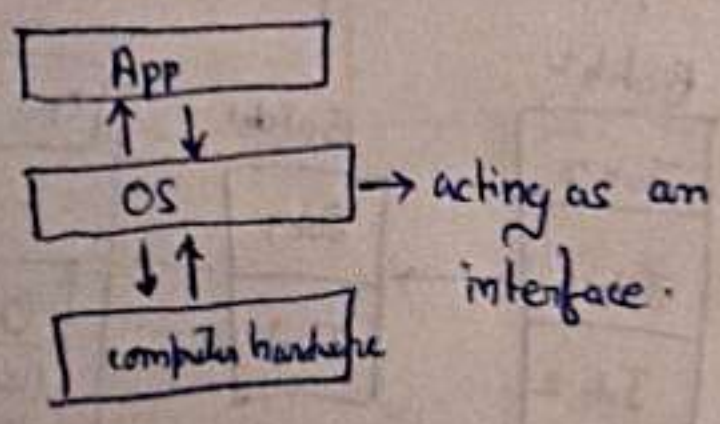
Operating System

Note → DRY → Do not Repeat Yourself

- Application Software → perform specific task.
- System Software → operates and controls computer system and provides platform to application software to run.
- Operating System → ⊡ A software that manages all resources of a computer system, both software and hardware and provides a platform/environment where user execute his program in a convecient way by abstraction.
  - ⊡ Also, acts as a resource manager.
  - ⊡ Os is made up of collection of System Software.
- No Operating System? ⊡ Bulky and complex App.
  - ⊡ Resource exploitation by 1 App
  - ⊡ No memory Protection.
- Functions of Operating System →
  - ⊡ Access to computer hardware
  - ⊡ interface between user and computer hardware.
  - ⊡ Resources management.
  - ⊡ Abstraction provided.
  - ⊡ provides execution of application program by providing isolation and protection.

⊡ 

⊙ Types of Operating System →

(i) Single process Operat Os.
(ii) Batch-processing Os.
(iii) Multiprogramming Os.
(iv) Multitasking Os.
(v) Multi-processing Os.
(vi) Distributed System.
(vii) Realtime Os.

(i) Single process Os.

→ only 1 process executes at a time from the ready queue.
→ Example · MS DOS 1.

(ii) Batch Processing Os →

Punch card concept

Example → ATLAS

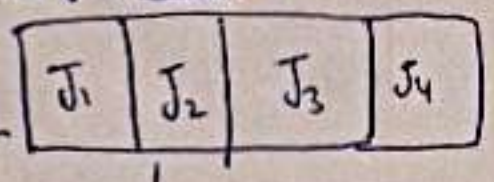
⊙ Os Goals →

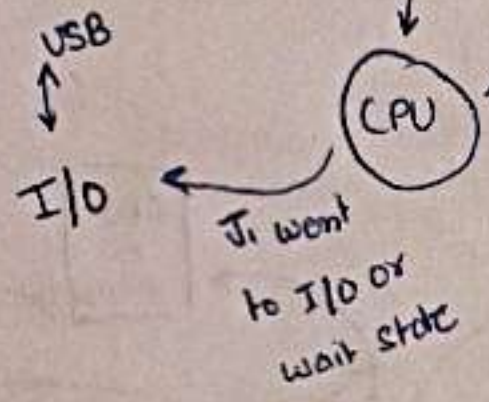
(i) Maximum CPU utilization.
(ii) Less process starvation.
(iii) High priority job execution.

problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time.

(iii) Multiprogramming Os →

- CPU is single as earlier 2 Os.
- Ready Queue (Queue having Jobs which ready for execution.) is present it.
- Example :→





- Example :→ THE
- Job : Task

⊙ Context Switching :→

(i) Our Job J_1 was first executing on CPU, from there when it move to I/O state and J_2 starts its execution on CPU, this process is called Context Switching.

(ii) It also ensures that 2 Jobs don't conflict with each other.

Features :→

(i) Single CPU.
(ii) Context switching happens.
(iii) Switch happens when 1 proces/Job goes to wait state.
(iv) CPU idle time reduces.

(iv) Multi Tasking Os →

Example - MIT

- logical extension of multiprogramming.
- Features :→
  (i) Single CPU.
  (ii) Can run more than 1 task/Job simultaneously.
  (iii) Context switching and time sharing used.
  (iv) ↑es responsiveness and ↓es CPU idle time.

⊙ Note →

Time sharing :→
Here we do the context Switching based on some amount of time alloted to a Jdo to execute through CPU.

(v) Multiprocessing Os → ▣ more than 1 CPU in a computer.

▣ Features → (i) Yes reliability (CPU fails, other can work.
(ii) Better throughput.
(iii) Less process Starvation (As more number of CPU more different Jobs can be executed at once.)

Example - Windows NT

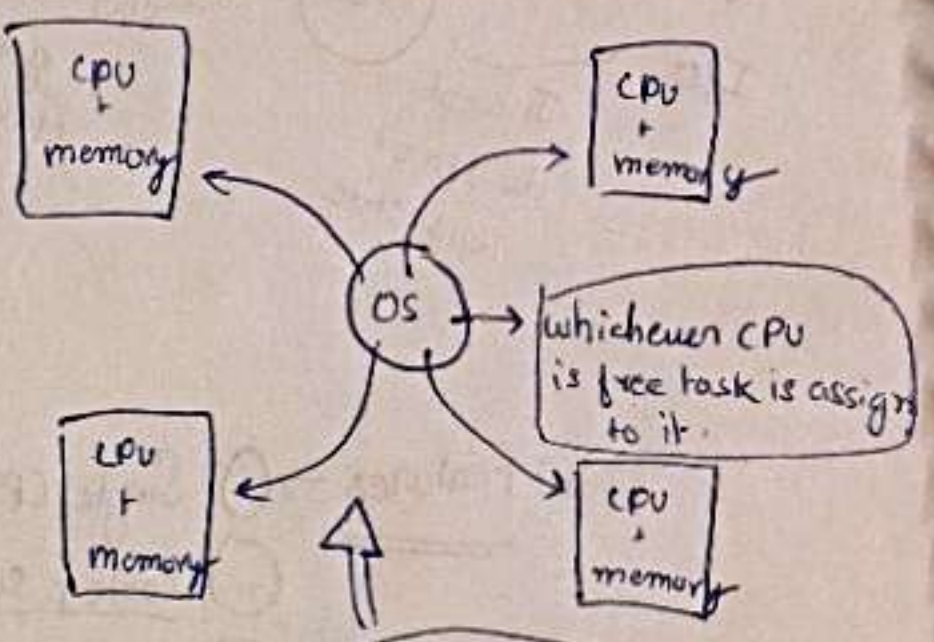
This is used nowadays.

(vi) Distributed Os →

→ manages many bunches of resources morethan 1 (CPU, memory, GPU, etc).

→ Loosely coupled Os.

~~Example~~ →



Example → LOCUS

Leetcode example our code getting checked by different CPU which are free.

(vii) RTOS → (Real time Os)

→ ▣ Error free, computations within tight time boundaries (Fast processing)

→ ▣ Used in AIR Traffic control system, Robots, etc.

Example → ATCS

⊙ Program → ▣ Executable file containing set of instruction to perform a specific task on our computer.

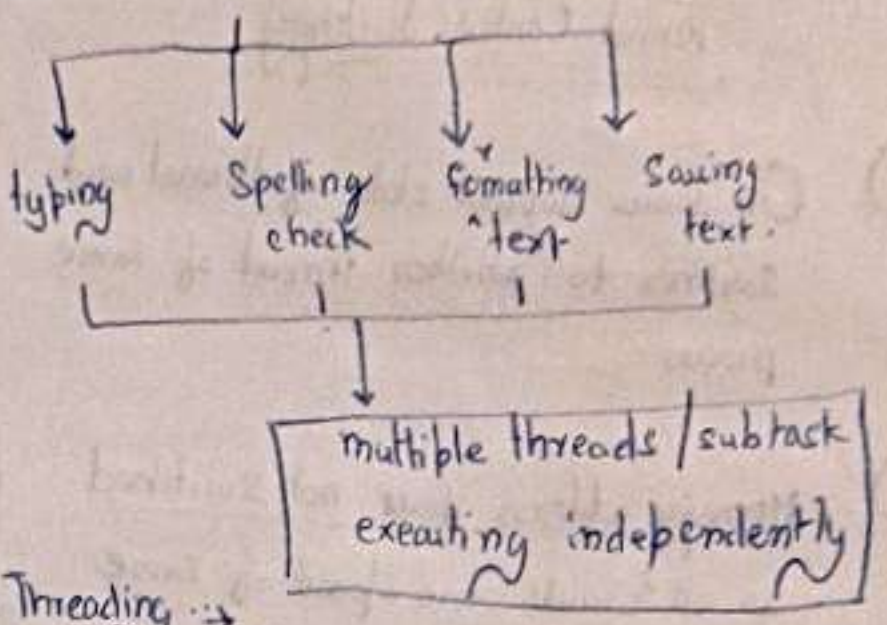
▣ It is a ready to be executed, compiled code, stored in disk.

⊙ Process → ▣ Program under execution. Resides in RAM.

⊙ Thread → ⊡ Light weight process executes independently. (or, a sub process).

⊡ ~~Single sequence stream within a process~~

⊡ Example → Multiple tabs in browser, text editor



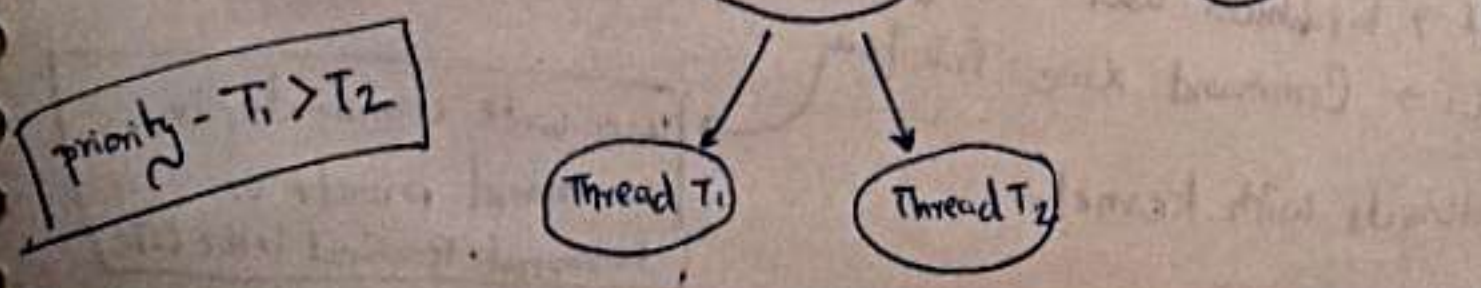
⊙ Difference TMulti-Tasking Vs Multi-Threading :→

Multi-Tasking	Multi-Threading
(i) MultiTasking is execution of more than 1 task simultaneously.	(i) Multi-Threading is a process divided into several sub-tasks called threads, which executes independently.
(ii) More than 1 process being context switched.	(ii) More than 1 threads being context switched.
(iii) No. of CPU = 1.	(iii) No. of CPU >= 1.
(iv) Isolation and memory protection exist. As here different process are there	(iv) No isolation and protection. As As here there is 1 process (main), and other process are its subparts sharing same memory and resources.

⊙ Thread Scheduling →

⊡ Threads are scheduled Based on their priority.

⊡ For example →



⊡ Now, here on the time alotted by CPU, OS will give max time to T_1, less to T_2. (on the time alloted by CPU).

⊙ Thread Context Switching Vs. Process Context Switching.

Thread Context Switching	Process Context Switching
(i) Os saves current state of thread and switches to another thread of same process.	(i) Os saves current state of process and switches to another process by restoring its state.
(ii) Memory address space not switched as all threads are apart of same process.	(ii) Memory address space is switched.
(iii) Fast Switching	(iii) Slow Switching.
(iv) CPU cache state is preserved as process is same.	(iv) CPU cache state is flushed as process is different.

Memory closed to cpu. It is faster than RAM.

⊙ Components of Os

- Userspace
  - Apps runs here conviniently.
- Kernel
  - Access to underlying hardwares.

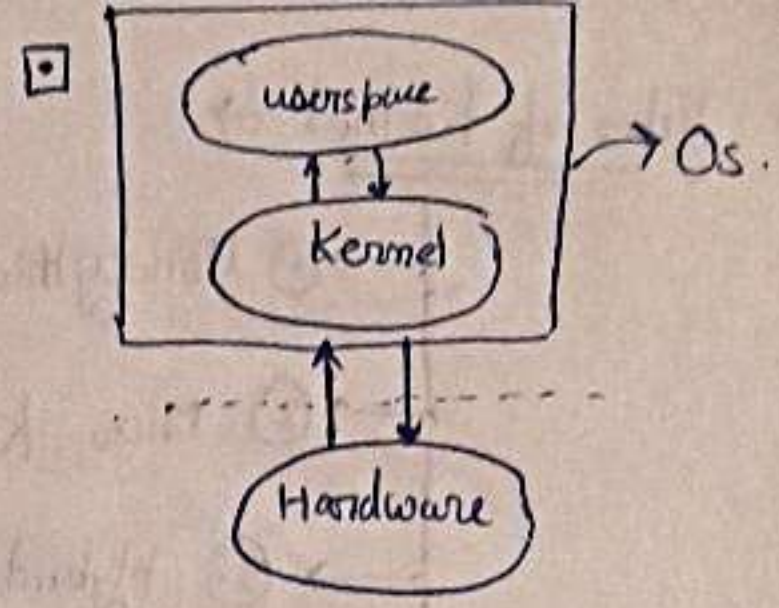
⊙ Userspace ⇒ ⊡ App runs here conviniently.

① GUI → Graphical user Interface.
② CLI → Command Line Interface

- User directly interacts with Frontend interface. Eg - making Folders with mouse click.
- User write commands in command prompt or terminal. Terminal is the CLI.

⊡ It interacts with kernel.

⊙ Kernel ⇒ ⊡ Heart of Os.
⊡ interact directly with Hardware.



⊙ Functions of kernel ⇒

→ ⊡ Process Management
- (i) Process creation and termination.
- (ii) Process and Thread Scheduling.
- (iii) Process Synchronization.
- (iv) Process communication.

→ ⊡ Memory Management
- (i) allocate and deallocate memory.
- (ii) Free space management. (Keep tracks of which part of memory is used by which process).

→ ⊡ File Management
- (i) Create, Delete files.
- (ii) directory management. (directory inside our system makes use of the concept of tree data structure).

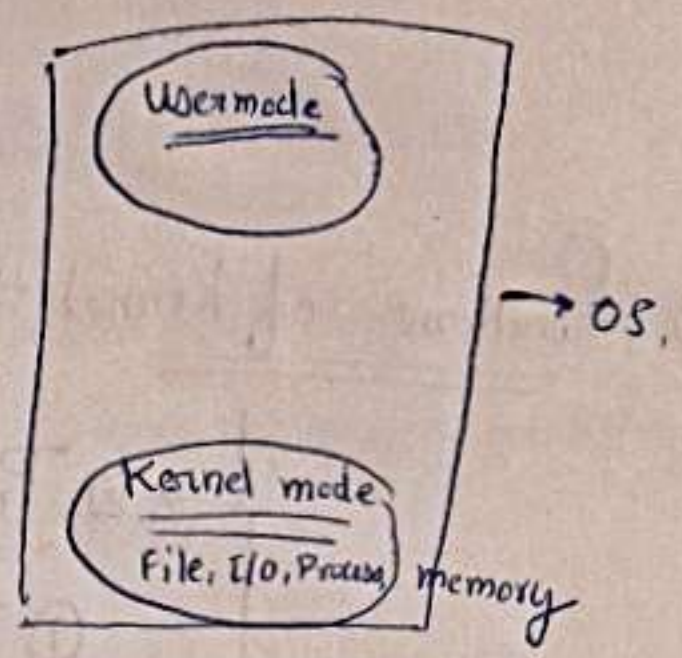
→ ⊡ I/O Management ⇒
- (i) Manage and control I/O devices and I/O operations.
- (ii) Spooling
- (iii) Buffering
- (iv) Caching

If saving multiple jobs in a particular area and executing them one by one at a different speed.
Example - Print spooling

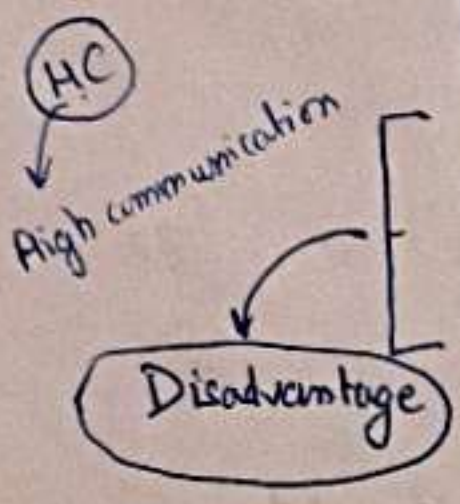
If we want to use a data after some so. we keep the data as buffer.
Ex - Youtube video buffering.

⊙ Types of Kernel :→

→ ① Monolythic Kernel.
→ ② Micro Kernel.
→ ③ Hybrid Kernel.
→ ④ Nano/Exo kernel.



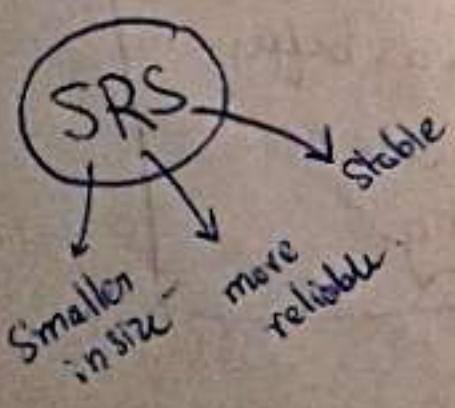
① Monolythic Kernel →



→ (i) All functions are in kernel itself (Process, memory, file, I/O)
→ (ii) Bulky in size.
→ (iii) Memory requirement high.
→ (iv) Less reliable, one module crashes → whole kernel is down.
→ (v) High performance as communication is fast.
→ (vi) Example → Linux, Unix, MS Dos.

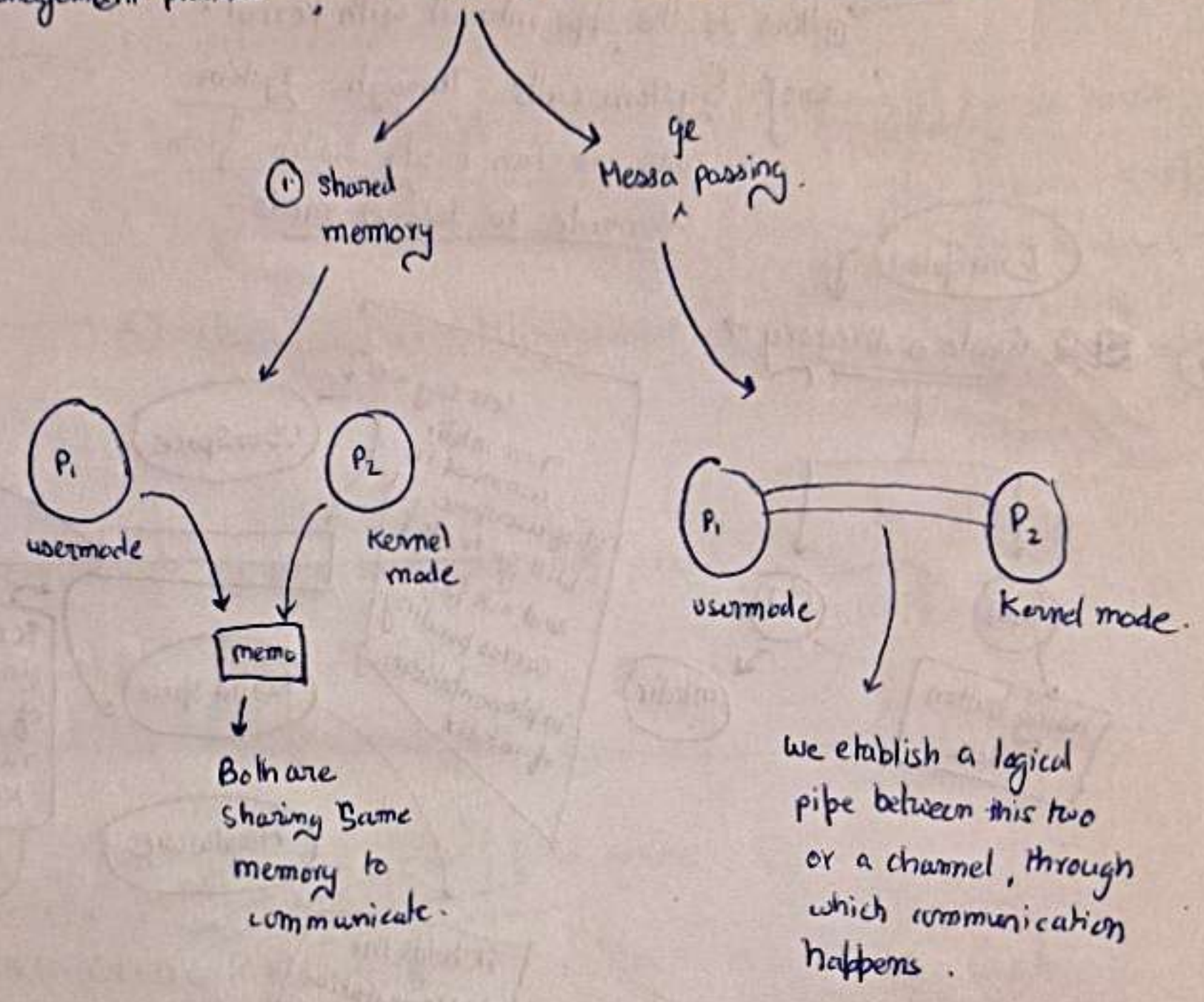
② Micro Kernel

→ (i) Major functions → Memory management
→ Process management.
→ (ii) Here userspace contains the rest (File, I/O).
→ (iii) Smaller in size, more reliable, more stable. → advantages
→ (iv) Performance less, overheading switching btw user and kernel mode. → disadvantages
→ (v) Example → L4 Linux.



