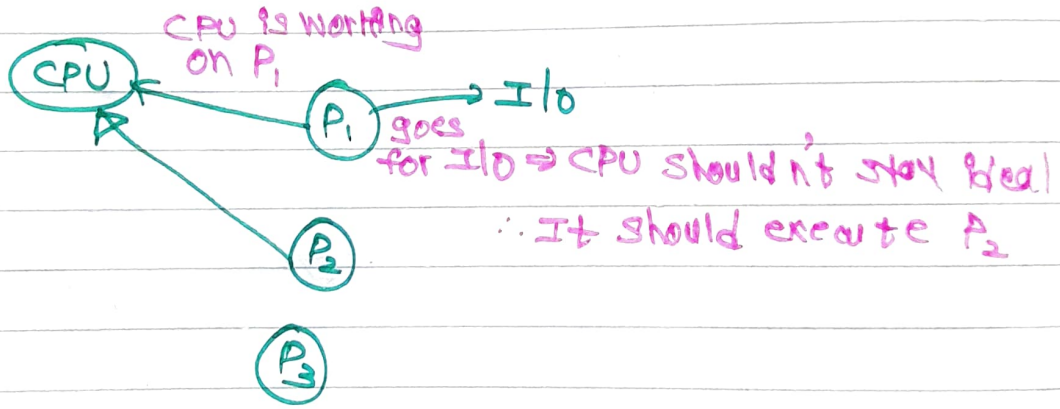
MAKE DIAGRAMS !!!!!

Love Bobbar

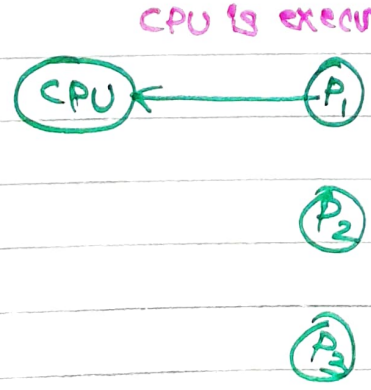
- What if there is no OS ?
  - Bulky & Complex app
  - Resource exploitation by 1 app
  - No Memory Protection

- OS goals :-

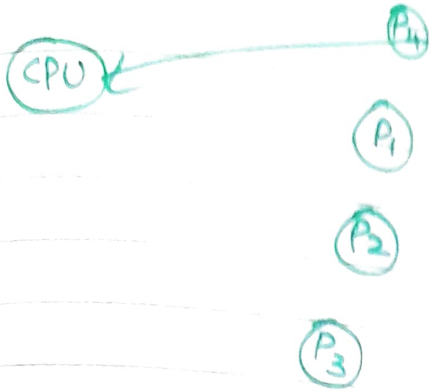
1. Max. CPU Utilization



2. Process Starvation X

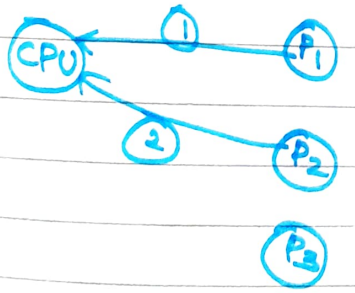


(3) High Priority Job execution (Eg Antivirus Scan)



- Types of OS:-

1. Single Process OS → Only 1 Process executes at a time from Ready Queue



→ Max. CPU Utilization ✗ since if P_1 goes for I/O, CPU can't execute P_2

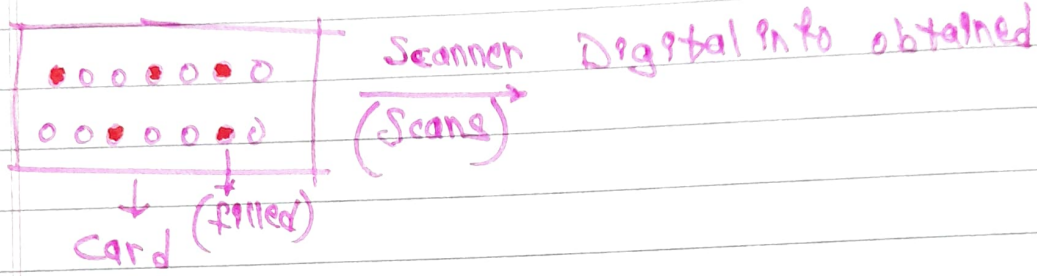
→ Process Starvation ✓ → If P_1 is long job, P_2 wont get a chance

→ H.P.J.E.✗ since P_1 keeps executing & H.P.J. doesn't get a chance

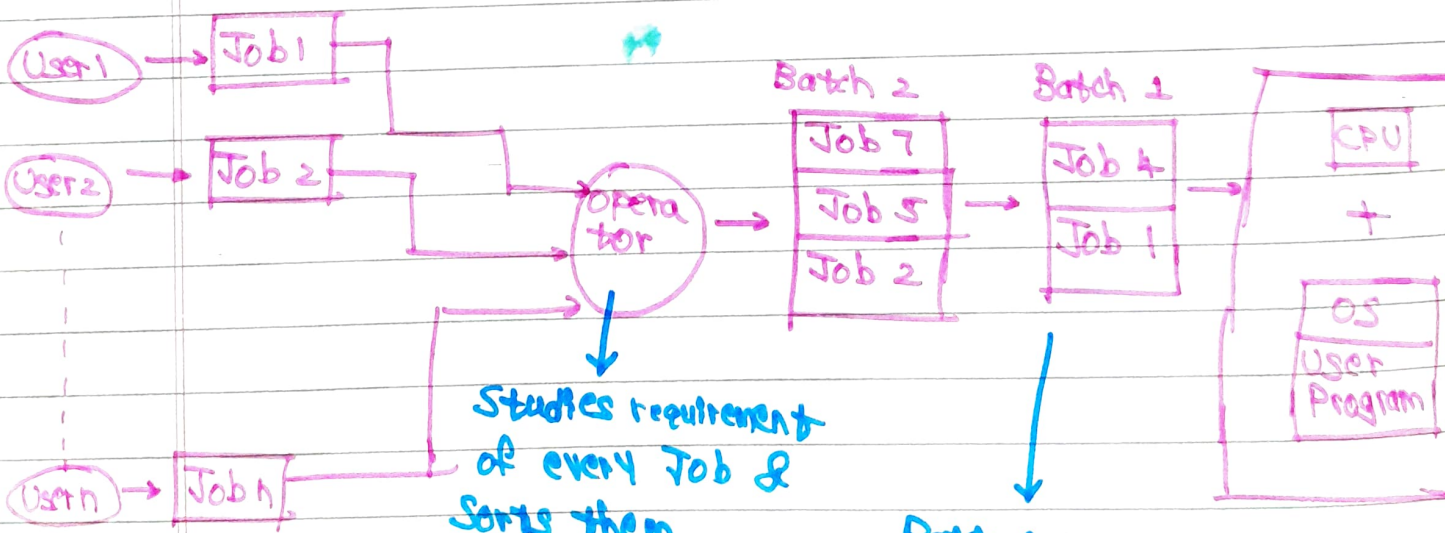
Eg MSDOS

2. Batch Processing OS →

Punch card



Here, user brings OS in form of a Punch card



→ Max. CPU Util X as each Batch is executed sequentially (& here if J_1 takes time, J_4 has to wait)

→ P.S. ✓ (Same reason)

→ Batch Starvation ✓ (If B-1 takes time)

→ H.P.J.E.X (Till B-1 is executing, B-2 can't be executed)

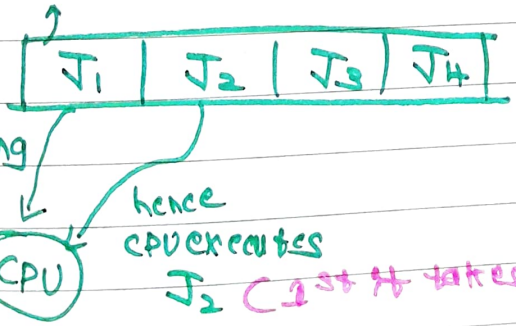
Eg ATLAS

3. Multi Programming OS →

→ Single CPU

• Ready Queue → Queue containing Jobs which are ready to be executed

Now goes for I/O → Enters Wait State ⇒ (Save its current content) in PCB



Process Control Block

→ This save & restore Process → Context Switching

Eg THE

4. Multi-Tasking OS

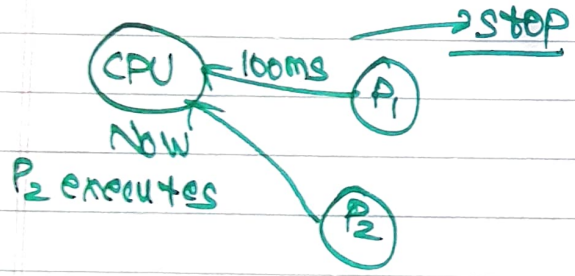
→ Single CPU

→ Context Switching

→ Time Sharing

⇓

Eg Time Quantum = 100ms
If any job executes for 100ms, 1 Quantum of execution is completed & next job should get a chance



→ Max CPU Util. ✓ P.S.X H.P.J.E. ✓ (If P_1 is executing, its current state is stored in PCB & High Priority Job executes)

Eg CTSS

5. Multi-processing OS

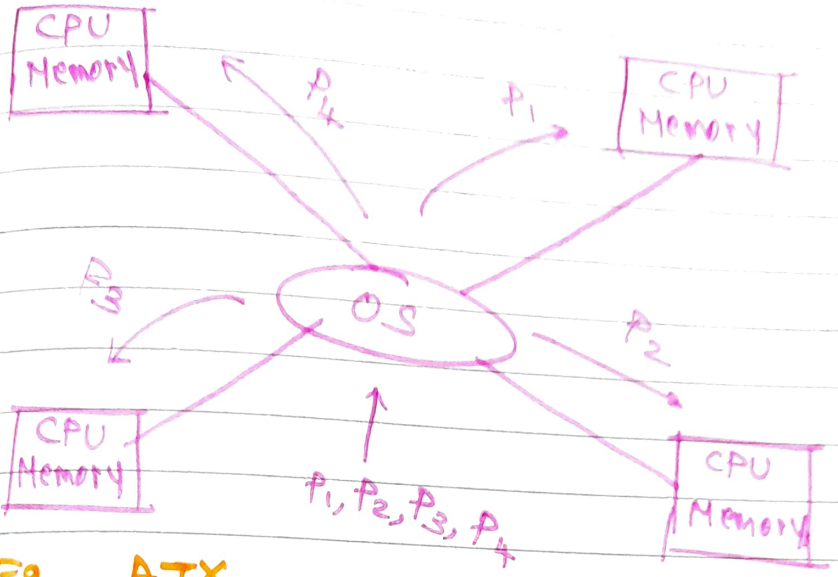
→ Context Switching, Time sharing, CPU ↑ 1

→ Increased Reliability, if 1 CPU fails, other is present

Eg Windows

6. Distributed OS

→ Loosely Coupled Autonomous interconnected computers



Eg AIX

7. Real Time OS (RTOS)

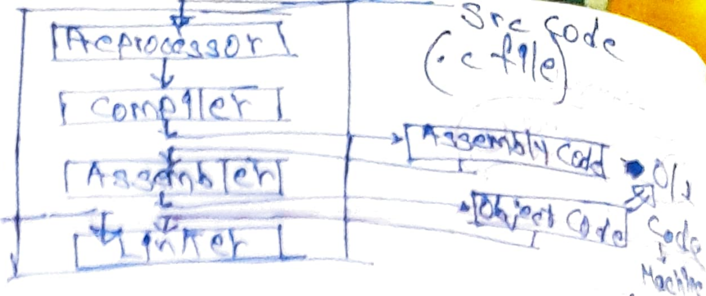
→ Used where Error chance ✗ & Speed of execution ↑ of Job

Eg ATC → Crucial Duty
Nuclear Plant

Eg Integrity

Process → Program under execution

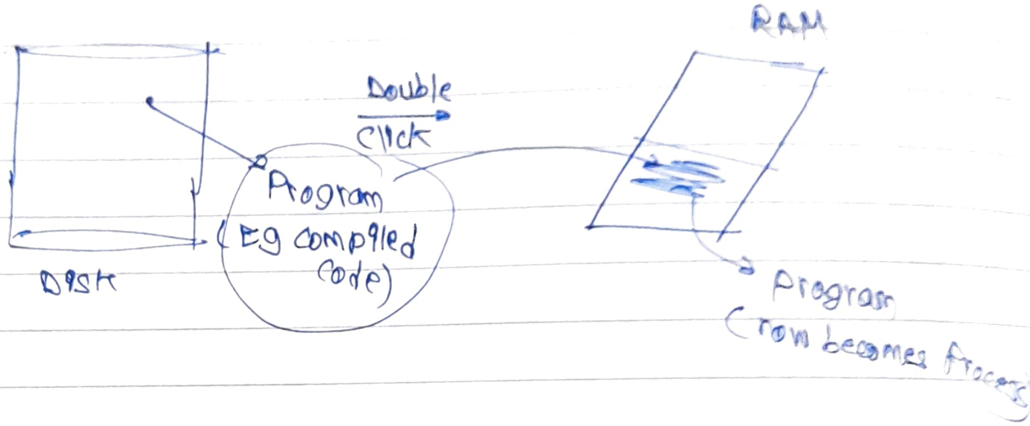
Compilation Process →



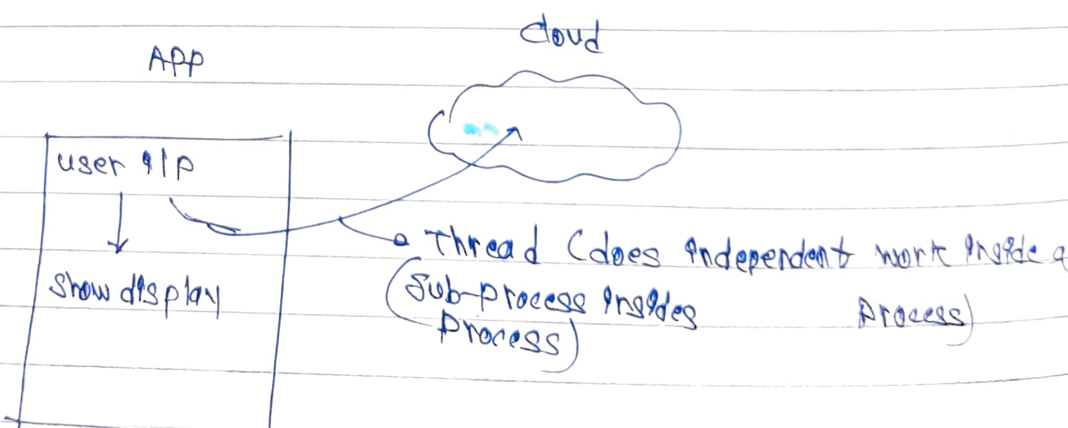
[Linker generates .exe file on windows & a.out for Linux] [Libraries]

Eg C++ Code ⇒ .cpp → compile → executable → .exe (in windows)
↓
Byte code (only OS can read)

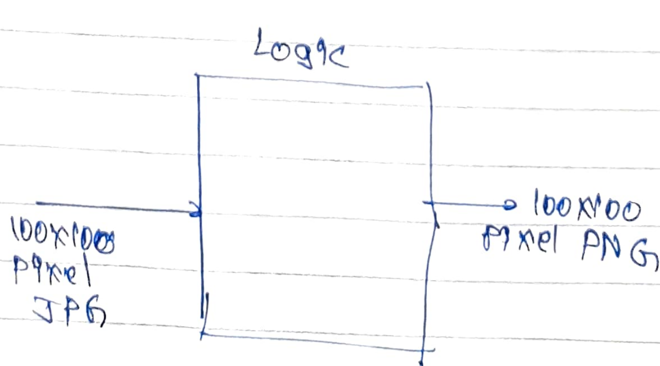
(double click) ←
When you open .exe say Paint.exe a process begins



- Thread → lightweight process
  → independently execute



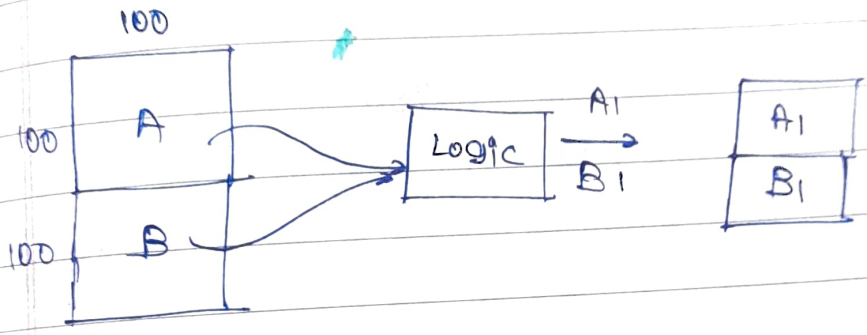
Eg



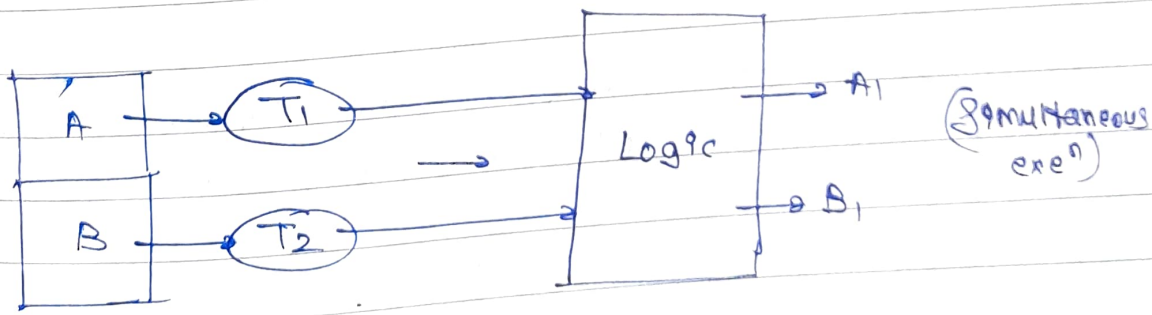
Preprocessor → Removes Comments
→ Replaces Header file with src Code
→ Replaces Macro Name with Code

If i/p → Image file → 100 (W) x 200 (H)

① Do this sequentially



But A is not dependent on B & vice versa, so it should happen asynchronously (simultaneously)
∴ Use Multi-Threading

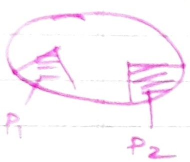


→ Can't occur if single CPU available, as 1 CPU takes hold of T_1 & other hold of T_2 (T_1 executes using CPU)

☆ A process is divided into several sub-tasks called threads, called Multi-threading. These threads have their own path of execution.

