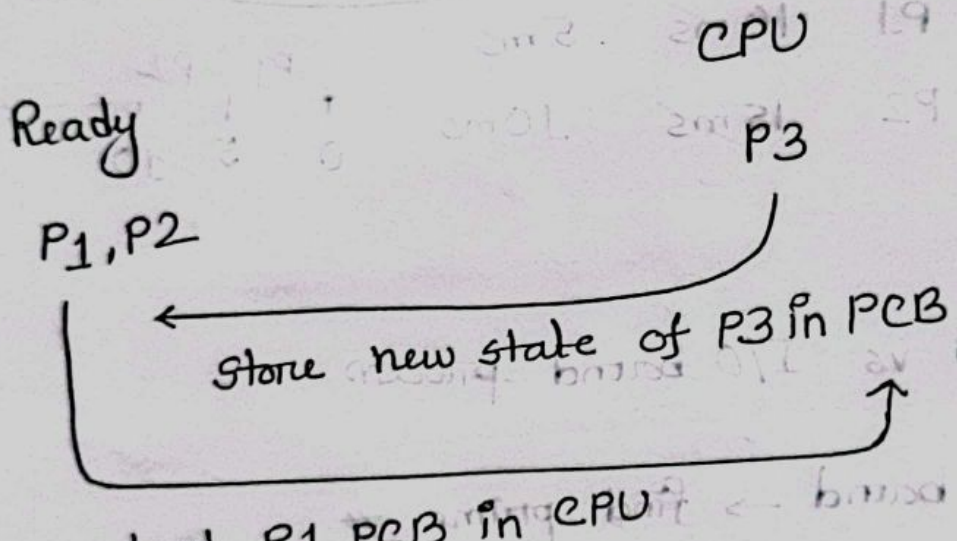
→ all CPU bound
→ I/O queue empty

☑ Medium term scheduler

→ time sharing system

→ partially executed process stay in secondary memory

☑ Context switch.



during this switch, CPU stays Idle. This is called context switch time. This time is overhead, no CPU utilization.

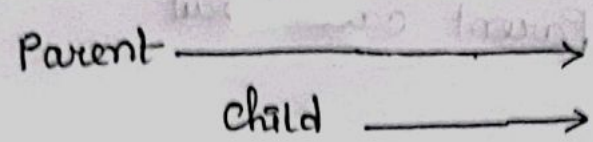
★ In slide scenario → 2 context switch.

Operations on process

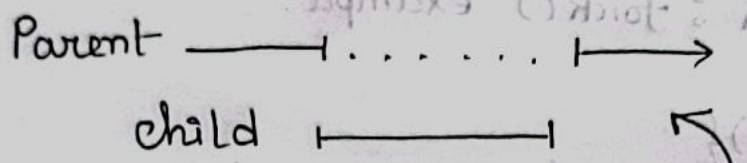
Unique PID

→ Process Creation

(I)



(II)



→ process creation in Unix

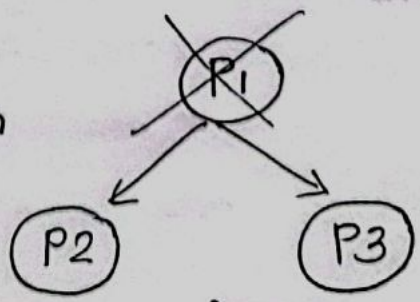
fork() → creates new child process

parent PID > 0

child PID = 0

→ process termination

cascading termination



orphan

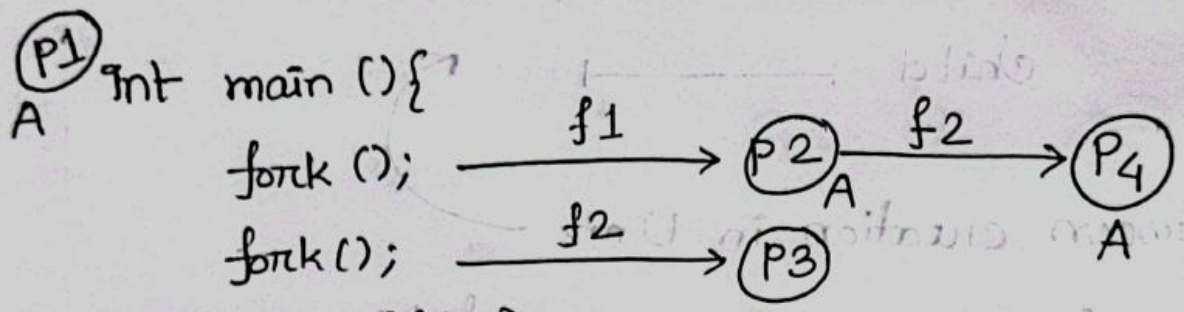
Zombie process in Unix

child process ends but parent doesn't call wait. System thinks child resources are running but in reality the resources are not running, hence zombie process. Parent calls wait function late.

Process Creation : fork() example.

1]

```
int main(){
    fork();
    fork();
    print f("A")
}
```



4 process

3 child process

4 A's

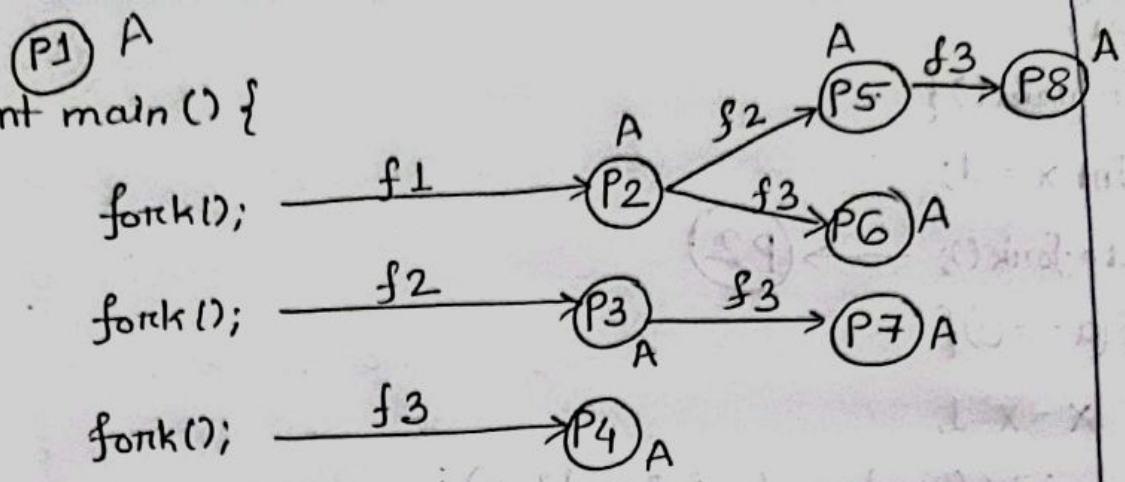
2] int main() {

fork(); f1

fork(); f2

fork(); f3

print f("A");



"AAAAAAAA"

8 A's

8 process.

3] int main() {

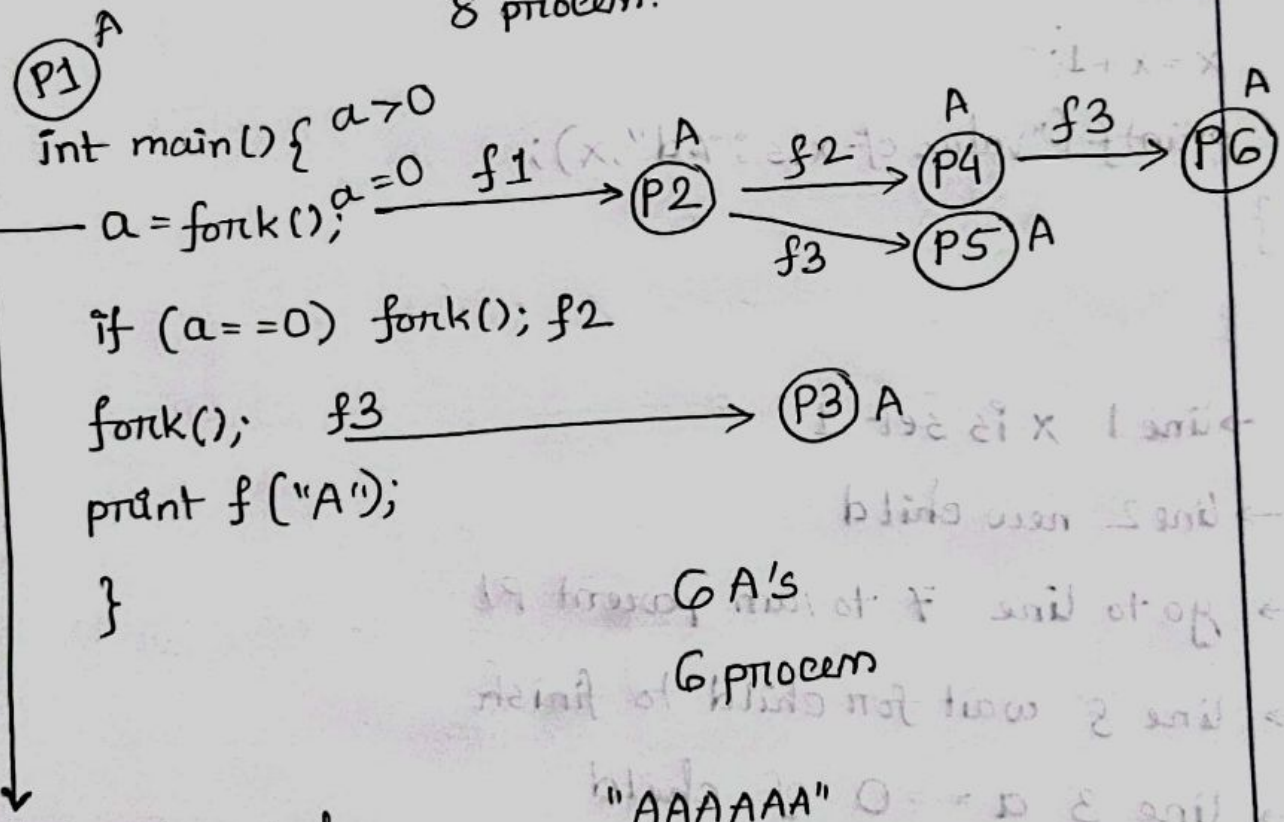
a = fork(); a=0 f1

if (a==0) fork(); f2

fork(); f3

print f("A");

}



6 A's

6 process

"AAAAAA"

a has fork return value

if a>0 → parent

if a=0 → child.

(P1)

4] int main(){

1. int x = 1;
2. a = fork(); ⟶ (P2)
3. if(a == 0){
4. x = x - 1;
5. printf("value of x is: %d", x);
6. }
7. else if (a > 0){
8. wait(Null);
9. x = x + 1;
10. printf("value of x is: %d", x);
11. }
12. }

→ line 1 x is set 1
→ line 2 new child
→ go to line 7 to run parent P1
→ line 8 wait for child to finish
→ line 3 a == 0 of child.
→ x = 1 - 1 = 0, print "value of x is 0"
→ return to line 9, x is now 1 → global var
→ x = 1 + 1 , print "value of x is 2"

(P1)

5] int main(){

a = fork(); ——f1——→ (P2)

if (a==0) {

print('B');

}

else if (a>0){

wait(NULL);

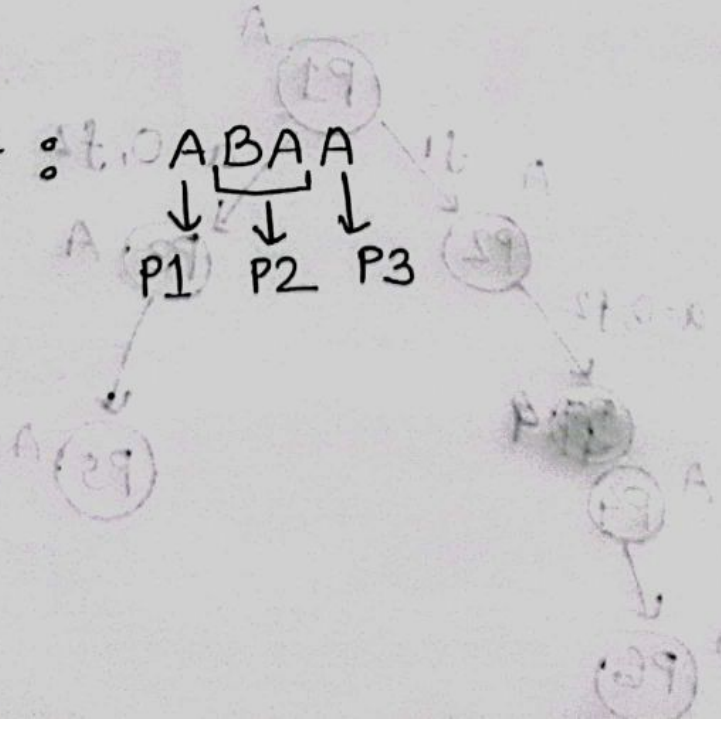
fork(); ——f2——→ (P3)

}

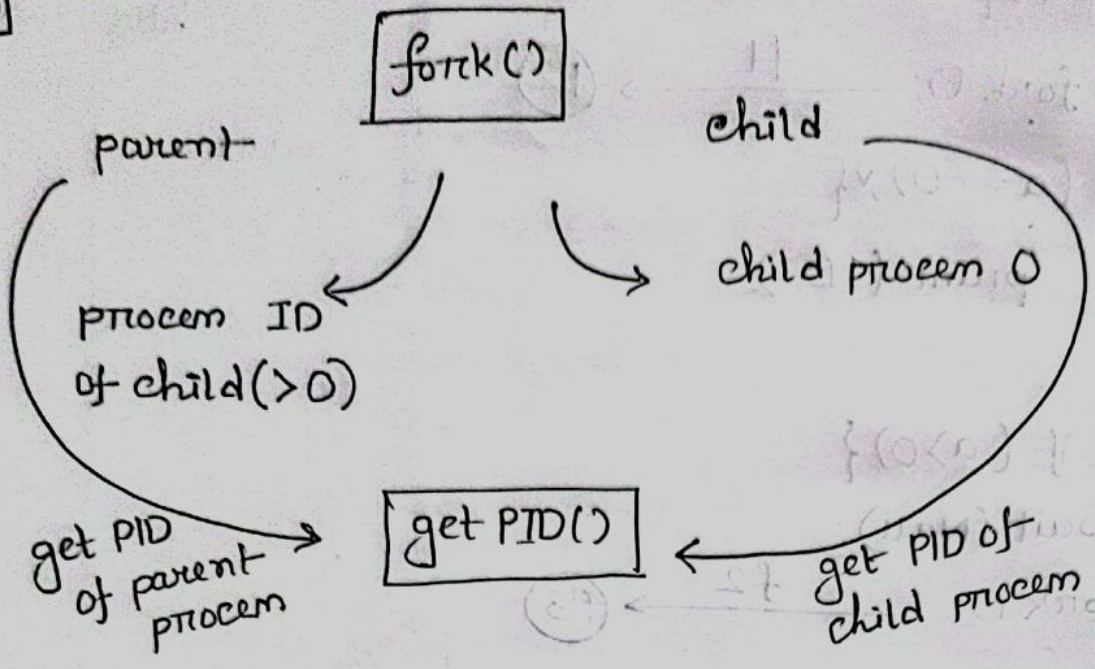
print('A');

B A A A
↓ ↓ ↓ ↓
P2 P2 P1 P3

If no wait : A B A A
↓ ↓ ↓
P1 P2 P3

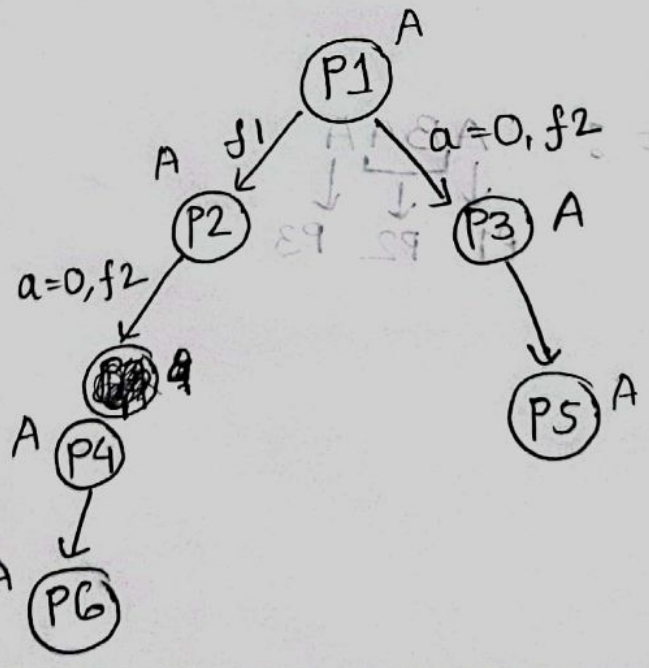


6



7

```
int main(){
    fork(); →f1
    a=fork(); →f2
    if (a==0) fork() →f3
    printf("A");
}
```



Interprocess Communication (IPC)

1) Independent
2) Cooperating → why we need it?
   - info sharing
   - computational speedup
   - modularity
   - convenience

two types of IPC:

1. Shared memory → fast
2. Message passing → slower because it needs kernel to pass messages. So it is costly as well

distributed system → small amount of data

→ Shared memory system

producer

buffer / shared space

consumer

☑ producer-consumer problem (Producer)

buffer : 2 types → Bounded (limited space)

→ Unbounded (unlimited space)

Code is not efficient : n size buffer, (n-1) full.