Q Usermode and Kernel mode does communication, how?

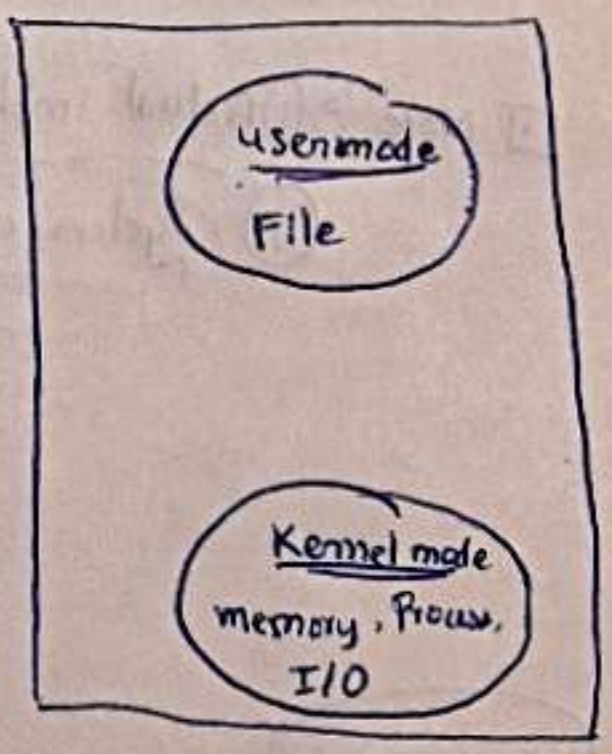
ans:- ⊡ Os → Process management provides ⟶ IPC (Inter Process Communication) to do it.



③ Hybrid Kernel ⇒

→ ⊡ Combined approach of Monolytic and and Micro

→ Example - MacOs, Windows NT (Windows 7+)



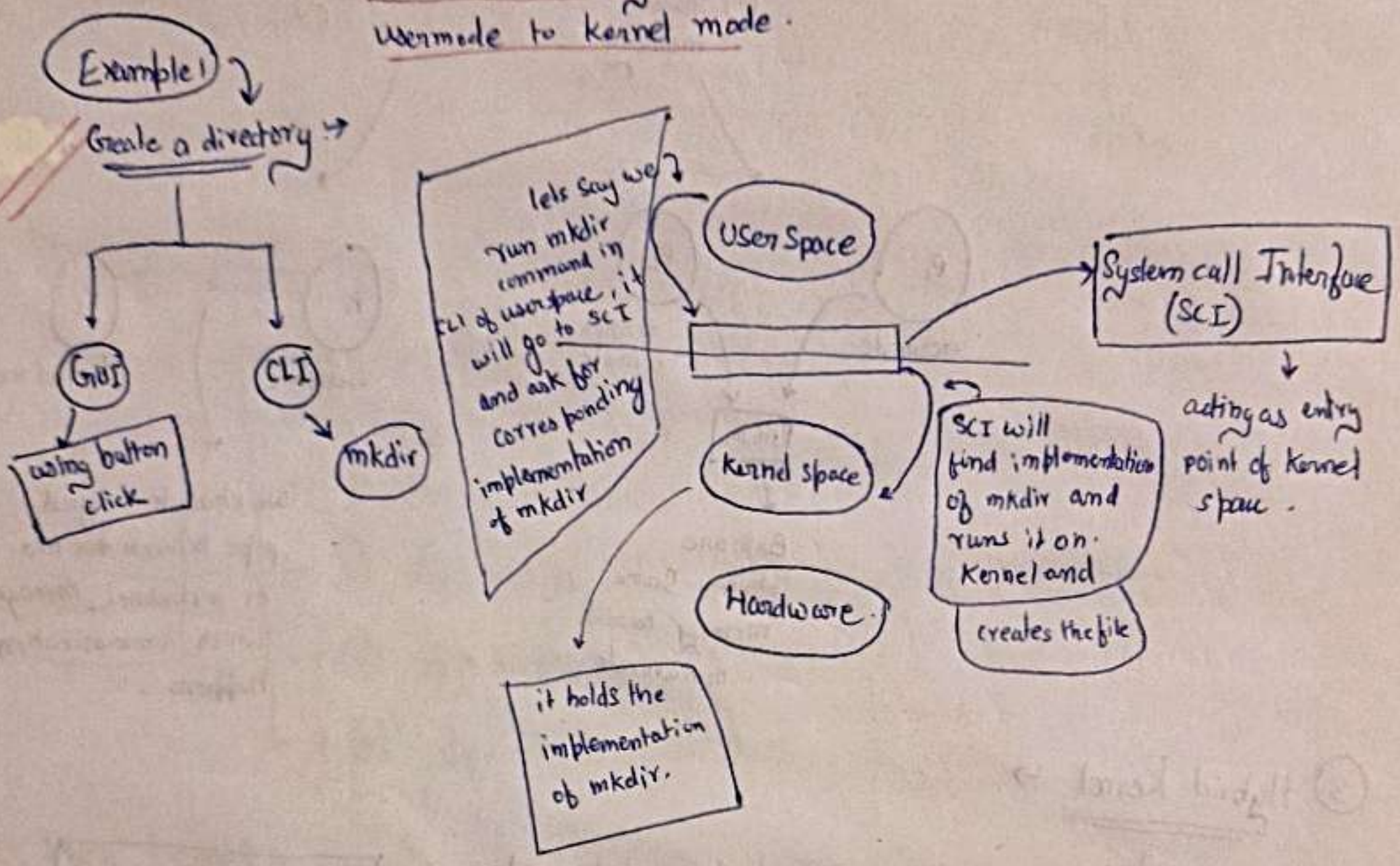
⊙ System Calls. →

⊡ How do the user apps interact with kernel?

ans:- System calls. Through system calls we can easily switch from usermode to kernel mode.

Example1 →

⊙ Create a directory →



⊡ Note :- (i) Actual implementation of mkdir is written in C.

(ii) System calls → implemented in C.

basically acts System calls acts as a interface in to C implementation.

Example2 →

⊙ Creating a process :-

STEP1 : Click on usermode (US) Ka Space .exe file.

STEP2 : System calls execute .exe by switching it to Kernel space. Exec

STEP3 : Now, the process gets created in Kernel space (KS).

STEP4 : Return to userspace (US).

⊙ Types of System calls →

1. Process control
2. File Management.
3. Device Management.
4. Information Maintenance.
5. Communication Management.

⊡ Notes → user mode ← kernel mode.

① Switching from UM to KM?

i. Execute Process
ii. Software interrupt.
calls to interrupt CPU in mid of its ongoing task and tells to perform the important recently arrived task.

① Process Control

- ⊡ end, execute, create, load (EECL) proces.
- ⊡ wait for Time and event.
- ⊡ allocate and free memory.

EWA

② Filemanagement →

- ⊡ (CDOCRW) a file. (create, delete, open, close, Read write)
- ⊡ get or set file attributes.

C GAs

③ Device Management →

- ⊡ read, write, request devices
- ⊡ get, set device attributes
- ⊡ logically attach or detach devices.

RL GAs

④ Information Maintenance →

- get and set
  - ⊡ time or date.
  - ⊡ system data
  - ⊡ files or devices.

TSF

⑤ Communication Management →

- ⊡ Create/Delete communications
- ⊡ Send/recieve messages
- ⊡ transfer status info.

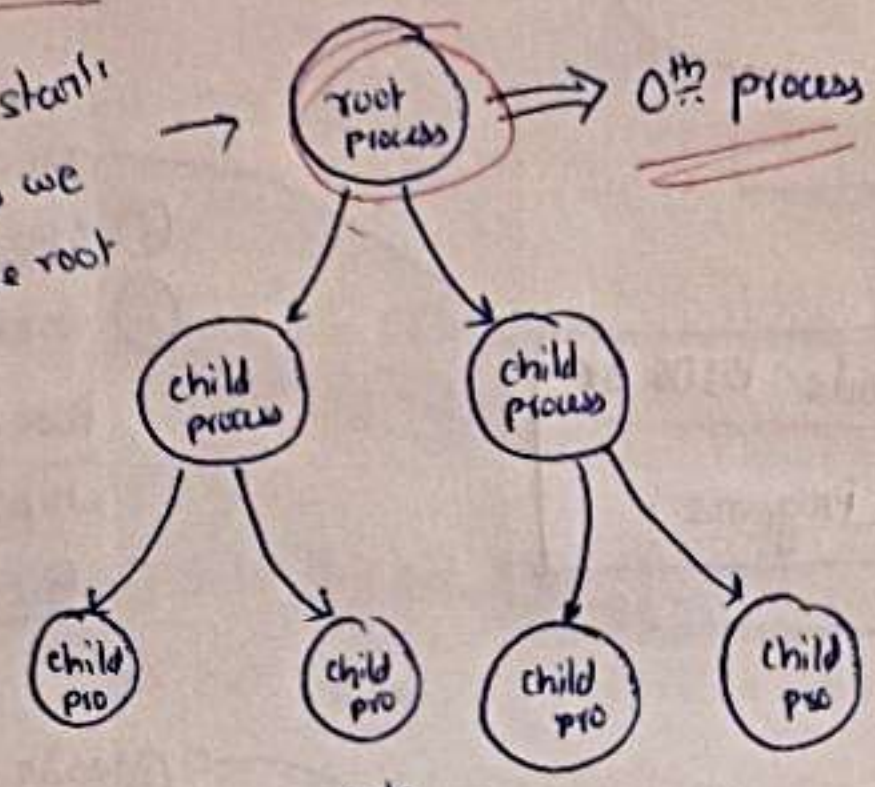
CST

⊙ Example of Window and Unix System calls

Category	Windows	Unix
① Process Control	Create Process()	fork()
	Exit Process()	exit()
	WaitForSingle Object()	wait()
② File Management	CreateFile()	open()
	Read File()	read()
	Write File()	write()
	Close Handle()	close()
	Set Security()	chmod()
	Initialize Security Description()	umask()
	Set Security Description Group()	chown()
③ Device Management	Set Console Mode()	ioctl()
	Read Console()	read()
	Write Console()	write()
④ Information Management	GetCurrent Process ID()	getpid()
	Set Timer()	alarm()
	Sleep()	sleep()
⑤ Communication	Create pipe()	pipe()
	Create file Mapping()	shmget()
	Map View Of file()	mmap()

Note ⇒ (i) fork() ⇒ Creates child Processes.

When our system starts the first process we do have is the root process starts



[1] Note:→

Every Process has a unique PID.

(ii) chmod() ⇒ sets mode of a file.

(iii) Umask() ⇒ sets default info for newly created file.

(iv) chown() ⇒ changes ownership of a File.

(vi) pipe() ⇒ make pipe for message passing.

(vii) shmget() ⇒ shared memory getting it is used.

(viii) mmap(). ⇒ maps or unmap devices to memory.

Q ⊙ What happen when you turn on your computer/ How the Operating System Boots up?

ansf. Next Page →

STEP1: Power on

STEP2: CPU Loads BIOS or UEFI Programs

- BIOS - BASIC INPUT OUTPUT System.
- UEFI - Unified Extensible Formware Interface. Basically BIOS 2.0.

- (i) CPU Initialise.
- (ii) ~~Does~~ CPU goes to Bios chip (non-volatile chip: Rome) and loads Bios Program (BIOS/UEFI)

STEP3: BIOS/UEFI runs tests and initialise hardware

- (i) loads ~~memory~~ settings from memory area, which is backed by CMOS Battery
- (ii) Bios Programs looks with Settings. Instructions in Bios Programs will POST

Power on Self Test

For every hardware we have test cases, if Success moves on to STEP, else display error message.
Ex → RAM - if not present CPU beeps.

STEP4: BIOS/UEFI hands off to Boot device (i.e., searches for Bootloader and hands off the next steps)

- Boot devices:
  - (i) CD, USB
  - (ii) SSD, HHD.
- Program which boot/on/start OS is called Boot loader.

- For old Systems, Boot loader present in MBR (Master Boot Record)
  - ↳ stored in 0th. index of disk.
- For new Systems, Boot loader present in EFI (Extensible Formware Interface), makes Partition in disk which stores Boot loader.

Bootloader loads the full OS.

Different Boot loaders
- (i) windows - bootmgr.exe.
- (ii) Mac - boot.efi
- (iii) Linux - GRUB.

⊙ 32 bit Vs 64 bit Operating System :→

⊙ 32 bit Os → ⊡ holds 32 bit registers.
⊡ can access 2^{32} (4GB) unique memory address.
⊡ 32 bit CPU architecture can process 32 bits of data and info.

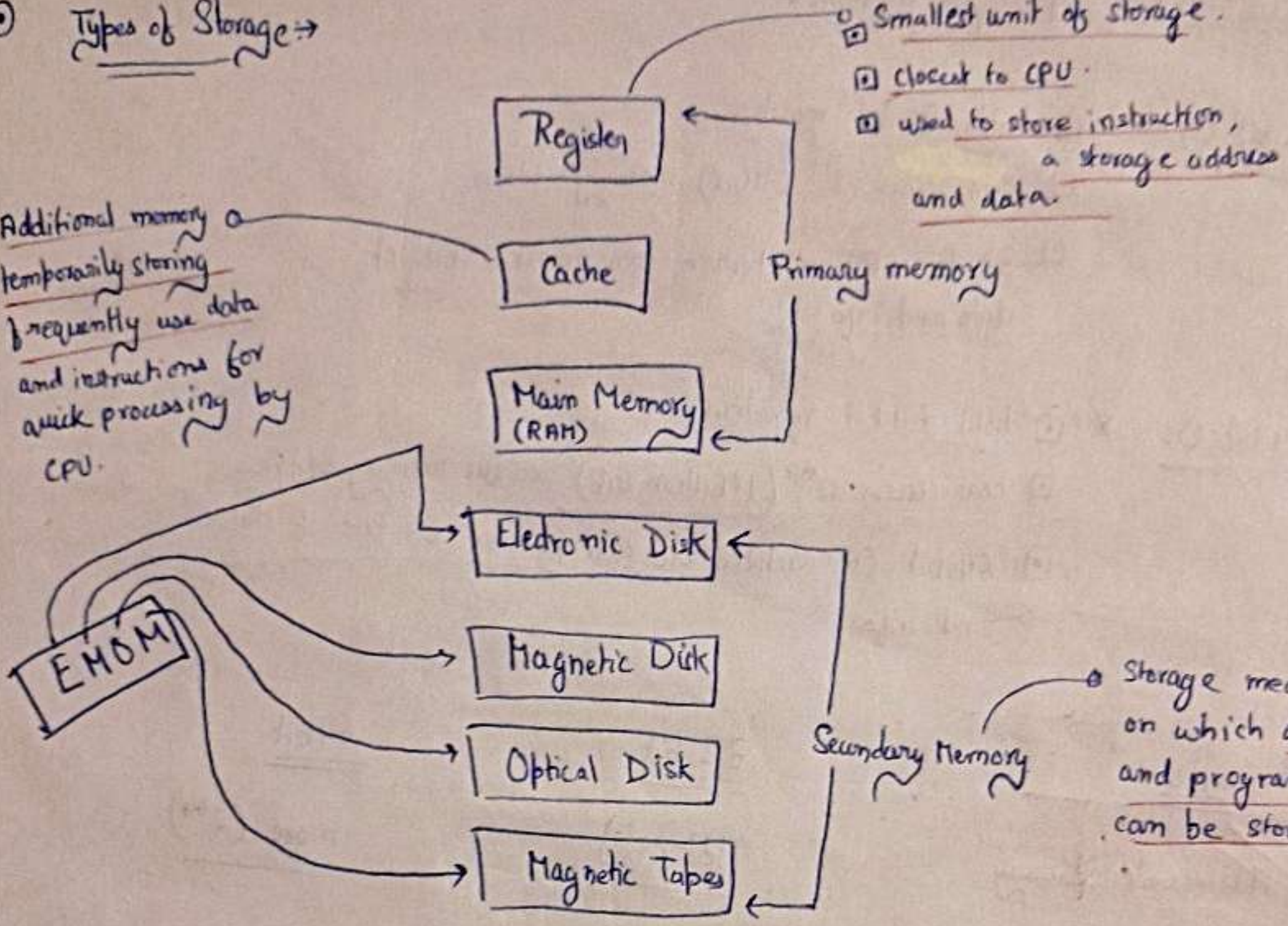
⊙ 64 bit Os :→ ⊡ hold 64 bit registers.
⊡ can access 2^{64} (17 Billion GB) unique memory address.
⊡ 64 bit CPU architecture can process 64 bits of data and info.

⊡ Parameters	32 bit	64 bit
(i) Addressable Memory	less (2^{32})	more (2^{64})
(ii) Resource Usage	less	more.
(iii) Performance	less	more

→ ⊡ 32-bit Process :→ 1 instruction cycle → 32 bit data/instruction process kar sakta hai.
→ ⊡ 64-bit Process :→ 1 instruction cycle → 64 bit data/instruction process kar sakta hai.

(iv) Compatibility	less as 32-bit CPU can run 32 bit Os only	more as 64 bit CPU can run both 32 bit and 64 bit Os.
(v) Graphic Performance	less	more

⊙ Types of Storage ⇒



⊙ Comparison ⇒

(i) Cost ⇒ Register > Cache > RAM > Secondary Memory.

(ii) Speed ⇒ Register > Cache > RAM > Secondary Memory.

(iii) Storage space ⇒ Register < Cache < RAM < Secondary Memory.

(iv) Volatility ⇒
- Primary memory - volatile
- Secondary memory - non-volatile.

↓

when computer is switched off all process gets flushed.

⊙ Why we need process?

↳ User → wants to do some work.

↓

to do this work process is only a way.

⊙ How OS creates a process?

⊙ why do we need process?

↳ User → perform some work

↓

for doing this work process is a way (zaria)

⊙ How OS creates a process?

↳ For converting Program to process

program

```
char name = "Rohit";
```

↳ when this program convents into process then, the name data will get initialised basically by static data, so thats why static data is also loaded.

(i) load the program & static data into memory

(ii) Allocate runtime stack

(iii) Allocate heap.

(iv) IO tasks.

(v) OS hands off control to main().

part of memory used for managing local variable, function argument and return value.

part of memory used for dynamic allocation

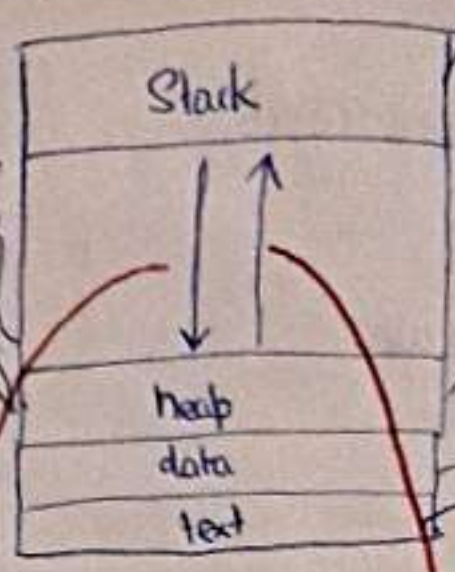
handle input, output, error tasks

main(){

}

return 0; → indicates successful execution.

⊙ Architecture of process :→



- local variable, function argument, return values.
- Dynamically allocated variables.
- Global and Static data
- compiled (Loaded disk) code.
- Such a big space as keeps between stack and heap as so as there is no creation problem of their respective programs.

⊙ Note :→

⊙ Stack overflow error occurs

when the stacks completely get filled and it reaches to heap position.