Multi-Tasking	Multi-Threading
Execution of Tasks Simultaneously	Execution Simultaneously
1. More than 1 Processes	More than 1 Threads.
2. Isolation & Memory Protection	No (Since Thread is a part of Single Process) → hence all have to the memory block allotted to that Process

Each process is separate entity & it had its own memory block (OS must allocate separate memory & resources to each program that CPU is executing)



3. Process Schedule	Threads Schedule
4. More than 1 Processes are context switched	More than 1 Threads are context switched
5. No. of CPU = 1	No. of CPU ≥ 1
	No. of Threads = No. of Cores in CPU

- Program → It is an executable file which contains a certain set of instructions written to complete specific operation on your computer
  - → It's a compiled code, ready to be executed
  - → Stored in Disk

- Process → Program under execution. Stored in Computer's RAM

- Thread
  - → Single Sequence Stream within a process
  - → independent path of execution in a process
  - → Light-weight process

Resources are shared among threads of that process. OS allocates memory to a process, multiple threads of that process share same memory & resources allocated to process

→ Used to achieve parallelism by dividing a process's tasks which are independent path of execution

Eg Multiple tabs in a browser, texteditor (When you are typing in a text editor, spell-checking, formatting of text, and saving the text are done concurrently by multiple threads)

- Thread Scheduling → Threads are scheduled for execution based on priority. Even though threads are executing within runtime, all threads are assigned processor time slices by OS

  time slices → allocation of time intervals by OS to individual threads for execution on CPU

Thread Context Switching	Process Context Switching
OS saves current state of thread and switches to another thread of same process	OS saves current state of process & switches to another process by restoring its state
Doesn't include switching of memory address space	Includes (as due to Isolation & Memory Protection, OS allocates separate memory & resources to each program that CPU is executing)
Fast Switching	Slow Switching
CPU's cache state is preserved	CPU's cache state is flushed (Cache is always related to a program, so program change hone par cache flush) Ek Process ka cache ka Area ek hi hota hai, so koi dusri process ... par pehle se jo item pada tha cache me vo delete karna padega

- File Descriptor → unsigned integer used by a process to identify an open file

⇓

It is a ptr to structures representing files

→ Standard In
→ Standard Out
→ Standard Error

File No.	
0	(Read from keyboard)
1	(Write to screen)
2	

OS

Kernel

- → Interacts directly with hardware
- → Performs crucial tasks
- → Heart of OS
- → Very 1st Part of OS to load on start-up

User Space

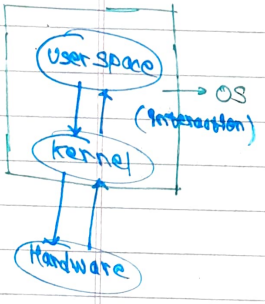
- → Application Software runs here
  - ↳ computer program designed for specific tasks
  - Eg Microsoft Word (for word processing)
  - Google Chrome (web browsing)

→ Apps dont have privileged access to underlying hardware

↳ CPU, RAM, Hard Drives, SSDs

→ Interacts with Kernel

→ Provides a convenient environment to user apps



→ User Space

- GUI
- CLI
  - → Terminal
  - → Powershell

Eg Helloworld.sh → script

./Helloworld.sh → executed on Terminal

Jis area me CLI me ispko run kar rahe ho → That's User Space

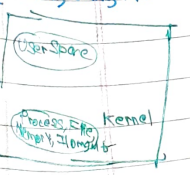
U.S. computation ke liye data hai Kernel ke pass, Kernel CPU se request karega kuch computation cycles ki, CPU vo computations kar dega & command line pe O/p dikhega

Kernel

- Process Management
  - Scheduling Processes & Threads to CPU
  - Creating & Deleting User & System Process
  - Suspending & Resuming Process
  - Provides mech for Process Synchronization or comm
- Memory Management
  - Allocate & Deallocate memory space as per need
  - Keep track of which part of memory is being used & by which process
- I/O Management
  - Manage & Control I/O op. & I/O devices
  - Spooling
  - Buffering
  - Caching
- File Management
  - → Create & Delete files & Directories
  - → Mapping files into secondary storage

Kernel

Monolithic
- All functions are in kernel itself
- Bulky in size
- High memory req. to run
- Less Reliable, one module crashes, whole kernel is down
- High performance coz comm^n is fast
- Linux, Unix, MS-DOS



Micro
- Only Mgmt functions are in Kernel
  - i. Memory mgmt
  - ii. Process mgmt
- → File & I/O mgmt are in User Space
- → Smaller in size
- → More Reliable
- → More Stable
- → Performance is slow
- → Overhead switching b/w user & kernel mode

Eg L4 Linux, MINIX

Hybrid
- → Advantages of both world
- → File mgmt in User Space Rest in Kernel Space
- → Speed & design of mono
- → Modularity & Stability of micro
- → IPC happens but lesser overheads

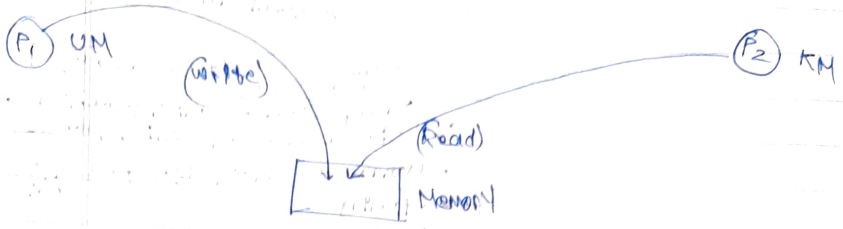
Eg Macos, Windows NT/7/10

Nano / EXO

UM & KM comm^n → done with help of Inter Process Comm^n (IPC)

2 ways of IPC

1) Shared Memory



2) Message Passing

- To switch b/w User Mode & Kernel Mode, system calls are used

Actual implementation ⇒ mkdir, System calls → C

- Apps interact with Kernel using System calls

Eg mkdir /abc

→ mkdir indirectly calls Kernel & asks file mgmt module to create a new directory

→ mkdir interacts with Kernel using system calls

Eg Creating a process

→ User executes a process (User space)

→ Gets System call (US)

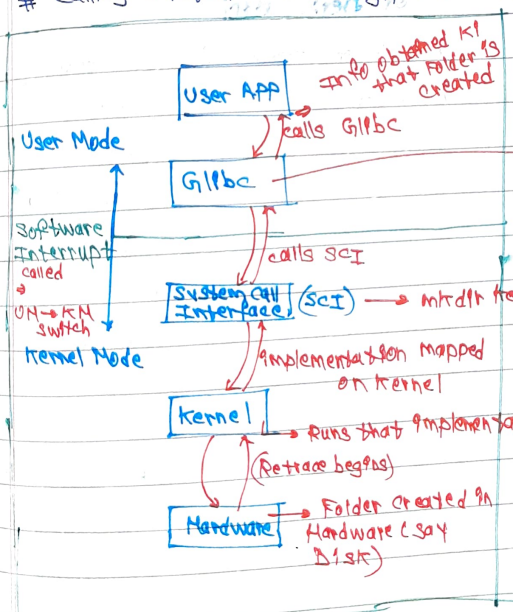
→ Executes System call to create a process (KS)

→ Returns to US

Transitions from US to KS done by software interrupts

System call :- A mech by which a user program can request a service from the kernel for which it does not have permission to perform.
User programs typically do not have permission to perform ~~I/O~~ operations like accessing I/O devices and communicating with other programs

Calling system call through User App AIM: Create a Directory



Eg Process Helloworld.exe → Run, Now it searches for system call which is provided by SCI, switches mode from user to kernel, kernel mode me process mgmt ke pass jayenge, jo process to create karega, phir kernel to user mode switch

W = Windows
U = Unix

Trick :- mgmt ⇓ CDF (Cumulative Distribution fn)

System Calls → Invoked through Software Interrupt instruction "INT"

Process Control (Functions)
- → end, abort
- → load, execute
- → Create, terminate process

W = Windows
U = Unix

W → CreateProcess()
U → fork()
↓
Creates a Process
when Computer Starts
↓

→ 0th process
→ All subsequent Process are child of 0th process

File mgmt
- → create, delete file
- → open, close
- → read, write

W → CreateFile()
U → open()
chmod()
↓
Changes mode of file
chown()
↓
changes ownership of file

Device mgmt
- → request, release device
- → read, write, reposition
- → logically attach or detach devices

W → SetConsoleMode()
U → read()

Info maintenance
- → Get Time or date, set time or date
- → Get System Data, set system Data
- → Get/Set Process, file or Device attri

W → GetCurrentProcessID()
U → getpid()

Comm^n mgmt
- → Create, Delete comm^n conn^n
- → Send, receive messages
- → Attach or detach remote devices

W → CreatePipe()
U → pipe()
Shmget()
↓
To get the shared memory through which multiple process are communicating

Eg Running .exe
inside Terminal
Terminal → Parent Process
.exe → child Process
⇓
Root creates all these child Process

exit()
↓
process ko exit karana
(memory deallocate karana)

System calls used
open ( )
write ( )
fork ( )

Q3 vi hello.sh
Write code & save

bash hello.sh
↓
creates a process hello.sh, which is child of that terminal i.e. fork ( ) is used

Run ps -a → You'll find a process bash hello.sh, along with its PID

Run ps -al → You'll find a PPID corr. to above PID → This child is created with help of fork
(PPID → parent process)

kill -9 PID
↓ (-9)
forceful kill

kill ( )
↓
kills the process

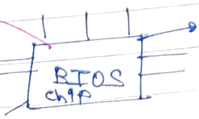
Q What happens when you turn on your computer ?

Ans 5 Steps :-

① Power ON

② CPU loads BIOS or UEFI → small program stored in BIOS chip
(Unified Extensible Firmware Interface) → More Adv. than BIOS

ROM chip found on motherboard that allows to access & setup computer system at most basic level



→ stores BIOS program
↓
(Basic Input Out System)

a) CPU initialize
b) CPU goes to a chip BIOS (to load BIOS program)

③ CPU runs BIOS & BIOS or UEFI run tests & init Hardware

→ BIOS loads ~~state~~ configuration settings from a memory area
(Backed by Complementary Metal Oxide Semiconductor)

→ BIOS program loads with settings, after that instructions in BIOS program does POST (Power On Self Test) process
⇐ There are some test cases corr. to each hardware (checks if all essential hardware are present)
If something is not appropriate, (like missing RAM), error is thrown & boot process is stopped

the responsibility for booting your PC

④ BIOS or UEFI hands off, to Boot Device → Disk (HDD or SSD), CD, USB Device
Stores instructions on how to load OS

→ When Bootloader is executed, it switches on OS (loads rest of OS)

present in
MBR (Master Boot Record)
present on 0th index of Disk
[BIOS uses MBR]

EFI
[UEFI uses EFI]
It is a partition in Disk where the

⑤ Bootloader loads full OS (Boots kernel, then User Space) into actual OS & turns it ON!

Windows → bootmgr.exe
Mac → boot.efi
Linux → GRUB
} Bootloaders for diff OS

→ Windows Boot Manager

- BIOS executes bootloader, which takes it from there & begins booting actual OS → Windows or Linux, for example

32-bit Vs 64-bit OS

32-bit OS → 32-bit registers → can access 2^{32} unique memory addresses
→ can process 32 bits of data

$$\text{No. of GB} = \frac{\text{No. of Addresses}}{2^{30}} = \frac{2^{32}}{2^{30}} = 4$$

∴ 32-bit OS can access 4 GB unique locations of physical memory

64-bit OS → 64-bit registers → can process 64 bits of data

- Adv. of 64-bit over 32 bit OS:

1. Addressable Memory ⇒ 32-bit CPU → 2^{32} unique memory addresses
   64 " " → 2^{64} " " "

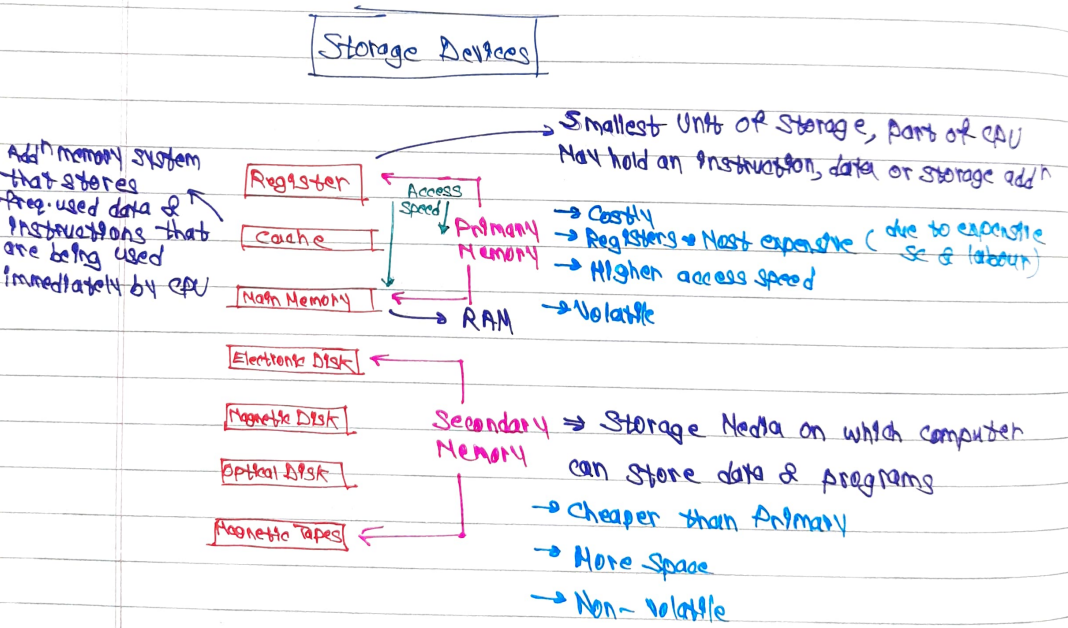
2. Resource Usage → Better in 64-bit OS, if I install add^n RAM in 32-bit OS, it's useless, since I can use only 4 RAM at a time in 32-bit OS (can't support more than 4 GB RAM)

3. Performance → Faster in 64-bit OS, since it can process more data at a time (64-bit)

4. Comparability → 64-bit CPU can run 32-bit & 64-bit OS, while 32-bit CPU can run 32-bit OS only.

5. Better Graphics Performance → In 64-bit CPU, you can make ~~8-bit~~ 8-bytes graphic calculations in 1 cycle, making graph intensive apps run faster

Storage Devices



Intro to Process

→ OS creates a process by converting a program into process

Steps :-

1. Load the Program & Static Data into memory (Main memory)
   located inside DISK — used for initialization

   Eg char *name = "Lakshay" → static data
   (ptr name var with Lakshay)

2. Allocate Runtime Stack → when you're calling a fn inside another you store all data of parent fn inside Stack & jump to chil

   Stack → Part of memory used for local variables, fn arguments & return values

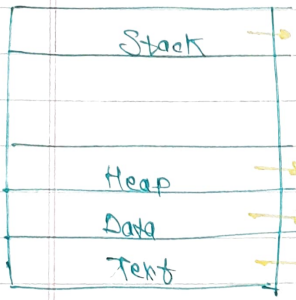
3. Allocate Heap → Part of memory used for dynamic allocn

4. I/O Tasks :-
   Unix :- I/P, O/P } I/P, O/P Handles are allocated to I/P, O/P haga
   Error → Error handle → prog. error generate karne par to error kaha gayga

5. OS handoffs control to main()

```
main()
{
    return 0; → means successful
}          exec done
```

Architecture of process:-



Local var, fn arguments, return values

Dynamically allocated var

Global & static data

Compiled code (loaded from disk)

beyond a limit)

- To prevent stack overflow, set base case (so that stack can't ↑ objects
- ——— out of memory error, deallocate unnecessary memory

→ All processes are being tracked by OS. It's using a table like Data Structure called Process Table

→ Each entry in this table is Process Control Block (PCB)
PCB stores info of a process such as process id, program counter, process state, priority etc

(PC) + register in CPU

Running / waiting / Ready / Terminated

PCB Structure:

Process ID	→ unique identifier
Program Counter (PC)	→ stores add^ of curr. inst^, incremented after inst^ executed
Process State	
Priority	
Registers	→ saves curr. value of registers
List of open files	
List of open devices	

Process State

process being created, hence can't be scheduled

fork() running, but not over. (Process is being created)

→ New → OS is abt to pick up the program & convert it to process

→ Ready ⇒ Process is in memory, present in Ready queue, waiting to be assigned to a processor

→ Running ⇒ Instructions are being executed, CPU is allocated

→ Waiting ⇒ for I/O

process table

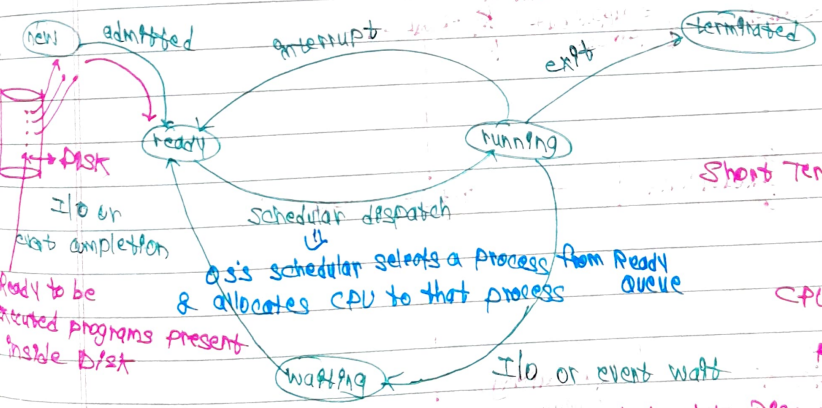
→ Terminated ⇒ Process has finished exec^n & PCB entry removed from

calls exit() when running, cleans PCB, takes resources, now terminated

Long Term Scheduler

then ready state)