☑ (Consumer)

☑ Message passing system

→ no shared space but distributed system.

→ systems reside in different locations

→ small amount of data.

Process Synchronization.

☒ Background.

Concurrent:

P1 — — — —

P2 — —

Issue:- shared variable } data integrity problem.

Parallel:

P1 ——————

P2 ——————

☒ Producer Consumer problem

counter ⟶ shared variable

during parallel execution, if both producer and consumer tries to update counter, data integrity problem arises.

☒ Data integrity problem.

☒ Race condition.

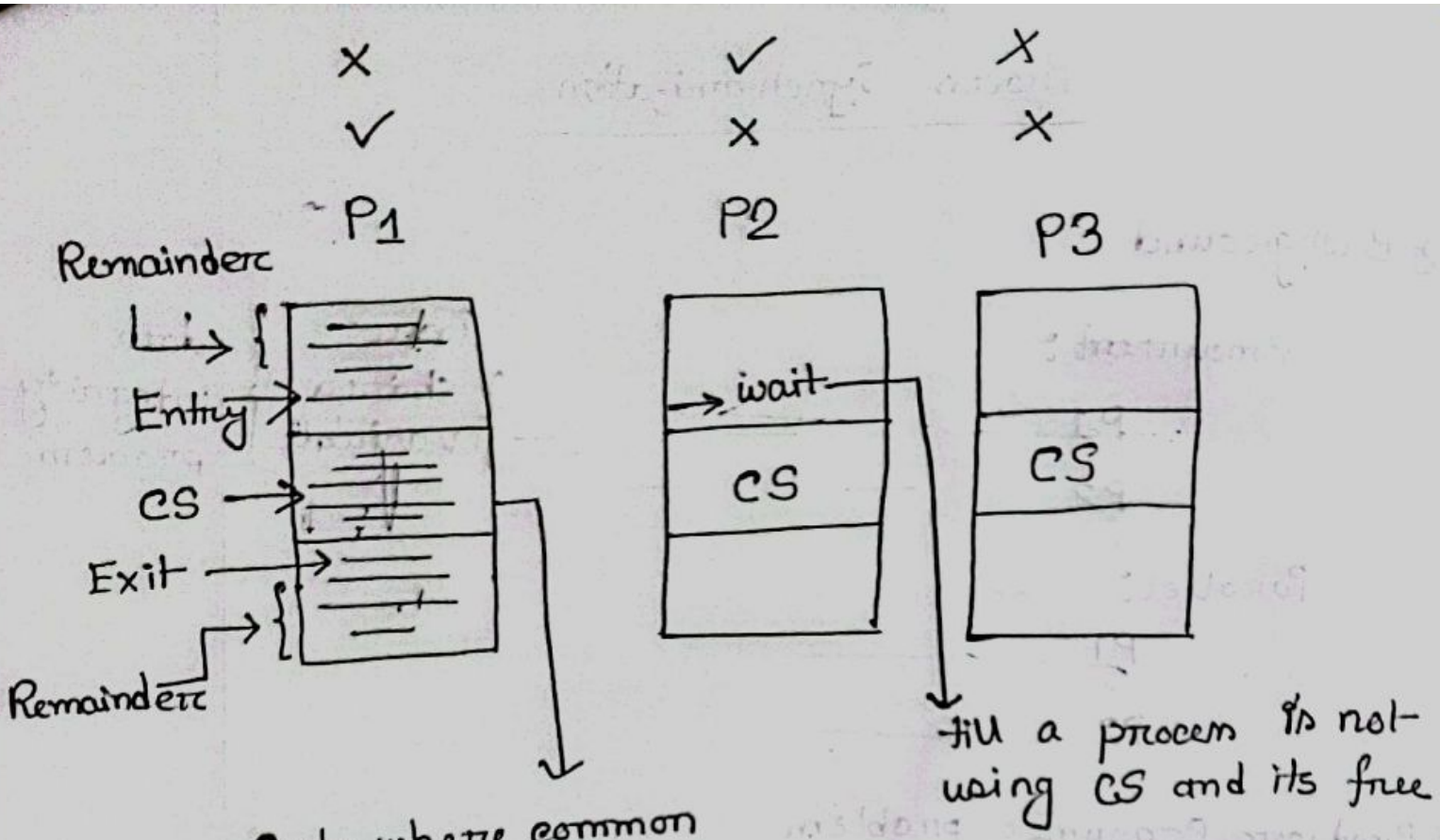
Solution

☒ Critical Section.

Entry Section

Exit Section

Remainder Section.



Code where common variable is shared

till a process is not using CS and its free

We can't let the CS of P1, P2, P3 run together

☑ Requirements of solution to CS problem.

(i) Mutual execution

(ii) Progress

(iii) Bounded waiting

Critical Sections in OS.

(i) Preemptive kernel → race condition

(ii) Non- " " → no " "

Peterson solution for CS problem:

Software based solution

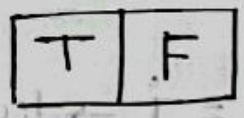
2 process only.

int turn → variable

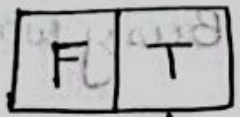
boolean flag [2] → array

Suppose 2 process P_i and P_j 

turn = □ next in line.

flag = 

ready to go to CS

Flag = 

↓

P_j ready for CS

limitation: can't be solved for multiple processes. 

flag true → ready to go to CS, false → not ready.

☑ Hardware based solution

locking → flag.

test and set(), compare and swap()

word → memory unit / location

atomically → 3 instructions in a single pact, no other instruction will go in between.

☑ Test and set()

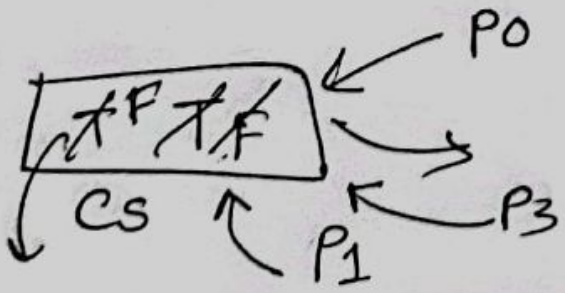
executed atomically.

* target → location of variable

* target = true → check and modify

& lock → flag location

True → already CS এ আছে wait till false.



Compare and swap()

1 → already cs এ আছে

0 → cs free

& lock → flag

0/1 → int * value → int variable

Mutex lock

Simplest solution of Hardware solution.

available	true	→ cs free	→ release lock
available	false	→ cs working ↓ acquire lock	

Semaphore

Synch technique

int variable

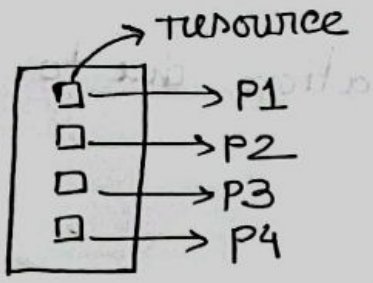
wait() , signal() → atomic operation

wait() → CS এ ঢোকার আগে call করে

signal() → CS এর কাজ শেষ করে call করে

2 types of semaphore

① Counting semaphore



S = 4 → 4 resource / instance

S = n = 5

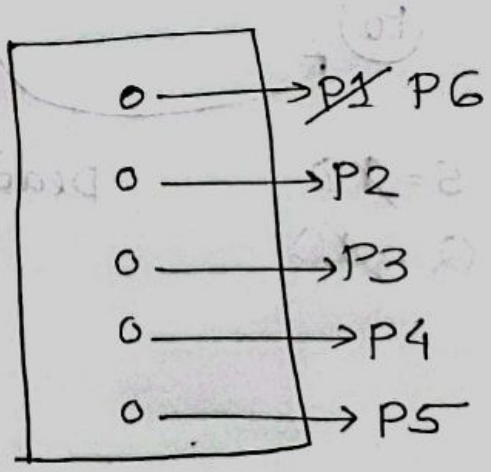
P_1. wait (S) → S = 4

P_2. wait (S) → S = 3

P_3. wait (S) → S = 2

P_4. wait (S) → S = 1

P_5. wait (S) → S = 0

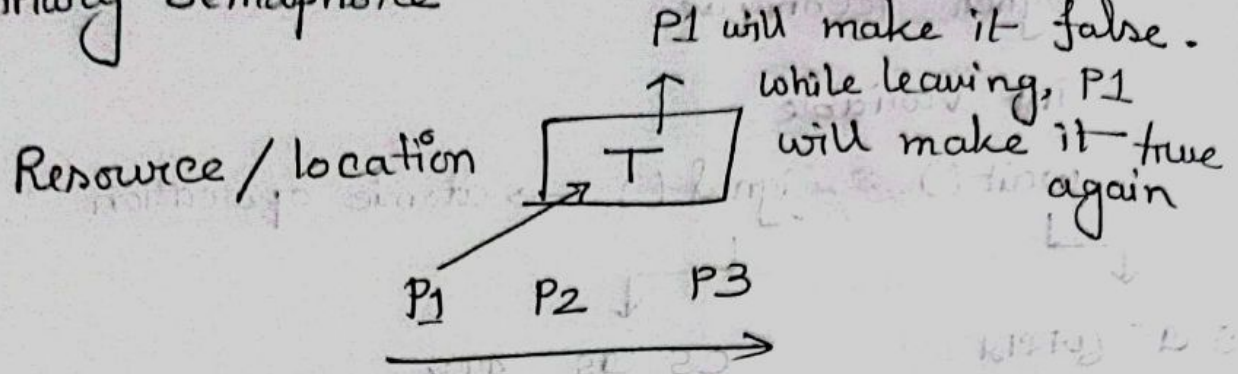


P_6. wait (S) → has to wait for signal() because S = 0

P_1. signal (S) → S = 1

P_6. wait (S) → S = 0

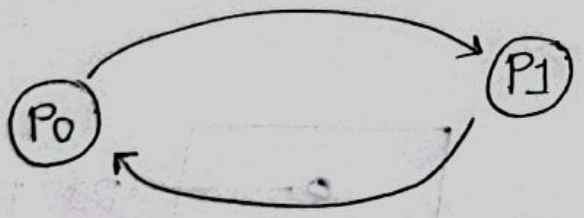
(11) Binary Semaphore



☒ Mutual exclusion with Semaphore

Binary semaphore → mutex

☒ Deadlock and starvation due to semaphore



Deadlock

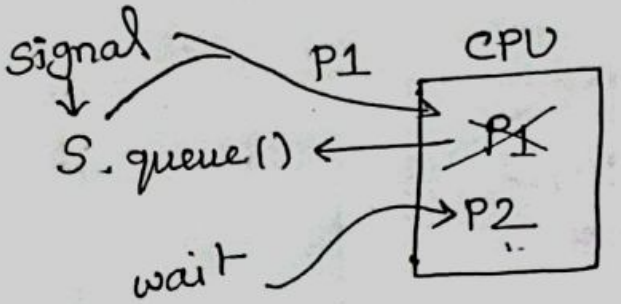
P0 → S = ~~1~~ 0

P1 → Q = ~~1~~ 0

Semaphore Implementation.

resource call → wait

exit → signal



P1

S → value = ∅ (-1 < 0)

S → list = { P1 } → block P1 so that another process can use CPU

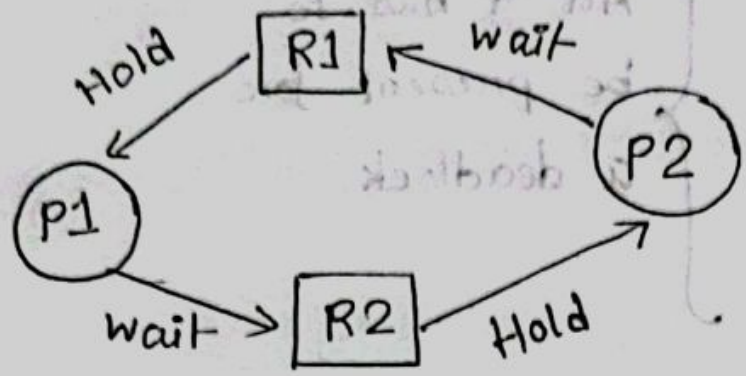
P1.wait()

P1.signal()

S → value = ∅ 1 0 → wake up P1 because S.value ≥ 0

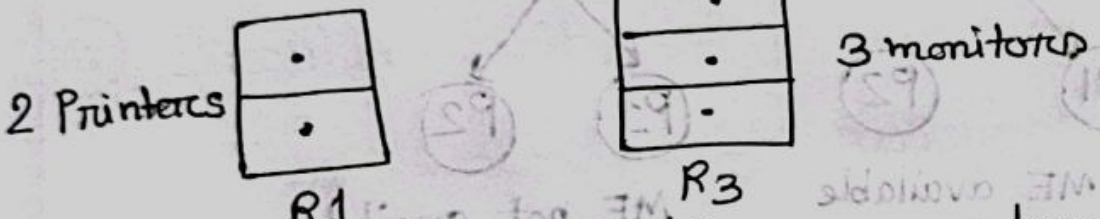
Deadlock

Deadlock



System model.

Finite resource → CPU cycles, file, I/O devices.

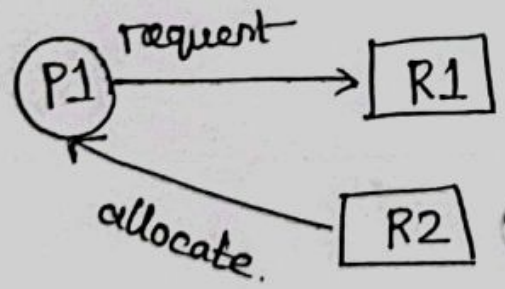


this resource has 2 instances

this resource has 3 instances.