Solved by Setting the Base case

⊙ Out of memory / memory insufficient

error occur when heaps completely gets filled and reaches to stack position.

Solved by deallocating unnecessary objects

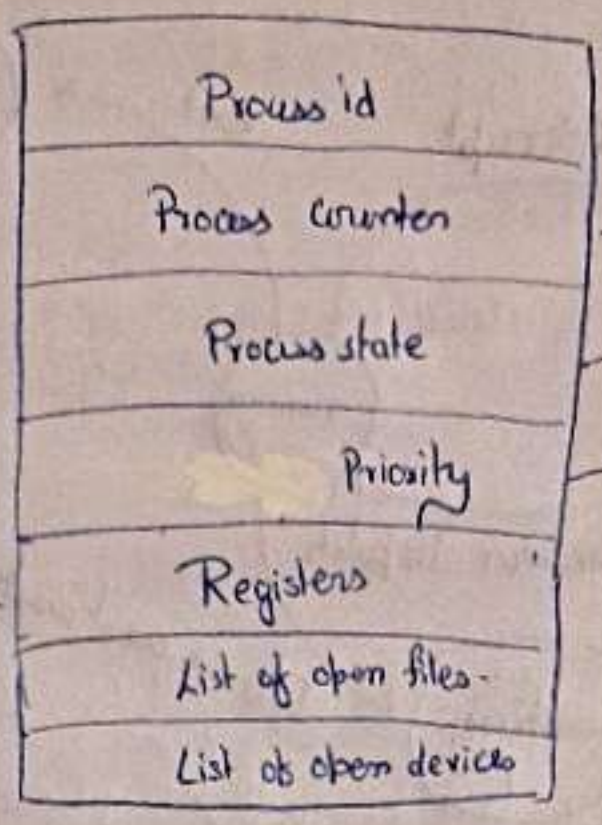
⊙ Attributes of Os :→

- Os tracks all process → table like data structure called Process table (PT).
- Each entry in PT is a PCB (Process control Block).

Feature that helps to identify process uniquely

Data structure used for storing info or all process like process id, program counter, process state, etc

⊙ PCB structure →



- Unique identifier.
- Next instruction address of program
- Stores process state
- Based on priority process gets cpu time.

ICSP 100
COR POST

- Register in PCB is a Data structure.
  - when a process runs and its time slice expires and its the time for it to go to wait state, before that the current value of process specific register from CPU gets stored in register block.

  Now, when that particular process is scheduled again to run, then we restore the current value/state what was stored before in register block from CPU, to our CPU.

⊙ Types of Process States →

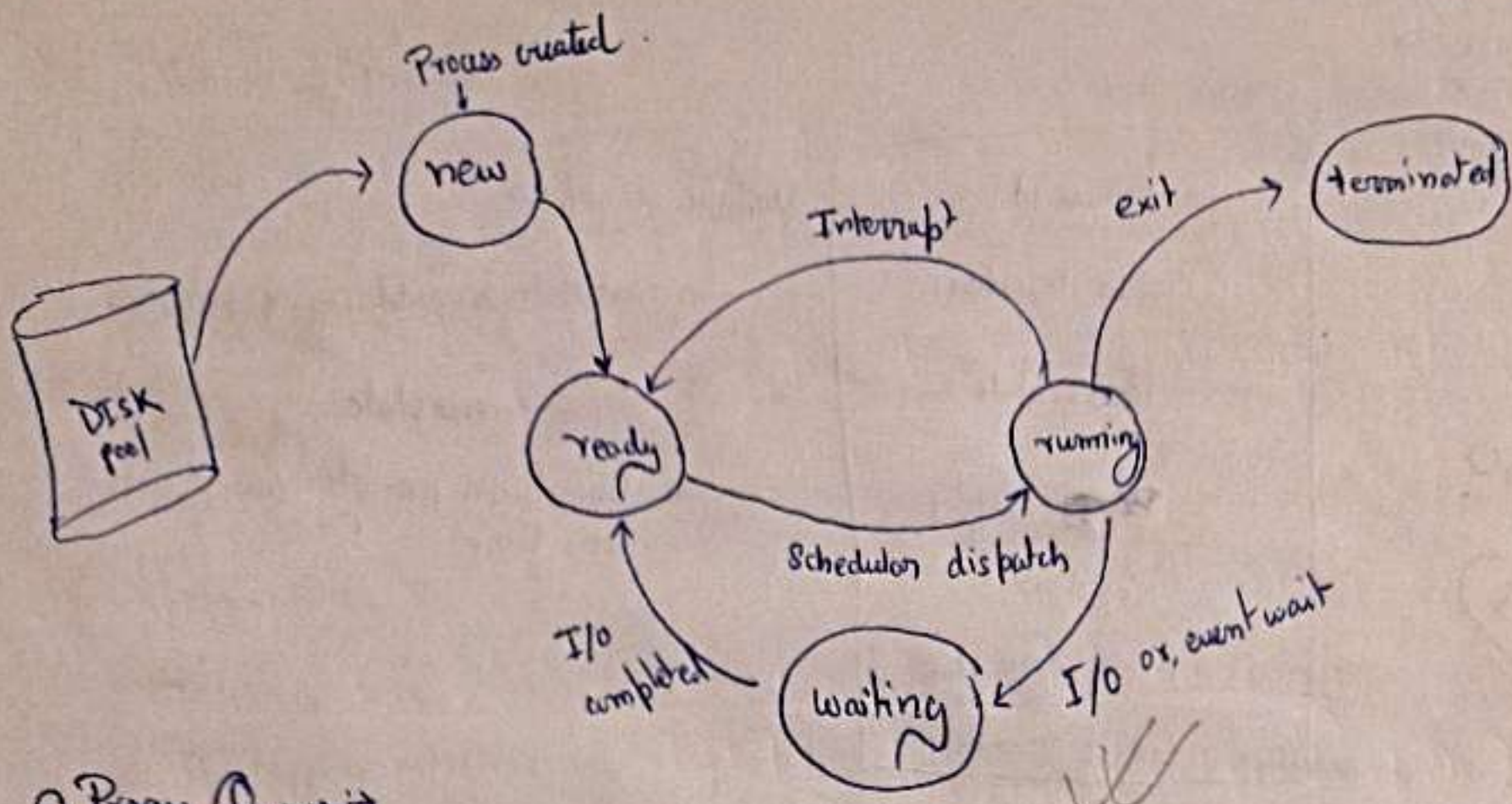
(i) New → Program → Process being done.

(ii) Ready → Process in memory (ready queue), waiting to be assigned to a processor (CPU).

(iii) Running → CPU allocated, instructions are being executed.

(iv) Waiting → Waiting for I/O completion.

(v) Terminated → Process finished.



⊙ Process Queues →

(i) Job Queue →

- Process in new state, present in secondary memory.
- Job Scheduler (LongTerm Scheduler (LTS)) picks process from pool and put them in ready state for execution.

idle time high and works at low frequency

(ii) Ready Queue →

- Process in Ready state, present in main memory.
- CPU Scheduler (Shortterm Scheduler) picks process from ready queue and dispatch it to CPU.

As it idle time is low and it works at High ~~process~~ frequency.

(iii) Waiting Queue →

- Process in wait state.

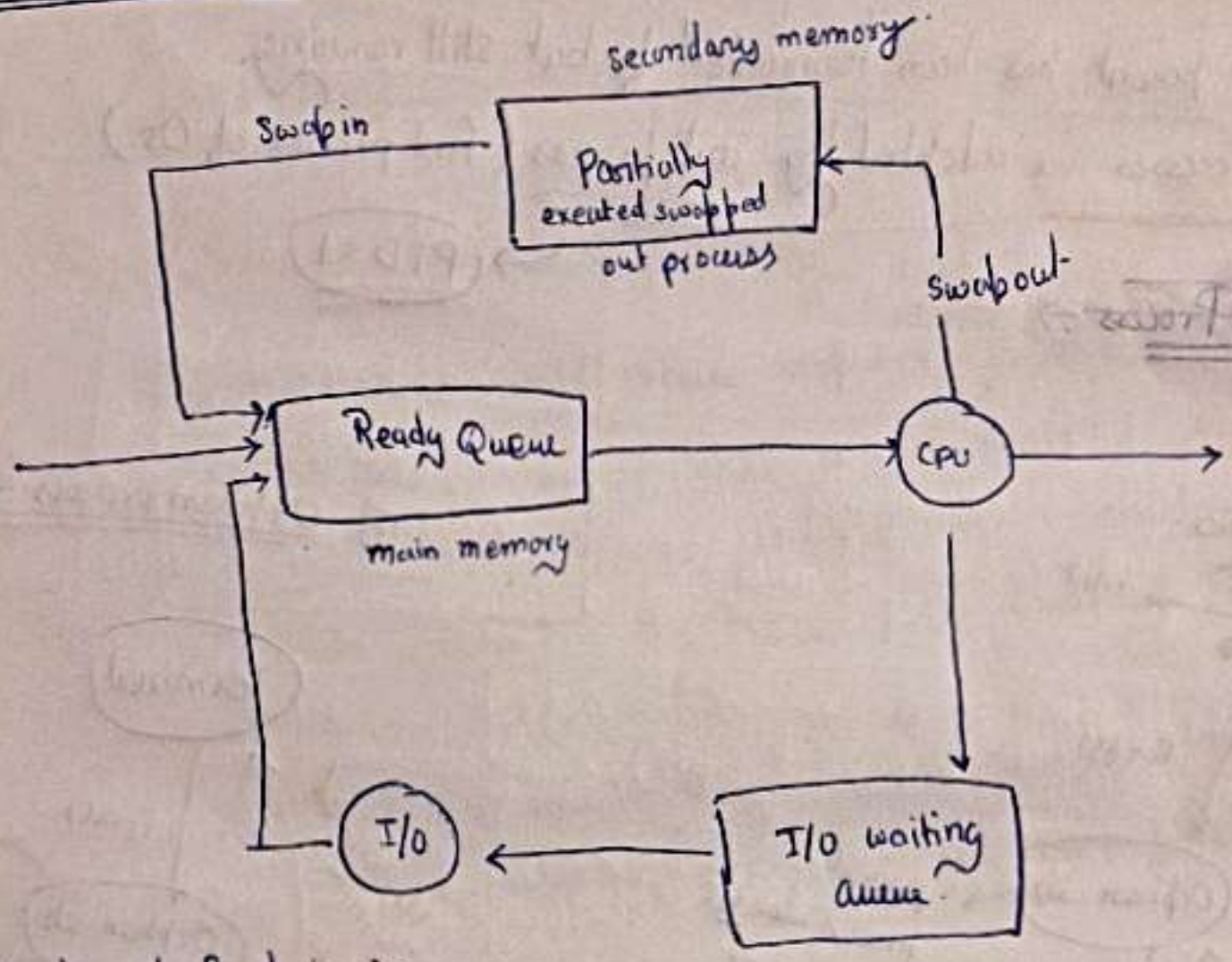
⊙ Degree of multiprogramming :→ ⊡ The number of process in memory.

⊡ Controlled by Job scheduler (LTS).

⊙ Dispatcher :→ module of OS giving control of CPU to a process selected by CPU scheduler (STS)

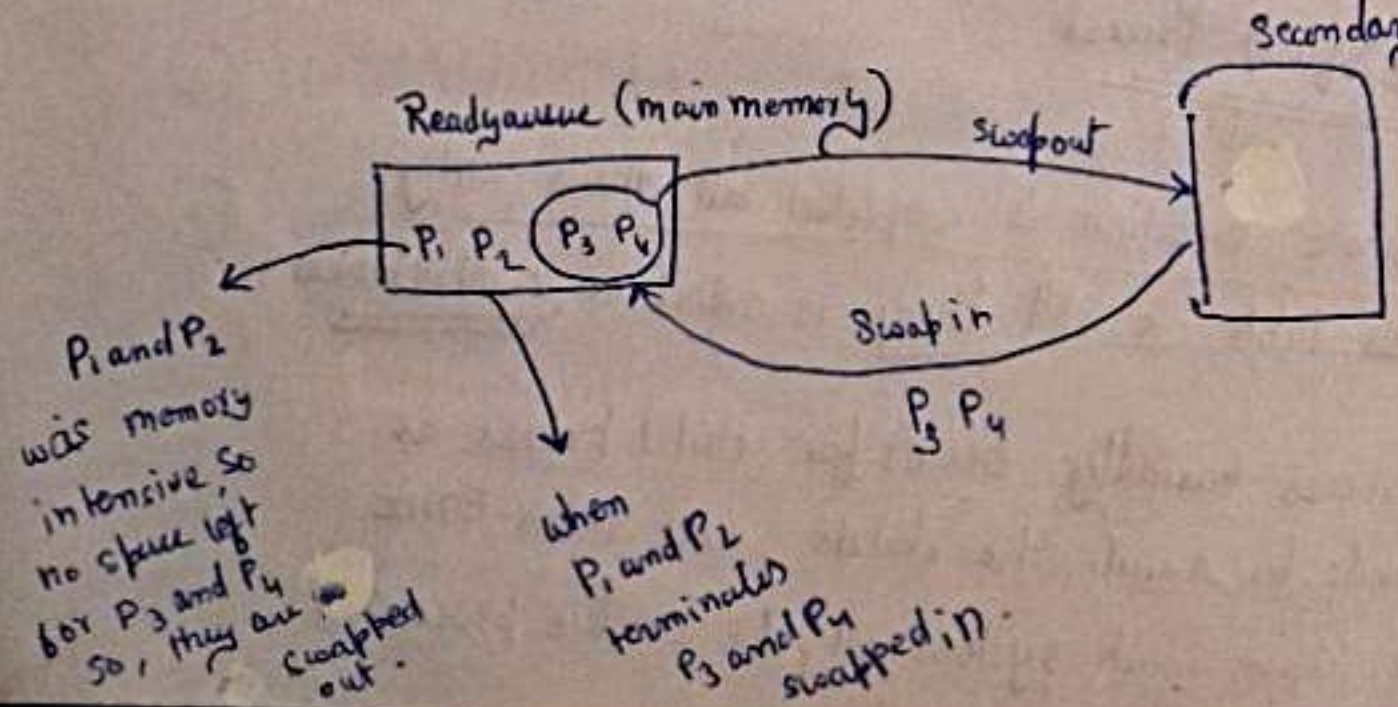
(1)

⊙ MTS (Medium term Scheduler). (Concept of Swapping).



⊡ Swap out and Swap in is done by MTS.

⊡ Swapping is a mechanism in which a process can be swapped temporarily out of main memory to secondary (Disk) ~~memory~~ storage and make the memory available to other process. At some time later, the system swaps back the process from secondary storage to main memory.



⊙ Context switching →

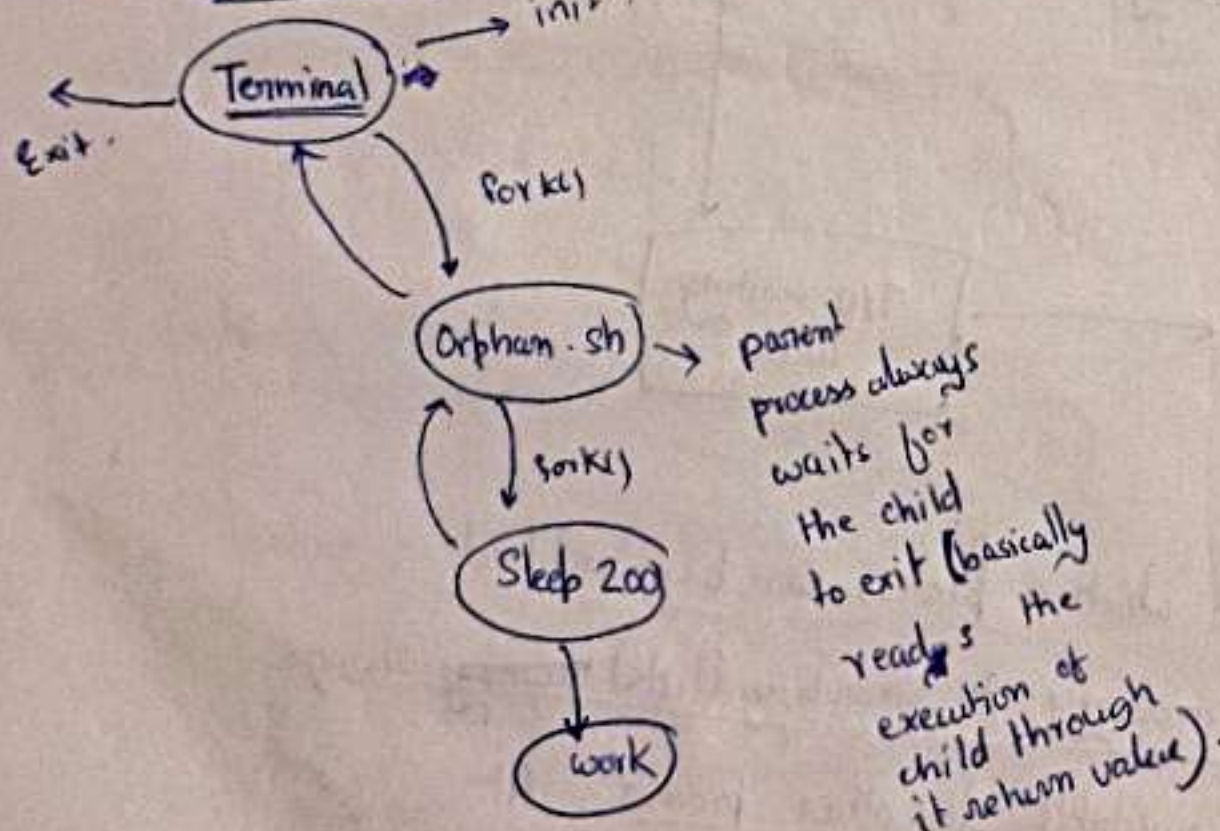
- Switching CPU to another process by saving a state of current process and restoring the state of other process.
- Kernel does this work.
- It is a pure overhead as system does no useful work while switching

⊙ Orphan process →

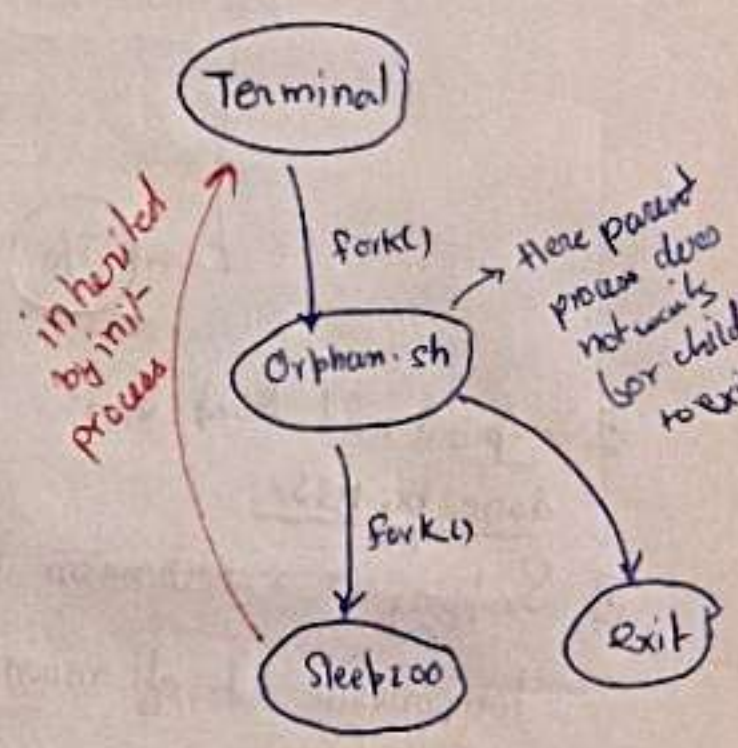
- process whose parent has been terminated but still running.
- Orphan processes are adopted by init process (first process of Os.)