Job Scheduler ⇒ picks these programs & schedules them (putting it into new, from pool



Short Term Scheduler

CPU Scheduler

Ready to be executed programs present inside Disk

Ready queue se process uthana & usko Running state tak Dispatch karna ⇒

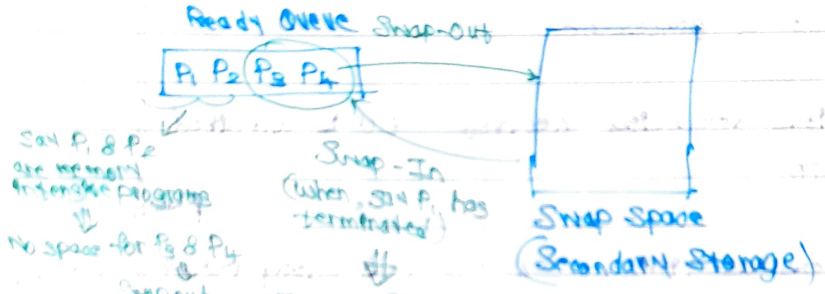
Job Scheduler → LTS → Long / Short depends on frequency

P ready se running pe aaya & waha few ms., vo I/O karne chala gaya, CPU khali ho gaya, but CPU shouldn't be idle!

for out process to no lagech running state pe dal dega at an instance

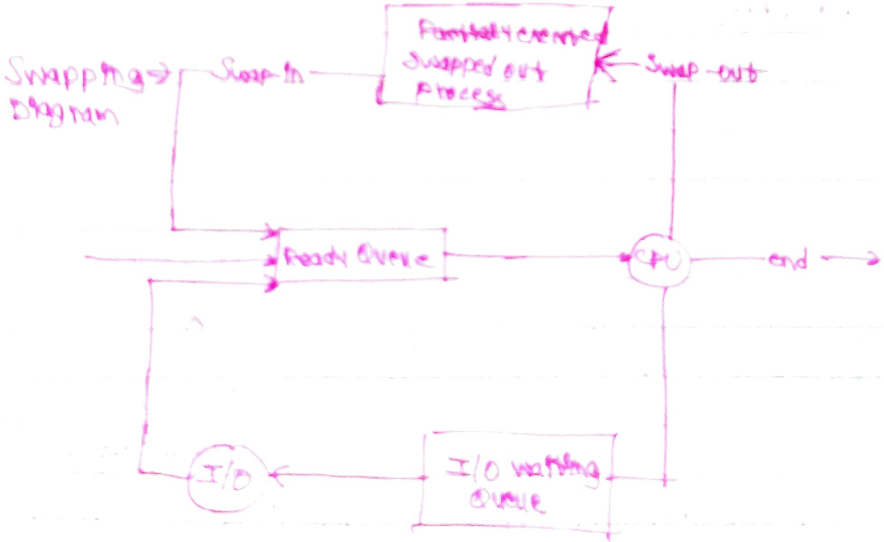
Degree of ~~multiprogramming~~ = No of processes in Ready queue ~~that is~~

↓
LTS handles this

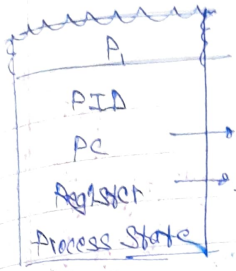


This is Swapping

↳ done by Medium Term Scheduler



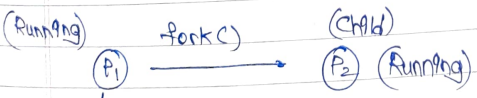
Context Switching



Saves state of current process & Restore state of next process
kernel does this

→ It is a pure overhead, since no process from user space is being executed, only C.S. is done using instr^n present in a program inside kernel (present in CPU) (i.e. No useful W.D)

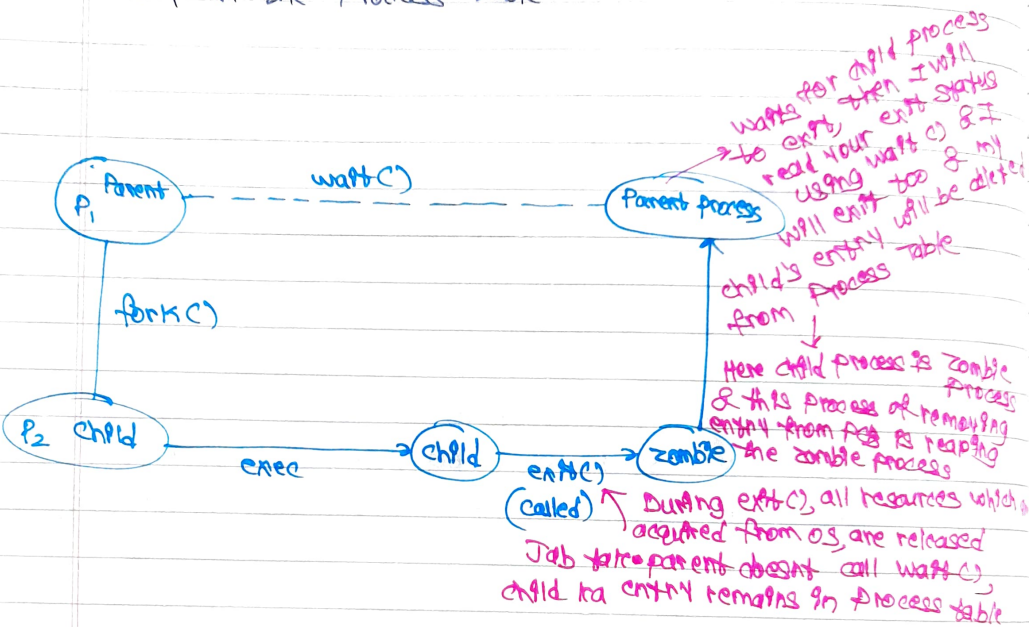
- Orphan process → Process whose parent process has terminated & it is still running. They are "adopted by init process, which is 1st process of OS



exception → terminate ⇒ Parent process terminated, hence P_2 becomes Orphan Process
So, OS makes init process, the parent process of P_2

- Zombie / Defunct Process

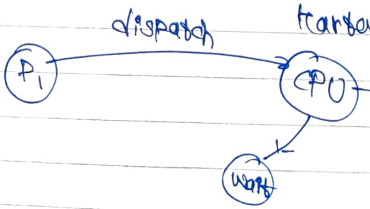
→ A process whose execution is completed but it still has an entry in the process table



Process Scheduling

① Non-preemptive Scheduling

Ek baar P_1 dispatch ho gaya hai, to vo CPU ko tab tak hold karega hai jab tak process terminate ho jaye ya fir wait state me chala jaye



→ Time Sharing X

② Preemptive Scheduling → Same as ① + Time-Quantum expire hone par process CPU ko free kar deta hai

Starvation → Non-Preemptive ↑ cause Time sharing part is missing if I encounter a job which is CPU intensive, then other processes face starvation

→ Preemptive ↓ ~~due to~~ cause Time Quantum expire hone par dusra process ko mauka milega

CPU Utilization → Preemptive ↑ as more processes CPU can work on

Overhead → Preemptive ↑, if Time Quantum = 1 sec, har 1 sec ke baad process change ho jayga yani har second ke baad CPU ko new process execute karni padegi

Goals of CPU Scheduling :-

① Max. CPU Utilization
② Min. Turnaround Time → TAT → Job Ready Queue me aana & usse exit karne ke beechka time
③ Min. Wait time → (for each process) ke beechka time
④ Min. Response Time → Process Ready Queue me aane ke baad & CPU milne
⑤ Max. Throughput → No. of processes completed per unit time

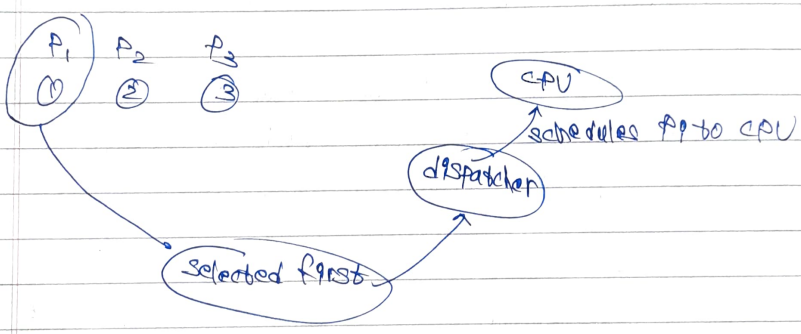
Arrival Time → Time when process is arrived at Ready Queue
Burst Time → Time req. by process for its execution
Wait Time → Time process spends waiting for CPU
Turnaround Time → Time taken from 1st time process enters ready state till it terminates

Response time → Time duration b/w process getting into ready queue & process getting CPU for 1st time
Completion time → Time taken till process gets terminated

1) $TAT = CT - AT$

2) $WT = TAT - BT$

- ~~FCSP~~ FCFS



* Convoy effect

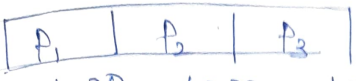
Eg

				$= (CT-AT)$	$= (TAT-BT)$
P no	AT	BT	CT	TAT	WT
1	0	20	20	20	0
2	1	2	22	21	19
3	2	2	24	22	20

Avg WT 13

Since P_1 has high BT than P_2, P_3
Hence Avg. WT ↑ → This is convoy effect

Gantt Chart

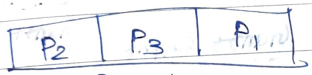


t=0 t=20 t=22 t=24

$= (CT - AT)$ $= (TAT - BT)$

PrO	AT	BT	CT	TAT	WT
2	0	2	2	2	0
3	1	2	4	3	1
1	2	20	24	22	2

Avg WT = 1 unit



t=0 t=2 t=4 t=24

18th June

CPU scheduling algos

- **FCFS**

→ Whichever process comes first in Ready Queue, will be given CPU first

→ If a process has longer BT, it has major effect on avg WT of diff processes, called Convoy effect

→ Convoy effect is a situation where many processes, who need to use a resource for short time, are blocked by another process holding that resource for long time causing poor Resource mgmt

Shortest Job First (SJF) (Criteria: AT, BT)

Non-Preemptive

(CT−AT) (TAT−BT)

P	AT	BT	CT	TAT	WT
1	0	8	8	(Complete)	
2	1	4	12		
3	2	9	26		
4	3	5	17		

Gantt Chart

P1	P2	P4	P3
t=0 – t=8	t=8 – t=12	t=12 – t=17	t=17 – t=26

Scheduled earlier than P_3 since at t=12 sec, both P_3 & P_4 had arrived & at Ready Queue, but P_4 has shorter BT

⇓

Yaha jo bhi process schedule ho rahi hai, vo pura execute hone tak schedule hoti hai. Jabki Preemptive me agar koi process ke exe^n ke dauran koi aur process Ready Queue me aa jati hai jiska BT ↓ hai, to current process ko rok ke, usko schedule kar dete hai!!

Preemptive

(Comment / TAT ...)

P	AT	BT	CT	TAT	WT
1	0	~~8~~ ~~7~~ 0	17	17	9
2	1	~~4~~ 0	5	4	0
3	2	9	26	24	15
4	3	~~5~~ 0	10	7	2

↳ acc. to Gantt Chart

[illegible] Avg

Gantt Chart

P1	P2	P4	P1	P3
t=0 – 1	1 – 5	5 – 10	10 – 17	17 – 26

P_1 preempted by P_2 since P_2 has lower BT!

At each instant, choose Pr with least BT (among all processes that have arrived)

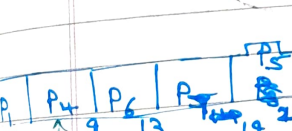
⇓

No Convoy effect ↳ Since the process with longer BT will be preempted by another process with ~~[illegible]~~ smaller BT

Priority Scheduling (Criteria: Priority)

Non-Preemptive

Priority	AT	BT	CT	TAT	WT
2	0	4	--	--	
4	1	2			
8	2	3	-	-	-
10	3	5			
8	4	1			
12	5	4			
9	6	6			



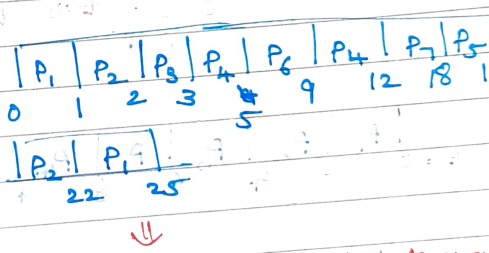
Amongst P_2, P_3, P_4, P_5

Highest Prior.

Shi consider karna!

Preemptive

P	P*	AT	BT	CT
1	2	0	~~+30~~	25
2	4	1	~~2+0~~	—
3	6	2	~~3+0~~	—
4	10	3	~~5+30~~	—
5	8	4	+0	—
6	12	5	~~+0~~	
7	9	6	~~6+0~~	



Avg. WT ≈ 11.4 sec which is even greater than its Non-Preemptive version. Here High Priority Job ~~gets~~ is always preferred leading to CONVOY EFFECT! since High Priority Job can have high

Both of these versions lead to Indefinite Waiting

Soln:- AGEING → Gradually ↑ Priority of Lowest priority (say every 15 min, Priority++ (so that L.P.J. also get

(Time Quantum)

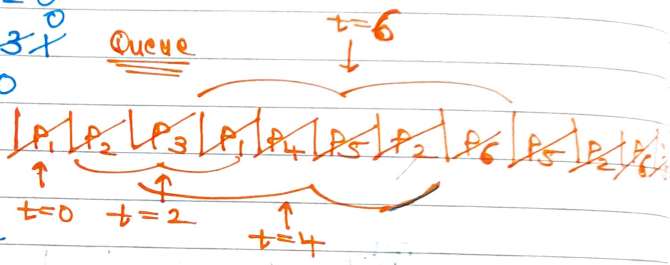
Round-Robin (RR) (Criteria: AT, TQ)

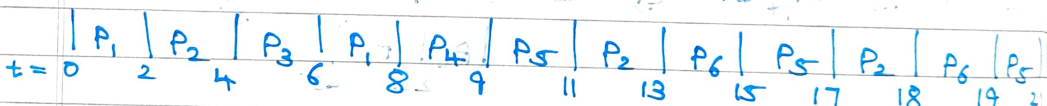
→ Response Time ↓
→ Starvation ↓ & Convoy Effect X
→ Preemptive version of FCFS
→ Designed for Time Sharing & we know Time Sharing is used in Multitasking OS

P	AT	BT
1	0	~~2~~ 0
2	1	~~5~~ ~~3~~ 0
3	2	~~2~~ 0
4	3	~~1~~ 0
5	4	~~6~~
6	6	~~3~~ 0

(Time Quantum = 2sec)

Queue





→ Starvation X but Overheads ↑

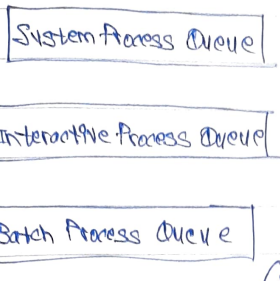
Multi-level Queue Scheduling (MLQ)

upon priority

→ Ready queue is divided into multiple queues depending
→ A process is permanently assigned to one of the queues
→ Each queue has its own scheduling algo

Eg SP → RR BP → FCFS
IP → RR

} — (1)



System Process: Created by OS

I.P: Needs User I/P (Foreground process)

B.P: Runs silently, no user i/p req (Background process)

- Scheduling inside sub-queues → (1)
  Scheduling amongst sub-queues → fixed priority preemptive
  SP > IP > BP

- Since after completion of all processes from top level ready queue, further level r.q will be executed → Causing Convoy Effect & Starvation for lower processes

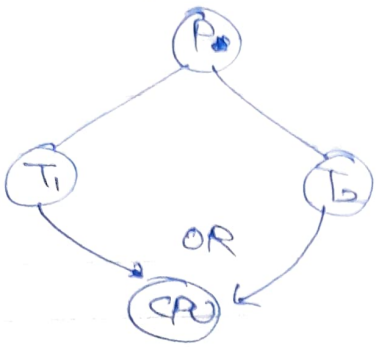
Concurrency → execution of multiple instruction sequences at same time. Happens in OS where there are several threads running in parallel

- Thread has a Thread control Block (TCB)

Q How each thread gets access to CPU?