Sequence of process using a resource :

Request → Use → Release

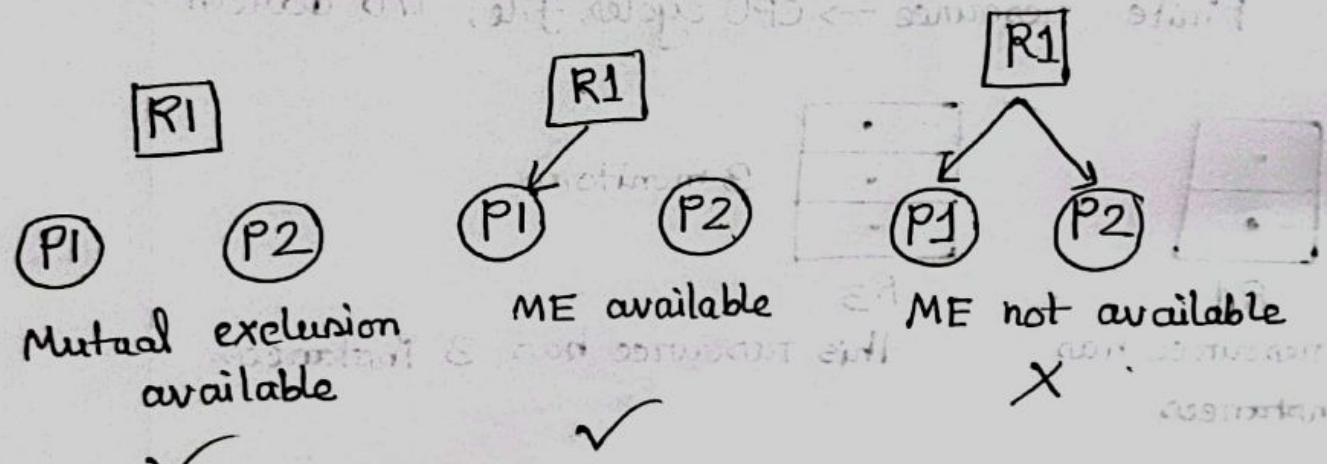


Conditions for Deadlock:

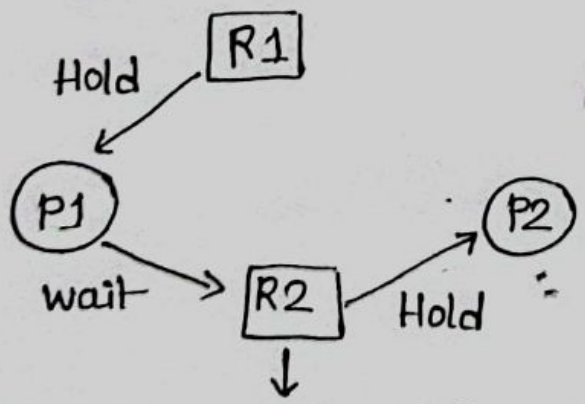
(I) Mutual exclusion.
(II) Hold and wait
(III) No preemption.
(IV) Circular wait

} All 4 has to be present for a deadlock.

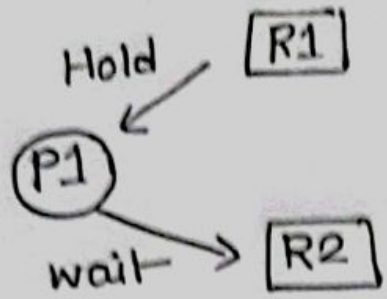
(I) Mutual exclusion



(II) Hold and wait

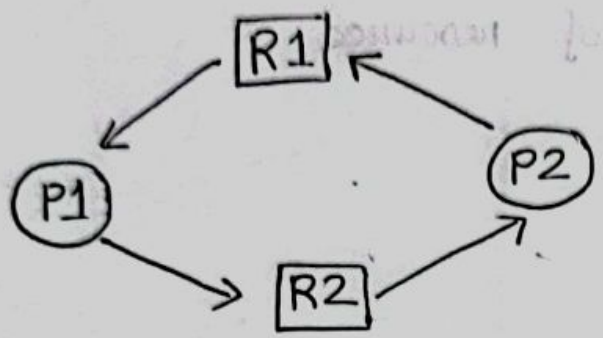


যার জন্য wait করছি সে যদি Hold এ থাকে



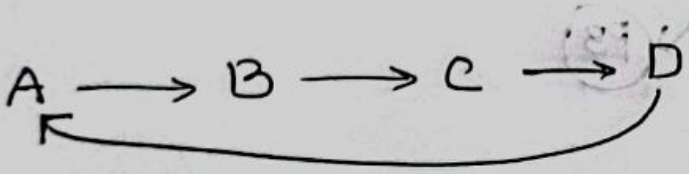
No hold and wait

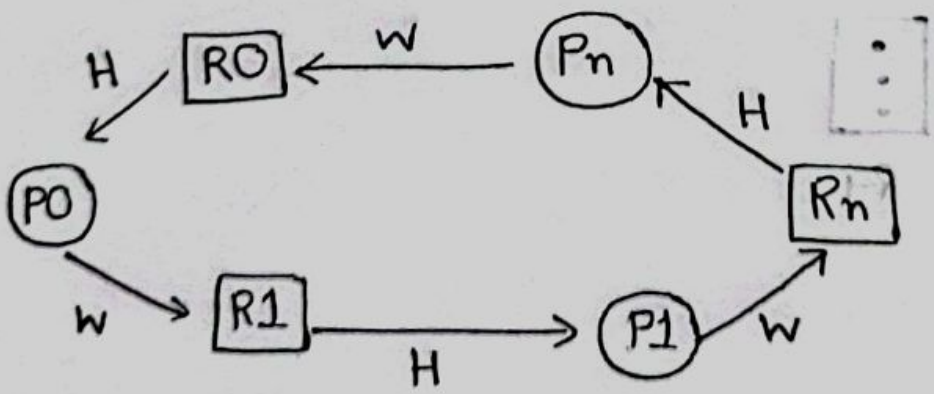
(III) No preemption



(IV) Circular wait

n number of resource and process

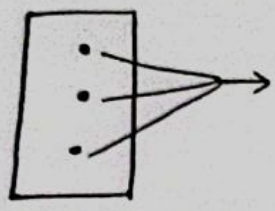




Resource allocation graph.

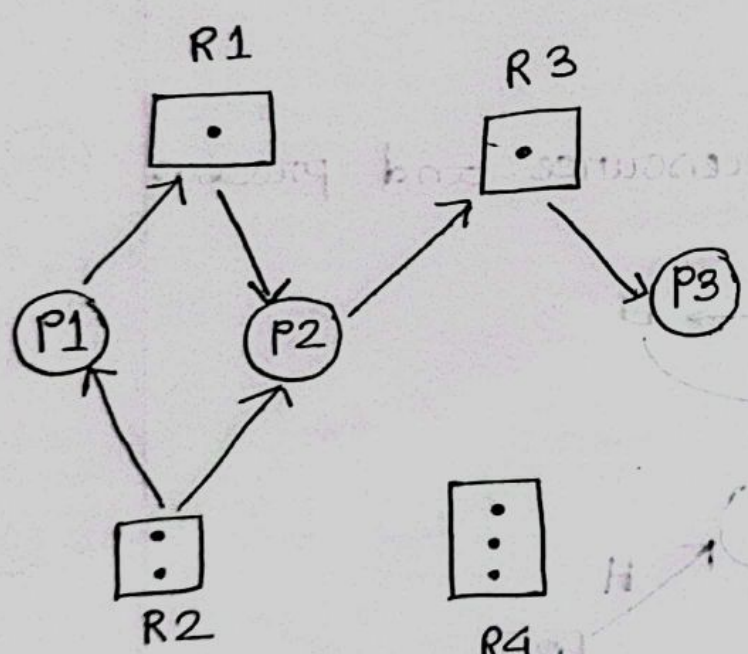
Request edge : $P_1 \rightarrow R_2$

Assignment edge : $R_1 \rightarrow P_2$



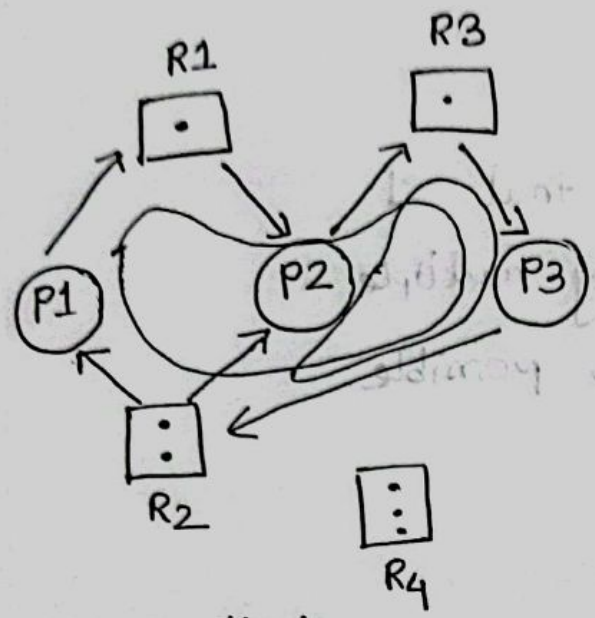
instances of resource

resource

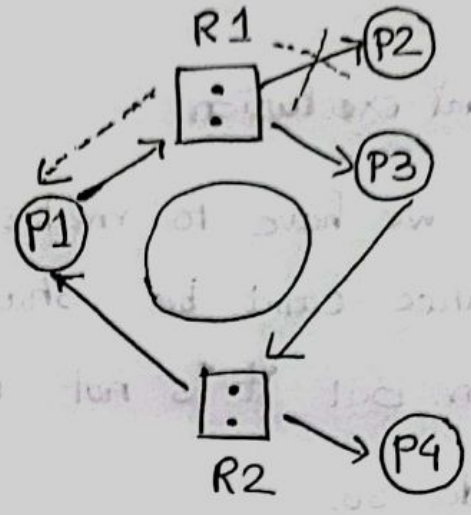


Cycle: Possibility of deadlock

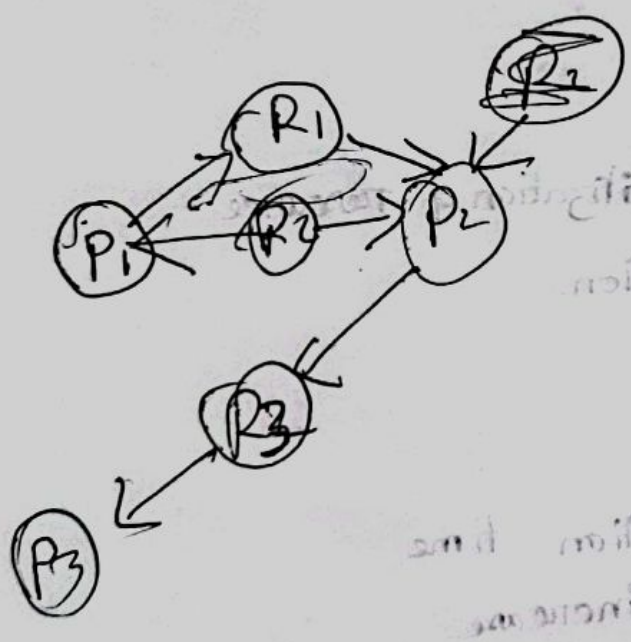
No cycle: No deadlock



Deadlock
2 cycles



Cycle, but no deadlock because P2 process do not request for any resource. So after some time, it will stop using R1 and release it. After that P1 process will be able to use R1 and break the cycle.



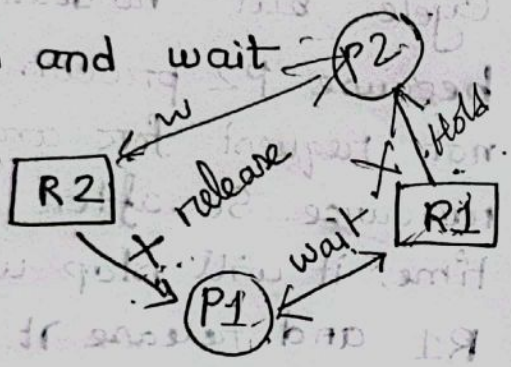
Methods for handling deadlock.

(1) Prevention

- Mutual exclusion

we have to make sure that 1 resource can't be shared by multiple procesn. But it is not always possible to do so.

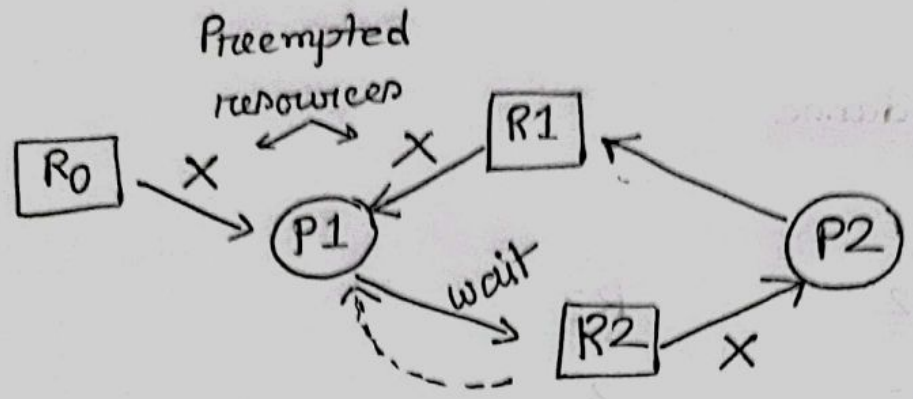
- Hold and wait



Disadvantage: - low utilization of resource.
- starvation.

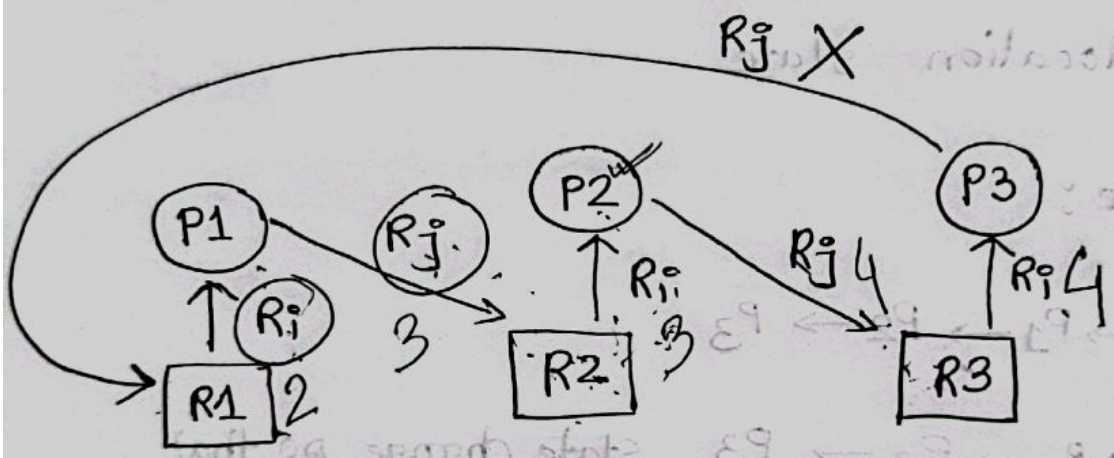
- No preemption

Disadvantage: - execution time may increase.



— Circular wait

- Break circle
- Most efficient



$R_i < R_j$ ✓

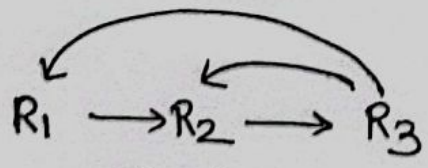
P1, R2

$R_i > R_j$ X

P3

$R_1 \rightarrow R_2 \rightarrow R_3$

Not circular



Circular.