↳ (PID = 1)

- Basic scenario



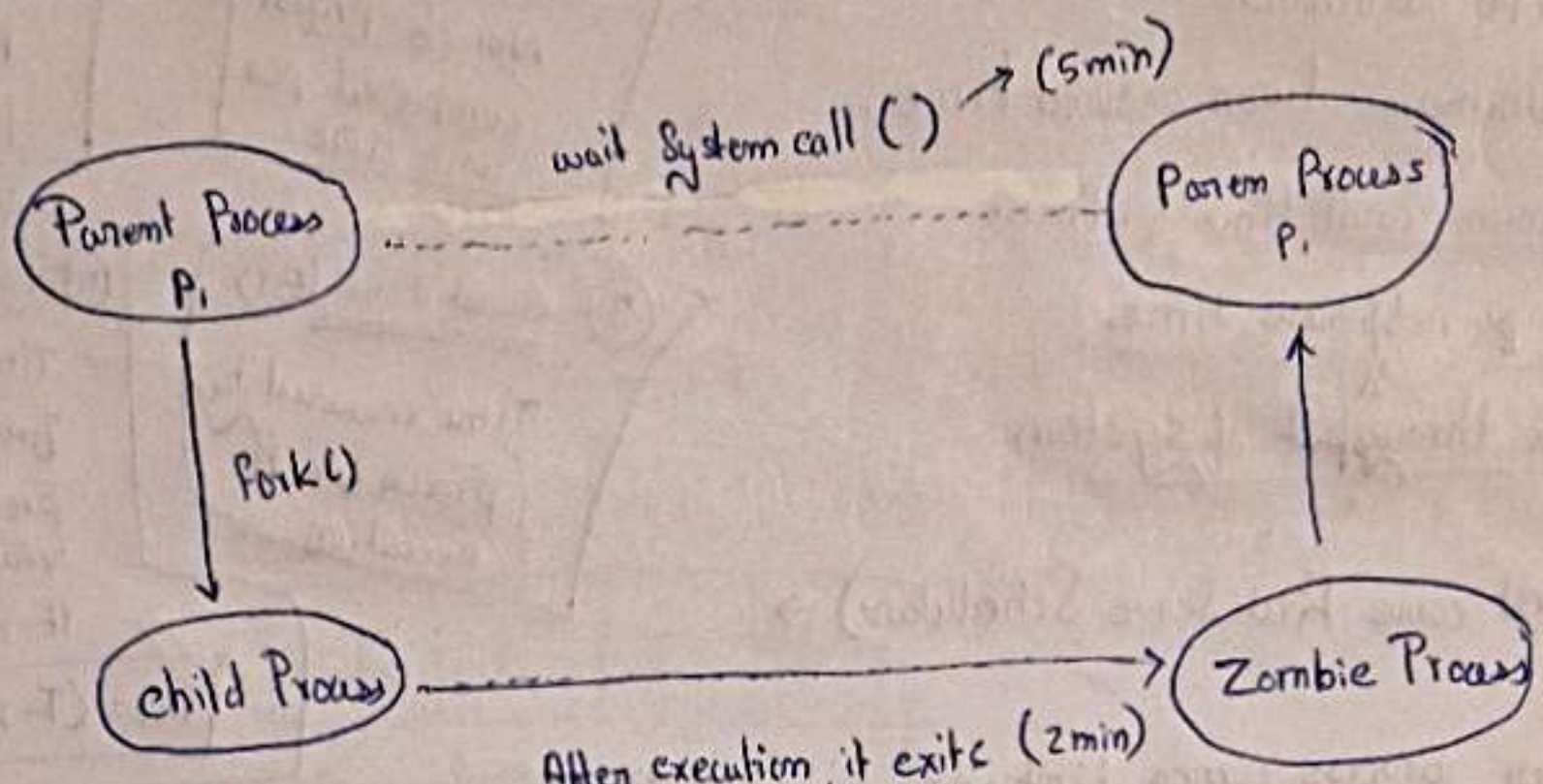
- Orphan process scenario →



⊙ Zombie process → / Defunct Process

- A process whose execution is completed but their entry in the process table is still there is called a zombie process.
- Zombie process basically occurs for child process, as parent needs to read the childs exit status. Once it is done using wait system call, the zombie process

is eliminated from the process table. This is known as reaping the zombie process.



Here the thing is child process completes its execution faster (2min). than the time, the parent takes to call wait() on the child process, as a result what happens is the child process remains as zombie process in the Process table until the wait() is call for it.

And, due to this reason there are chances that process table fills completely as it has got some fixed capacity (4.1 million)

① Non-Preamptive Scheduling

1. Process leaves CPU ----
   i. on termination
   ii. on going to wait state for I/O.
2. Process starvation more
3. CPU utilization less.
4. Overhead less

Preemtive Scheduling

1. Process leaves CPU ----
   i. on termination
   ii. on going to wait state for I/O.
   iii. when time quantum expires.
2. Process starvation less.
3. CPU utilization high
4. Overhead more.

⊙ Goals of CPU Scheduling ⇒

(i) Max CPU utilization.
(ii) Minimum turn around time.
(iii) Minimum wait time for Process.
(iv) Min ^h response time
(v) Max throughput of system.

(1) ⊙ FCFS (First come First Serve Scheduler) ⇒

(i) whichever process comes first in queue will be given CPU first.

(ii) Convoy Effect:

It is a situation where many processes, who need to use a resource for a short time is blocked by one process, (which holds resource for a long time (i.e has high BT).

Drawback

① Throughput
No. of process completed per unit time.

② Arrival time (AT)
Time when process arrived in ready queue

③ Burst Time (BT)
Time required by process for its execution

④ Turnaround time (TAT)
Time taken by from first time process enters ready state till it terminates
TAT = CT - AT

⑤ Wait time (WT)
Time process spends waiting for CPU
WT = TAT - BT

⑥ Completion Time (CT)
Time taken till process gets terminated.

⑦ Response time
Time duration between process getting into ready queue and process getting CPU for first time
↓
Time first getting CPU − AT

(2) SJF (Shortest Job First) ⇒

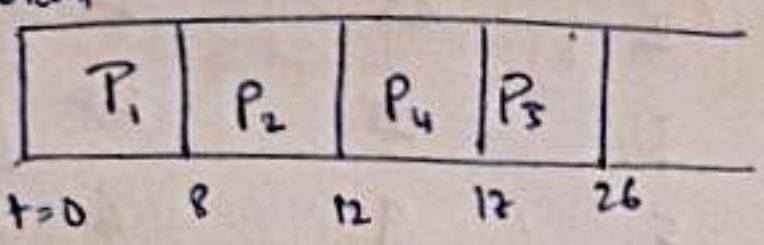
⊡ Process with less BT will get CPU.
⊡ Might suffer with convoy effect if first process has a large BT.
⊡ Process of starvation might happen.
⊡ Criteria of for SJF Algo, AT+BT.

Non-preemptive

Process	AT	BT	CT	TAT	WT
P_1	0	8	8	8	0
P_2	1	4	12	11	7
P_3	2	9	26	24	15
P_4	3	5	17	14	9

Avg Time: 7.75s

Grant chart



(ii) Preemptive → (SJF) / SRTF)

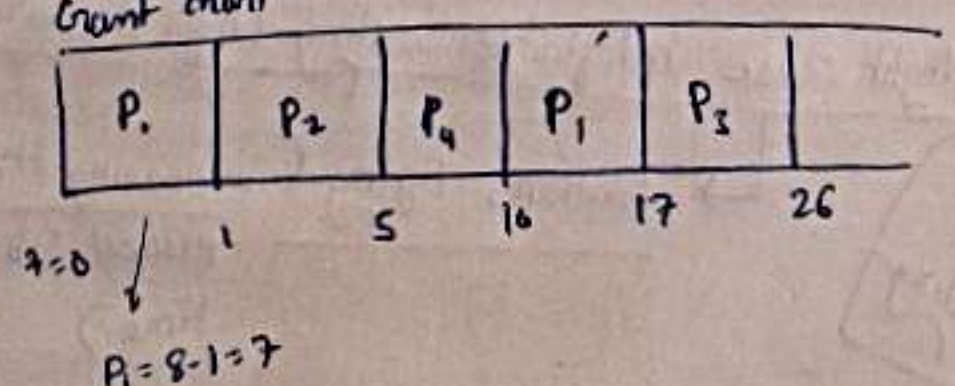
→ Shortest Remaining Time First.

- Less starvation
- No convoy effect
- Gives less WT as compared to SJF (preemptive)

Process	AT	BT	CT	TAT	WT
P_1	0	8	17	17	9
P_2	1	4	5	4	0
P_3	2	9	26	24	15
P_4	3	5	10	7	2

Avg. Time: 6.5s.

Grant chart.



Note →

In SJF, it is nearly impossible to implement that because we cannot know the BT before hand.

Drawback

⊙ Priority Scheduling

↳ Here priority is assigned to each process when created.

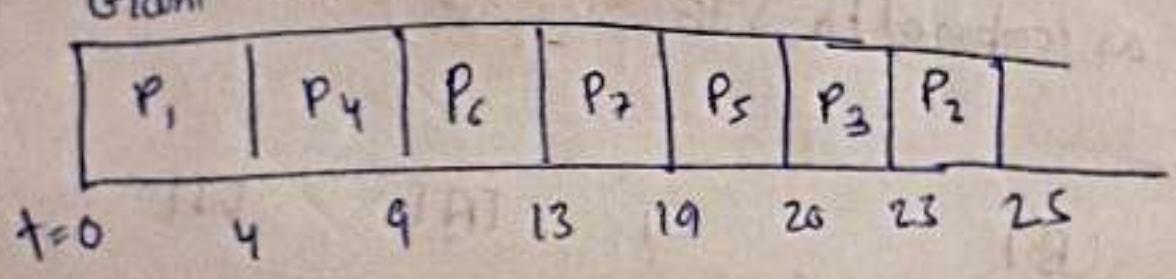
☐ Non-Preemptive →

↳ SJF is a special case of general priority scheduling with priority

Process	Priority	AT	BT	CT	TAT	WT	$\propto \frac{1}{BT}$
P_1	2	0	4	4	4	0	
P_2	4	1	2	25	24	22	
P_3	6	2	3	23	22	19	
P_4	10	3	5	9	6	1	
P_5	8	4	1	20	16	15	
P_6	12	5	4	13	8	4	
P_7	9	6	6	19	13	7	

Avg. 9.71 s.

Grant chart.



☐ Preemptive Priority Scheduling →

i. Current run state of a Job/Process gets preempted if next Job has higher priority.

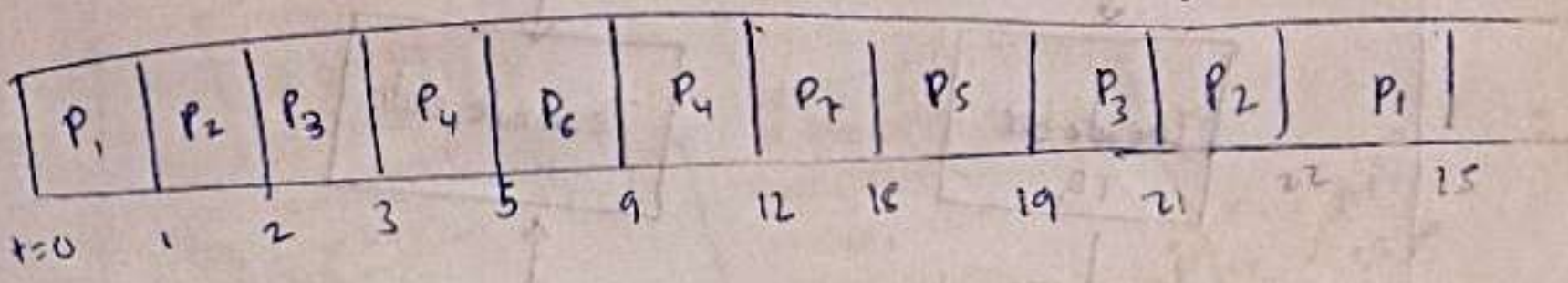
ii. Might cause indefinite waiting (high starvation)

↳ Solution: Aging (increasing the priority of others Process/ Job after a period of time)

Biggest Drawback of Priority Scheduling

Process	~~Priority~~	AT	BT	CT	TAT	WT
P_1	2	0	~~4~~ 3	25	25	21
P_2	4	1	~~2~~ 1	22	21	19
P_3	6	2	~~3~~ 2	21	19	16
P_4	10	3	~~5~~ 3	12	9	4
P_5	8	4	1	19	15	14
P_6	12	5	4	9	4	0
P_7	9	6	6	18	12	6

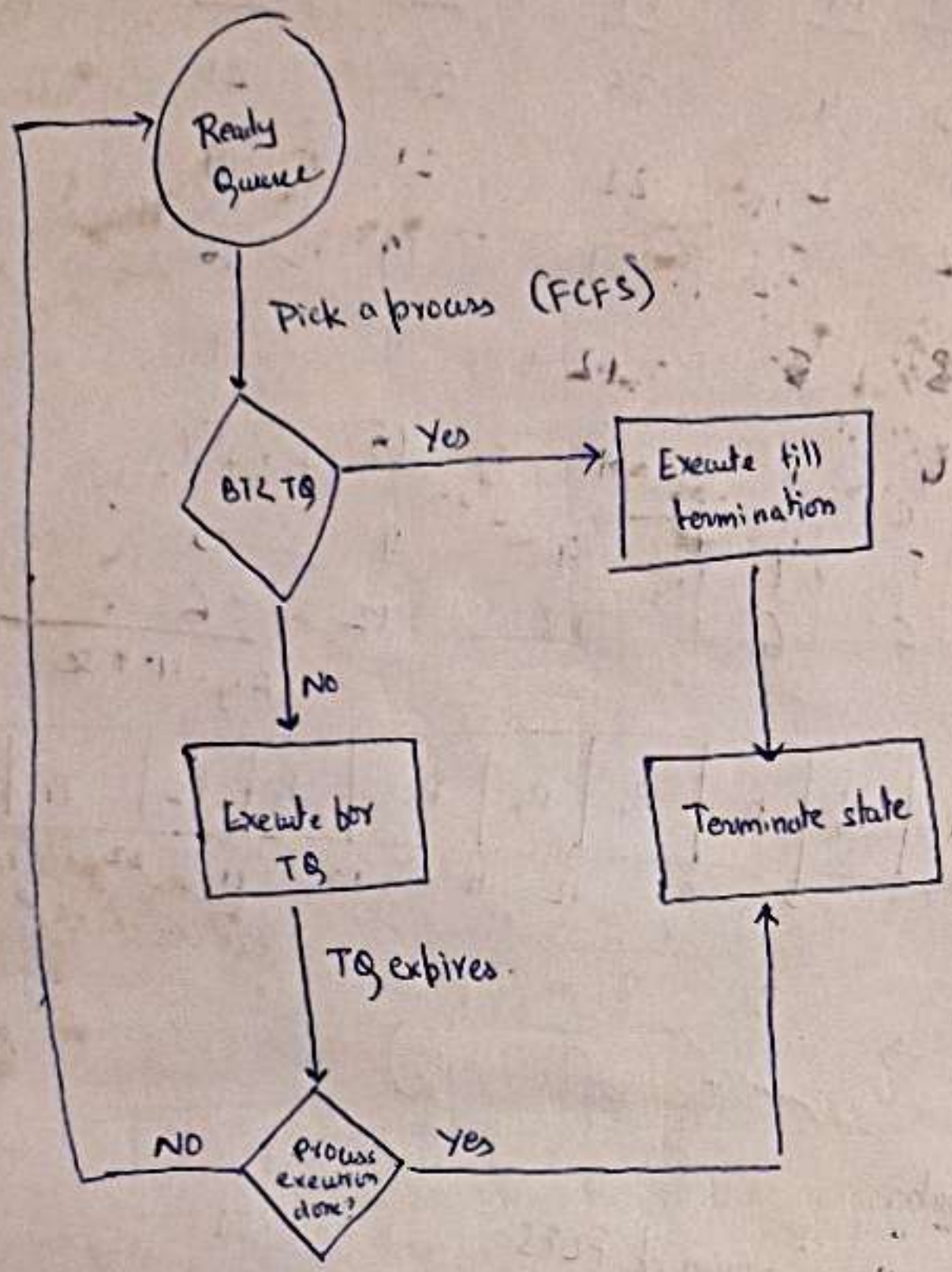
Avg - 11.4 sec



⊙ Round Robin Scheduling →

- (i) Most Popular.
- (ii) Preemptive version of FCFS
- (iii) Criteria :→ AT + Time Quantum, no BT like SJF.
- (iv) design for Time sharing System. (~~Multi~~ tasking OS). mai used nota hai.
- (v) Less $ starvation, no convoy effect.
- (vi) Easy to implement.
- (vii) More context switching/ overhead, overhead $\propto \frac{1}{\text{Time Quantum}}$.

Drawback.



Process	AT	BT	CT	TAT	WT.
P_1	0	~~4~~ ~~2~~ 0	8	8	4
P_2	1	~~5~~ ~~3~~ ~~1~~ 0	18	17	12
P_3	2	~~2~~ 0	6	4	2
P_4	3	~~1~~ 0	9	6	5
P_5	4	~~6~~ ~~4~~ ~~2~~ 0	21	17	11
P_6	6	~~3~~ ~~1~~ 0	19	13	10

Avg Time = 7.33 s

Queue = | ~~P_1~~ | ~~P_2~~ | ~~P_3~~ | ~~P_1~~ | ~~P_4~~ | ~~P_5~~ | ~~P_6~~ | ~~P_5~~ | ~~P_2~~ | ~~P_6~~ | ~~P_5~~ |

Gant chart

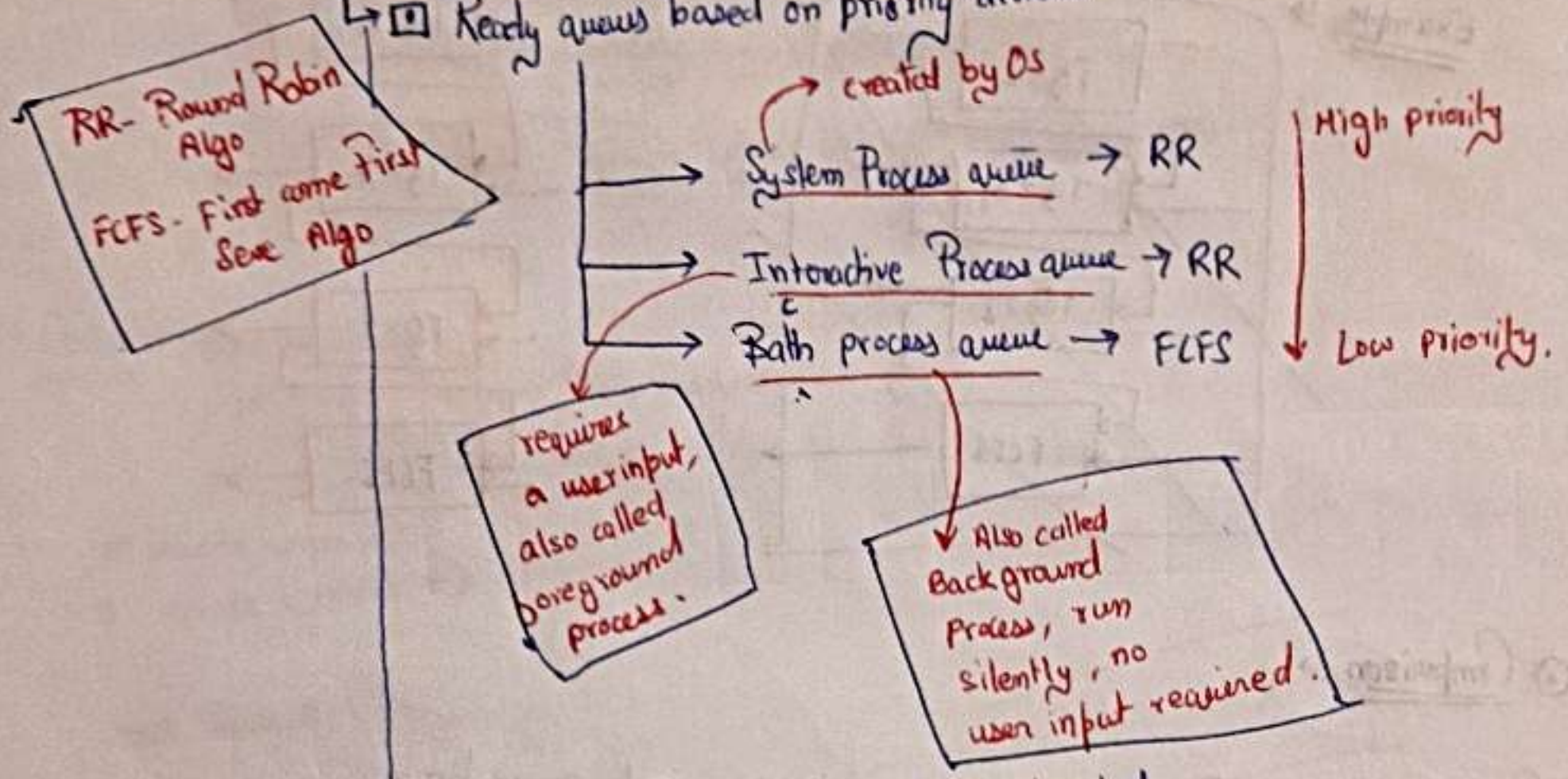
TQ = 2

P_1	P_2	P_3	P_1	P_4	P_5	P_2	P_6	P_5	P_2	P_6	P_5

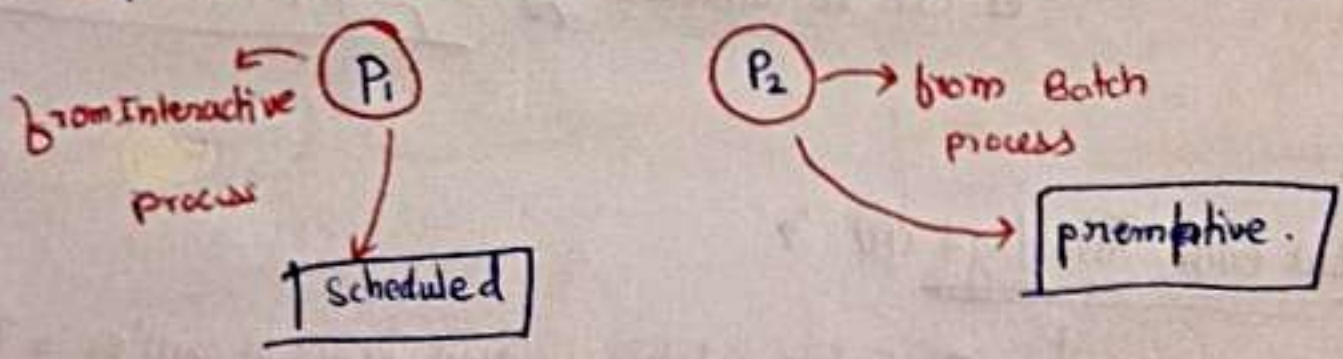
T=0 2 4 6 8 9 11 13 15 17 19 19 21

Multilevel Queue Scheduling

- Ready queues based on priority division ...