Ans
1) Each thread has its own Program Counter → add'r of next instn
2) Depending upon thread Scheduling algo, OS schedules these threads
3) OS fetches instructions corr. to PC of that thread & execute instn

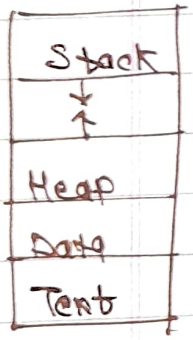
- There is no point of Multi-Threading for Single CPU system



⇒ Either T_1 gets CPU
OR
T_2 gets CPU
⇒ No parallelism Sequential

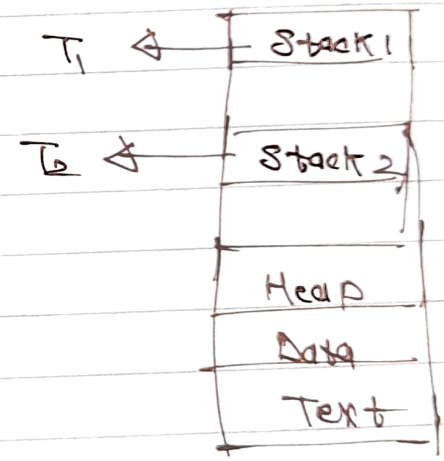
Memory organization

Process



No. of Threads
=
No. of Stacks

Thread



- Benefits of Multi-Threading

① Responsiveness → Interactive app jisme user i/p req., Ho ho rahi internet se kuch fetching-decoding chalu, background me koi aur task chalu, ye independent paths Thread me divide ho jate hai, so agar 1 Thread i/p le rahi hai, to koi aur Thread kuch aur kaam kar sakti hai, & otherwise ye tasks sequentially karne padte

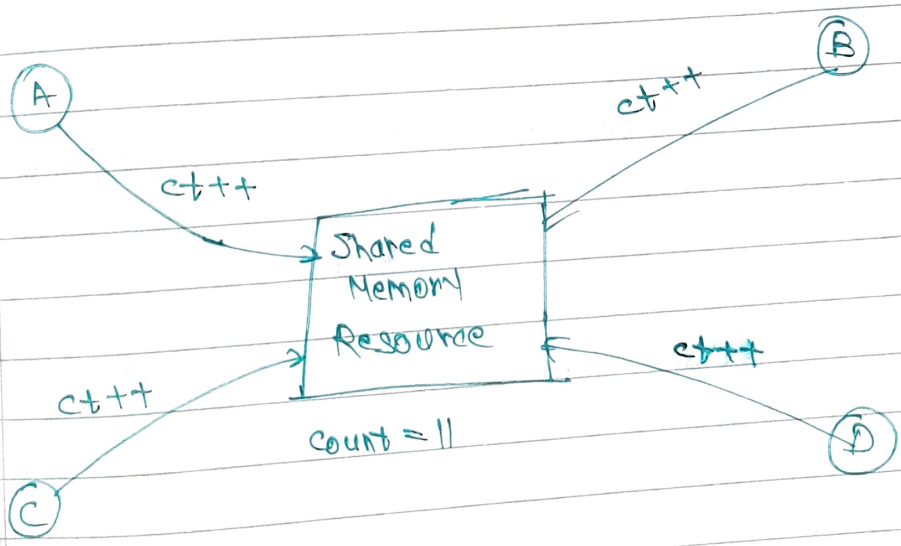
(2) Resource Sharing → Efficient, Overhead ↓ (Threads un resources me sharing karte hai jo ek process ko mile hai)

(3) Economical → Threads creation & context switching is more economical than Processes (Cause process ke liye alag se resources, add^n space allocate karna padega)

(4) Threads better utilize Multi Core CPU
Eg 8 cores hone par, ek process ko 8 independent tasks me divide kar sakta & har ek ko ek CPU provide kar sakta

7:04:36 → Code

Race Cond^n



A → ct++ → tmp = ct+1 = 12, then ct = temp but before ct = t
happens B → ct++ → tmp = 12, same parallel exe^n is going
Hence we find out that count ↑ just by 1, instead of 4

↓↓

Race Cond^n

- Critical Section → Segment of code where threads access shared resources, such as common files & variables, and perform write op. on them.

- Race cond^n → Occurs, when 2 or more threads access shared data & try to change it at same time. Result of change in data is dependent on Thread Scheduling algo (Since the algo can swap b/w Threads at any time & hence we dont know order in which Threads attempt to access data)

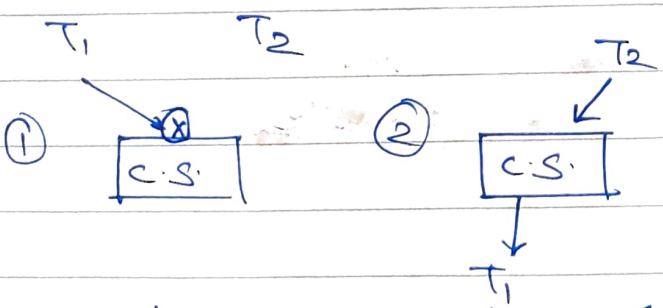
- Soln :-

1) Atomic Operations → Make critical code section an atomic operation i.e. executed in 1 CPU cycle

	Earlier	Atomic
count++ ⇒ tmp = count + 1	1 CPU cycle	Combined 1 CPU cycle
count = tmp	1 CPU cycle	

We can make Atomic Variable in C++

2) Mutual Exclusion



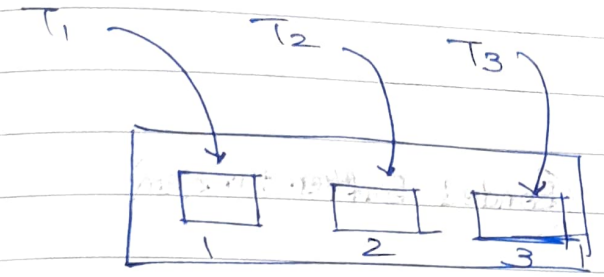
Only 1 Thread should execute C.S. at a time (Sequential)

It can done using Locks (Mutex)
Only after T_1 unlocks, T_2 can access C.S.

3) Semaphores

→ It is an integer variable

→ 1 Resource ke multiple instances hai



$T_1 \rightarrow T_{10}$ → 10 Thread

Sem = 3, 2, 1, 0

Jab tak koi bhi T_i apna lock release nahi karti & resource ko free nahi karti, tab tak unko koi aur thread access nahi karti hai

If $T_1 \rightarrow$ exists, Sem++, Now T_4 can access that resource

- Thread calls wait() for acquiring resources & calls signal() while releasing resources

```
wait(S)
{
  S->value --
  if (S->value < 0)
  {
    add to S->block list
    block();
  }
}
```

```
signal(S):
{
  S->value++
  if (S->value <= 0) {
    remove P from
    Block list
    wakeup(P)
  }
}
```

2 instances of a single Resource

Now T_3 can get CPU access

Eg Semaphore S(2);

$T_1 \rightarrow$ wait() → S->val = 1 → After exiting from C.S → signal()

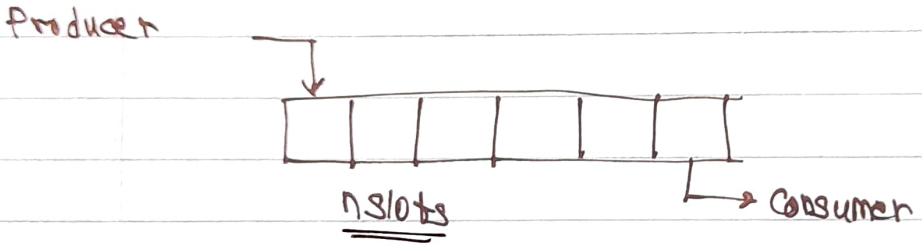
$T_2 \rightarrow$ wait() → S->val = 0 → ——— → signal()

T_3 ——— = -1 → block() (Thread is Blocked)

- Semaphore
  - Binary ($n=1$) (It's actually Mutex)
    - ↳ Used to acquire lock on buffer
  - Counting ($n>1$)

Producer- Consumer Problem (Bonded Buffer Problem)

1. Producer Thread
2. Consumer Thread



Problem?

1) Since buffer is shared resource, I want synchronization b/w producer thread & consumer thread
2) Producer must not insert data when buffer is full
3) Consumer must not remove ——— empty

Sol^n → Semaphores

1. m, mutex → Used to acquire lock on Buffer
2. empty → counting semaphore, initial value = n → Tracks empty slots
3. full → Tracks filled slots —— —— = 0

Producer

```
do E  empty=n ?0 → move on to (1)
wait(empty); // wait until empty ?0,
then empty → value--
wait(mutex); —(1)
// C.S; add data to buffer
signal(mutex);
signal(full); // increment
full → value.
} while(1)
```

Now that can & hence Producer is able to access C.S. do E whereas consumer cant

Consumer

```
[full=0 (in main), so full--
same par wait pe chala jayega]
wait(full); // wait until
full ?0, then full-
wait(mutex);
// remove data from buffer
signal(mutex);
signal(empty); // increment empty
} while(1)
```

Jab tuna Buffer filled hai → ~~A~~ empty=0 → Now Producer has to wait & Consumer accesses C.S.

wait(mutex):

1. Thread calls wait(mutex) & releases associated mutex
2. Thread enters a waiting state, where it is paused
3. Another thread, when appropriate, sends a signal to wake up waiting thread
4. waiting thread is then reawakened ~~& resumes its execution~~ & attempts to reacquire mutex before resuming exec^n

Reader-writer problem

① Reader Thread → Read & Writer Thread → write
② If ?1 ~~Thread~~ Readers are reading → No issue
③ ?1 writers OR 1 writer & some other thread (R/W) in parallel, leads to race cond^n & data inconsistency

Soln:- Semaphores due to R-W problems

1. mutex → Binary semaphore
   → To ensure mutual exclusion, when read count is updated
   → No 2 Threads modify read count at same time

2. wrt (write) → Binary semaphore
   → common for both reader & writer
   → Synchronized access of c.s. b/w reader & writer

3. readcount (rc) → Integer (initial value = 0);

Writer Soln

```
do {
    wait(wrt);   → Look kar dena i.e. koi aur thread enter nahi ho gaya;
// do write operation
    signal(wrt);   → Now koi aur thread enter ho sakta hai
} while(true);
```

Reader Soln

Multiple reader can read

~~wait(mutex)~~

```
do {
    wait(mutex); // To mutex reader var, cause ct++ can also cause race cond^n, use mutex to prevent it
    rc++;
    if(rc==1) wait(wrt);
```

[Agar writer pehle se c.s. me hoga, to Reader thread yaha block ho jayga; else writer block ho jayga]

So, it ensures no writer can enter even if there is 1 Reader

```
    signal(mutex);
// c.s.: Reader is reading
    wait(mutex);
    rc--; // reader leaves
    if(rc==0) // no reader is left in cs
        signal(wrt); // writer can enter
    signal(mutex); // reader leaves
} while(1)
```

→ notifies waiting thread that it can resume exec^n

Single

Q Why cant we use, flags instead of locks / semaphores ?

Ans Sol^n of C.S. should have:

1. Mutual Exclusion
2. Progress

No ordering b/w T_1 & T_2

T_1 T_2

Say jab tak T_1 naha gaya, tab tak T_2 nahi hona chahiye

C.S.

3 Bonded waiting (indefinite waiting) → Every thread should get a chance to access C.S. & hence waiting time should be limited

Eg Single flag → Turn = 0/1

T_1

```
while (1) {
while (turn != 0);
  C.S.
turn = 1
  R.S.
}
```

T_2

```
while (1) {
while (turn != 1);
  C.S.
turn = 0
  R.S.
}
```

Say, turn = 1, T_1 executes, turn = 0, T_2 executes

—— = 0, T_2 —— = 1, T_1 ——

∴ Mutual Exclusion ✓ (only 1 thread is accessing C.S. at a time)

Progress ✗ since initial value of turn decides ordering b/w T_1 & T_2

Iska Improved Version is Peterson's Sol^n

Peterson's Soln

- flag[2]
- turn

flag[2] → indicates if a thread is ready to enter c.s.
flag[i] = true → P_i is ready to enter c.s.

turn → indicates whose turn it is to enter c.s.

Structure of process P_i (in this soln)

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn = [j]) [;]
    critical section
    flag[i] = false;
    remainder section
}
while (TRUE);
```

(Means No body after this loop)

Now let's say P_j is ready to enter c.s. → flag[j] = true, hence the while cond^n in above code → True → Jab tak ye cond^n False nahi hota, tab tak c.s. → not accessible

P_i

```
do {
flag[i] = true;
turn = j;
while(flag[j] && turn == [j]);
critical section
flag[i] = false;
remainder section
} while (TRUE);
```

P_j

```
do {
    flag[j] = true;
    turn = i;
while(flag[i] && turn == [i])
 c.s.
flag[j] = false;
...
...
```

Let's says both P_i & P_j try to access c.s. at same time

1) flag[i] = true
2) turn = j
3) flag[j] = true
4) turn = i ⇒ Final value of turn !!

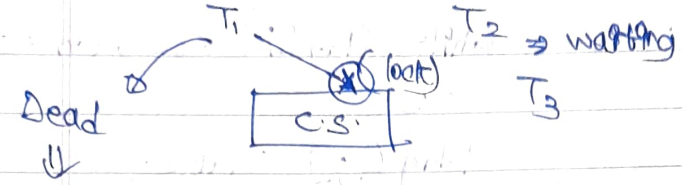
Hence, while loop of P_j → ∞ loop (with No Body !)
Hence, P_i is able to access c.s.
Sets flag[i] = false → now P_j ka while loop cond^n → false
∴ Now P_j can access c.s.

Here, Mutual Exclusion ✓ Progress ✓ Bounded waiting ✓
when 1 process wanted to access c.s., it was allowed to en
& wasn't waiting indefinitely

Problem → Safe Only for 2 Threads

Locks Disadv

① Contention.



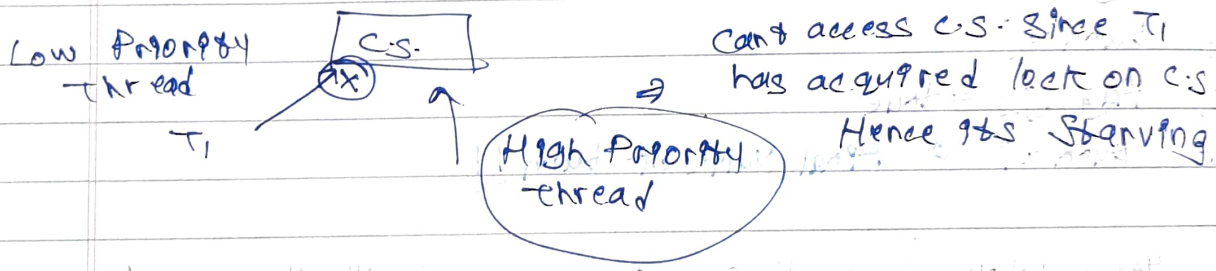
⇓

Causes T_2, T_3 to wait infinitely since the lock was never released

② Deadlock

③ Debugging issue

④ Starvation



Summary:

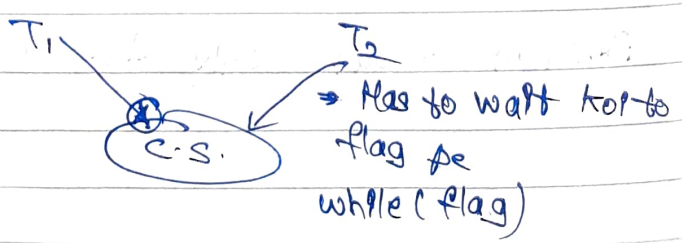
Problem → Race Cond^n

Sol^n → Single Flag ✗ (since Progress ✗)

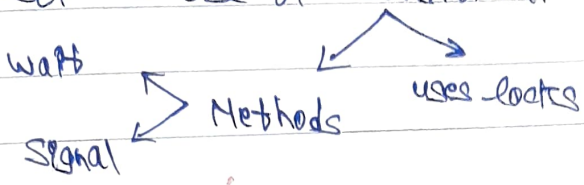
Peterson's Sol^n (only for 2 Threads)

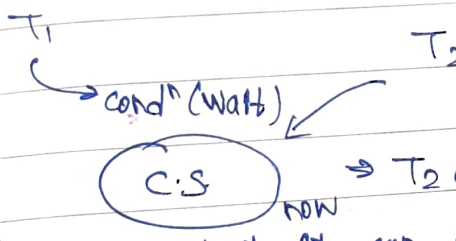
Locks / Mutex (causes Busy waiting)

Busy Waiting →



Improved Soln → Use of Conditional Variables

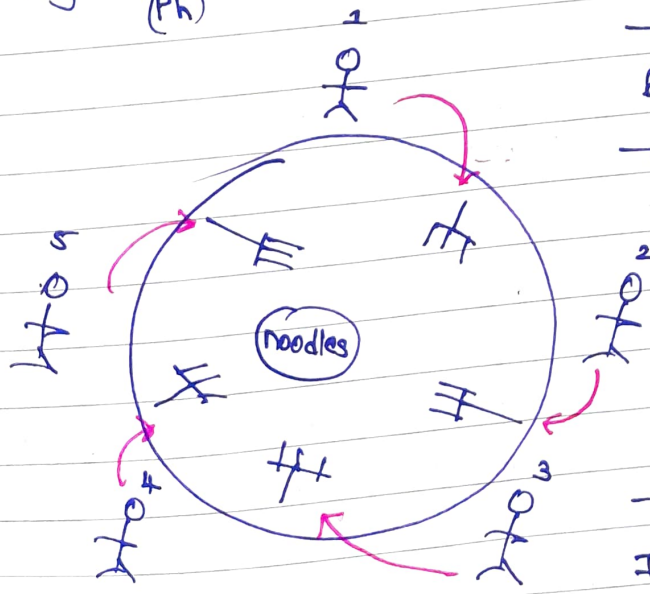




→ T_2 executes C.S. & then calls signal (), which informs T_1 that now it can access C.S

→ No Busy waiting

Dining Philosopher's Problem (Ph)



→ 2 forks req. to eat noodles
→ Ideally 5×2 = 10 forks req
but we have only 5
→ Use semaphores to solve prob^m

→ Each fork — Binary semaphore
Semaphore fork[5] ∈ {1}
↑ Data type ↑ value

→ Ph picks a fork & acquires lock on it
If
→ wait is called on fork[i]
→ ph[i] has acquired fork[i]

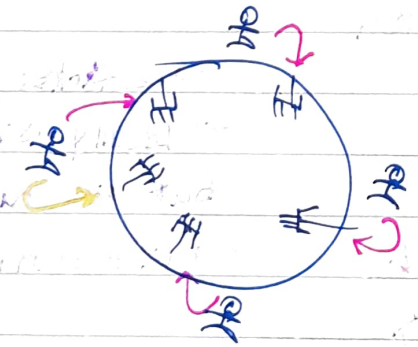
Soln →