Deadlock avoidance

	R1	R2	R3
	2	5	3
P1 =	1	2	1
P2 =	1	1	2

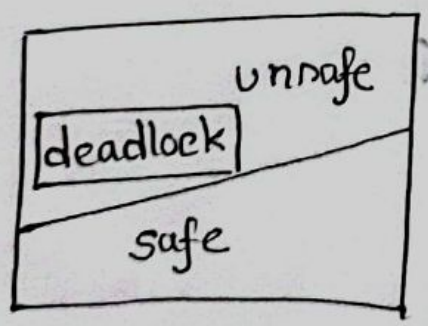
- Resource allocation state

- safe state :

$P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3$

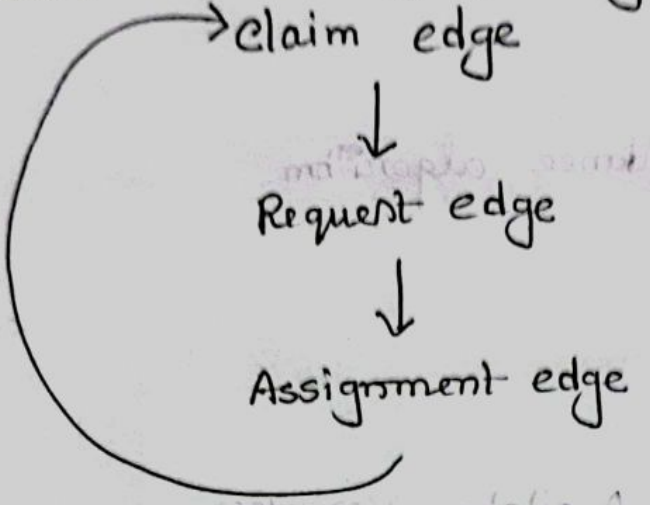
$P_1 \rightarrow P_2 \rightarrow P_0 \rightarrow P_3$ state change so that system comes in safe state

safe sequence



deadlock is a subset of unsafe state

Resource allocation graph algorithm.



Claim edge info দিয়ে আমরা বুঝতে পারি কখন কোন resource কে কোন process এ allocate করতে হবে যাতে cycle create না হয়। This is deadlock avoidance algo.

Banker's algorithm

Deadlock avoidance algorithm.

Available

R1	R2	R3
2	2	3

Availabe resources

Max

	R1	R2	R3
P1	2	3	0
P2	6	5	2

P1 কোন হতে max 2টা R1, 3টা R2 এবং 0টা R3 লাগে

Allocation

	R1	R2	R3
P1	1	0	0
P2	2	3	2

currently কয়টা resource use হচ্ছে

Need

	R1	R2	R3
P1	1	3	0
P2	4	2	0

বাকি কয়টা resource দরকার

If not enough resource available : deadlock

Example 1

$P1 \rightarrow P3 \rightarrow P4 \rightarrow P2 \rightarrow P0$
safe sequence

Available

R1	R2	R3
3	3	2

Max

	R1	R2	R3
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Allocation

	R1	R2	R3
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Need

	R1	R2	R3	
P0	7	4	3	x wait ✓
P1	1	2	2	✓
P2	6	0	0	x wait ✓
P3	0	1	1	✓
P4	4	3	1	✓

Work : (Copy of available)

3	3	2

Finish :

P0	P1	P2	P3	P4
F	F	F	F	F

F	T	F	F	F
P_0	P_1	P_2	P_3	P_4

3+2=5	3+0=3	2+0=2

F	T	F	T	F

5+2=7	3+1=4	2+1=3

F	T	F	T	T

7+0=7	4+0=4	3+2=5

T	T	F	T	T

7+0=7	4+1=5	5+0=5

T	T	T	T	T

7+3=10	5+0=5	5+2=7

Amount of total number of instances of resources

No deadlock

Example 2

	Allocation			Max			Available			Need			
	A	B	C	A	B	C	A	B	C	A	B	C	
P0	1	0	1	2	1	1	2	1	1	1	1	0	✓
P1	2	1	2	5	4	4				3	3	2	✗ → deadlock
P2	3	0	0	3	1	1				0	1	1	✓
P3	1	0	1	1	1	1				0	1	0	✓

Work:

2	1	1

2+1	1+0	1+1

3+3	1+0	2+0

6+1	1+0	2+1

7	1	3

Finish:

F	F	F	F
P0	P1	P2	P3

T	F	F	F
P0	P1	P2	P3

T	F	T	F
P0	P1	P2	P3

T	F	T	T
P0	P1	P2	P3

$P_0 \longrightarrow P_2 \longrightarrow P_3$

Avoid P1 because it is deadlock.

☑ Resource-request algorithm.

Process sends new resource request

→ Request <= Need

→ Request <= Available

→ Available = Available – Request

→ Allocation = Allocation + Request

→ Need = Need – Request

→ Run Bankers algo

→ if we find safe sequence → accept request

Example 3

Suppose in example 1, P1 requests more (1, 0, 2)

→ request <= Need → (1, 0, 2) ≤ (1, 2, 2) True

→ request <= Available → (1, 0, 2) ≤ (3, 3, 2) True

→ If both true : Available : (3, 3, 2) − (1, 0, 2)

= (2, 3, 0)

Allocation : (2, 0, 0) + (1, 0, 2)

= (3, 0, 2)

Need : (1, 2, 2) − (1, 0, 2)

= (0, 2, 0)

Available

2	3	0

	Allocation			Max			Need		
P0	0	1	0	7	5	3	7	4	3
P1	3	0	2	3	2	2	0	2	0
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

Work :

2	3	0

2+3	0+3	0+2

5+2	3+1	2+1

7+0	4+0	3+2

7+0	4+1	5+0

7+3	5+0	5+2

→

10	5	7

Finish :

F	F	F	F	F
P0	P1	P2	P3	P4

F	T	F	F	F

F	T	F	T	F

F	T	F	T	T

T	T	F	T	T

T	T	T	T	T

P1 → P3 → P4 → P0 → P2

Safe Sequence

→ If P4 requests for (3,3,0) in example 1

request <= need → (3,3,0) ≤ (4,3,1) → True

request <= available → (3,3,0) ≤ (3,3,2) → True

∴ Available: (3,3,2) − (3,3,0) = (0,0,2)

Allocation : (0,0,2) + (3,3,0) = (3,3,2)

Need : (4,3,1) − (3,3,0) = (1,0,1)

Available:

0	0	2

	Max			Allocation			Need		
P0	7	5	3	0	1	0	7	4	3
P1	3	2	2	2	0	0	1	2	2
P2	9	0	2	3	0	2	6	0	0
P3	2	2	2	2	1	1	0	1	1
P4	4	3	3	3	3	2	1	0	1

Work

0	0	2

Finish

F	F	F	F	F

Can't run Bankers algo.

→ If P0 requests for (0,2,0)

request <= need → (0,2,0) ≤ (7,4,3) → true

request <= available → (0,2,0) ≤ (3,3,2) → true

∴ Available : (3,3,2) − (0,2,0) = (3,1,2)

Allocation : (0,1,0) + (0,2,0) = (0,3,0)

Need : (7,4,3) − (0,2,0) = (7,2,3)

	Max			Allocation			Need					
P0	7	5	3	0	3	0	7	2	3	X	X	✓
P1	3	2	2	2	0	0	1	2	2	X	✓	—
P2	9	0	2	3	0	2	6	0	0	X	✓	—
P3	2	2	2	2	1	1	0	1	1	✓	—	—
P4	4	3	3	0	0	2	4	3	1	X	X	✓

Available : (3,1,2)

Work :

3	1	2
3+2	1+1	2+1
5+2	2+0	3+0
7+3	2+0	3+2
10+0	2+3	5+0
10+0	5+0	5+2
10	5	7

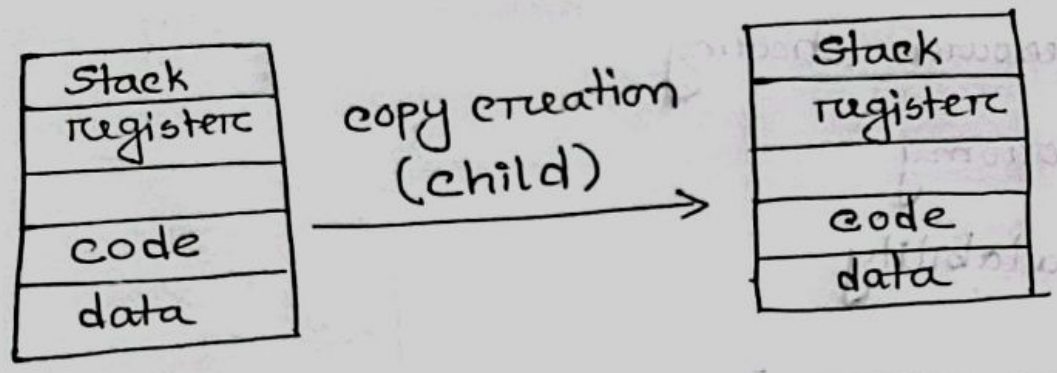
Finish :

P0	P1	P2	P3	P4
F	F	F	F	F
F	F	F	T	F
F	T	F	T	F
F	T	T	T	F
T	T	T	T	F
T	T	T	T	T

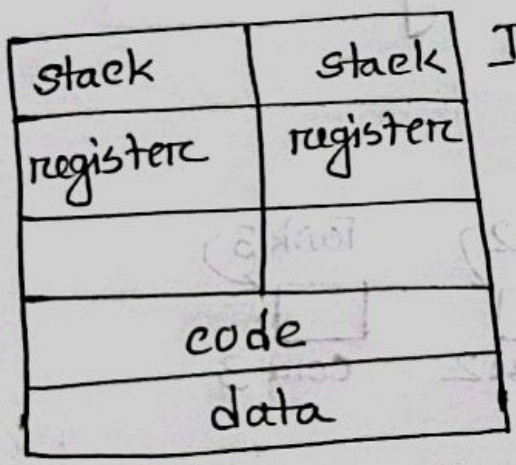
P3 ↓ P1 ↓ P2 ↓ P0 ↓ P4

Thread

- Basic unit of CPU utilization
  - Thread id
  - program counter
  - register set
  - Stack
- A process can have single/multiple threads
  - code section
  - data section
  - OS resources.



if 1 process goes to wait state, other process follows it.



Independent process

Shared resources

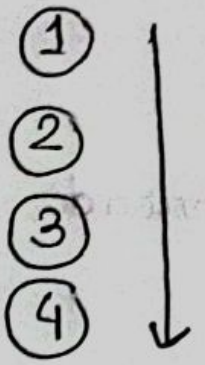
create own thread

Shared items: code, data, files

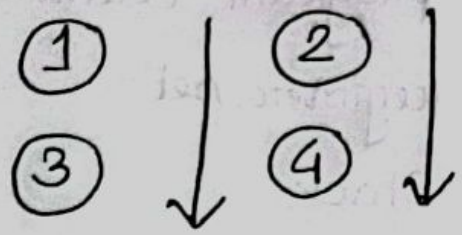
Individual items: register, stack.

☑ Suppose a process has to do 4 tasks:

Single thread:



Multi thread:

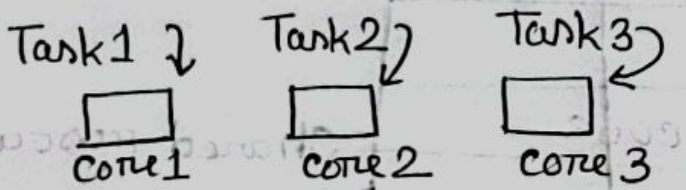


☑ Benefits

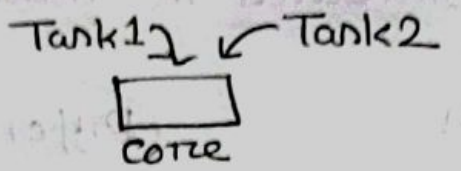
- responsiveness
- resource sharing
- economy
- scalability

☑ Multicore programing

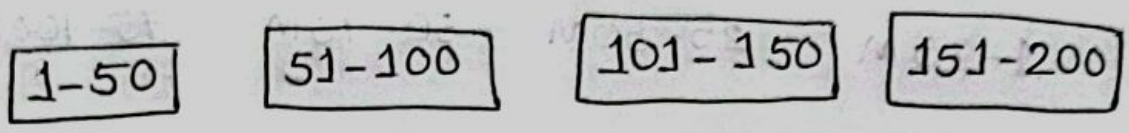
- parallelism



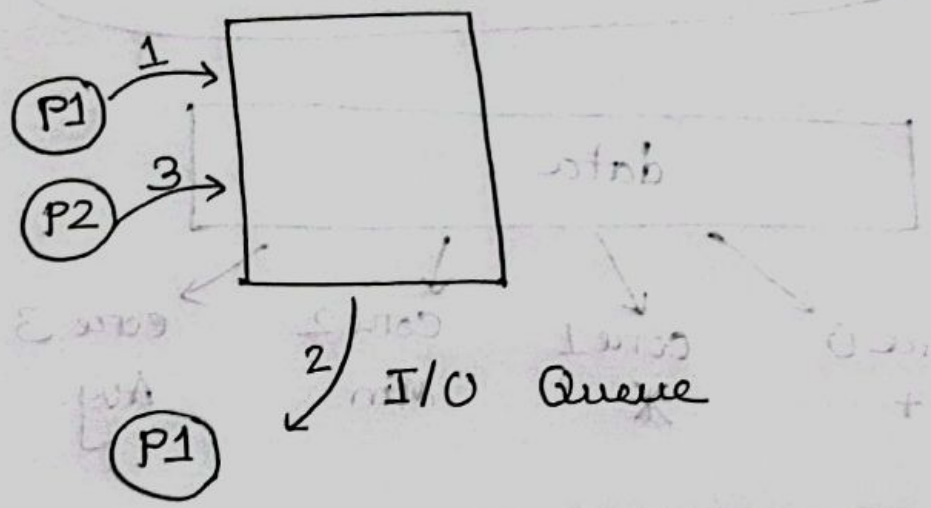
- Concurrency (scheduler provides concurrency)



→ parallel task : Add 1-200



Thread context switch is not expensive child process creatio doesn't cost bad.