- Among the Queues, the queue scheduled using fixed priority premptive scheduling (SP > IP > BP)



- Drawbacks ⇒ (i) Starvation can occurs as only after completion of process of top-level queues, the process in low-level queues gets chance.

(ii) Convoy Effect present.

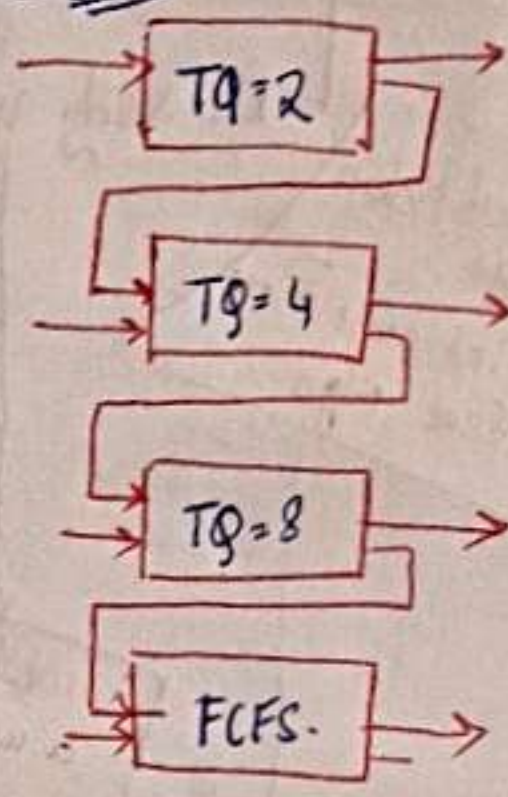
Multi-level Feedback Queue Scheduling ⇒

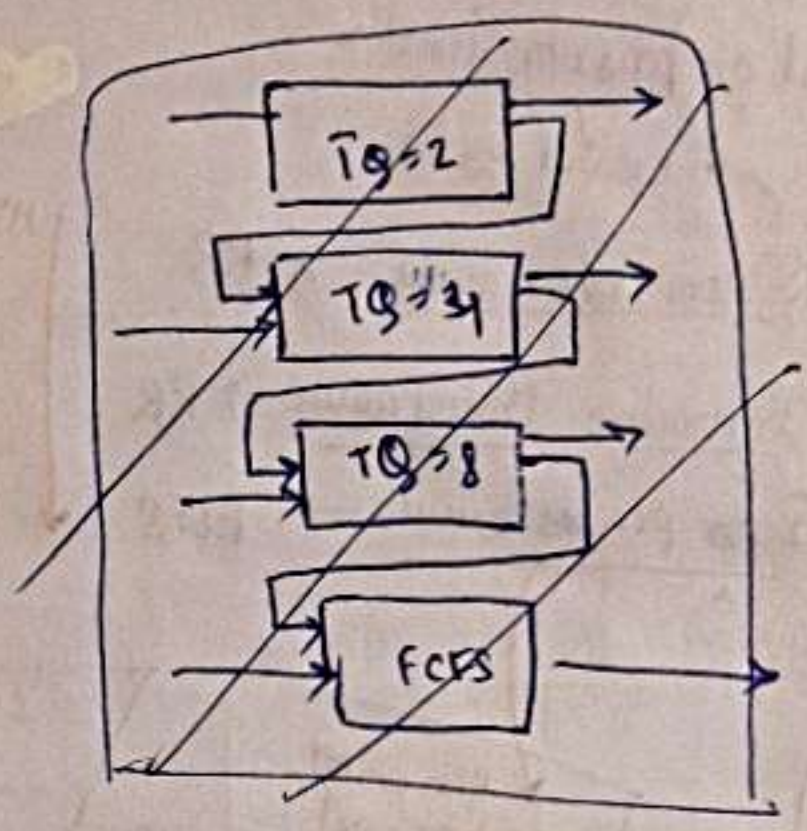
Nowadays used.

- multiple sub queues present, having interqueue movement allowed.
- Separate process based on BT (Burst Time)
  - (i) BT High - low queues
  - (ii) I/O bound and interactive process - high priority / top queues.

- ⊡ Aging method used to lower starvation, as due to inter queue movement.
- ⊡ flexible and configurable.

⊙ Example →





⊙ Concurrency ⇒ ⊡ Executing multiple instructions at same time.

⊡ Can be achieved by dividing process into threads.

⊙ How each Thread get CPU ⇒

TSA: Thread Scheduling Algo

TCB: Thread control Block

↓ provides ability to context switch between threads.

- ⊡ Each thread has its own program counter in TCB.
- ⊡ Depending on TSA, OS schedule these threads.
- ⊡ Os fetches ~~info~~ instruction corresponding to PC of that thread and execute instruction.

⊙ Will single CPU would gain by multi-threading techniques?

ans. No, as 2 threads have to context switch for that single CPU, which won't give any gain

⊙ Benefits of Multi threading ⇒

- (i) Responsive ↑es.
- (ii) Resource sharing efficient due to same memory address.
- (iii) Economy - Fast context switching -
- (iv) Threads better utilizes Multi core CPU.

⊙ Code ⇒ How to divide a Process into multiple threads and let them perform their independent task.

```
#include <thread>
#include <unistd.h>

void taskA(){
    for(int i=0; i<10; i++){
        sleep(1);
        printf("Task A: %d\n", i);
        fflush(stdout);
    }
}

void taskB(){
    for(int i=0; i<10; i++){
        Sleep(1);
        printf("TaskB: %d\n", i);
        fflush(stdout);
    }
}

int main(){
    thread t1(taskA);
    thread t2(taskB);
    t1.join();
    t2.join();
}
```

Keep thread waiting for 1 sec

Clears the output buffer and move the buffered data to console.

Creating thread by assigning task.

main thread waits until t1 and t2 exists. if this joincy function not written then main threads exist at very first and program throws an error.

⊙ Process Synchronization techniques help to maintain consistency of shared data.

⊙ Critical Section :→

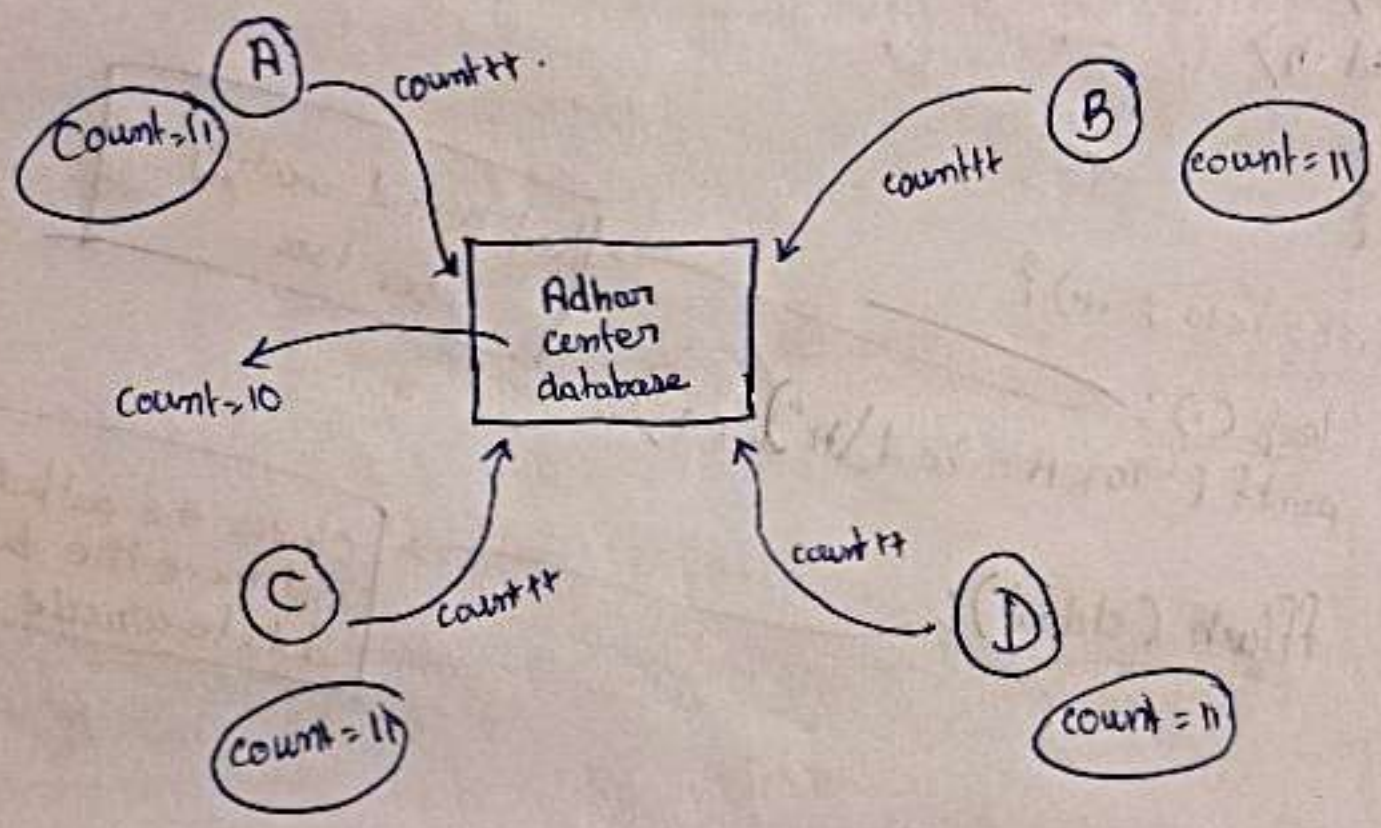
Segment of code where process/thread access shared recources, such as common variables and files, and write operations on them. Since they execute concurrently, any process can be interrupted in mid-section.

⊙ Race condition :→

↓

Thread Scheduling issue.

⊡ Occurs when 2 or more threads can access shared data and they try to change it at same time.



Here the value of count should have been (0+1+1+1)+1 = (14),

But due to data inconsistency count = (11), which is in appropriate leading to Race condition.

⊙ Solution to Race condition :→

Execute in 1 CPU cycle

(i) Atomic operations: Critical section code → make atomic operation

(ii) Mutual Exclusion using locks. (One thread will go and lock critical section, until its execution is over, then only other goes, chance) threads.

(iii) Semaphores.

⊙ Can we use single flag to avoid race condition, why?

ans: ⊡ Solution to critical section must have 3 important conditions .....

(i) Mutual exclusion
(ii) Progress → threads should not have order any thread can go and start its execution asap.
(iii) Bounded Waiting → should not be indefinite waiting

⊡ No, as in flag, Progress law is violated, i.e, the threads are having a order of execution.

⊡ Peterson's solution extended form of flag holds goods for race avoidance, but it holds good for case of 2 Process/threads.

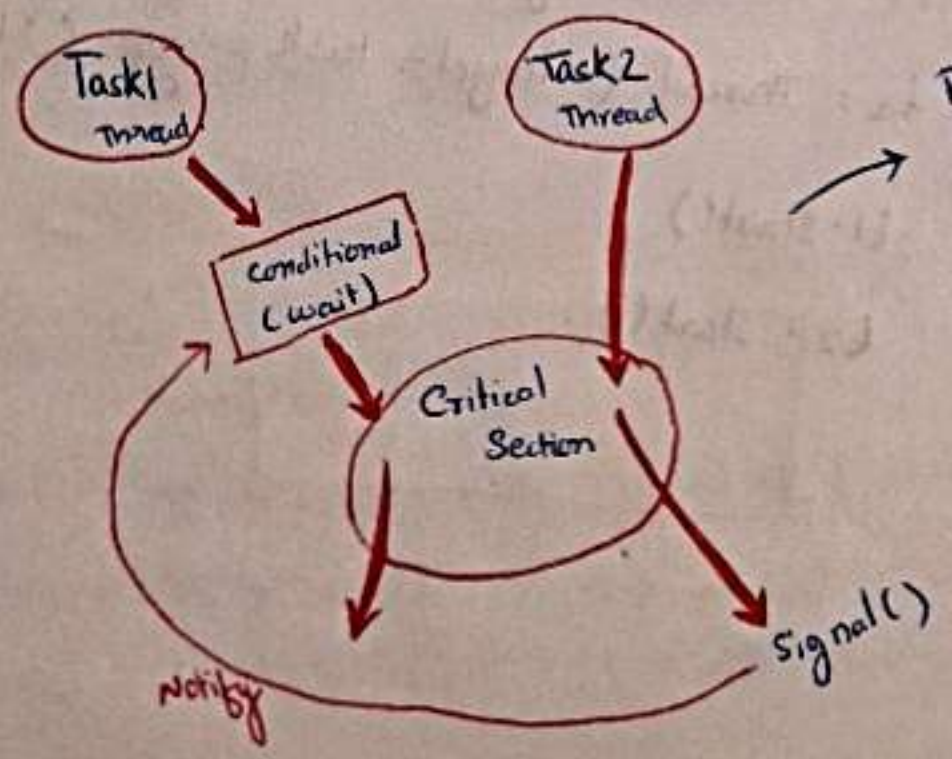
⊙ Mutual Exclusion using Locks / Mutex →

↳ disadvantage

- Contention → One thread acquire lock, other threads are bussy wait, for some exception that thread in lock dies, the other threads goes into indefinite waiting.
- Deadlock
- Debugging
- Starvation of high priority threads.

CDDS

⊙ Conditional Variables →



Basically what happens is here we have got two threads T_1 and T_2.

T_1 will wait on a conditional wait first. Now, T_2 Thread will go into Critical section and execute. After T_2 thread completes it execution it notifies thread t_1 by a signal. Then, thread t_1 can start its execution on critical section.

(ii) works with a lock.

(iii) conditional waiting is used to avoid busy waiting.

(iv) Contention is not here.

Code:

```
cond = condition ()
done = 1
def task (name):
    global done
    with cond:
        if done == 1:
            done = 2:
            print ("waiting on condition variable cond:", name)
            cond. wait()
            print ("Condition met:", name)
    else:
        for i in range(5):
            print ('.')
            time. sleep(1)
        print .("Signaling condition variable cond", name)
        cond. notify_all (1)
        print ("Notification done", name)

if __name__ == '__main__':
    t1 = Thread (target = task, args ('t1',))
    t2 = Thread (target = task, args ('t2',))
    t1. start()
    t2. start().
```

Output: →

Waiting on conditional variable cond: t1

.
.
.
.
.

Signaling conditional variable cond: t2

Notification done t2

condition met ; t1

⊙ Semaphores →

- ⊡ Integer Variables = Number of Resources (Synchronizat Method.)
- ⊡ Mutiple threads ~~and~~ can execute Critical Section consequetively, due to it. (C.S.)
- ⊡ Types
  - ⊡ Binary Semaphores → Only 1 thread can access C.S. (Internal implementation of mutex locks.)
  - ⊡ Counting Semaphores → Multiple but fixed number of threads can access C.S.

⊡ Internal implementation of wait and Signal →

⊡ wait

```
wait (S){
    S→value --;
    if (S→value <0){
        add P→ blocklist
        block();
    }
}
```

⊡ Signal.

```
Signal (S){
    S→value ++;
    if (S →value =0){
        remove P from blocklist.
        wake(P);
    }
}
```

⊡ Example →

Semaphore S(2);

T_1 → wait() → S→val = 1 → C.S → Signal(S→val++)

T_2 → wait() → S→val = 0 → C.S

T_3 → wait() → S→val = -1 → Block state → After signal T_1 → wake up T_3 and will get C.S.

⊡ code →

```
from threading import *
import time
sem = Semaphore(2)

def task(name):
    sem.acquire();
    for i in range(5):
        print("{} working".format(name))
        time.sleep(1)
    sem.release()

if __name__ == '__main__':
    t1 = Thread(target=task, args=('Thread-1',))
    t2 = Thread(target=task, args=('Thread-2',))
    t3 = Thread(target=task, args=('Thread-3',))

    t1.start()
    t2.start()
    t3.start()

    t1.join()
    t2.join()
    t3.join()
```

→ As Buffer has a finite size.

⊙ Producer / Consumer Problem ⇒ (Also called Bounded Buffer Problem)

→ Two Threads → (i) Producer Thread
→ (ii) Consumer Thread.

Problems that needs to be resolved here ....

(i) Critical Section (Buffer) ☆ act as shared memory between Producer and Consumer, which may lead to data inconsistency ⇒ Syncronization between Producer and Consumer thread.

(ii) Producer thread must not insert data when buffer is full.

(iii) Consumer thread must not pick/remove data when buffer is empty.

▫ Some important variables →.

(i) m, mutex → Binary semaphore used to acquire lock on buffer.

(ii) empty → ▫ counting semaphore tracks empty slots.
▫ initial value = n

(iii) fill → ▫ tracks filled slots.
▫ initial value : 0.

Producer

```
do {
    wait(empty); // wait until empty > 0,
                 then empty → value --
    wait (mutex);
    // c.s, add data to buffer
    signal (mutex);
    signal (full); // increment
                   full → value ++
} while (1).
```

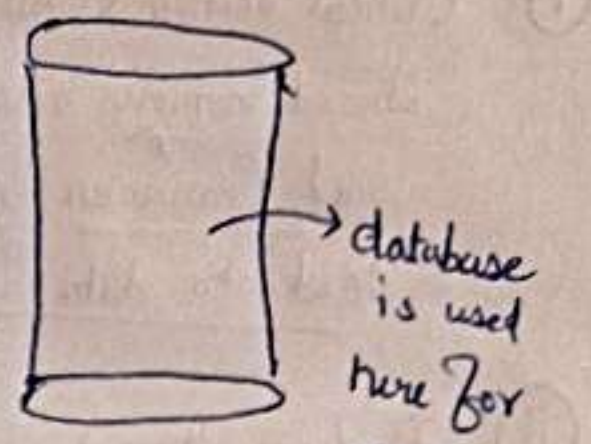
Consumer

```
do {
    wait (full); // wait until full > 0, then
    wait (mutex); full → value --
    // c.s, remove data from buffer.
    signal (mutex);
    signal (empty); // increment
} while (1)
```

empty → value ++

⊙ Reader - writer Problem ⇒

- (1.) Reader thread → Read.
- (2.) Writer thread → write, update.



⊙ Problems

- (i) If more than 1 Readers are reading ⇒ No issue.
- (ii) If more than 1 writers OR 1 writer and some other thread (R/W) parallel ⇒ Race Condition and data inconsistency.

⊙ Solution ⇒ Semaphores ⇒

- Important variables
  - mutex ⇒ Binary Semaphore
    - To ensure mutual exclusion when (RC) Read count is updated.
  - wrt ⇒ Binary Semaphore
    - Common for reader and writer.
  - Read count (RC) ⇒ Integer value initialised with 0.
    - Keep track of number of readers reading the critical section / databases.