```
do {  wait (fork[i]);
      wait ( (fork[i]+1)%5);
      wait ( fork[(i+1)%5]);
      // eat
      signal (fork[i]);
      signal ( fork[(i+1)%5]);
      // think
} while (1)
```

Problem →
ensures 2 adjacent ph can't eat simultaneously but here Deadlock can happen
But if all of them are hungry simultaneously & decide to pick up their left fork & hence acquire a lock on it (See ↓ in fig)
Then, 2 is waiting ti 5 will release lock on 5, so that it can eat noodles, simult^n for others
∴ Deadlock

To avoid this problem:

1. Allow at Most 4 ph to sit simultaneously



once eating done, it releases locks on both forks & hence problem solved. No Deadlock

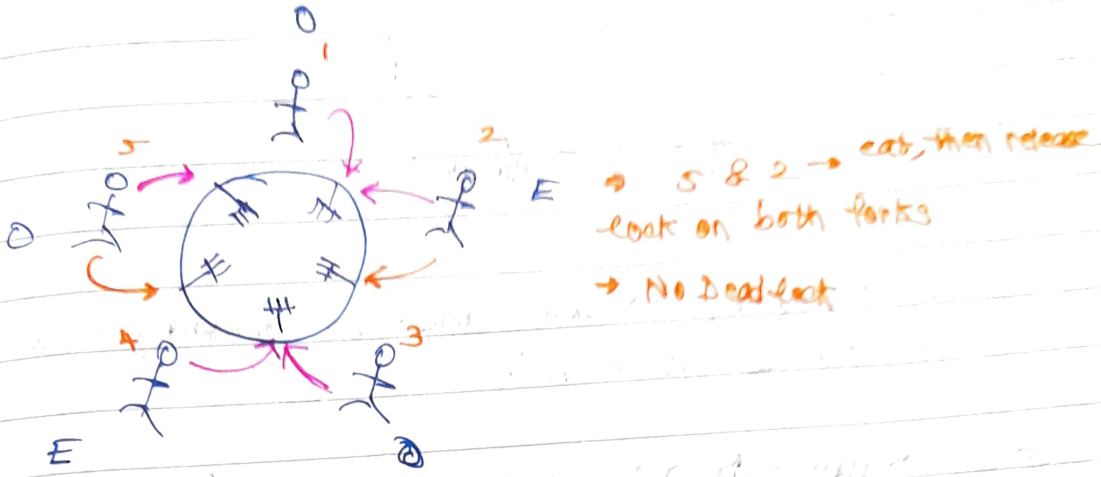
2. Allow ph to pick up fork if both forks available & he must pick them up atomically (in a critical section)

Mutex