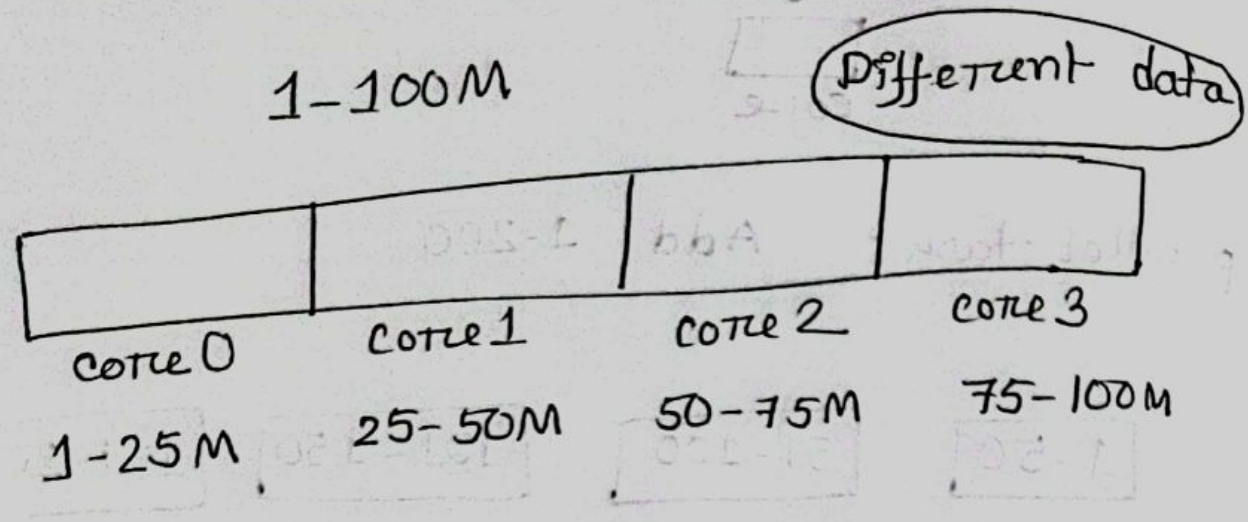


Single core ত threading possible না -

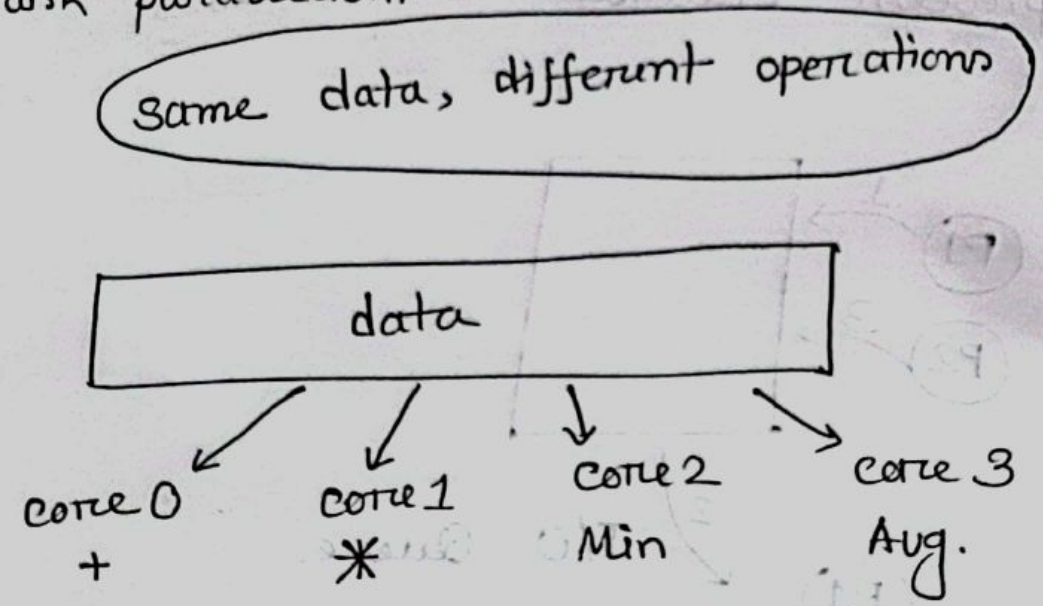
→ In slide, core 1 has 2 threads T1 and T3
core 2 " 2 " T2 " T4

Data Parallelism

1 operation distributed in multiple cores.



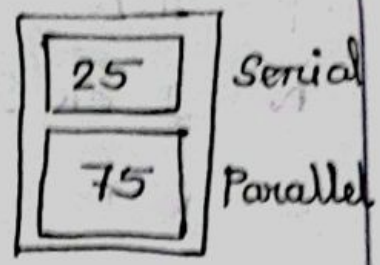
Task parallelism



☑ Amdahl's Law.

S → serial portion

N → Number of processing cores.



If N = 1, S = 0.25

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

$$= \frac{1}{0.25 + \frac{1-0.25}{1}} = 1$$

If N = 2,

$$\text{speedup} \leq \frac{1}{0.25 + \frac{(1-0.25)}{2}}$$

$$= 1.6$$

speedup of 1.6 times

If N is ∞, $\frac{1-S}{N}$ becomes 0. So speed depends only on S. Speed is inversely proportional to S.

→ P = 40%.

N = 2 or 8

S = 60%.

If N = 2; $\frac{1}{0.6 + \frac{1-0.6}{2}} = 1.25$

If N = 8; $\frac{1}{0.6 + \frac{1-0.6}{8}} = 1.5465$

$\frac{1.5465}{1.25} = 1.23$ times increase

Serial + parallel → N

Multithreading model

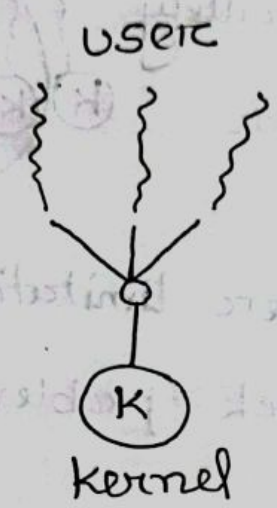
- user thread → function
- kernel thread → system call

} thread library.

4 models :

① Many to one

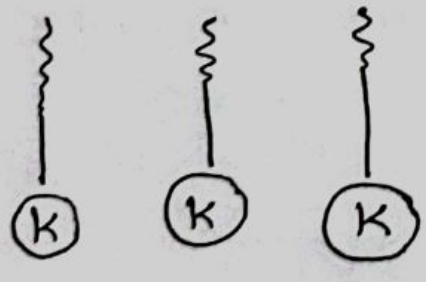
- If one user invokes block, whole system stops working



- kernel executes one task at a time. So no parallelism possible

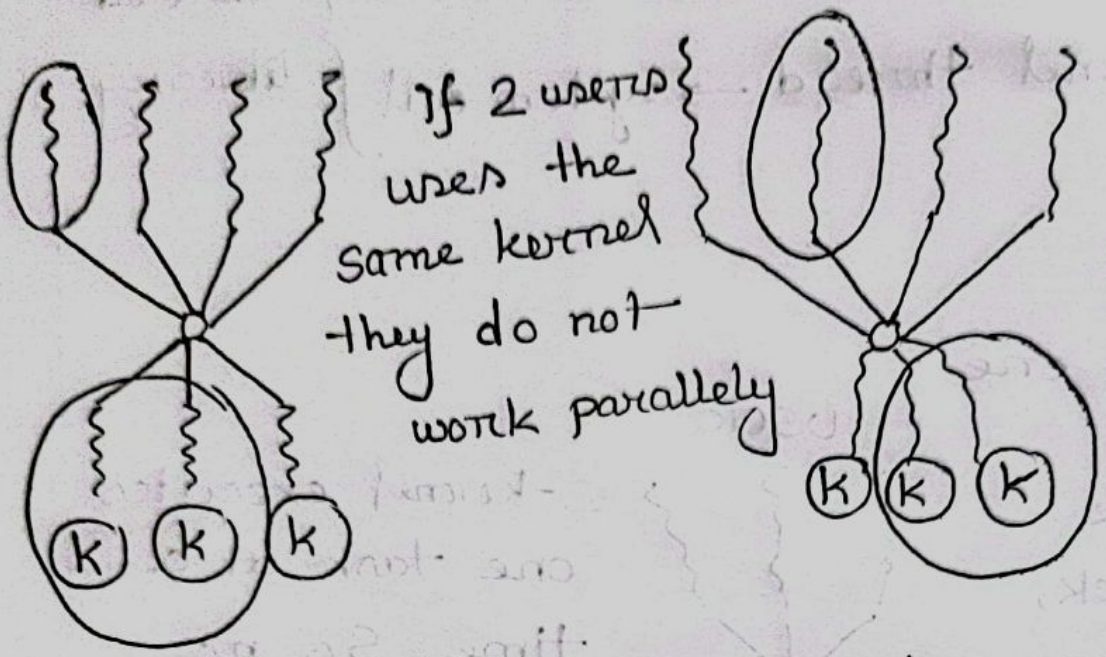
② One to One

windows, Linux



+ parallelism
- we can't create infinite user because we have limited kernel.
- overhead.

(III) Many to Many.

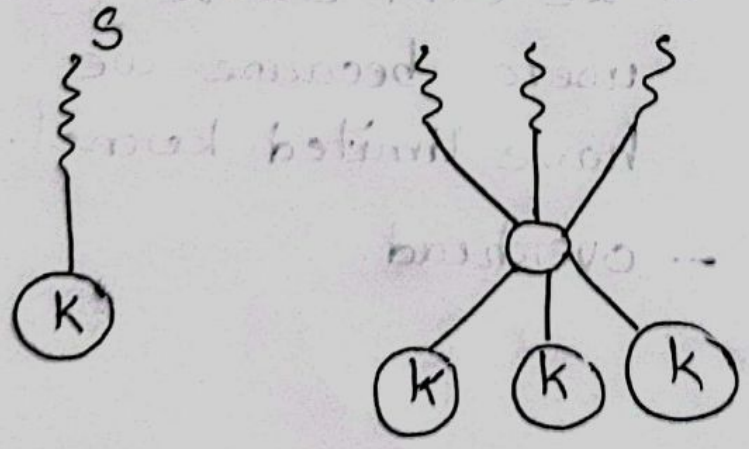


If 2 users uses the same kernel they do not work parallely

+ Solves users limitation
+ solves block problem
+ parallel
− Slowers

(IV) Two-level model

mix of (II) and (III)



☑ Thread Library

API → creates bridge between services.
provides API for creating / managing

☑ Java threads
managed by JVM

- extending thread class
- implementing the runnable interface (standard)

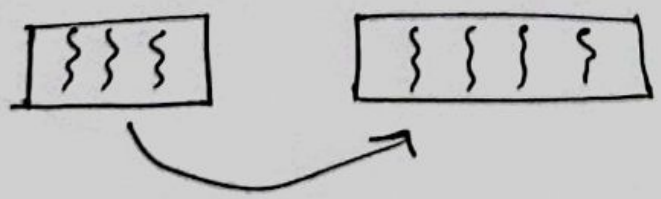
☑ Threading Issues

① fork() → duplicates
exec() → replace process

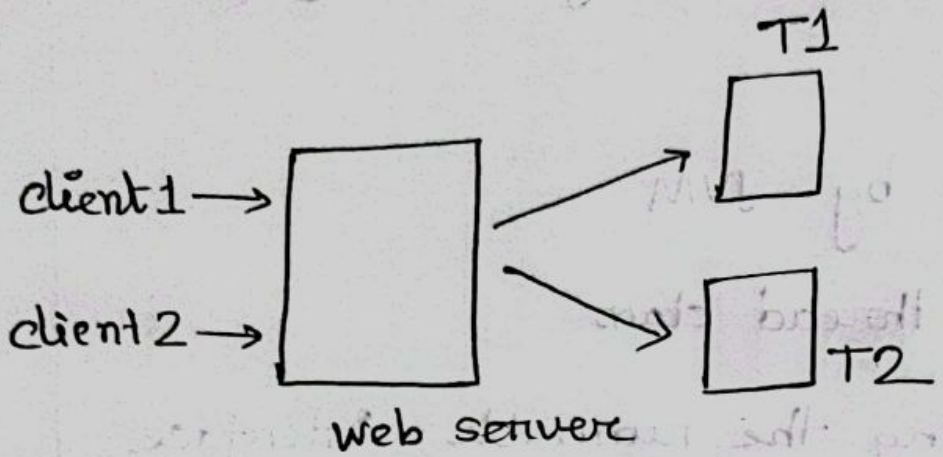
② Thread cancel → terminate thread

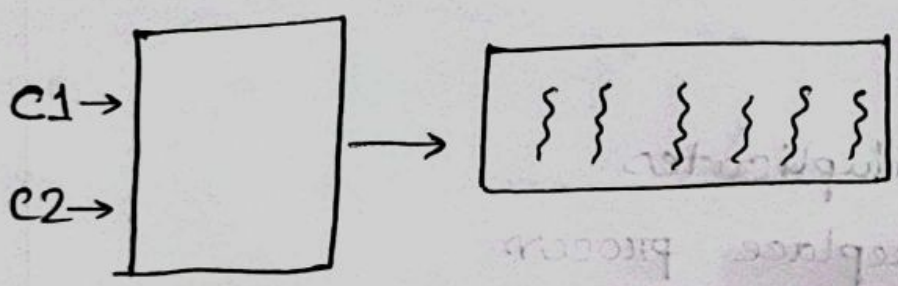
Asynchronized : call করার সাথে সাথে cancel
Deferred : wait till task is finished

③ ~~Single~~ Signal Handling :



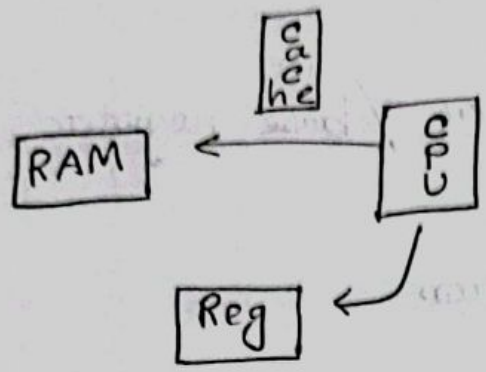
(iv) Thread pool → create a number of threads at the process start-up

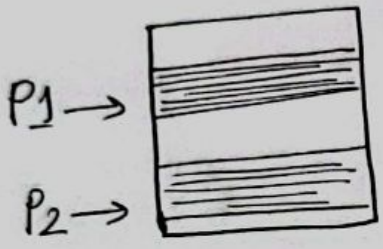




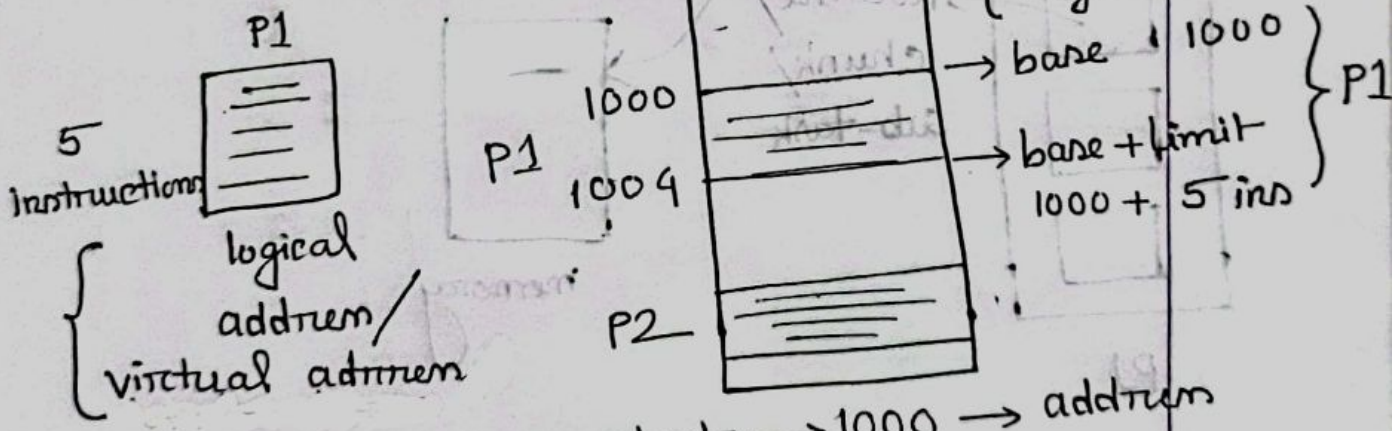
(v) Thread specific data → each thread might need it's own copy of certain data.

Main Memory

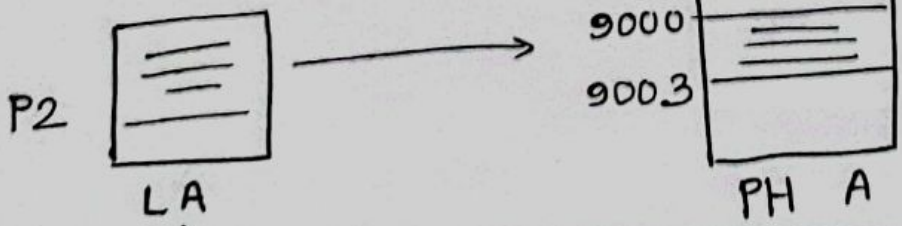




Protection has to be ensured so that P1 can never accen the adrren of P2



only OS can accen this.
- Base → addrren start → 1000 → addrren
- limit → 5 → integerc



↓

human readable errrore detection.