⊙ Writer Code :→

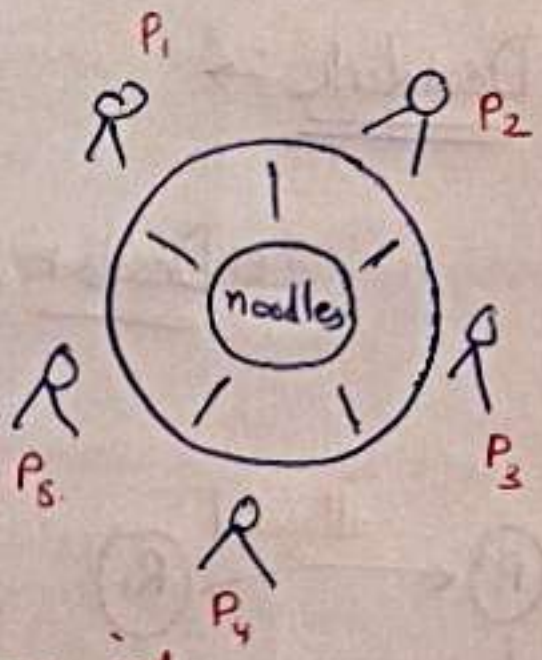
```
do {
    wait (wrt);
    // do write operation
    signal (wrt);
} while (true);
```

⊙ Reader Code Solution ⇒

```
do {
    wait (mutex); // to mutex read count variable.
    Rc++;
    if (Rc==1)
        wait (wrt); // Ensure no writer enter even if we have one reader.
    Signal (mutex);
    // C.S.; Reader is reading
    wait (mutex);
    Rc--; // a reader leaves
    if (Rc==0)
        signal (wrt); } // if no Reader left writer can enter.
    signal (mutex); // reader leaves.
} while (1)
```

⊙ Dinning Philosopher ⇒

- ⊡ 5 philosopher, 5 fork and noodles
- ⊡ No, 2 philosopher can use same fork at same time.
- ⊡ Solution Semaphore ⇒ Semaphore
  - Each fork: Semaphore ⇒ fork [5] {1}
  - wait() → fork[i] → philosopher[i] acquired.
  - Signal() → fork[i] → fork ⇒ Release.



⊙ Code ⇒

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

⊡ In this solution we make sure that no 2 adjacent neighbours are eating simultaneously. But, it can lead to a Deadlock. (Suppose all philosopher tries to pick right fork.)

⊙ Solution to Deadlock issue :→

→ ⊡ Allow at most 4 philosophers to sit simultaneously

→ ⊡ Allow a philosopher to pick two forks at same time. Now, here we will be putting our 2 forks together in a critical section by calling waiting and then we will also be releasing them together.

→ ⊡ Odd - Even : (i) Odd Philosophers picks → Left fork.
(ii) Even Philosophers picks → Right fork

Note →
only using Semaphores is not sufficient, here, must add some enhancement for deadlock.

⊙ Deadlock :→

⊡ Deadlock → (i) Deadlock is a situation in which a Process/Thread requests for a resource and if the resources aren't available they enter into waiting state. Sometimes, this waiting is never able to change its state as the resources they requested remain busy forever.

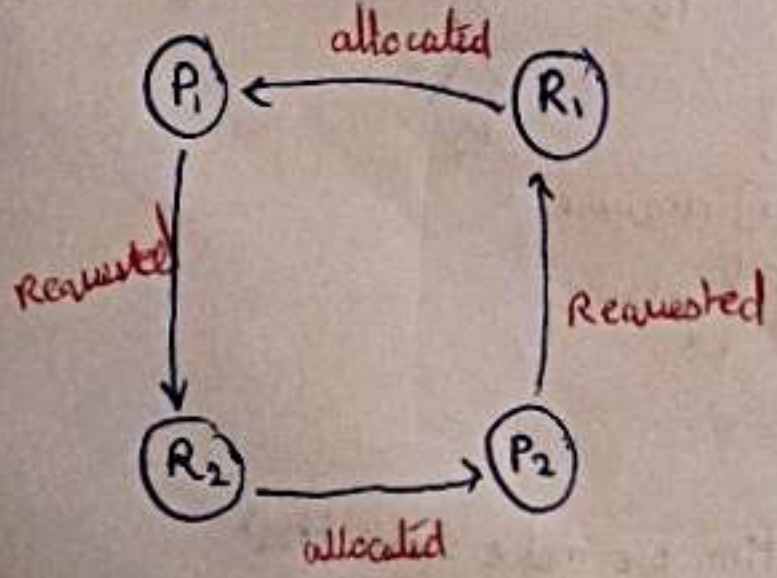


Fig 1.1

(ii) Deadlock → Bug of Process/Thread Synchronization.

(iii) Resources (Ex) → Memory Spaces, CPU cycles, locks, etc.

⊙ How a process / thread utilize a Resource?

(i) Request :→ If resource available then lock, else wait.

(ii) Use

(iii) Release → Release resource and make it available for some other process.

⊙ Necessary condition for Deadlock

(i) Mutual Exclusion ⇒ Only 1 Process can use a Resource at a time, if another process comes and requests for the same resource, then it has to wait until resource is released.

(ii) Hold and wait ⇒ A Process must atleast hold 1 Resource and simultaneously wait for other Resource to be released by other process.

(iii) No-preemption ⇒ Resources must be voluntarily released by the Process after complete of execution. (No Resource preemption)

(iv) Circular wait ⇒ Waiting of Resources by Process must be in a circular fassion.

Example → Fig 1.1

One Resource cannot be forcefully taken by one Process from another Process before its complete execution

⊙ Resource Allocation Graph :→

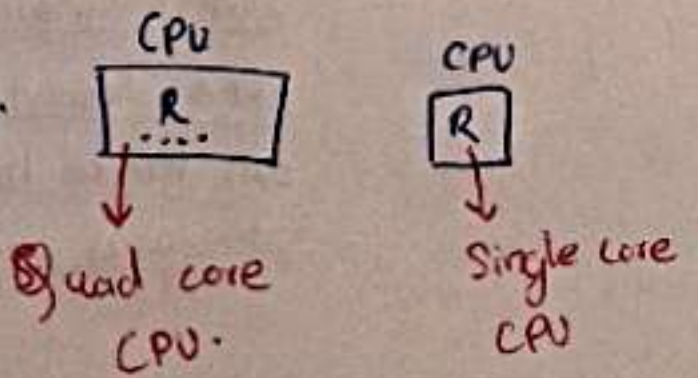
⊡ Vertex ⇒

- Process vertex
- Resource vertex.

⊡ Edges ⇒

- Assignment
- Request.

⊡ To show multiple instances ⇒ Eg.



⊙ By definition of RAG ⇒

(i) No cycle (RAG) ⇒ No Deadlock.

(ii) Cycle (RAG) ⇒ may be Deadlock

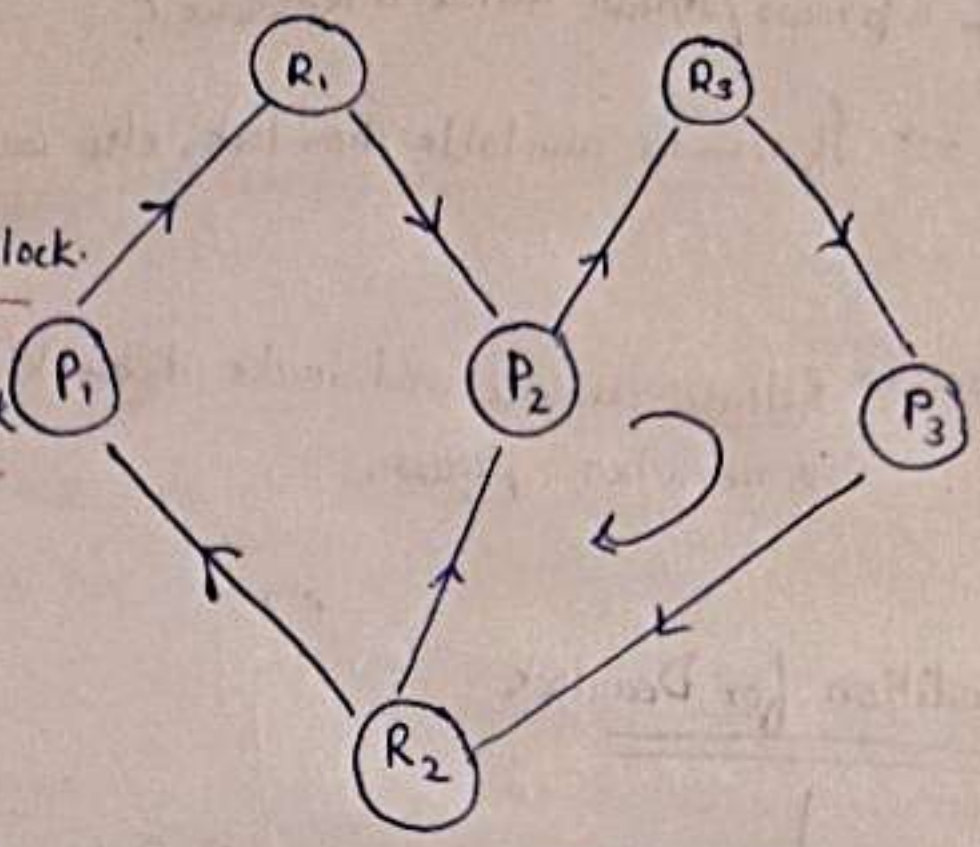


Fig 1.2 Has a cycle Has a deadlock. but not necessary : if cycle present, then deadlock will be there.

⊙ Methods to handle Deadlock ⇒

(i) Prevent or avoid Deadlock

(ii) Allow system to go in Deadlock.
→ Detect → Recover it.

(iii) Ostrich Algorithm ⇒ (Deadlock Ignorance).

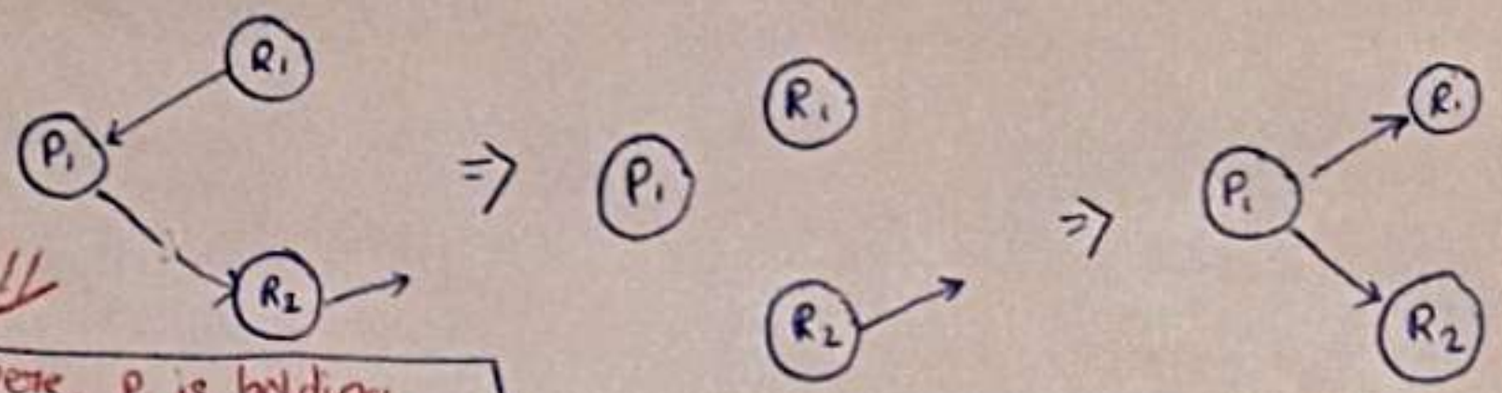
⊙ Deadlock Prevention Techniques ⇒ (Prevent atleast 1 necessary condition of Deadlock)

(i) Mutual Exclusion ⇒
- Avoid making all resources as non-Sharable using locks.
- Example → Read Only file → Sharable Resource.
- Can't prevent Deadlocks by denying mutual-exclusion conditions because some resources are intrinsically non-Sharable.

(ii) Hold and wait ⇒
- Either at beginning hold all resources and start execution or else while requesting for a resource we must ensure that our Process has released all the earlier resources which it was holding.

→ (iii) No-preemption ⇒

(i)



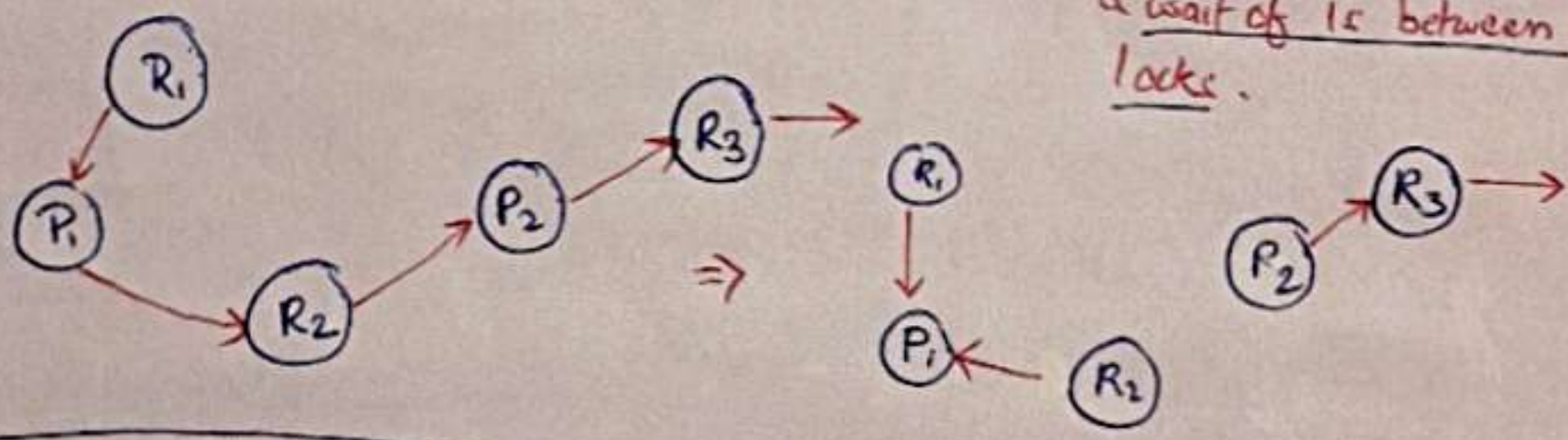
▣ Sohere, P_1 is holding R_1 and requesting for R_2. So, what we will do here is release R_1, and when R_2 is available we gonna request for R_1 and R_2 together.

P_1, P_2 : Process
R_1, R_2 : Resources

▣ Request 2 resources at same time might lead to a problem called Live Lock, in which we suffer with collision of locks if multiple done at same time. due to which after collision also locks come back to same state do collision again.

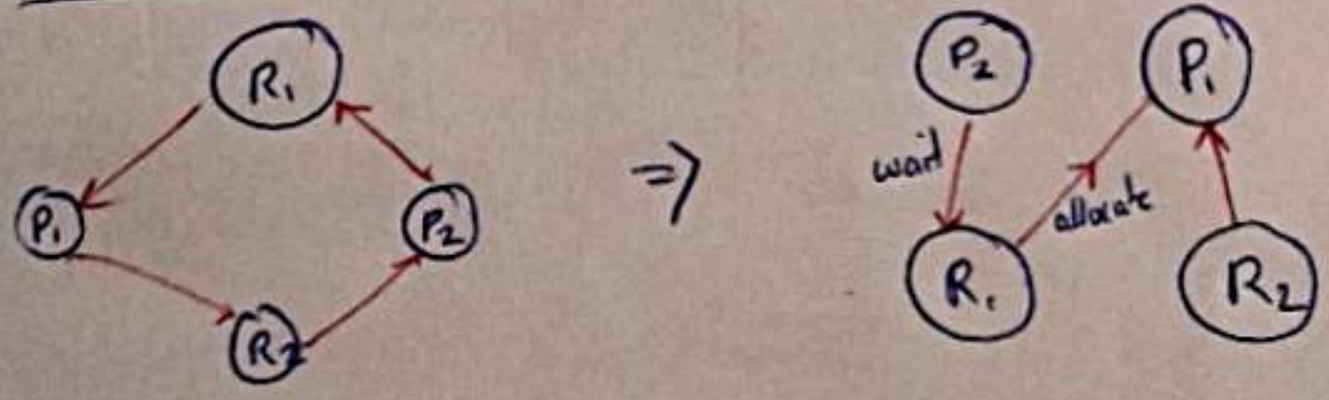
▣ can be avoided by keeping a wait of 1s between the locks.

(ii)



▣ Here we first prempt R_2 and provided it to Process P_1, then Process P_1 got released R_2, Now, P_2 acquire R_2 and will complete its execution.

(iv) Circular Wait →



For P_2's execution ⇐

▣ Here what we have done is let P_1 and P_2 both request for R_1 first, Since P_1 did first request So it acquired R_1 and P_2 went in wait. Now P_1 will also acquire R_2 and execute. After, that will release R_1 and R_2

⊙ Deadlock Avoidance ⇒

↳ when it happens we Know the current state of system. →

(i) Number of Process.
(ii) Max need of Resources of each Process.
(iii) Currently allocated amount of Resource to each Process
(iv) Max amount of Each Resources.

Target of Deadlock Avoidance ⇐ sthedule Process and Resources ⇐ So, that system is in safe state, or, deadlock free.

① Banker's Algorithm ⇒

Total Resource :→ A=10, B=5, C=7.

(max − allocated)

Process	Allocated Resource A	B	C	Max need Resource A	B	C	Availble Resource A	B	C	Remaining needed Resource A	B	C	
P_1	0	1	0	7	5	3	3	3	2	~~7~~	~~4~~	~~3~~	④
P_2	2	0	0	3	2	2	5	3	2	~~1~~	~~2~~	~~2~~	①
P_3	3	0	2	9	0	2	7	4	3	~~6~~	~~0~~	~~0~~	⑤
P_4	2	1	1	4	2	2	7	4	5	2	1	1	②
P_5	0	0	2	5	3	3	7	5	5	~~5~~	~~3~~	~~1~~	③
Total already	7	2	5				10	5	7				

▣ Safe state : $P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_3$

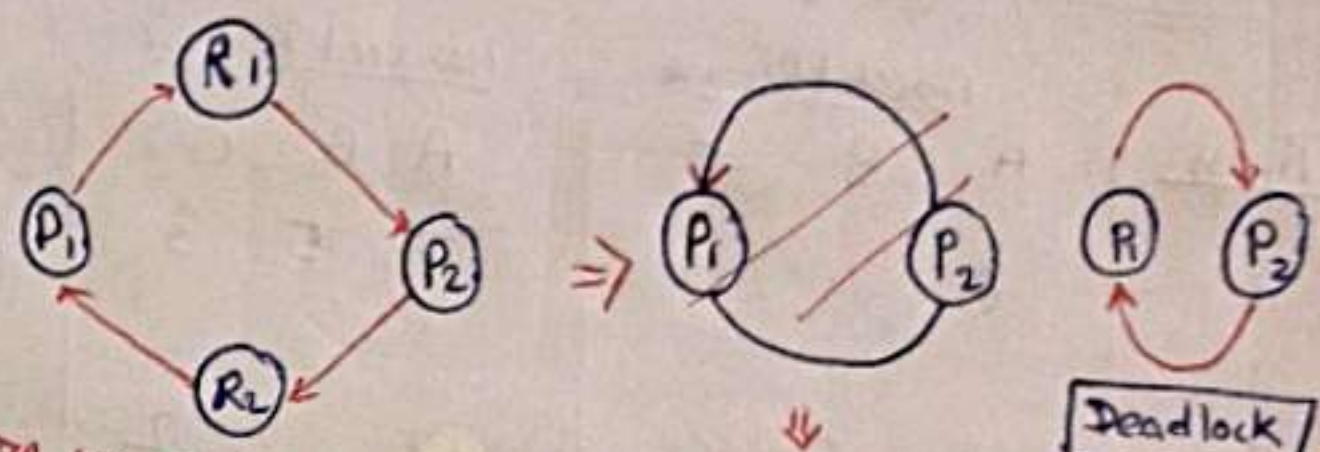
(i) Safe state : If we are able to schedule all process.

(ii) Unsafe state : State in which we are not able to schedule any 1 Process or more.

⊙ Deadlock Detection :→

- System → Algorithm → deadlock ? → NO (back to Algorithm)
- deadlock ? → Yes → Recovery

- ⊡ & Dection Techniques ⇒
  - (i) Single instance of Each Resource Type



Here, we remove the resources from RAG and check for cycle.

Cycle ⇒ Deadlock
No-cycle ⇒ No Deadlock

  - (ii) Multiple Instances of Each Resource
    - → Banker's Algorithm.

- ⊡ Recovery from Deadlock :→
  - (i) Process Termination
    - (i) Abort all Deadlock process.
    - (ii) Abort 1 low priority process from deadlock cycle.
  - (ii) Resource Preemption
    - → Prempt Resources of low priority process in deadlock cycle and make it available for other process.