```
wait (fork[i])
wait ( fork[(i+1)%5])
```

→ C.S.

3. Odd-even rule → Odd ph: ① Left fork ② Right
→ Even ph: ① Right ② Left



Q How a process/thread utilises a Resource?

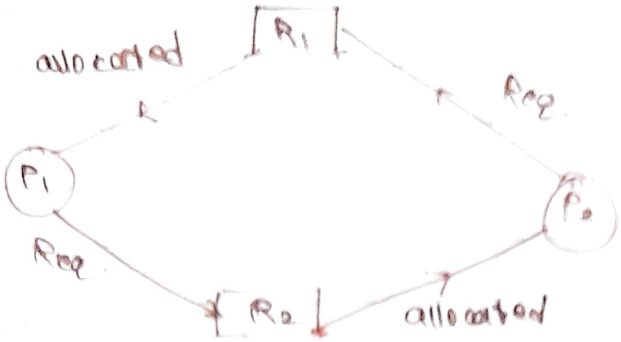
Ans ① Request → Check if lock available, yes? acquire lock on resource, then ② Use resource, ③ Release resource by unlocking the lock, now some other process can use resource

V. IMP for JEE Mains (2024)

Deadlock Cond's :- (All 4 should hold simultaneously)

① Mutual Exclusion → Only 1 process can use a resource at a time, if another process requests that resource, it must wait until resource has been released

② Hold & wait → A process must be holding at least 1 resource & waiting to acquire addⁿ resources, that are held by other processes



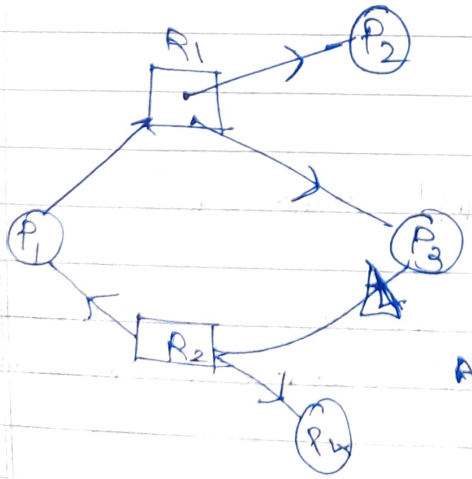
(3) No Preemption → Resource must be voluntarily released by process after exec^n

(4) Circular Wait → A set $\{P_0, P_1, \dots P_n\}$ of waiting processes must exist s.t. P_0 is waiting for P_1, P_1 for P_2 & so on

- Resource Allocation Graph → Cycle ✗ ⇒ Deadlock ✗
  → Cycle ✓ ⇒ Can't tell

- Cycle without Deadlock

RAG ⇒



$t = 0$

P_4 → releases R_2
Now R_2 assigns to P_3
P_3 wants R_1 & R_2, it has both now
So, P_3 execute karke done ho jayega
After exec^n of P_3 is done, P_1 gets R_1

Methods to handle Deadlock

1. Prevent or avoid Deadlock
2. Allow system to go in DL → Detect → Recover
3. Ostrich Algo → Ignore DL

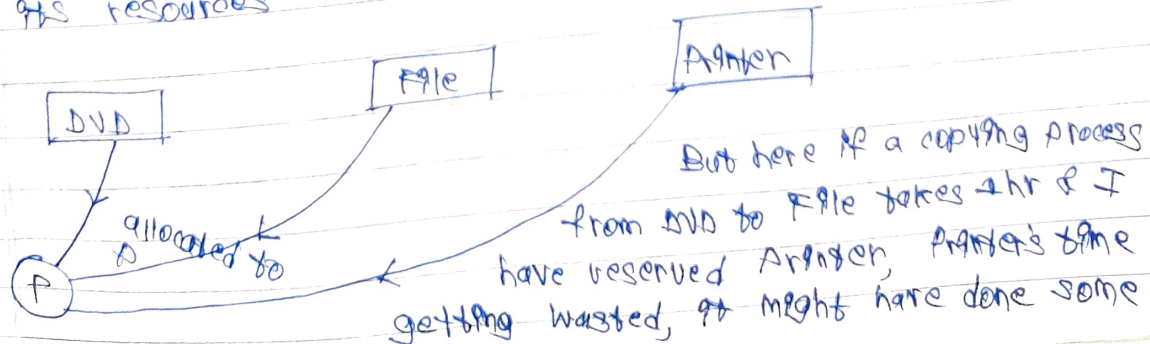
Deadlock Prevention:

(1) Mutual Exclusion → Non sharable resource me use mutex,
C.S. → Memory Space → To ek baar me ek hi resource use kare
Eg Read Only file → Sharable resource (R4 can be written, SC inconsistency can't be achieved) → Don't use Locks
But, say Printer → It can handle only 1 process at a time → Non-Sharable Resource (Since, consistency na toote?)
But, ROF se multiple users can read simultaneously

→ Use locks only for Non-Sharable Resource
Eg ROF → Sharable Printer → Not Sharable resource

→ Some resources are intrinsically non-sharable, hence ismme mutual exclusion can't prevent Deadlock

(2) Hold & wait →
→ Whenever a process requests a resource, it must not hold any other resource
↳ Protocol (A) → Each process has to request & be allocated all its resources before exec^n

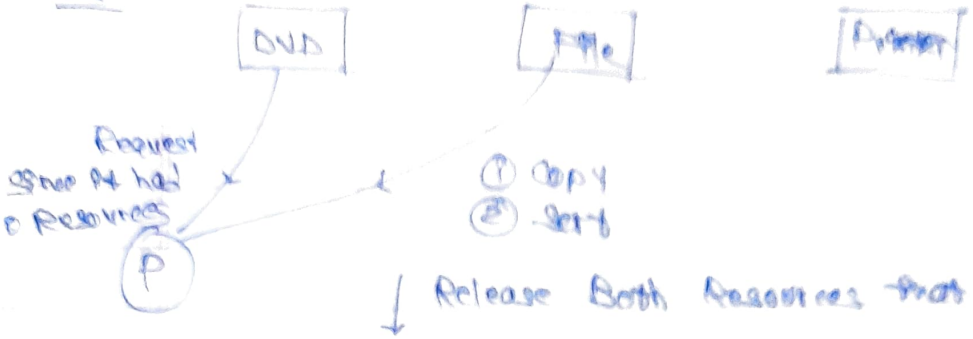


But here if a copying process from DVD to File takes 1hr & I have reserved Printer, Printer's time getting wasted, it might have done some

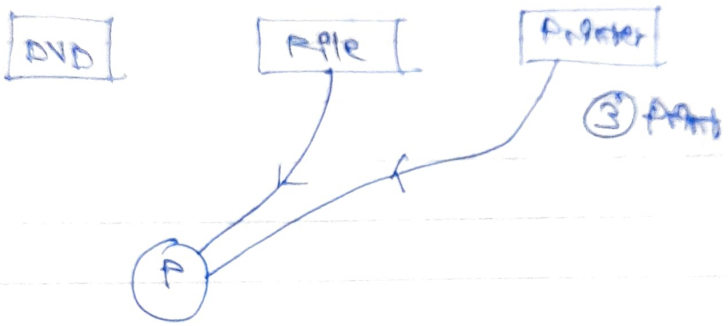
has none

→ Protocol (B) ⇒ A process can request a resource only when it has none. It can request addⁿ resources only when it has released all the resources that it is currently allocated

Eg

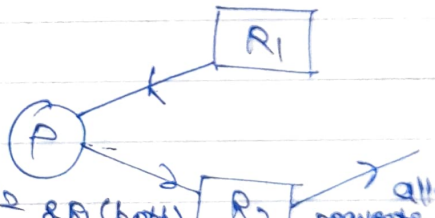


↓ Release Both Resources that are Allocated



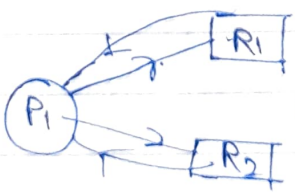
③ Non-preemption ⇒

1) R_1 → allocated but it is waiting for R_2 & R_1 (both)



Hence, release R_1 & when R_2 is free, acquire lock on both R_1 & R_2 and carry out req task

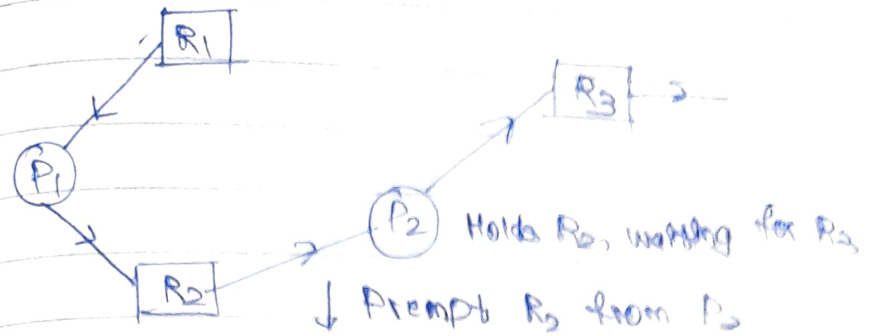
Problem ⇒

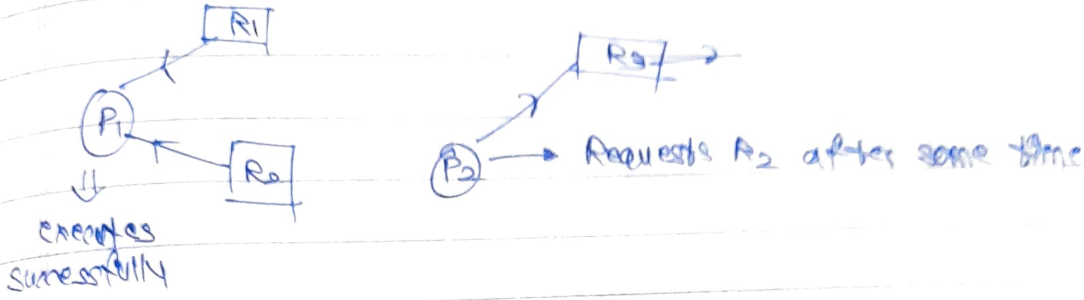


If a process tries to acquire locks on 2 process simultaneously, locks may collide causing Live Lock condⁿ where some same process tries to acquire lock on those processes again & if the locks collide again, this process keeps repeating

Hence Lock1 & Lock2 acquire karne ke beechme let's say 1 seconds

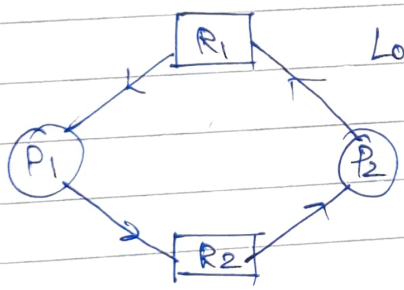
2)



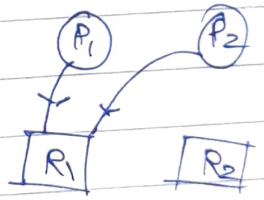


④ Circular Wait

$P_1 \rightarrow R_1$ & R_2 $\quad P_2 \rightarrow R_1$ & R_2



Locks should be ~~ordered~~ acquired in an order
Say, 1st R_1, then R_2



Say P_2 acquires R_1 1st
∴ P_1 waits for R_1
After P_2 ~~ends~~'s work with R_1 is done, P_1 acquires R_2 & ~~P1 acquires~~ req. resources

* Fix Order of Locks on Resources

~~schedule~~

- Safe ~~state~~ → ~~Future allocated resources hai, unko aise allocate karo~~
  ↳ Each process to resource alse du ki at the end, system Deadlock free rahe (All processes should be executed without Deadlock)
  Unsafe → System ki aisi configuration hai, ki iske aage agar Resources ka allocation hota hai, to Deadlock occur ho sakta hai

Deadlock Avoidance

→ Main key of D.A. is that if request is made for resources then request must be approved only if resulting state is a safe state

→ Scheduling algo to avoid DL is Banker's Algo

P	allocated (Given)			max need (Given)			available resources (Total − Max need)			Rem need (Max need − allocated)		
	A	B	C	A	B	C	A	B	C	A	B	C
P_1	0	1	0	7	5	3	3	5	2	7	4	3
P_2	2	0	0	3	2	2	7	3	5	1	2	2
P_3	3	0	2	9	0	2	1	5	5	6	0	0
P_4	2	1	1	4	2	2	6	3	5	2	1	1
P_5	0	0	2	5	3	3	5	2	4	5	3	1

Total : 7 2 5

already available resources

Total ⇒ A = 10, B = 5, C = 7

	A	B	C
	3	3	2
	5	3	2
(P_4 allocated)	7	4	3
(P_5 —)	7	4	5
(P_1 —)	7	5	5
(P_3 —)	10	5	7

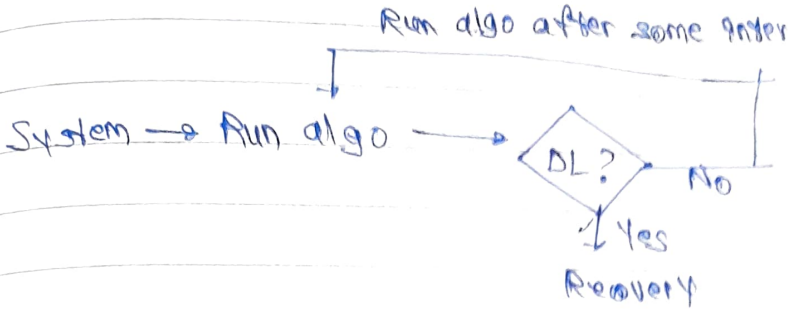
since P_2 ka rem. need is (P_2 allocated) (1,2,2 & 3,3,2 available ⇒ 3+2, 3+0, 2+0 (allocated already))

Available → Same as Total

∴ Scheduling done, hence $P_2 \to P_4 \to P_5 \to P_1 \to P_3$ is safe state

If last allocated process P_3 demanded 8 resources, instead of 6, we had only 7, so P_3 can't be scheduled ⇒ Unsafe state

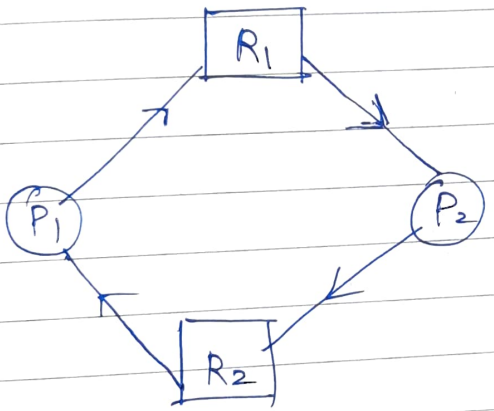
Deadlock Detection



→ Used if Deadlock prevention or Deadlock avoidance cannot be implemented

① Single instance of each resource

Resource allocation graph



Wait for graph

Remove R's

Cycle ✓ Deadlock ✓

cycle ✗ Deadlock ✗

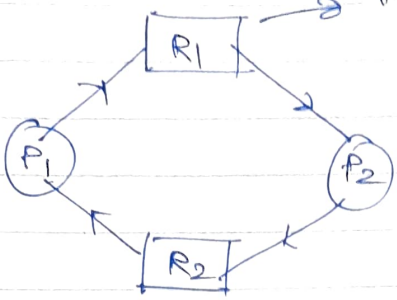
② Multiple instances of each resource

→ Banker algo
→ Safe sequence available ⇒ No Deadlock
→ No ⇒ DL ✓

- Recovery from Deadlock

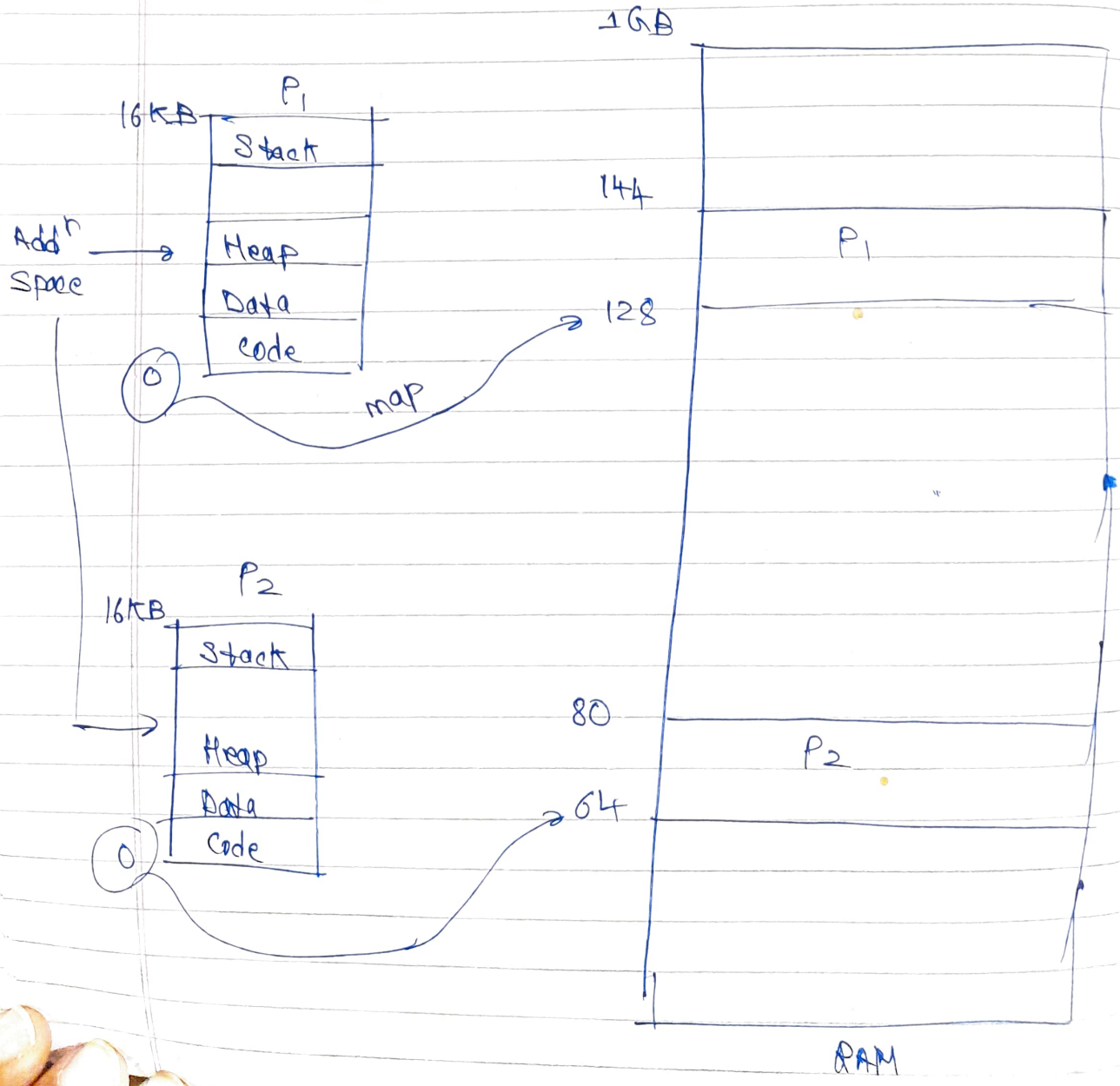
① Process termination
→ Abort all Deadlock processes
→ Abort 1 process at a time (low priority wise) until DL cycle is eliminated