Memory management Unit

Relocation registers / base registers

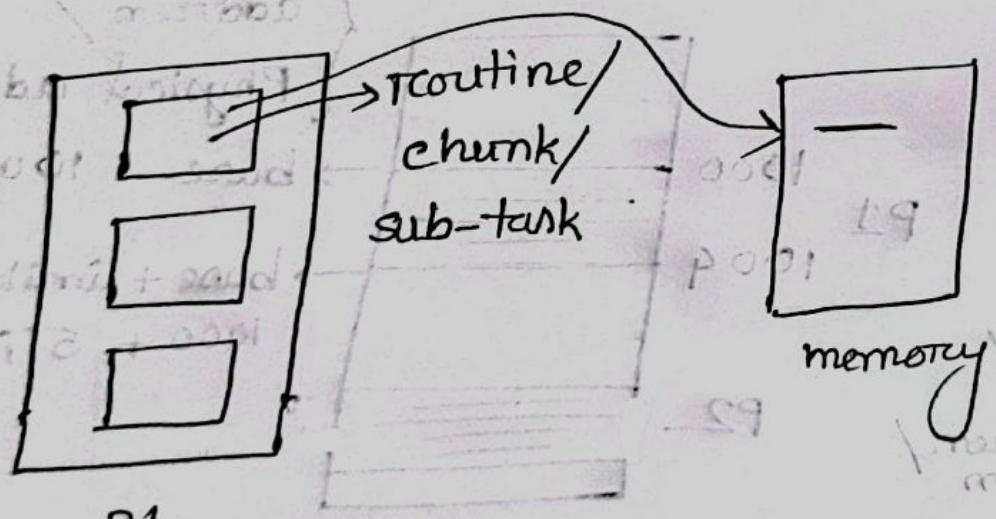
+

logical address

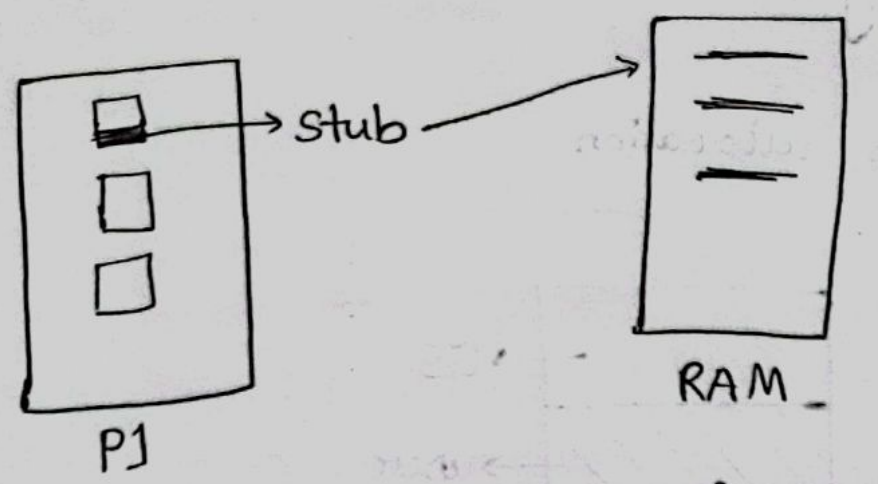
↓

physical address.

Dynamic loading

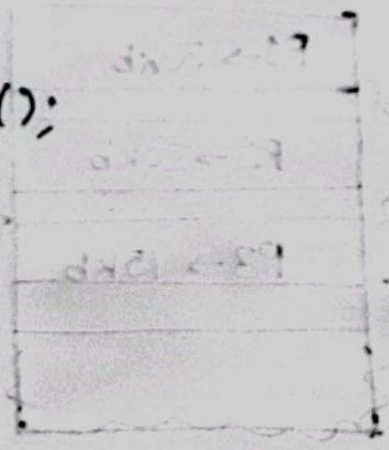


Dynamic Linking.

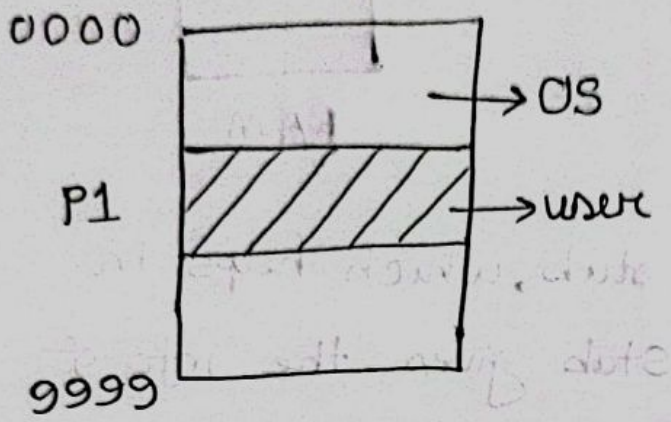


each routine has stub, which helps in dynamic linking. Stub gives the info of which function will be used now.

```
if (True){
    f1();
    }
else {
    f2();
    }
void function1();
```



☑ Contiguous allocation

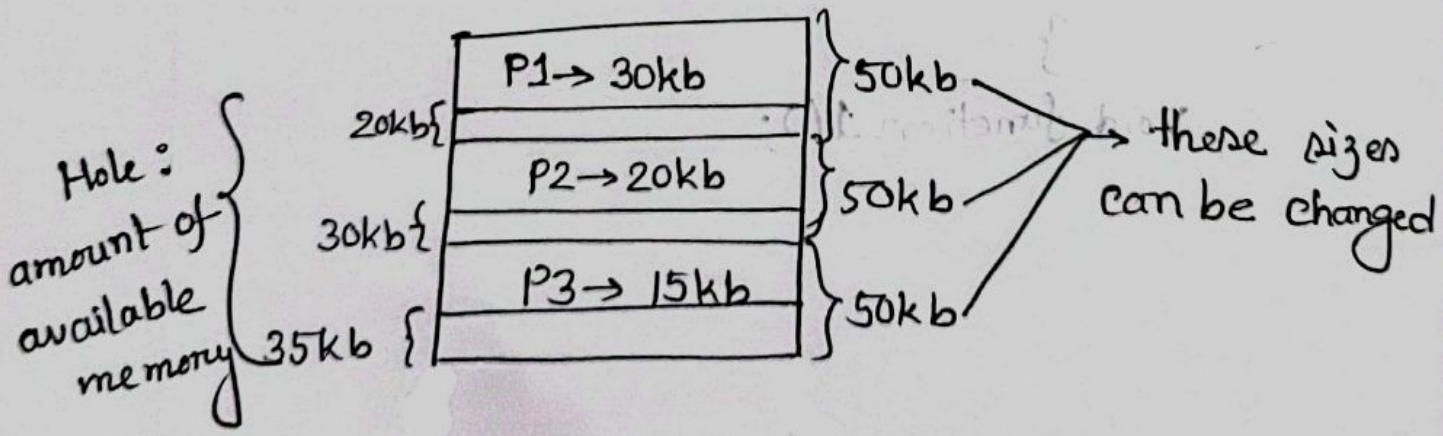


P1 has to be continuous, no break

Logical / local address

easier to find address because of C.A.

☑ ~~Mutual~~ Multiple partition allocation

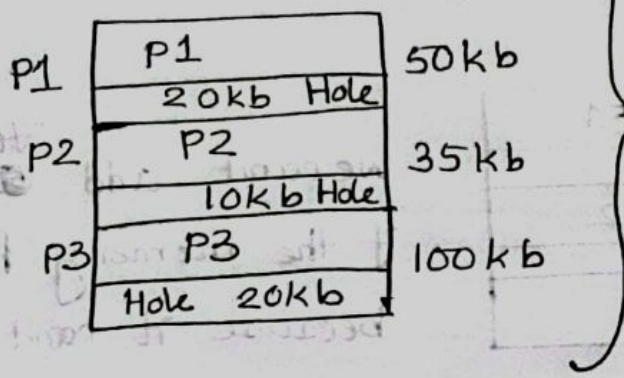


P4 → 50kb but CA is no possible.

Solution → variable partition

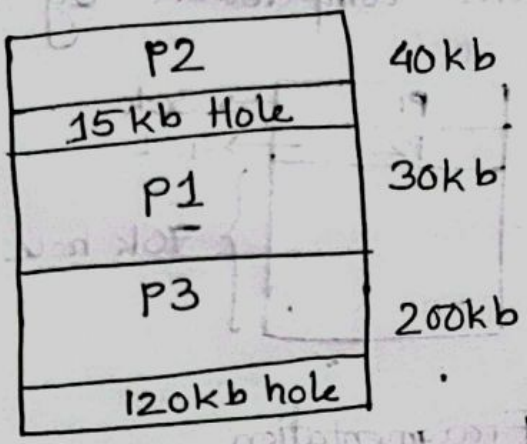
Dynamic storage allocation

P1 → 30kb
P2 → 25kb
P3 → 80kb



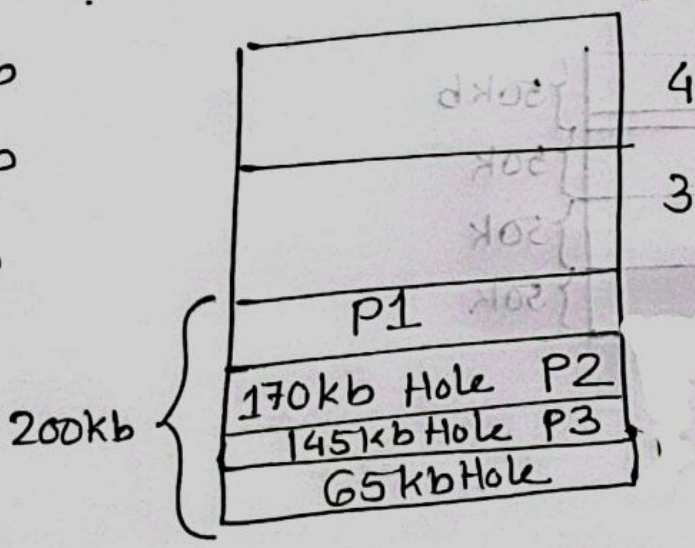
(I) First Fit

P1 → 30kb
P2 → 25kb
P3 → 80kb



(II) Best Fit

P1 → 30kb
P2 → 25kb
P3 → 80kb

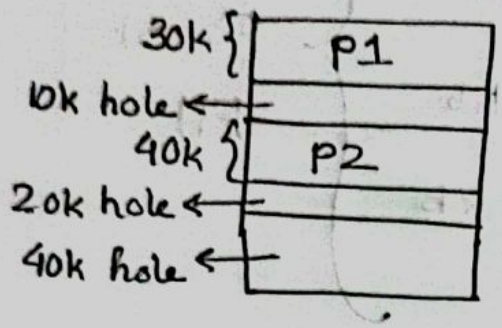


(III) Worst Fit

Problem : Fragmentation

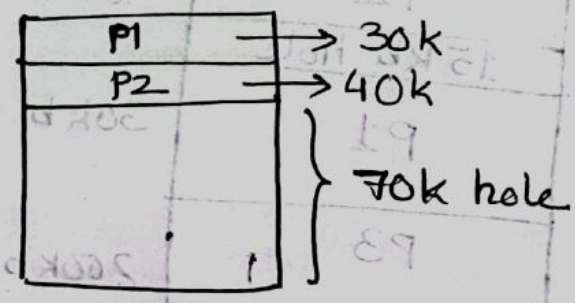
Fragmentation : 2 types

(I) External fragmentation

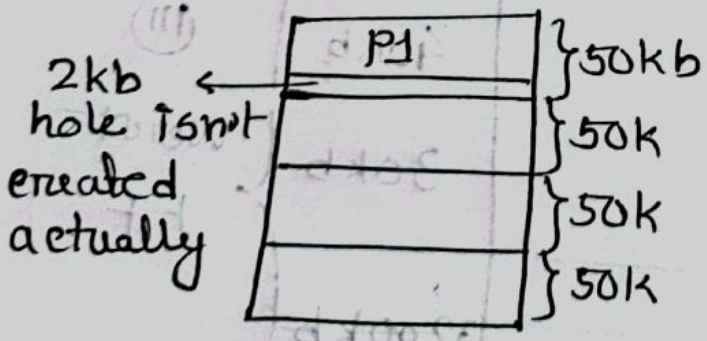


We can't add 70 kb even if the memory have space because it cant be C.A.

Solution: Compaction by shuffle

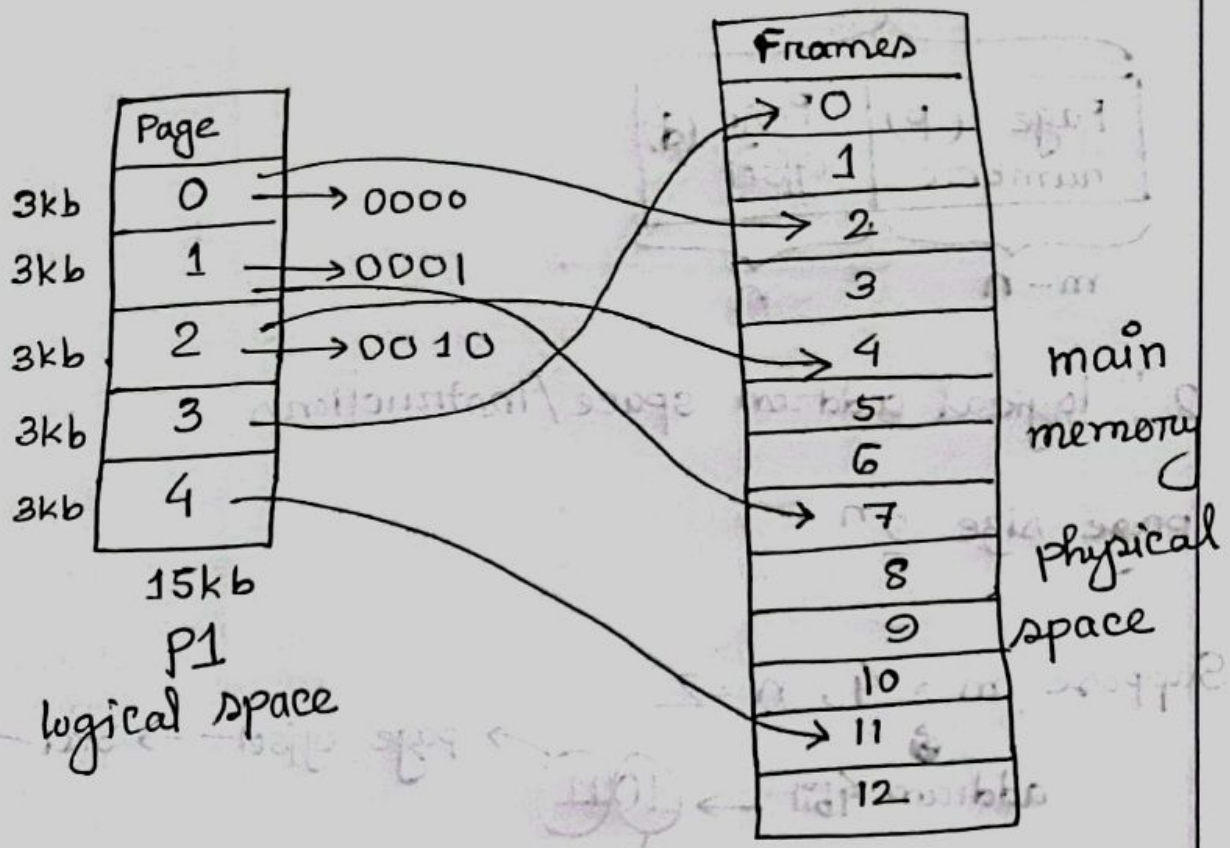


(II) Internal Fragmentation.