⊙ Memory Management Techniques :→

⊡ Logical Address Space (L·A·)

An address

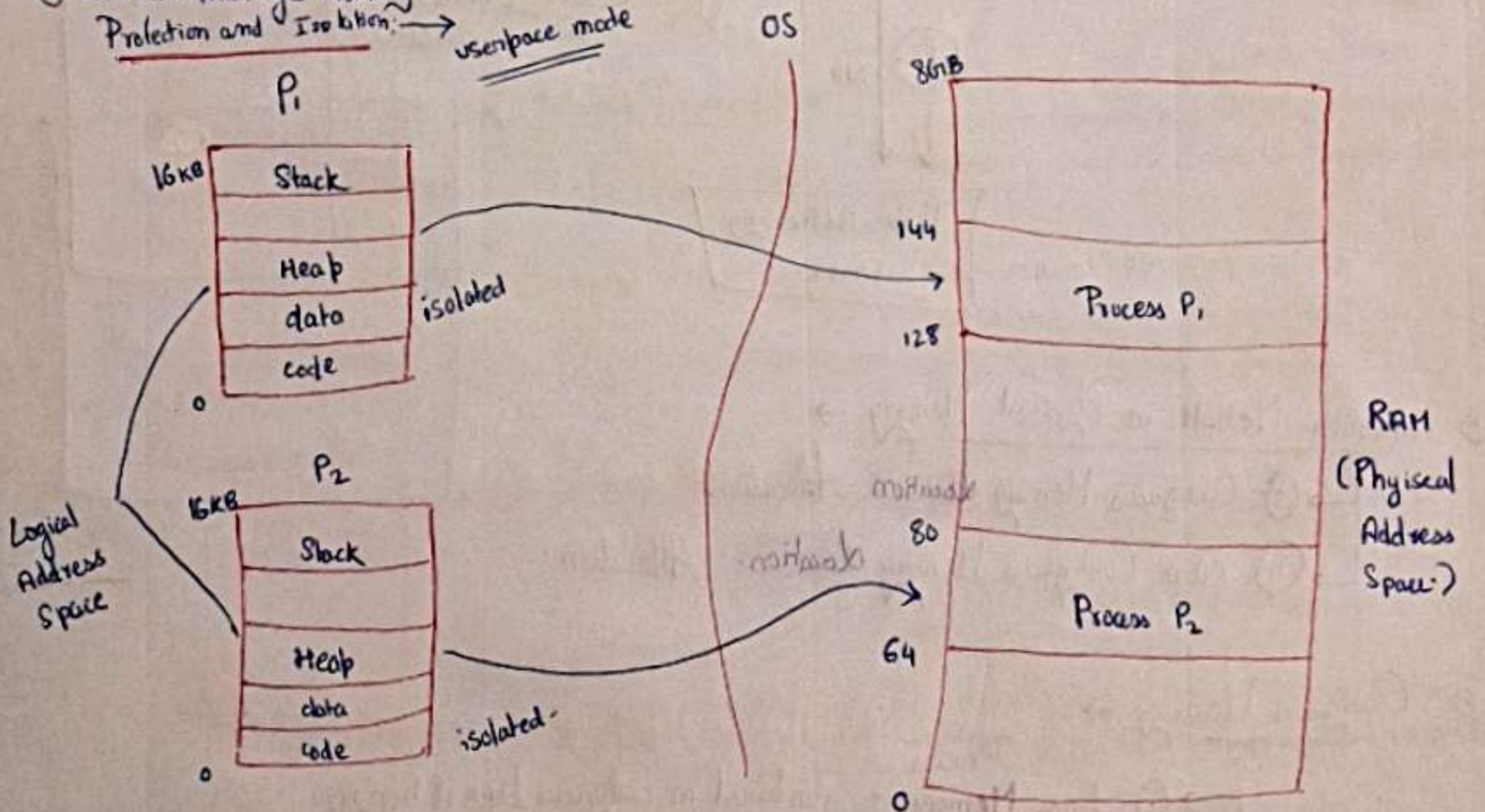
- (i) Generated by CPU.
- (ii) It is the address of instruction/ data used by a process.
- (iii) User has access to it and also has indirect access to P.A. through it.
- (iv) Not exist physically (Also known as virtual address).
- (v) Range → 0 - Max.

⊡ Physical Address Space (P·A·)

- (i) Not accessiable by user.
- (ii) P·A· is in memory unit, located in RAM.
- (iii) Set of all physical address corresponding to L·A· is called P·A· space.
- (iv) Computed by MMU (Memory Management Unit)
- (v) Range: (R+0) → (R + max)

R → base value.

⊙ How Os manages memory Protection and Isolation: → user space mode



Process P_1

Base → 128

offset → 16 KB

Process P_2

Base → 64

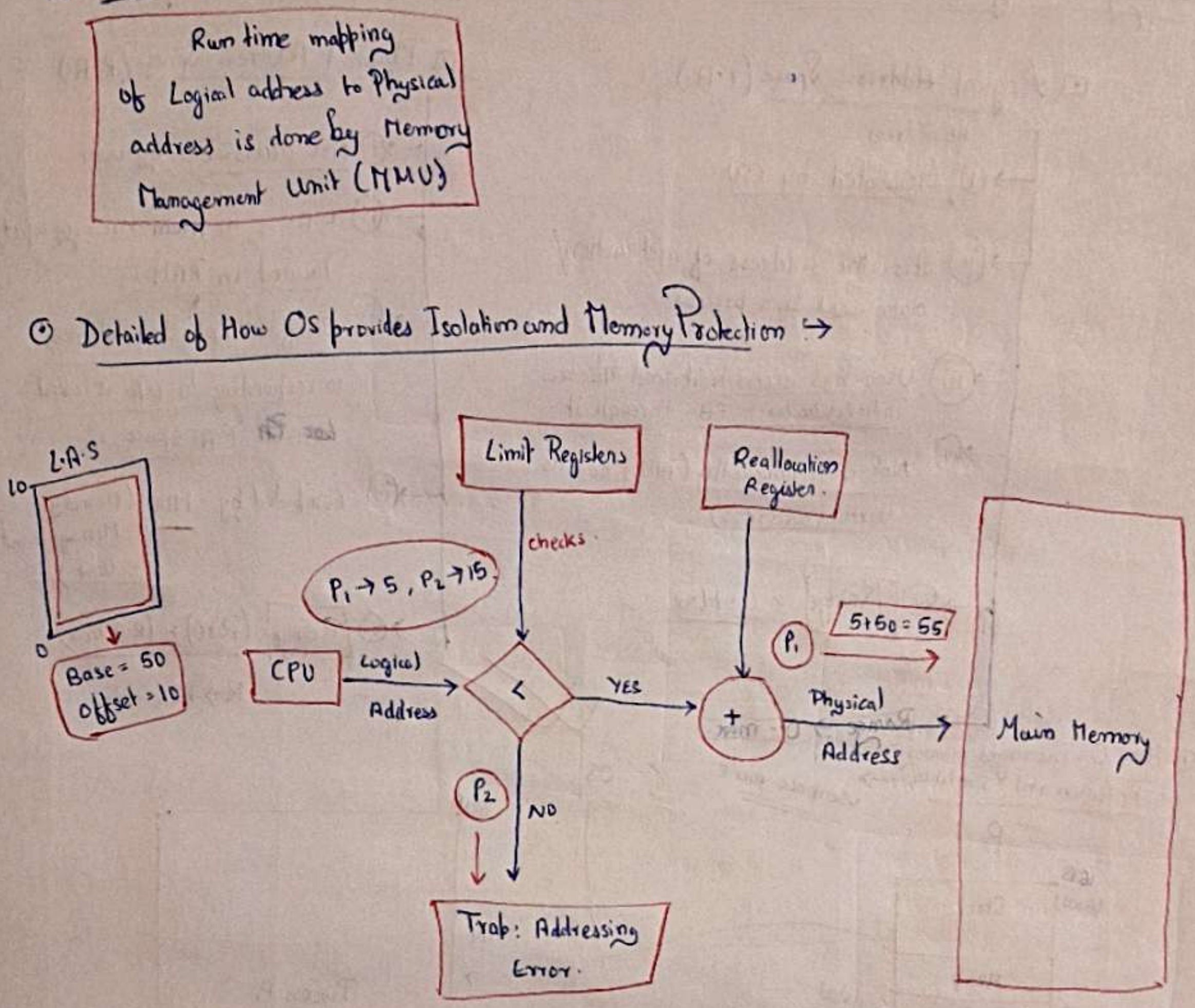
offset → 16 KB

Here Os takes L·A and stores is somewhere in Physical address. If program wants to access a particular P·A, then Os do Base + offset of that program and check whether P·A· required by program is within the P·A· range of the program or not.

⊡ Note :→

Run time mapping of Logical address to Physical address is done by Memory Management Unit (MMU)

⊙ Detailed of How OS provides Isolation and Memory Protection →



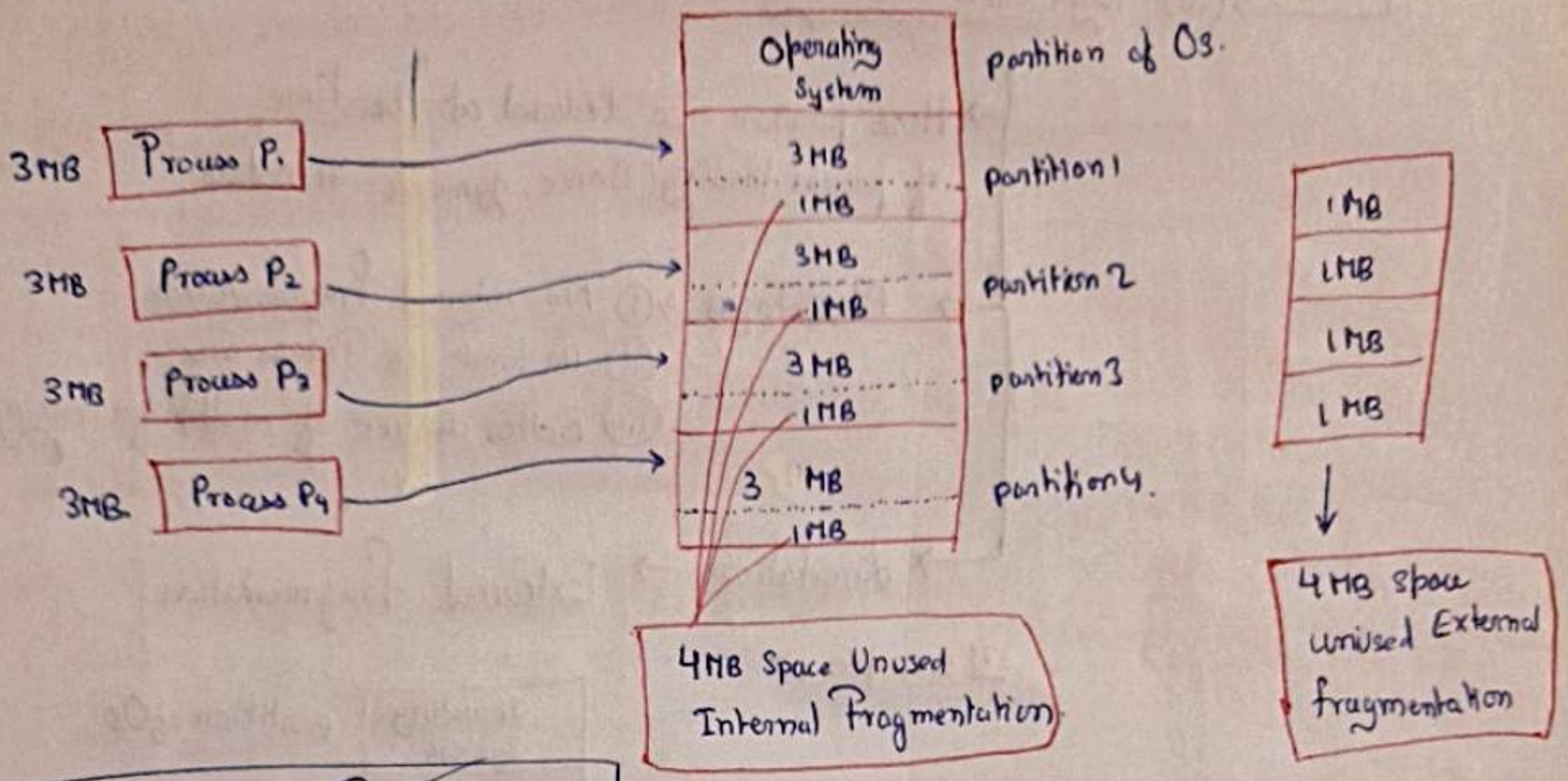
⊙ Allocation Methods in Physical Memory →

- (i) Contiguous Memory — Allocation.
- (ii) Non-Contiguous Memory — Allocation.

⊙ Contiguous Memory →

- (i) Each Process is contained in contiguous block of Memory.
- (ii) Fixed Partioning ⇒ Main Memory divided into partitions of equal or different sizes.

□ Example (FP) :→



□ Limitations of Fragmentation :→

□ Limitation of Fixed Partioning :→

- (i) Internal Fragmentation :→ size of Process < size of partition, some partition remain unused. This wastage of memory is called Internal Fragmentation.
- (ii) External Fragmentation :→ Total unused space of various partitions cannot be used to load a process even though memory is available.
- (iii) Process Size :→ Sometimes, if Process size > size of partition then process cannot be loaded in memory.
- (iv) Low Degree of Multiprogramming (DOM) :→ D.O.M. is fixed and less as Partion size does not changes with Process size.

→ (iii) Dynamic Partioning →

→ Here partion size declared at the time of process loading. Hence, dynamic in nature.

→ Advantages → (i) No Internal fragmentation.
(ii) No limit of Process Size.
(iii) Better degree of multiprogramming.

→ Limitation → External fragmentation.

⊡ Example →

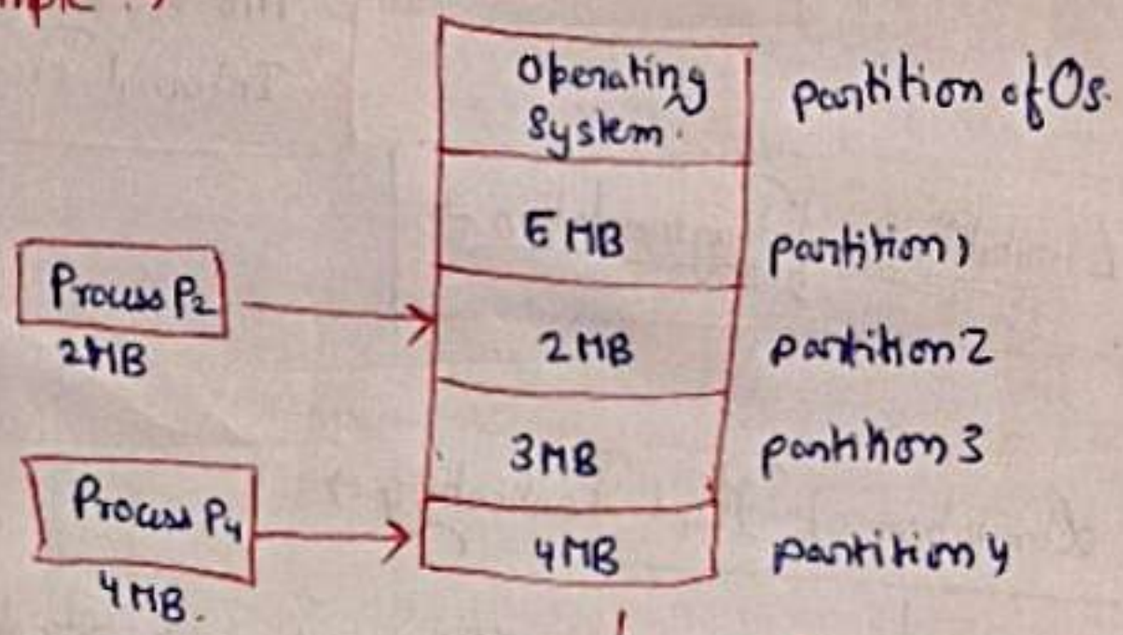


Fig 1.1

Now, here if process5 comes with requirement of 8MB, it does not get loaded in memory though 8MB is present as 8MB memory present is not contiguous.

⊙ Free Space Management →

(i) Dynamic fragmentation suffers from External fragmentation.

(ii) Defragmentation → ⊡ In this process we merge the free spaces and keep them together. And, we sort free spaces and loaded memory spaces and keep them apart.

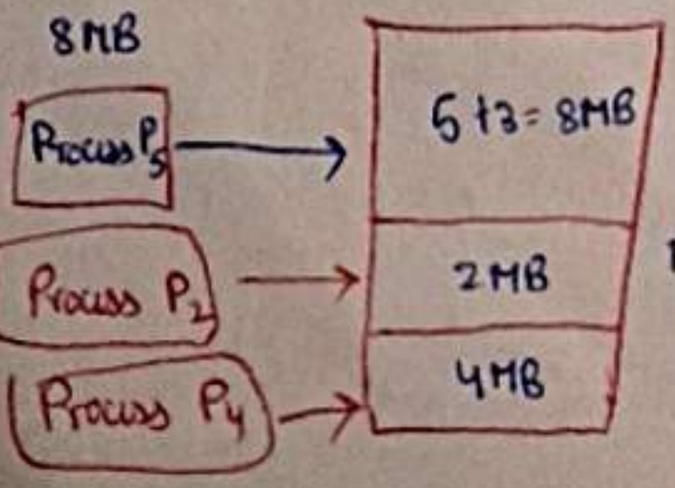


Fig 1.1 after defragmentation.

⊡ Limitation → Its a very time consuming process due to alot of overheads. (because of searching freespaces and merging it)

Note :→ How freespace stored/represented in Os?

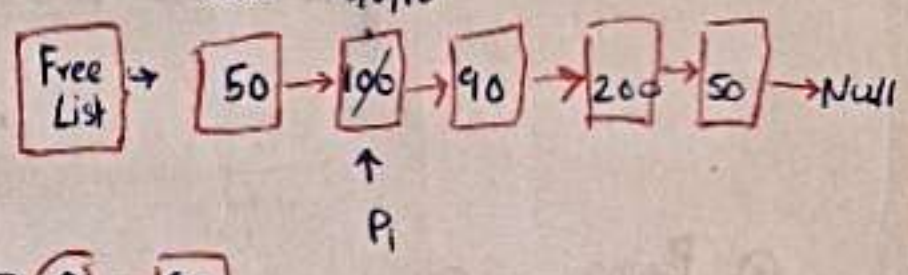
Free holes in memory are represented by a free list. (Linked-List Datastructure used)

How to satisfy request of nsize from a list of free holes?

ans:- Various Algorithms implemented by Os to find free holes in LL and allocate them to process are - - -

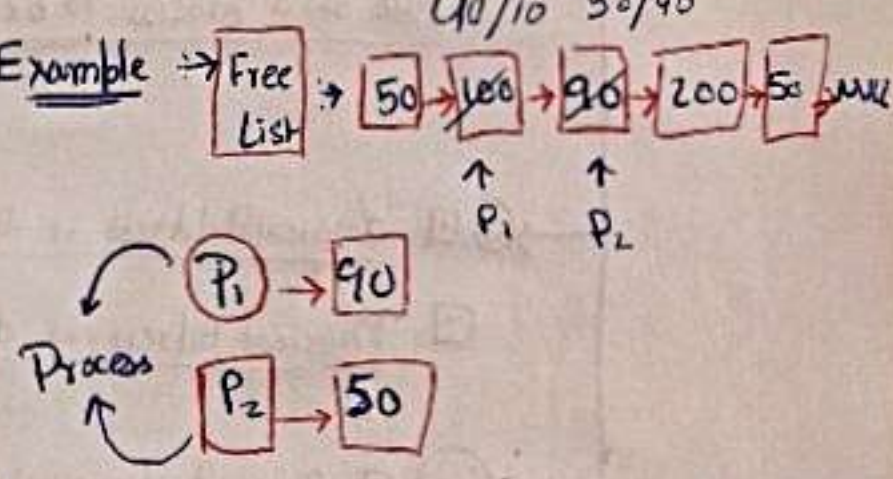
(i) **First fit** →
- (i) Allocate first hole that is big enough.
- (ii) Simple and easy implementation.
- (iii) Fast/less Time Complexity.

Example :→ allocate 90/10 → left



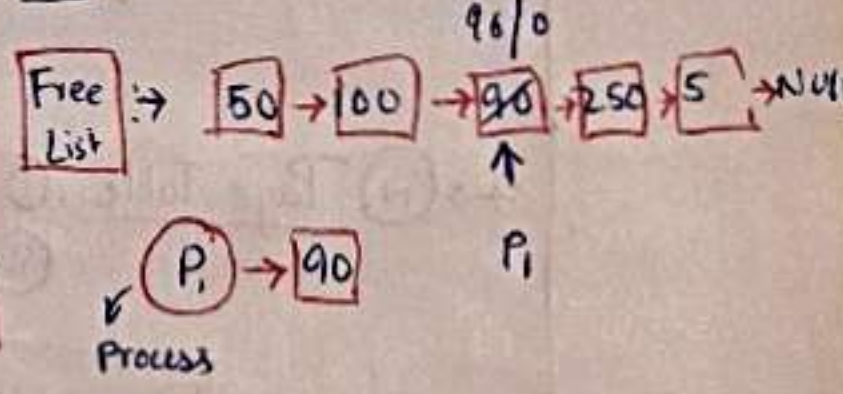
(ii) **Next fit** →
- (i) Enhancement of first fit but search always from lastly allocated hole.