② Resource preemption



Preempt R_1 from P_2
Now P_1 has both R_1 & R_2, so once its exec is complete, P_2 acquires R_1 & R_2 and completes

Memory Mgmt



① $P_1 \rightarrow$ Base at RAM (Physical Add^r Space) offset = 16
$P_2 \rightarrow B = 64$, offset = 16

$B = 128$

Add^n =

Say, P_2 the code me aisa kuch hai, 0 + Random value
= 0 + 129
Add^r = 129

$P_2 \Rightarrow 0 \xrightarrow[to]{map} 64$

OS stores Base & offset

$129 \xrightarrow[to]{map} 64 + 129 = 193 \rightarrow$ actual physical add^r

OS checks if add^r = 129 belongs to P_2
So, it does 64 + 129 = 193 & checks Max value of P_2 in RAM, which is 80 → 193 add^r loc^n may be allocated to some other program
⇒ OS throws exception → process trying to do illegal op.; hence OS provides Isolation & Memory Protection

- Allocation Methods in Physical Memory

Contiguous Allocation
→ Each process is contained in a single contiguous block of memory

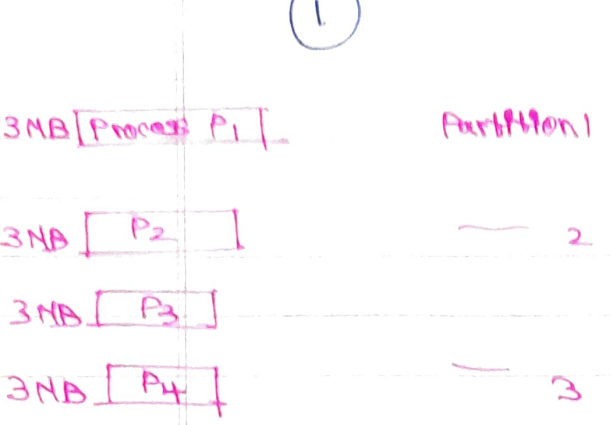
① Fixed Partitioning
→ Main memory is divided into partitions of equal sizes

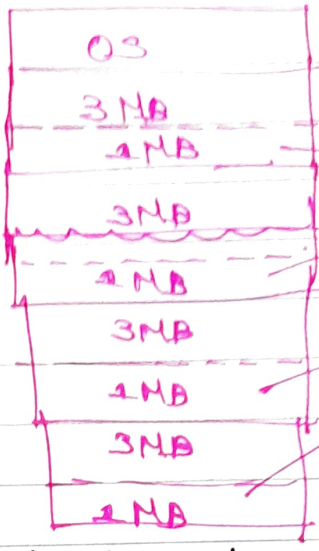
② Dynamic Partitioning
→ Partition size is declared at time of process loading

Non-Contiguous Allocation
→ Paging (V.IMP for Interviews)

③

(1)





Internal Fragmentation
⇓
Size of Process < Size of Partition

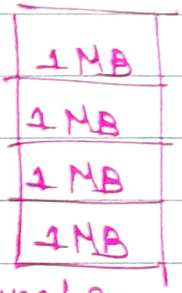
Problems

① Limitation on Process Size
↳ Has to be less than Partition size

② Degree of Multiprogramming ↓
since No. of processes = No. of partitions

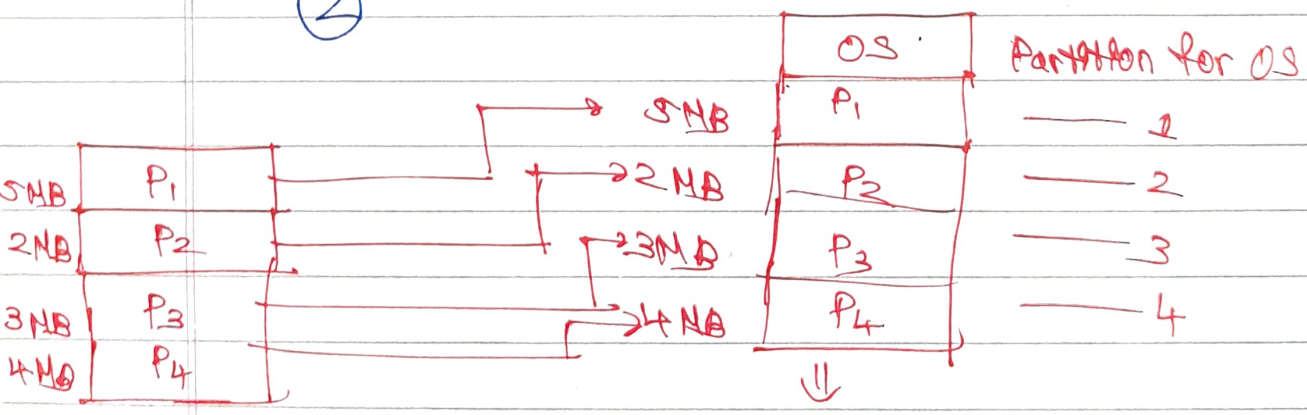
③ Internal & External Fragmentation

Total unused space of various partitions can't be used to load the processes even though there is space but not in Contiguous form



Unused Space
External Fragmentation

(2)



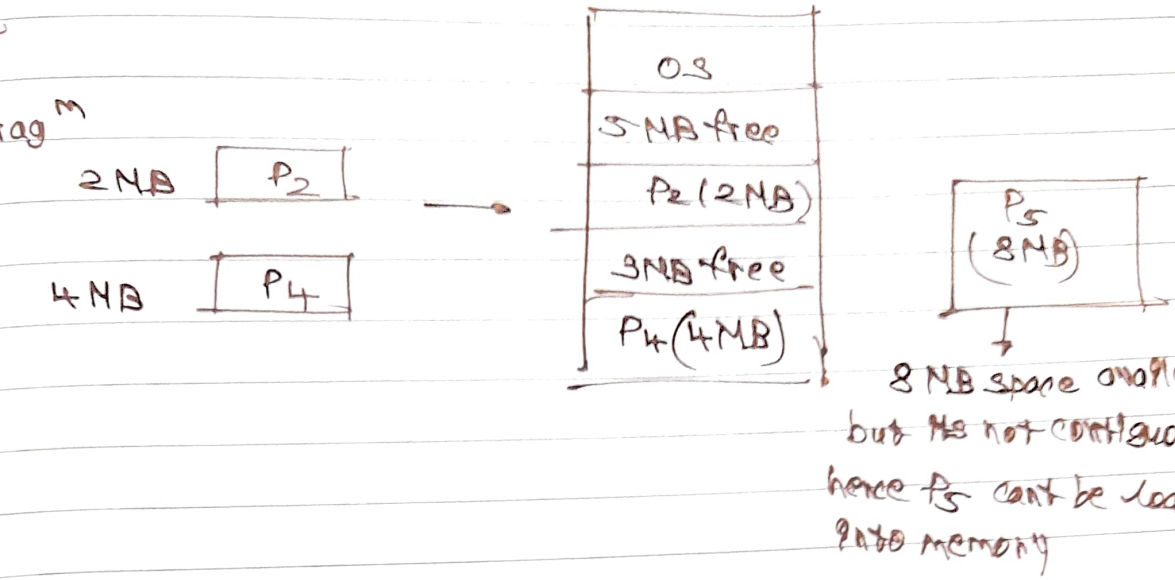
⇓

Process Size = Partition Size

→ No internal fragmentation
→ No limit on size of process
→ Better degree of multiprogramming

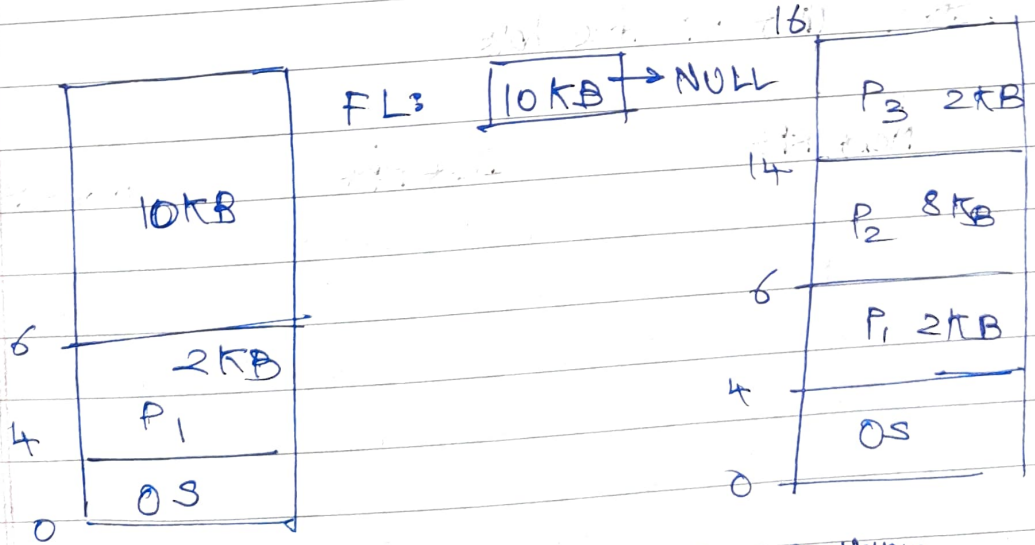
Problem

Ext. Frag^{n}

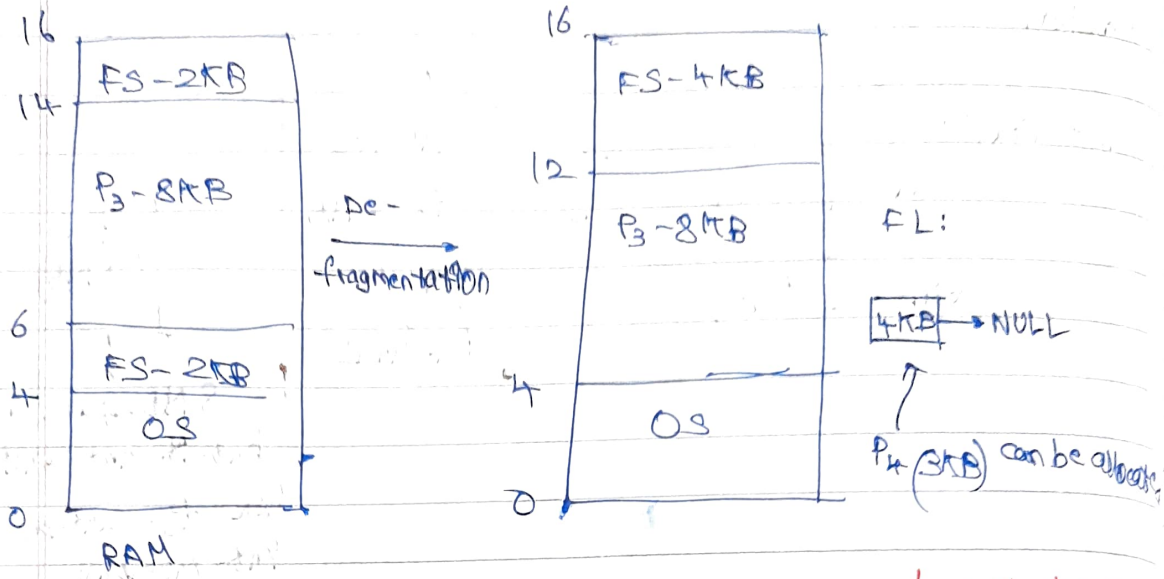


- OS uses freelist (linked list) to manage freespace → stores starting addr

- Defragmentation ⇒ Free space → to one side & used space to other side move nearer so that F.S. contiguous form one main hai



Say, Total free space = 4KB (2+2) → contiguous ⇒ Process 3KB to cant be allocated → This is Fragmentation

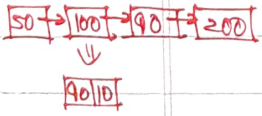


brought together

→ All free partitions are made contiguous & All loaded partitions
→ We can store bigger process in memory
→ Eff. of system ↓ since free space is transferred from several places to single place

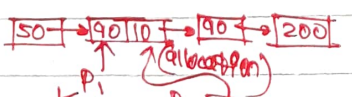
To satisfy request of n size from list of free holes

First Fit
→ Allocate 1st hole that is big enough
→ Simple & fast
P_1: 90 KB



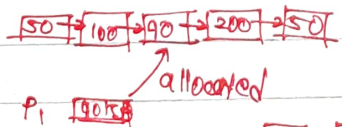
FL:

Next Fit
→ Enhancement of First Fit
→ Search always starts from last allocated hole
→ Same Adv as First Fit

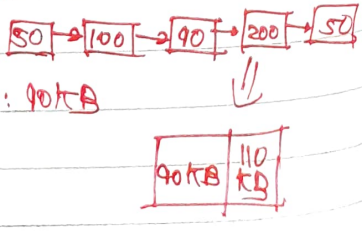


Next allocation starts from $P_1 + 1$

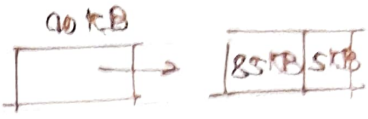
Best Fit
→ Allocate smallest hole that is big enough
→ Less Internal Frag^m
→ External Frag^m
→ Slow



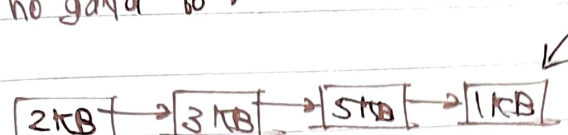
Worst Fit
→ Allocate largest hole that is big enough
→ Slow
→ Leaves larger holes that may accomodate other processes hence ext. Frag^m ↓



Best Fit → External Frag^{n} since → P: 85 KB



agar ye bhi agar aur process ke saath ho gaya to ⇒



→ external Frag^{n}

ye hota hai due to choote chunks of unused space, ye 2 KB agar 7 KB me tabdil ho jaye, to problem solved i.e. ↑ space of every free space → This is done by Worst Fit

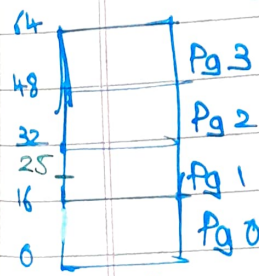
W.F:

→ 110 KB is free space hence all $P_i \leq 110$ KB can be allocated

- **Paging**

Page size = Frame size !!

Non-contiguous allocation of a process

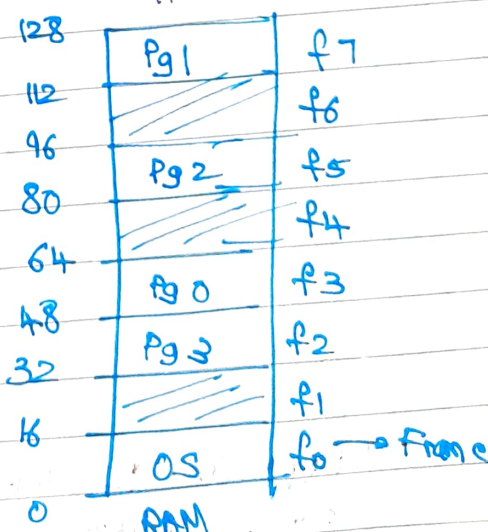


→ 64 KB

Logical add^{r} space of P_0

Logical add^{r} space
↓
Fixed size division
↓
Page

Kisi process ko non-contiguous allocation ke zariye allocate karna
⇓
ext^{n} frag^{m} ✗



(Physical add^{n})

Each process has its own page table

stores base add^{r} of each page

Page table →

(Binary)	Logical Page no
00	pg 0
01	1
10	2
11	3

(Binary)	Physical Page no.
	f3
111	f7
101	f5
	f2

P_0 → Logical add^{r} → 64 bytes = 2^6 ⇒ 6 bits req. to identify each add^{r} uniquely

Let, add^{r} = 25 = 01|100

page no (p) | offset (d)

(p) → No. of pages = 4 = 2^2 ⇒ 2 bits for page

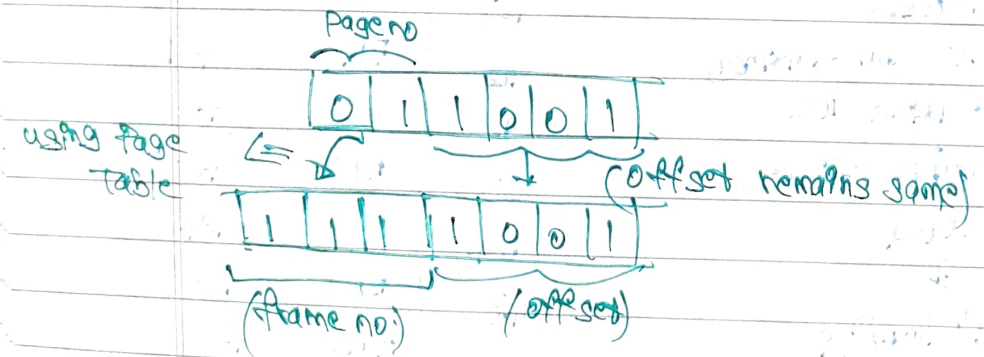
25 → p ka base add^{r} + offset = 16 + 9

25th loc^{n} in Logical add^{r} space ≡ (Pg 1 in f7) 112 + 9 = 121

(112 = Base, 9 = offset)

121 = 111|100| → frame no. 7, → offset

No. of frames = 8 = 2^3 ⇒ 3 bits for frame

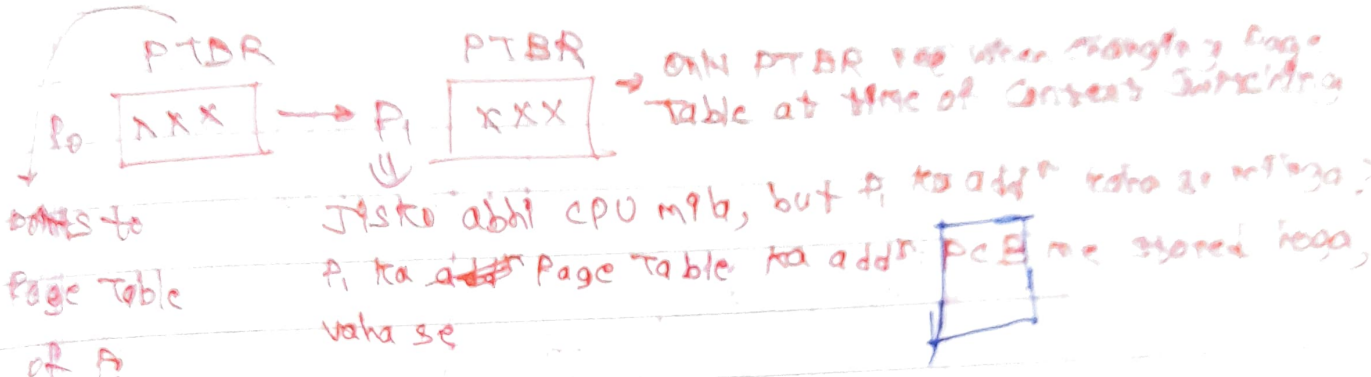


→ Page Table is stored in main memory at the time of process creation & uska base add^{r} → stored in Process Control Block (PCB)

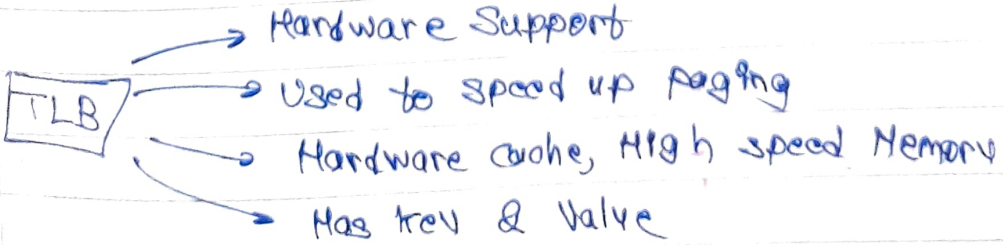
Context Switching me PTBR ki value change ho jayegi

→ Page Table Base Register (PTBR) is present in CPU that points to current page table

→ Jab P0 process I/O karne jayegi, to uski our process ko CPU ka access milega, hence page table bhi change hona chahiye

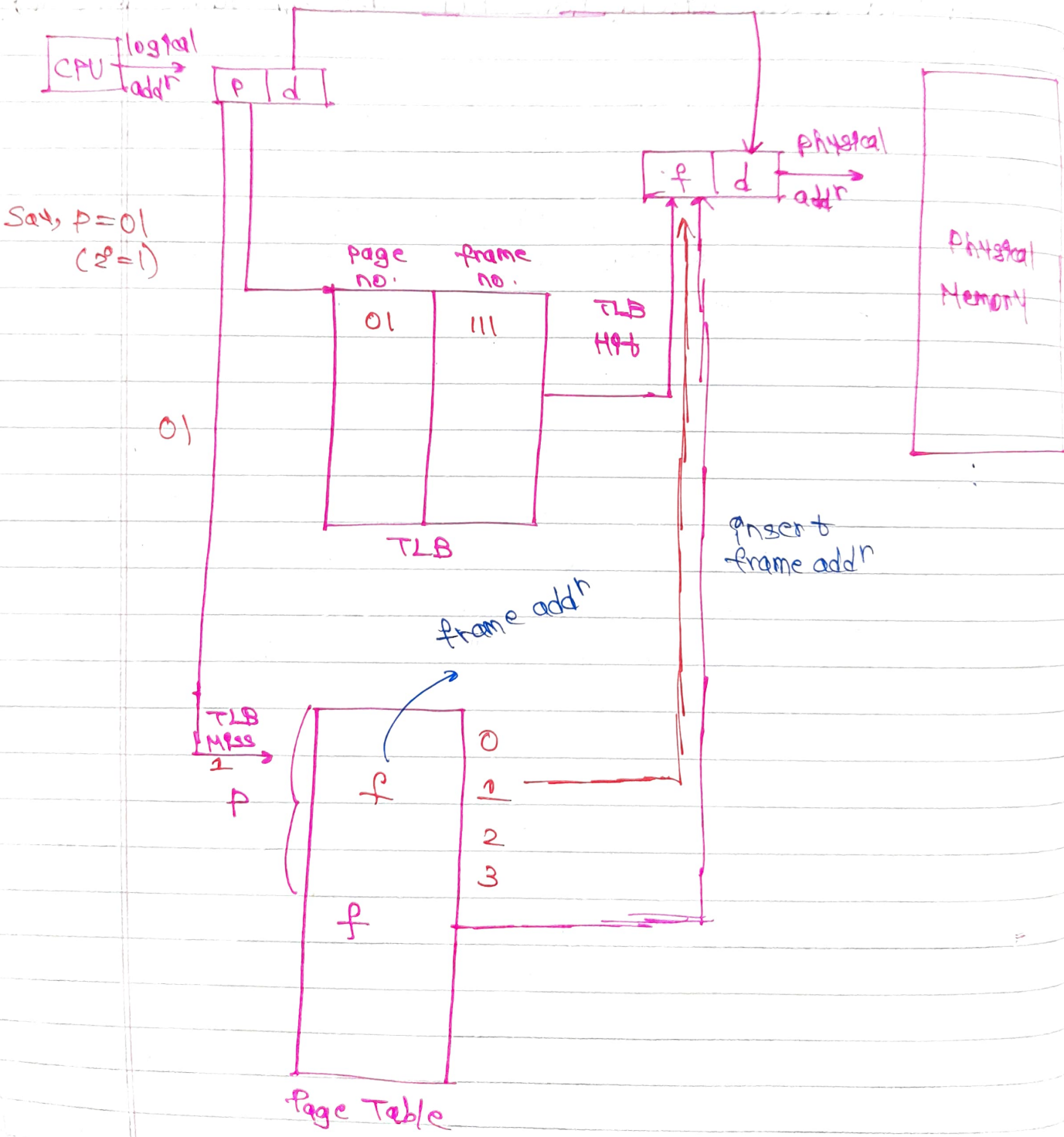


• Memory Refrences → Page Table → stored in RAM, logical se physical addr mapping me page table ko ma ek refrence dekha hu, vaha se mujhe frame mila (corr. to that page) (frame ka base addr mil gya), then I add offset ⇒ Overhead
Paging → slow ⇒ Use TLB



Paging Hardware with TLB

"d" remains as it is



Parantu, jab bhi Context Switching hogi, Page Table go pehle P_0 ke corr. tha, vo P_1 ke corr. ho jayga, to kya har baar TLB ke entries flush karna? Charges nahi kiye to Isolation & Memory Protection ka problem ho jayga

Sol^n:-

① TLB Reset → Baar Baar flush karne par, Cache ka purpose hi defeat ho gaya, cause enough entries dali hi nahi