P1 → 48kb

Paging → non-contiguous technique → tough to locate



page table

P	F
0	2
1	7
2	4
3	0
4	11

Free frames

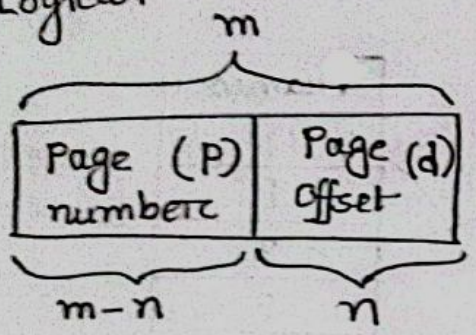
1
3
5
6
8
9
10
12

Solves external fr

but there might be internal fr

☒ Address translation scheme

Logical address:



2^m logical address space/instructions

page size 2^n

Suppose m = 4, n = 2

address 4bit → 1011 → page offset → 3

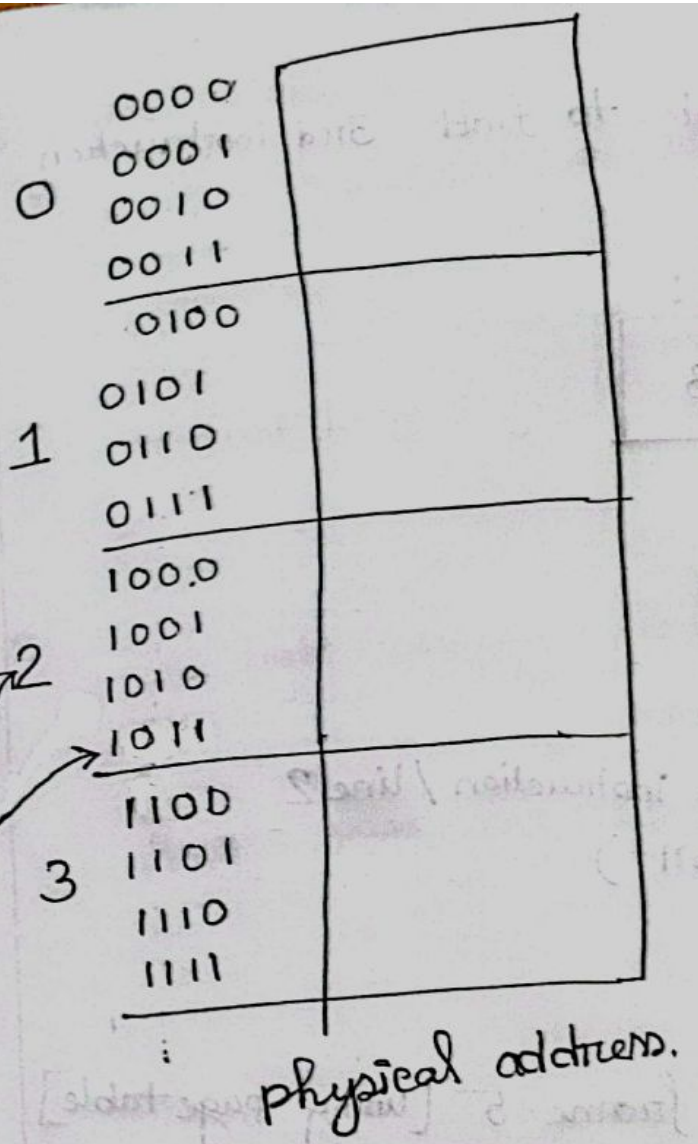
page number → 2 no ঘর এ আছে

$2^4 = 16$ address

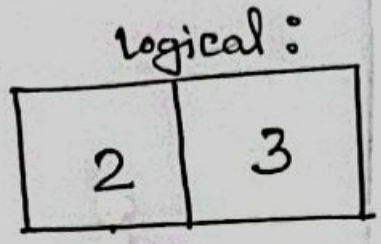
page size $2^2 = 4$ pages [কয় জায়গা ভাগ হচ্ছে]

↓

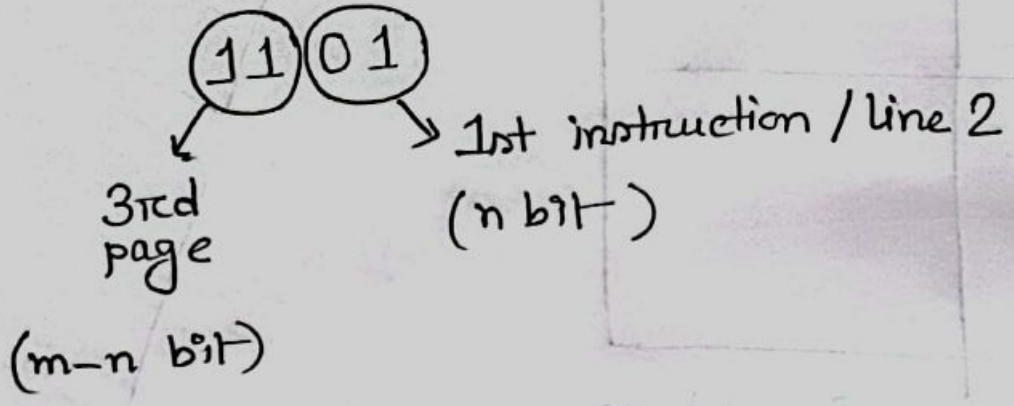
4 ঘর পর পর নতুন address



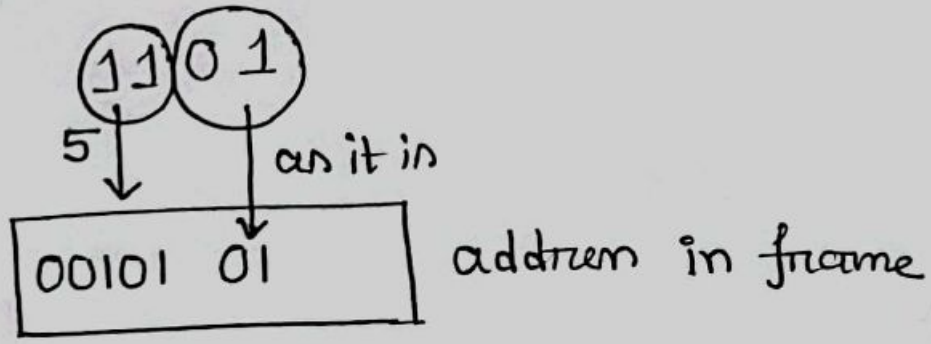
☒ Suppose we want to find 3rd instruction of page 2



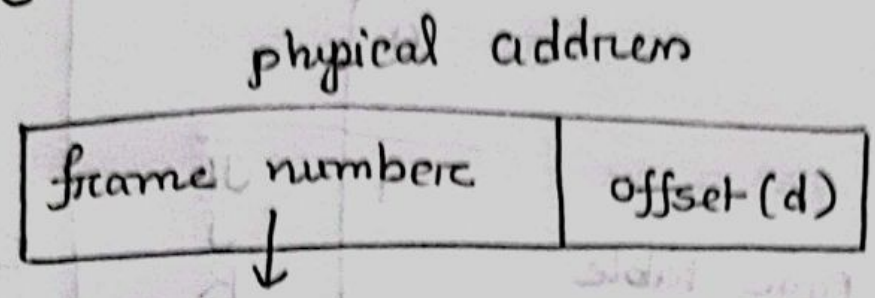
☒ Where is 1101?



Suppose 1101 is in frame 5 [using page table]



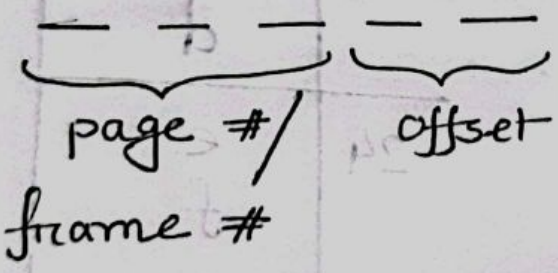
☒ Paging Hardware



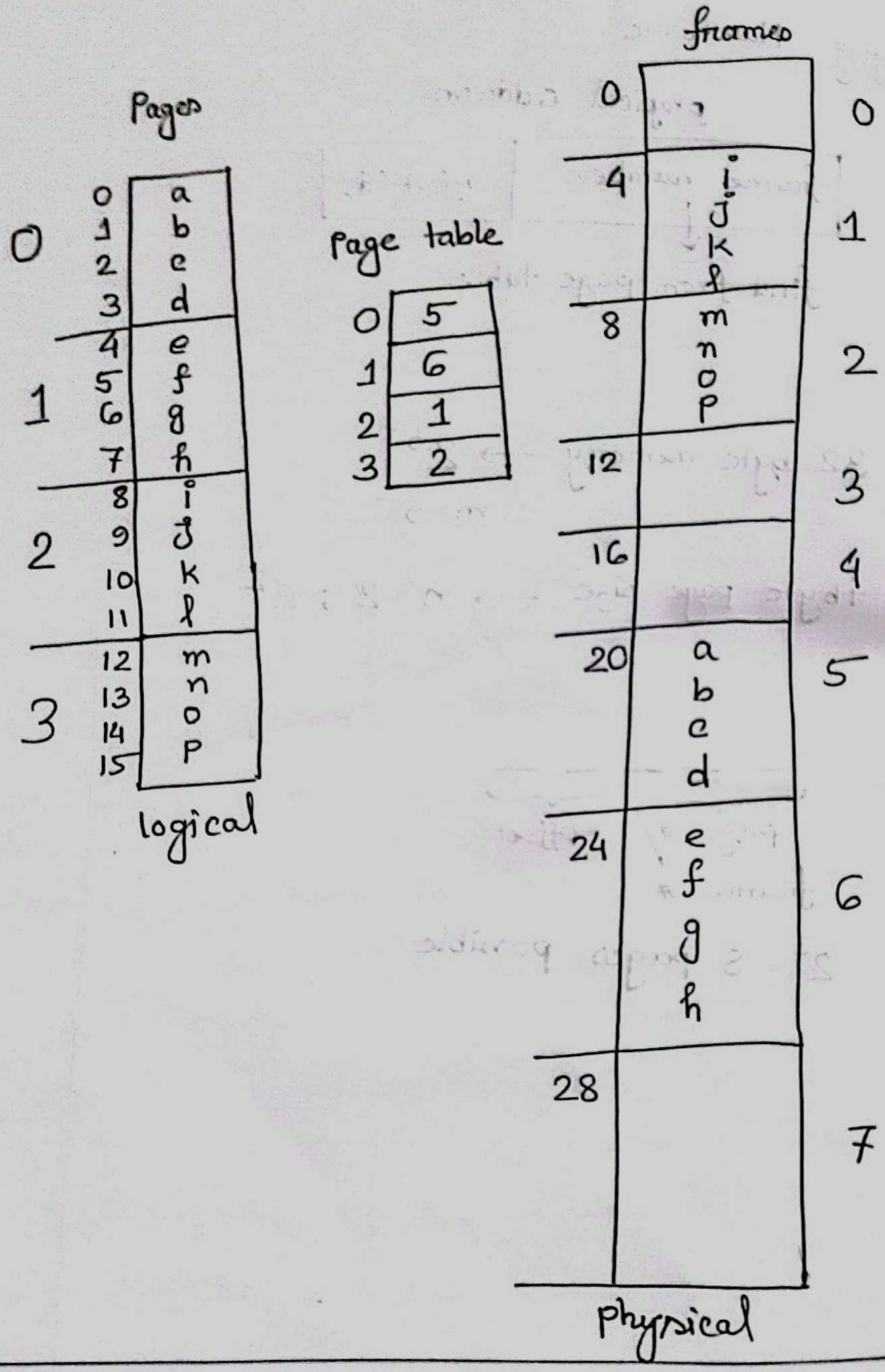
☒ 32 byte memory → 2^5

$\therefore m = 5$

4byte page size → $n = 2$; 2^2



$2^3 = 8$ pages possible


Pages
0
1
2
3
0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h
8 i
9 j
10 k
11 l
12 m
13 n
14 o
15 p
logical
Page table
0 5
1 6
2 1
3 2
frames
0
4 i j k l
8 m n o p
12
16
20 a b c d
24 e f g h
28
0
1
2
3
4
5
6
7
Physical

☑ Size of page

page size → 2048 bytes

process size → 72766 bytes

$\frac{72766}{2048}$ = 35 + 1 new frame

35 pages + 1086 bytes

Internal fragmentation : 2048 − 1086

= 962 bytes empty

If we reduce page size, we can reduce this waste but then page table size would increase, which also has to be stored in main memory. So we can't reduce page size as we want.

☑ Free frame

☑ Implementation of page table

solve repetetive search by cache memory (associative memory / translation look-aside buffers (TLBs))

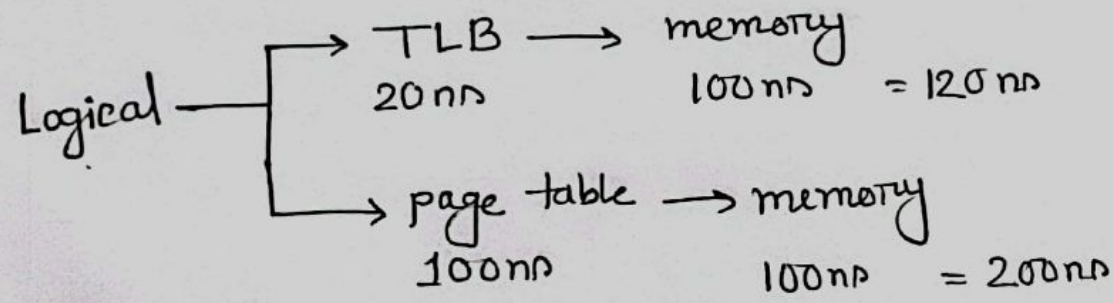
☑ Associative memory → do not need to go to page table.

☑ TLB → buffer.

Data থাকলে → TLB hit

না থাকলে → TLB miss

☑ Effective Accen Time (EAT)

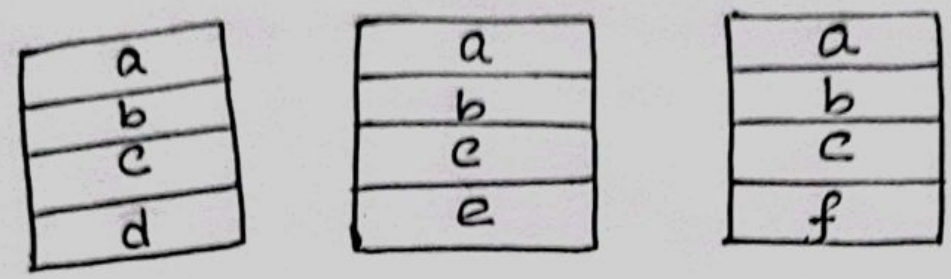


EAT : $\{(0.80 \times 120) + (0.2 \times 200)\}$

$= 136$ ns

α → Hit ratio, ε → cache accen.

Shared pages



Shared page slide example :

9 common page → 3 frame

memory oplimized

Virtual Memory

☑ virtual memory: process exceeds main memory, computers runs out of physical memory and writes its requirements to the hard disc in a swap file as 'VM'.

☑ Demand paging

paging system with swapping

page table with valid - invalid bit

secondary memory.

☑ Page fault

trap means the OS failed to bring the desired page into memory.

Access to a page marked invalid causes page fault. Trap → set → 0

- valid → loaded in memory frame
- invalid → not in logical address space of process / valid but currently on disk and hasn't been loaded.

* How to solve page fault

Page replacement Algorithm

① FIFO

time	1	2	3	4	5	6	7	8	9	10	11	12
page	P2	P3	P2	P1	P5	P2	P4	P5	P3	P2	P5	P2
	P2	P2	P2	P2	P5	P5	P5	P5	P3	P3	P3	P3
		P3	P3	P3	P3	P2	P2	P2	P2	P2	P5	P5
				P1	P1	P1	P4	P4	P4	P4	P4	P2
	*	*	hit	*	*	*	*	hit	*	hit	*	*

main memory size → 3 ⟶ frame

pages → 12

hit ratio : $\frac{3}{12} \times 100\%$

=

fault → 9

LRU

time →	1	2	3	4	5	6	7	8	9	10	11	12
page	P2	P3	P2	P1	P5	P2	P4	P5	P3	P2	P5	P2
	P2	P2	P2	P2	P2	P2	P2	P2	P3	P3	P3	P3
		P3	P3	P3	P5	P5	P5	P5	P5	P5	P5	P5
				P1	P1	P1	P4	P4	P4	P2	P2	P2
	*	*	hit	*	*	hit	*	hit	*	*	hit	hit

hit ratio : $\frac{5}{12} \times 100\%$

fault → 7

Optimal

time →	1	2	3	4	5	6	7	8	9	10	11	12
page	P2	P3	P2	P1	P5	P2	P4	P5	P3	P2	P5	P2
	P2	P2	P2	P2	P2	P2	P4	P4	P4	P2	P2	P2
		P3	P3	P3	P3	P3	P3	P3	P3	P3	P3	P3
				P1	P5	P5	P5	P5	P5	P5	P5	P5
	*	*	hit	*	*	hit	*	hit	hit	*	hit	hit

hit ratio : $\frac{6}{12} \times 100\%$

fault → 6

* hit ratio বেশি হওয়া ভালো।

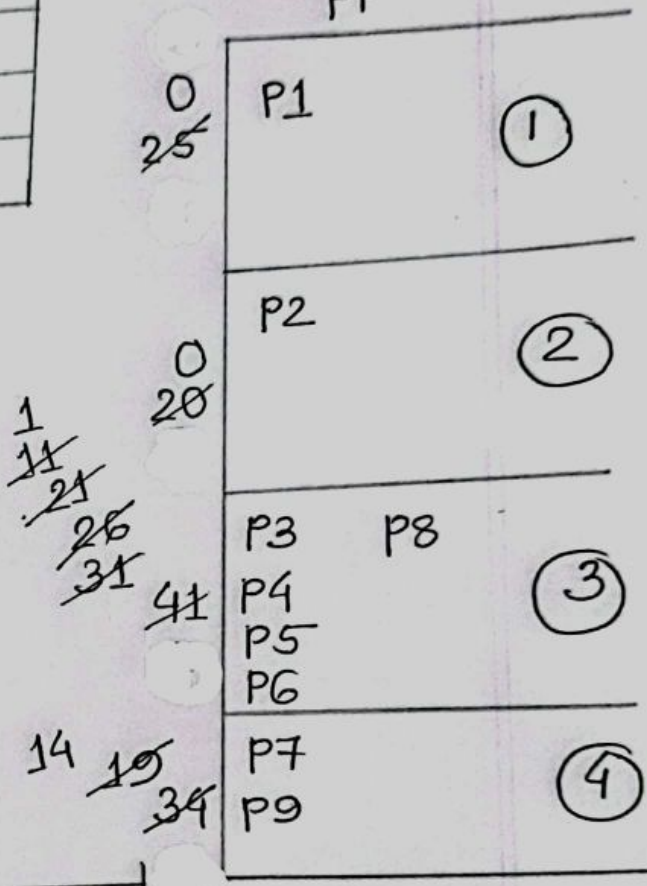
P5	P5	P1	P5	P2	P1	P3	P3	P0	P7	P0	P1	FIFO
P5	P5	P5	P5	P5	P5	P5	P5	P0	P0	P0	P0	5 hits
		P1	P1	P1	P1	P1	P1	P1	P7	P7	P7	
				P2	P2	P2	P2	P2	P2	P2	P1	
						P3	P3	P3	P3	P3	P3	
	hit		hit		hit		hit			hit		
P5	P5	P1	P5	P2	P1	P3	P3	P0	P7	P0	P1	LIFO
P5	P5	P5	P5	P5	P5	P5	P5	P0	P0	P0	P0	
		P1	P1	P1	P1	P1	P1	P1	P1	P1	P1	6 hits
				P2	P2	P2	P2	P2	P7	P7	P7	
						P3	P3	P3	P3	P3	P3	
	h		h		h		h			h	h	
P5	P5	P1	P5	P2	P1	P3	P3	P0	P7	P0	P1	Optimal
P5	P5	P5	P5	P5	P5	P5	P5	P0	P0	P0	P0	
		P1	P1	P1	P1	P1	P1	P1	P1	P1	P1	6 hits
				P2	P2	P2	P2	P2	P7	P7	P7	
						P3	P3	P3	P3	P3	P3	
	h		h		h		h			h	h	

Dynamic Storage Allocation : First Fit, Best Fit, Worst Fit

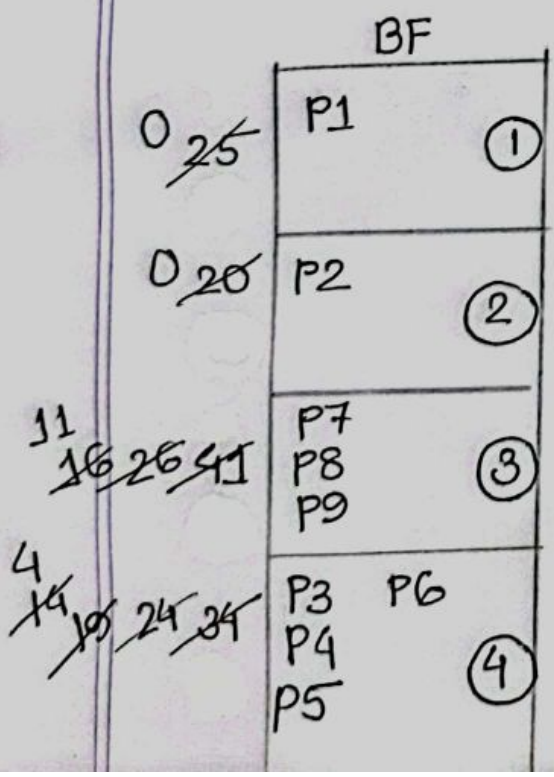
P	Size	FF	BF	WF
P1	25	1	1	3
P2	20	2	2	4
P3	10	3	4	1
P4	5	3	4	2
P5	5	3	4	3
P6	10	3	4	1
P7	15	4	3	2
P8	10	3	3	4
P9	5	4	3	3
P10	15	N/a	N/a	N/a

25
20
41
34

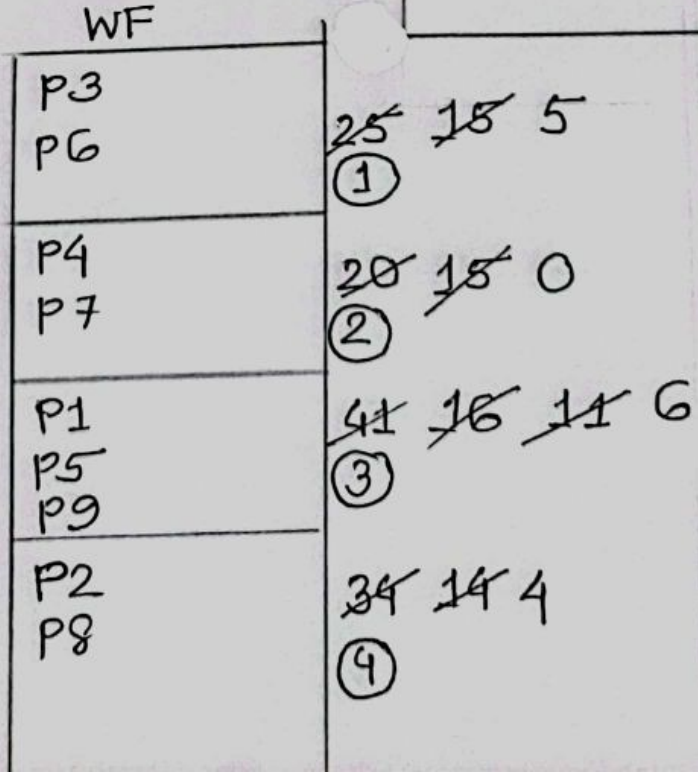
FF



BF



WF

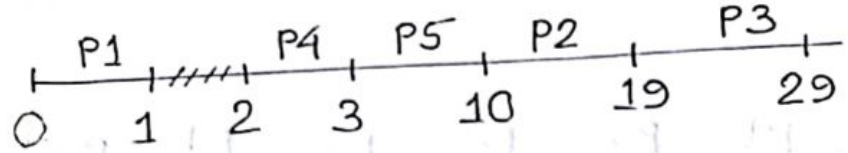


CSE 321

Scheduling Algorithms

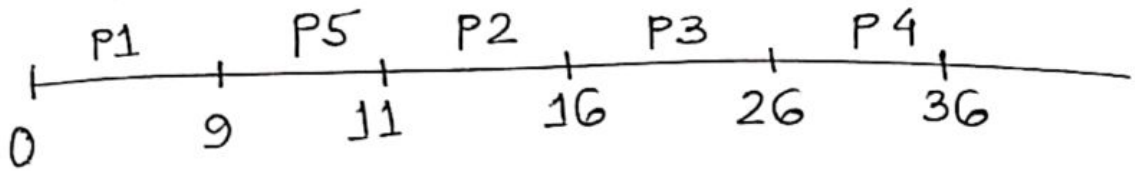
☒ FCFS

PID	AT	BT	WT	TT
P1	0	1	0	1
P2	3	9	7	16
P3	9	10	10	20
P4	2	1	0	1
P5	2	7	1	8



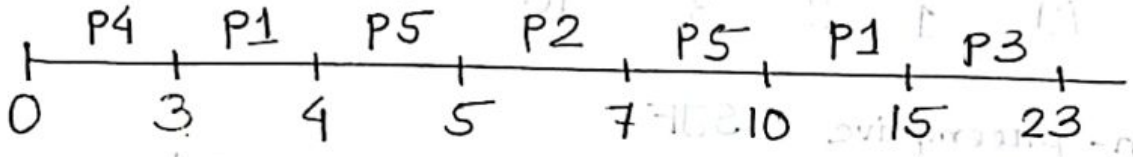
☒ Non-preemptive SJF

PID	AT	BT	WT	TT
P1	0	9	0	9
P2	2	5	9	14
P3	2	10	14	24
P4	1	10	25	35
P5	3	2	6	8



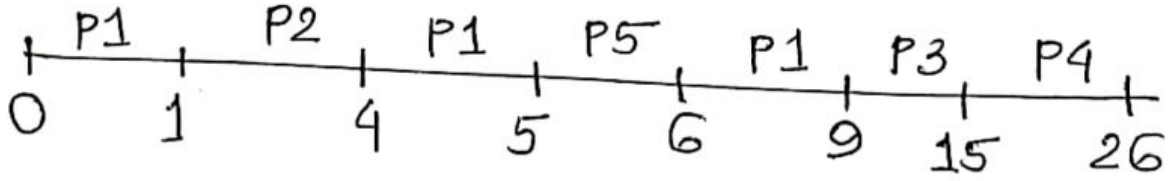
Preemptive SJF

PID	AT	BT	WT	TT
P1	2	6 8 0	1+6	13
P2	5	2 0	0	5
P3	1	8 0	14	22
P4	0	3 0	0	3
P5	4	4 3 0	2	6



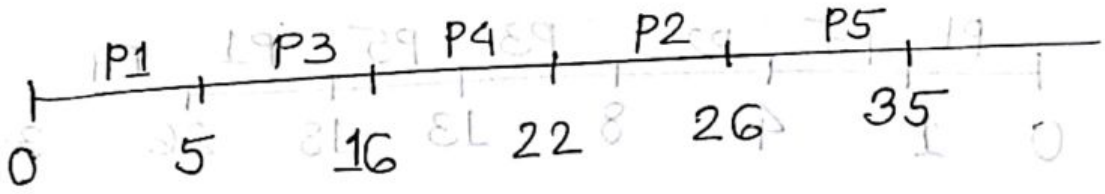
Preemptive SRTF

PID	AT	BT	WT	TT
P1	0	5 4 3 0	0+3+4 = 4	9
P2	1	3 0	0	3
P3	1	6 0	8	14
P4	5	1 1 0	10	21
P5	5	1 0	0	1



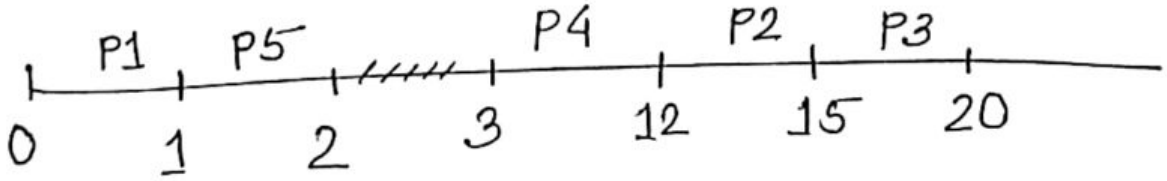
Non-Preemptive Priority Queue

PID	AT	BT	PQ	WT	TT
P1	0	5	2	0	5
P2	2	9	4	20	24
P3	2	11	1	3	14
P4	1	6	3	15	21
P5	5	9	5	21	30



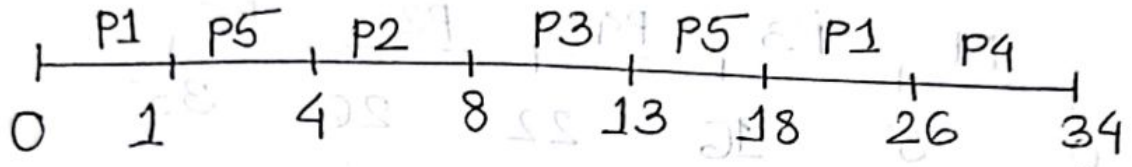
Preemptive Priority Queue

PID	AT	BT	PQ	WT	TT
P1	0	1	3	0	1
P2	4	3	4	8	11
P3	5	5	5	10	15
P4	3	9	1	0	9
P5	1	1	2	0	1



Preemptive Priority Queue.

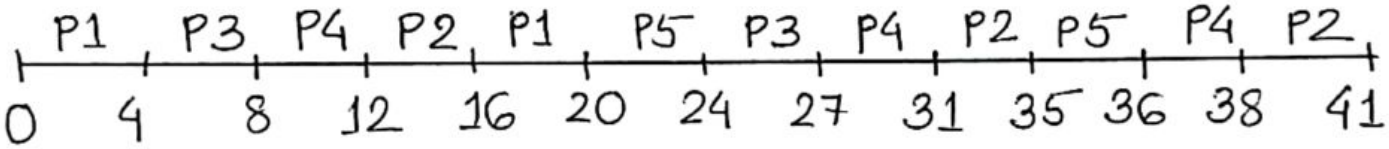
PID	AT	BT	PQ	WT	TT
P1	0	9 8	4	17	26
P2	4	4 0	2	0	4
P3	8	5 0	1	0	5
P4	5	8	5	21	29
P5	1	8 5	3	9	17



☐ Round Robin

$q = 4$

PID	AT	BT	WT	TT
P1	0	8 4 0	12	20
P2	3	11 7 3 0	27	38
P3	2	7 3 0	18	25
P4	2	10 6 2 0	26	36
P5	6	5 1 0	25	30



P1 P3 P4 P2 P1 P5 P3 P4 P2

P5 P4 P2