② Unique Identifier in TLB that identifies unique processes

Say, P_0 ke liye Pg = 10 ≡ frame = 100

TLB →

ASID	Pg no.	frame no
0	10	100

(Add^r space Identifier)

→ Now Context Switching hone par, jab P_1 Pg no 10 ke corr. entry dekhta hai TLB me, to kyuki uska ASID match nahi ho raha ⇒ TLB Miss. So, Page Table ke pass jaya

0	10	100
1	10	45

- fun() ε 3 — 4KB → OS divides it into 2KB & 2KB ke Pages

2KB | fun() ε | Pg 0 → f5 (say)

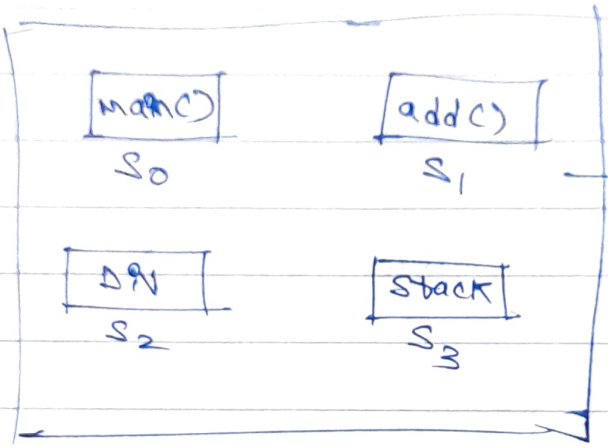
2KB | 3 | Pg 1 → f3 (say)

fun ε Partial part3 — f5
fun() second part — f4, f3, f2, f1, f0

⇒ access inefficient ho jata → issue in Paging has fun ka (overhead aa gaya)

Soln :- Segmentation

Diff^n Segments for diff^n P's



Segment size is different for each segment

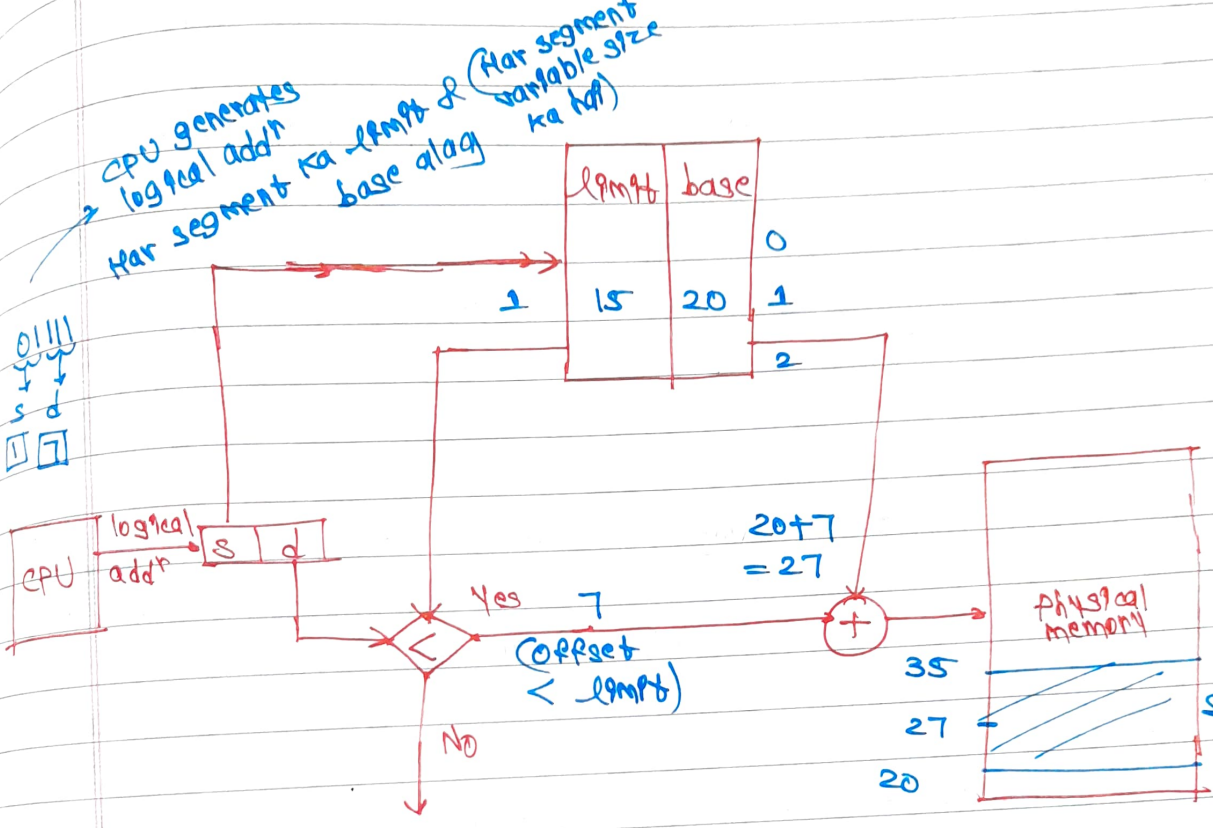
MMU → add^n Translation (Logical → Physical)

Logical add^r → | Seg. no. | Offset |

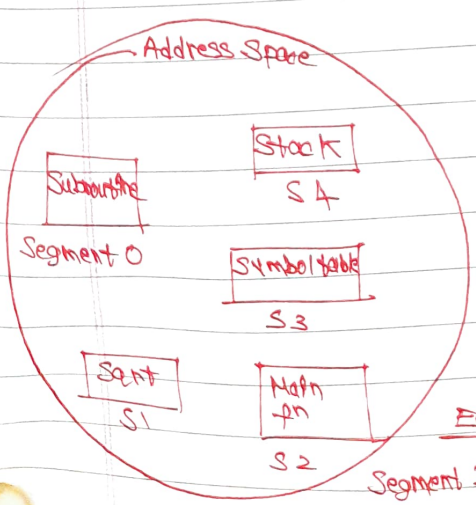
Seg. no. → S

Offset → d

Segmentation Hardware



Example of Segmentation



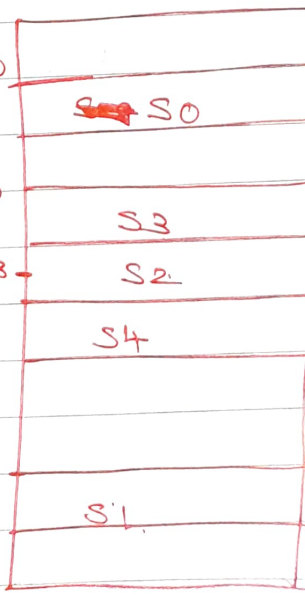
Segment Table

Limit	Base
1000	1400
400	6300
400	4300
1100	3200
1000	4700

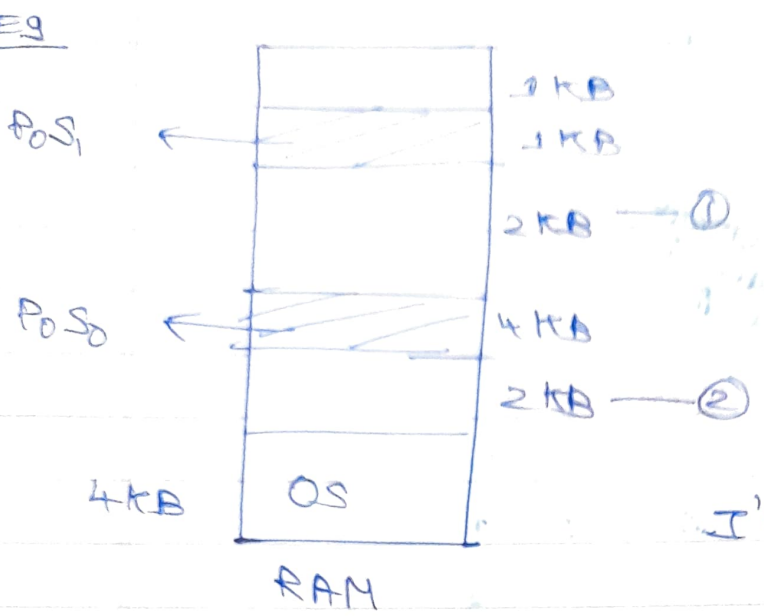
Eg

Segment 2 = 4300 + 53 = 4353

(Since 53 < 400)



Eg



P1 — S0 — 2 KB
S1 — 3 KB

can be allocated (at 1/2)

But not S1

∴ In Segmentation, Ext. fragm ✓

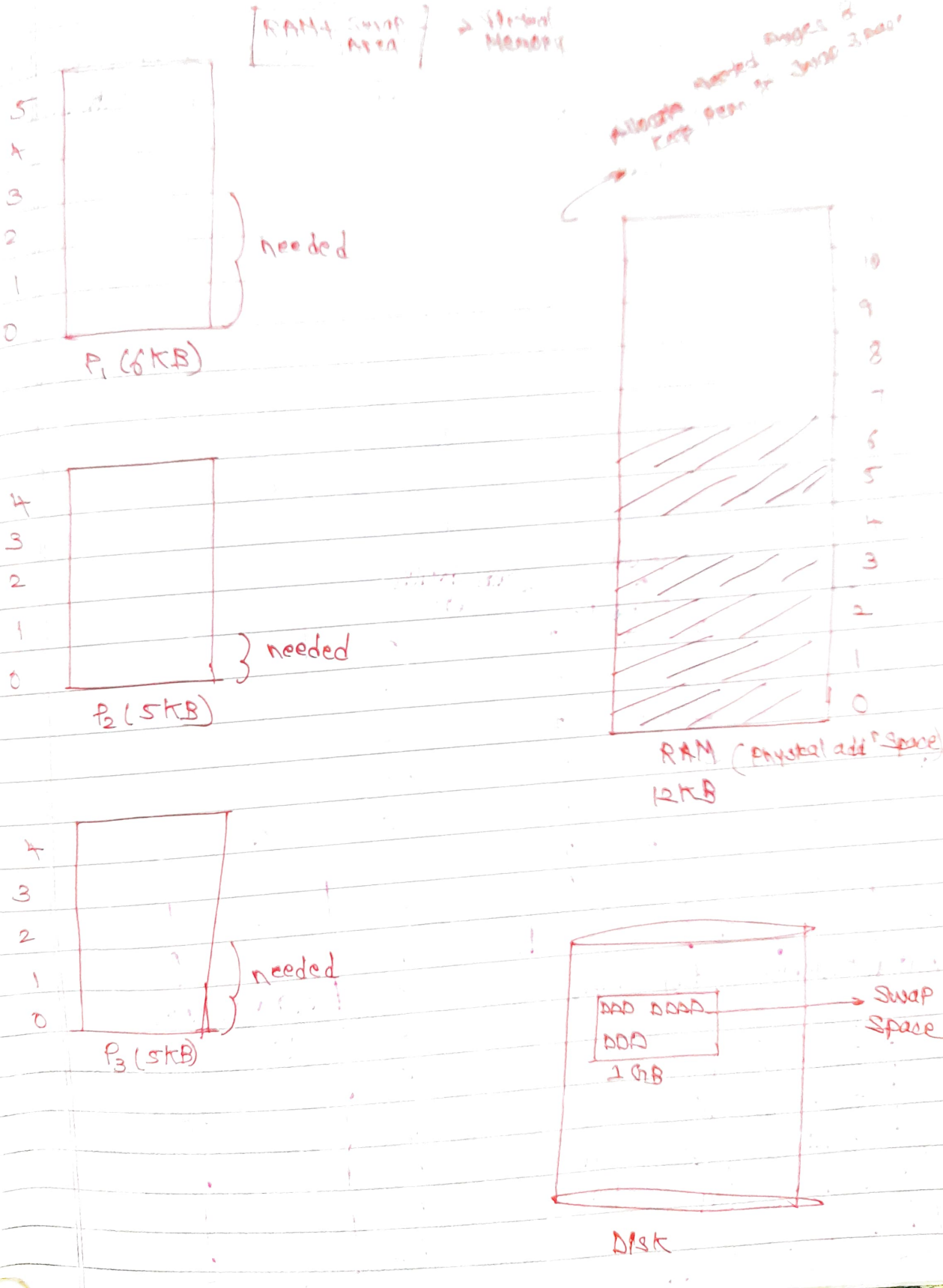
Int fragm ✗ cause I'll allocate pura segment

Say, 100 KB ka program, 2 KB each ke pages me divide
⇒ No. of pages = $\frac{100}{2}$ = 50 ⇒ Page Table → len = 50, but (o
program may just contain 2 segments ⇒ Segment Table →
only 2 entries

- Diff^n size of segment → Not good for swapping
  Segment Size ∝ Time req. for swapping

- Internal fragm → Paging ✓
  → Segmentation ✗

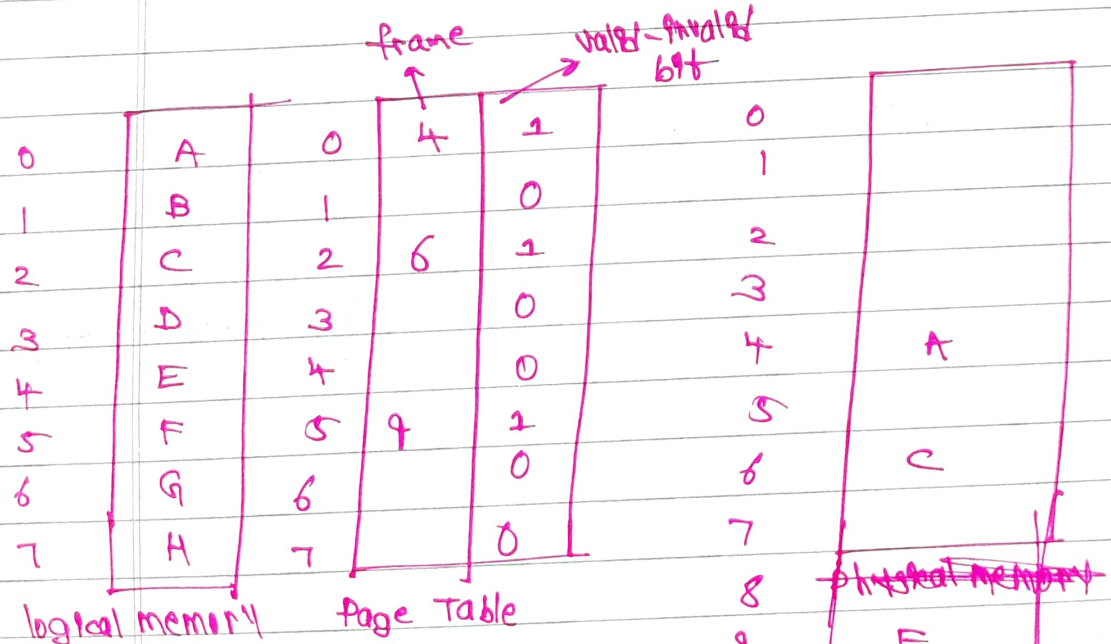
Virtual Memory Management

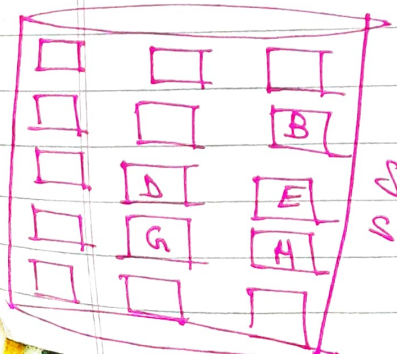


→ User is able to run multiple programs using single RAM
→ Degree of Multiprogramming ↑
→ User can run a program which is larger than physical memory
→ Bohot saare processes ke thodi needed pages rakh ke, CPU Utilization ↑
→ Used Nowadays

- Pure Process ko Swap In - Swap out karna → Swapper
  ~~Blk~~ Page → Pager

- Valid - Invalid Bit = 1 → Associated page is both legal & in memory
  = 0 → Page is either not valid (not in LAS of process) or is valid but is currently on disk

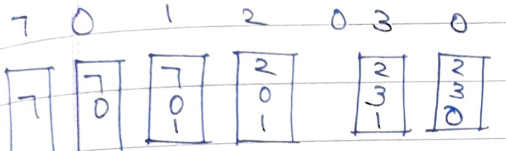




locality of reference → Vo pages jo pagger ko log rakha hai, ki inki need hogi

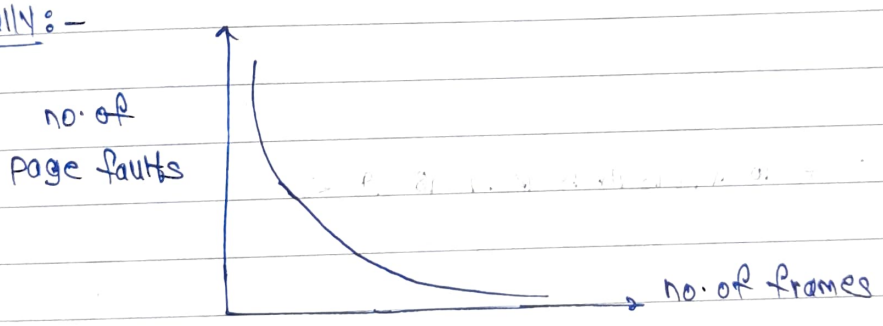
Page Replacement Algos

① FIFO

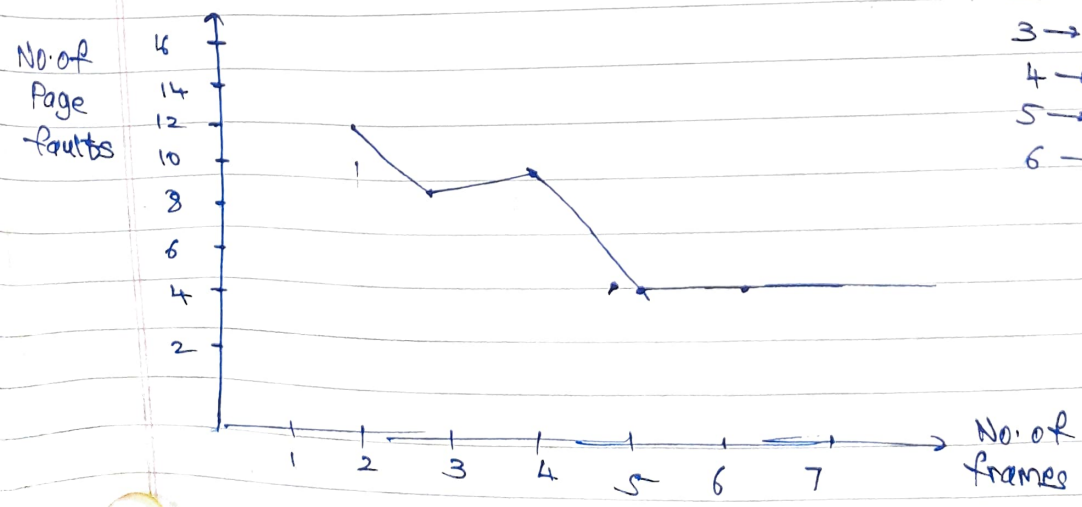


1 2 3 4 4 5 6 → ~~Total~~ no. of page faults

Normally :-



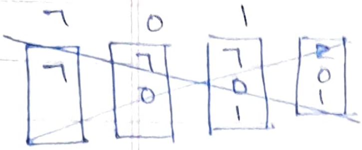
- Belady's anomaly on FIFO



2 → 12
3 → 9
4 → 10
5 → 5
6 → 5

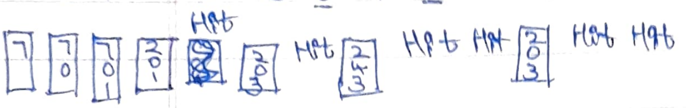
② Optimal Page Replacement

→ Cant be implemented



1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

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



↓
Now full

Most Recent Access

7 → 18 → Sabse late
0 → 5
1 → 14

↓
So Replace 7

③ LRU → Implement using Stack (Top being Most Recently Used)

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

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

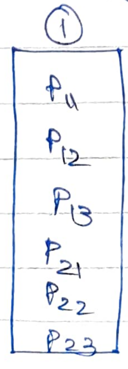
(Jaha Hit vaha Hit likha rest Page faults)

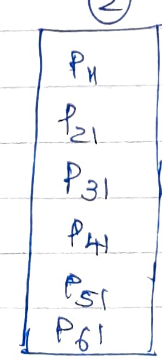
7:1
0:2
1 8:3

↓
0 is used at 5
So its not LRU

Thrashing

① ②





① → Page Faults ↓ cause P_1 ka Page 1, then 2 then 3 lene me time lagega

but ② me har process ka 1st page hi hai & baaki page ko access karne jaye to miss ho jaata hai

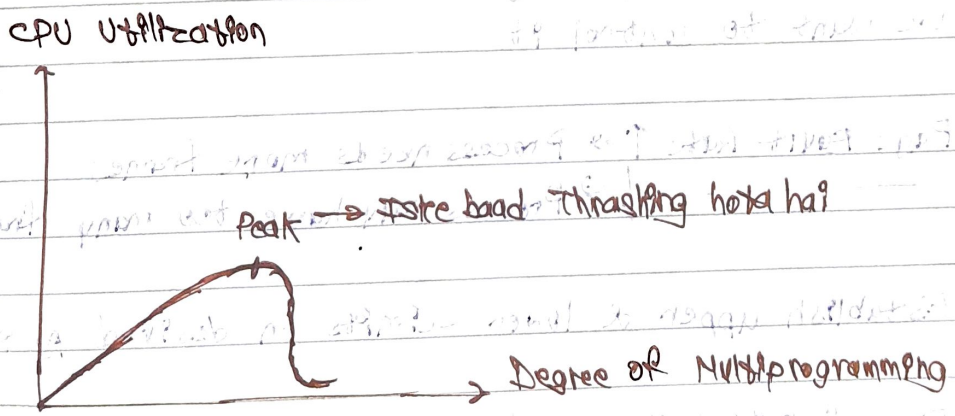
Problem: A process doesn't have no. of frames to support pages in active use leading to Page Faults

Soln: Replace some pages

Problem in Soln: All Pages are in active use, hence it must replace a page that will be needed again right away Hence it faults again & again, replacing pages that it must bring back in immediately

→ Spends more time servicing page faults, than executing process

If Degree of Multiprogramming ↓, say Degree = 2, & both processes I/O karne chale gaye, to CPU free hai, hence CPU utilization ↓



Hence Processes to khup padi hai, but saare hi page fault service karne me busy hai, P2 → exec chalu hone par koi aur page chahiye → service page fault → koi aur process ka page uda dete ho → dusri process ko vo page jab vapis chahiye to fir se page fault generate hota hai & swap-in, swap-out me hi pura time chale jata hai

- Locality → Ek process ka local area (main ke pass ke f's) hamesha rahte hai bhi process ek particular locality me hi to & fro.

Techniques
→ To Handle Thrashing

1) Working Set Model
→ Allocate enough frames to accomodate a process' current locality
→ It'll only fault when it moves to a new locality
→ If allocated frames < size of current locality, process is bound to thrash

2) Page Fault Frequency
→ Thrashing has High Page Fault Rate
→ We want to control it
→

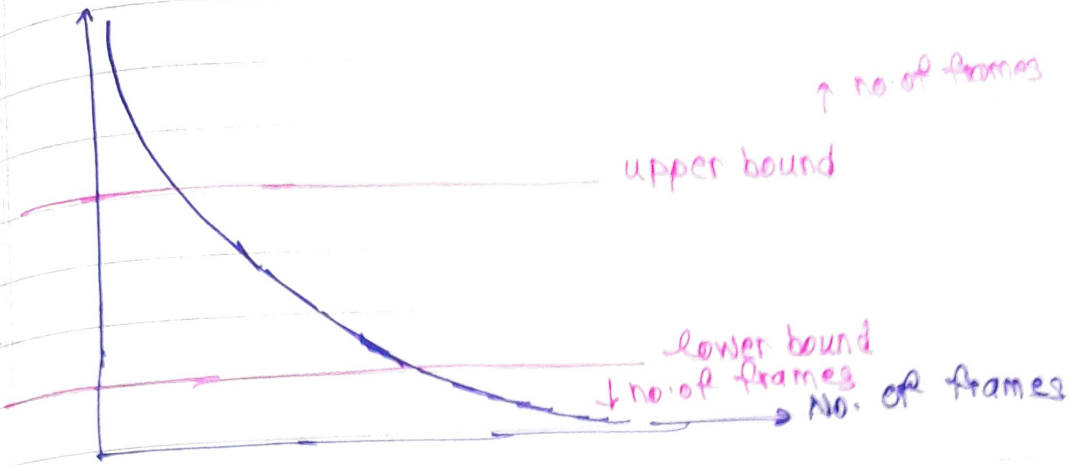
→ Page Fault Rate ↑ ⇒ Process needs more frames
↓ ⇒ Process may have too many frames

→ Establish upper & lower limits on desired pg fault rate

→ If pf-rate > upper lt ⇒ Allocate the process another frame
If pf-rate < lower lt ⇒ Remove a frame from process

→ By controlling pf-rate, Thrashing can be prevented

page fault rate



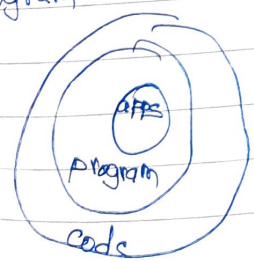
storage mgmt → self study

Lecture

File is a sequence of bytes lying on hard disk (SSD or CD or PD)
It has a name, owner, size etc

Application is a program that runs on an environment created by OS, under the control of OS
Hence called User Application

Code → any piece of complete / incomplete programming language
Program → piece of complete code



Some app opens file (it doesn't open itself)
File is displayed by program which owns it