Example →



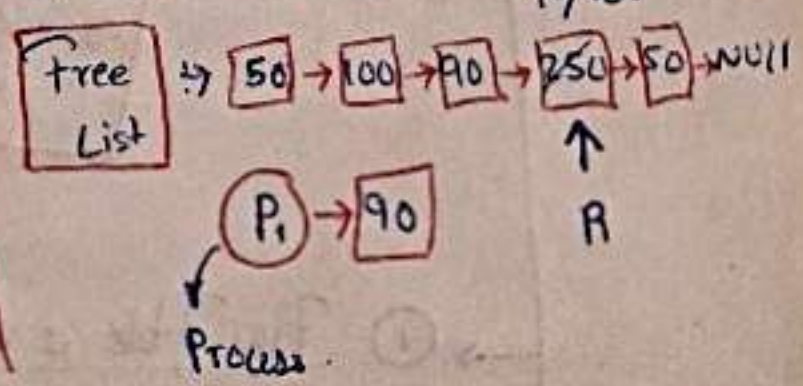
(iii) **Best fit** →
- (i) Allocate smallest hole that is big enough.
- (ii) Less Internal Fragmentation, but more external fragmentation due to large number of small free holes.
- (iii) Slow, as have to iterate over whole free list.

Example →



(iv) **Worst fit** →
- (i) Allocate the largest hole that is big enough.
- (ii) Slow → as iterate over whole free list.
- (iii) Less External fragmentation due to larger left wholes.

Example :→



⊙ Non-Continuous Memory Allocation ⇒ (NCM)

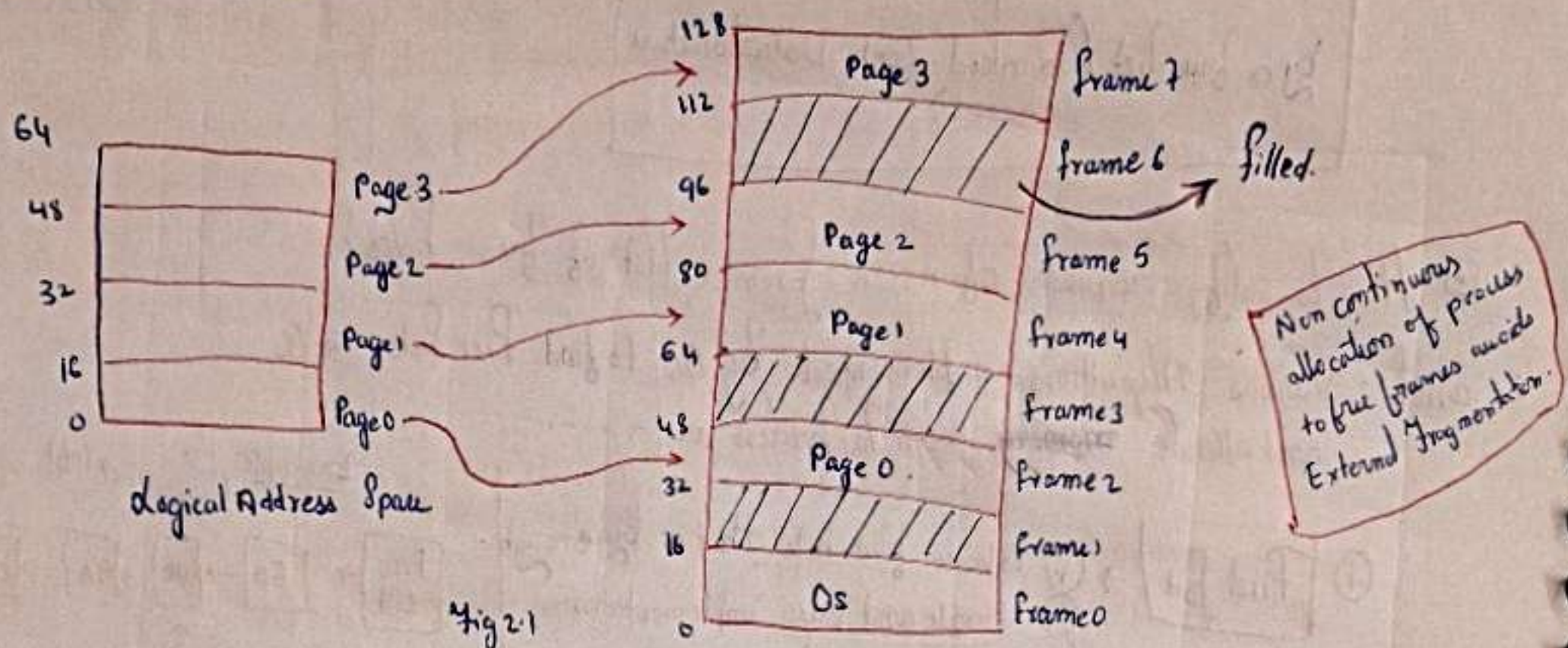


Fig 2.1

⊙ Paging :→ (i)

(i) It is an NCM Scheme that allows the physical address space of a process to be non-continuous.

(ii) ⊡ LogicalAddress is divided in fixed size partition, called as Pages.
⊡ Physical Adress is divided in fixed size partition, called as Frame.

(iii) ⊡ Size of pages is Page Size
⊡ Size of Frame is Frame Size
} Page Size = Frame Size

(iv) Page Table : (i) Datastructure storing mapping of Page → Frame.
(ii) Each process has different Page Table (depend Process size).

Note ⇒ Page table consists of base address of each page in physical memory.

Page No.	Frame no.
0	2
1	4
2	5
3	7

→ Process Table of Fig 2.1

(v) Page Table is stored in main memory at time of process creation and its base address is stored in PCB.

(vi) Page Table Base Register (PTBR) —points→ current page table. Changing page table requires only this register at time of context switching.

Example ⇒ 25 bytes. ∴ L·A·S = 64 bytes = 2^6 ⇒ 6 bits

L·A·S. ⇒ 6 bits ⇒ 25 = | 01 | 1001 |

Page no. (01) → Page 1 ; offset (1001) = 9

⇒ 16 + 9 = 25

2|25 1
2|12 0
2|6 0
2|3 1
2|1 1
0

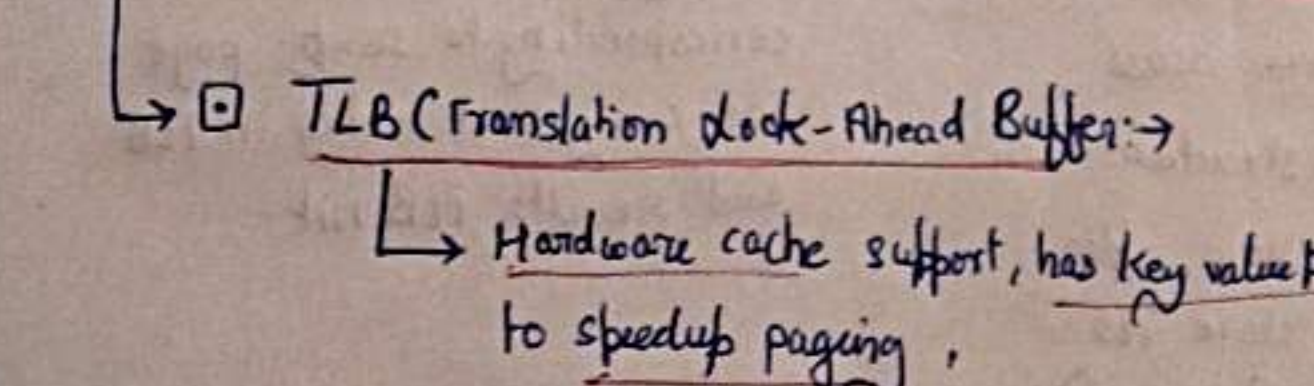
P·A·S ⇒ 128 bytes = 2^7 = 7 bits

7 bits ⇒ 73 = | 100 | 1001 |

frame number (100) = 4 ; offset (1001) = 9

⇒ 64 + 9 = 73

2|73 1
2|36 0
2|18 0
2|9 1
2|4 0
2|2 0
1



① Why Paging is slow and how to make it fast?

↳ ⊡ There are lot of memory references to access a desired location in physical memory, which leads to lot of overheads.

Example

context switch → New Process → New Page table stored in RAM → desired location in physical memory.

Ref1 Ref2

↳ ⊡ TLB (Translation Look-Ahead Buffer) →

↳ Hardware cache support, has key value pair, to speedup paging.

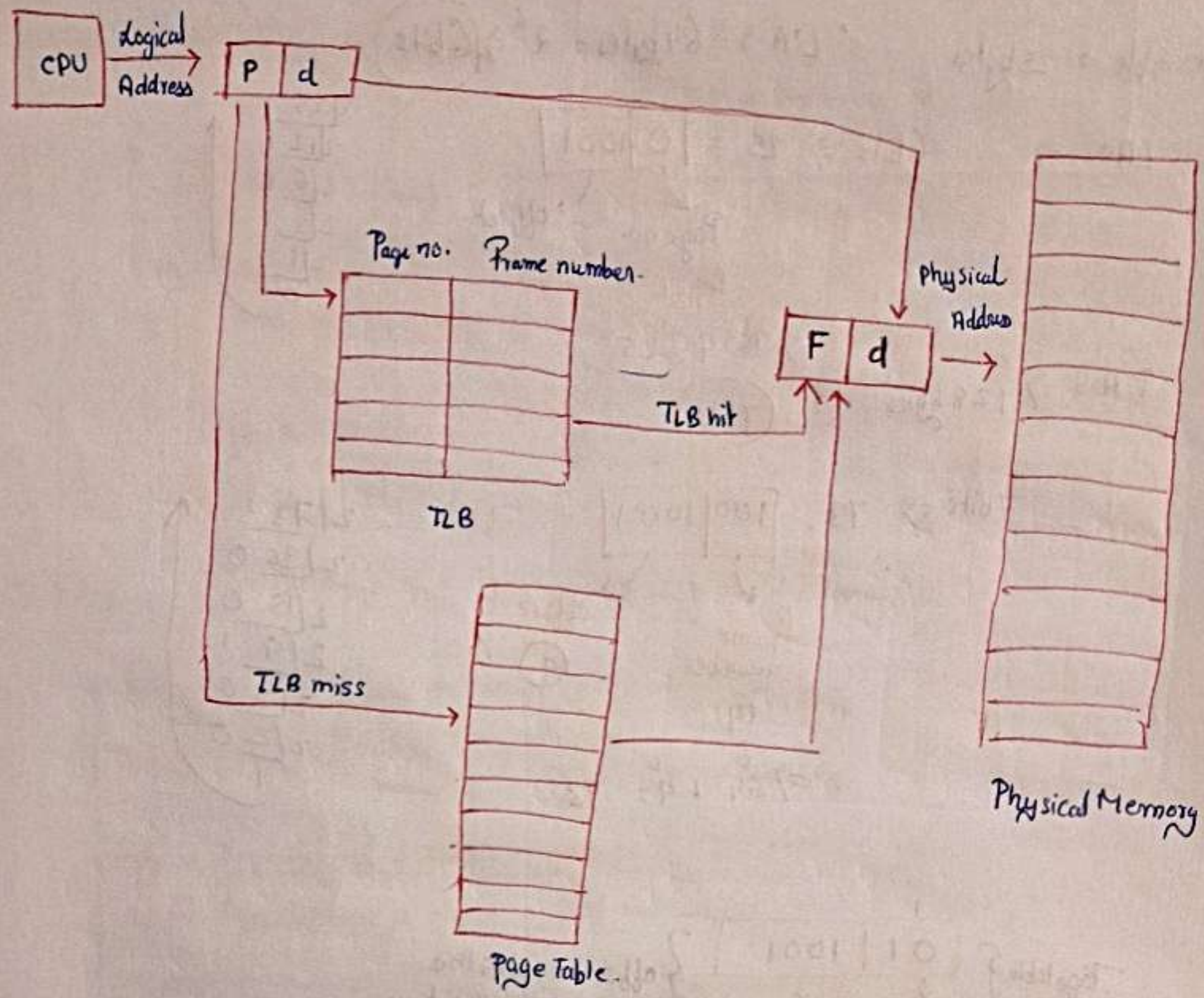


Fig 3.1 Paging with TLB hardware.

Here what happens is CPU first goes to Logical Address and then Try to find Frame no. corresponding to page no. present in L.A.S inside TLB. If it does not gets the entry in TLB, then it is a TLB Miss, then we go to page tab and ~~acces~~ access frame no. like array data structure a[Page no.] = frame no. Now, after getting frame no., we store its a key-value pair (Page no: Frame no.) in our TLB for later usage and Through it we also get our desired location of Frame in physical memory corresponding to particular ~~Page~~ Page.

later, if we search for frame corresponding to same page we can get it entry in TLB and results TLB hit.

⊙ TLB at time of Process Context Switching :→

2 solution → (i) flush entire TLB. (Lot of overhead) ⊗

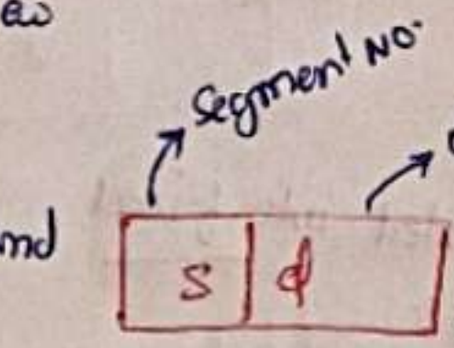
(ii) Use of additional parameter ASIDs to uniquely identify each process.

ASIDs	Page no	Frame no

TLB

⊙ Segmentation :→

- Consists of Divides a process into different Segements having different sizes based upon user's views.
- In paging, we don't have any user view of process like Segmentation.
- Each segment has a segment no. and offset like paging.



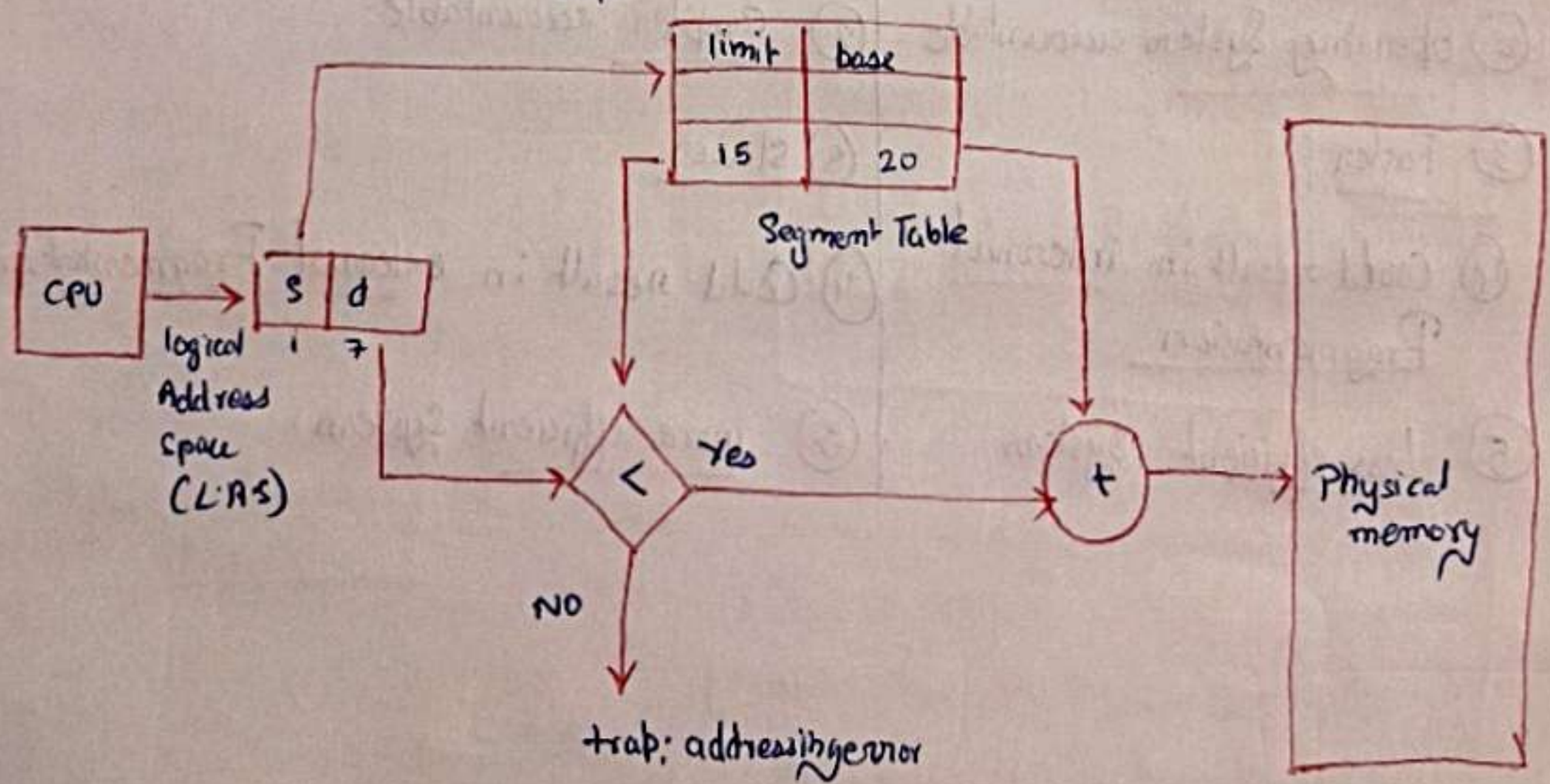


Fig 4.1 Segmentation Hardware.

Example :→

process :→ Segment :→ | 01 | 111 | (s = 1, d = 7) ⇒ Firstly CPU takes L.A.S, then checks limit > offset or not, if yes than adds offset + base and stores it in physical memory else throws an error.

⊙ Advantage ⇒ (Segmentation)

- (i) No internal fragmentation like Paging.
- (ii) One segment has continuous allocation. (Unlike Paging we use to divide a function if its size was greater than page size.)
- (iii) Size of Segment table < Page table.
- (iv) More effecient as compiler keep same type of function is same one segment.

⊙ Disadvantage:→

- (i) External Fragmentation
- (ii) Different segment size not good for scaffing. → learnt in MTS.

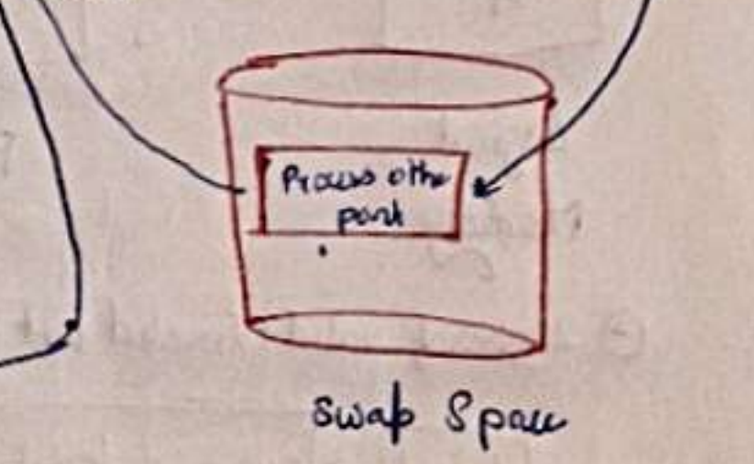
⊙ Paging	Segmentation
(1) Process divided into fixed size pages.	(1) Divided in variable size pages.
(2) operating System accountable	(2) Compiler accountable.
(3) Faster.	(3) Slower
(4) Could result in internal Fragmentation	(4) Could result in external Fragmentation.
(5) Less efficient system	(5) more efficient system.

⊙ Virtual Memory Management → (VMM)

⊡ It is a technique by which we provide a user an illusion of having a big memory by executing a process that is not completely present in the memory.

⊡ RAM + Swap space : Virtual memory

In swap space, we keep the part of processes which are not required in present, and in Ram we keep the required ones. Now, when we require a particular part of processes in RAM for execution. Then our RAM swaps out the old part of process and from Swaps Spaces the required new part of process Swaps In



⊡ Note → Through concept of Virtual Memory we can create run many process/ program, at same time, leading to increase degree of Multiprogramming.

⊡ Advantage →

① User benefit: → user can run a program having size greater than that of physical memory.

② System benefit: → Hes in CPU utilization and throughput.

⊡ Disadvantage →

① System may become slow due to lot of swapping

② Thrasing may occur.

⊙ Demand Paging →

① Helps to Implement VMM concept.

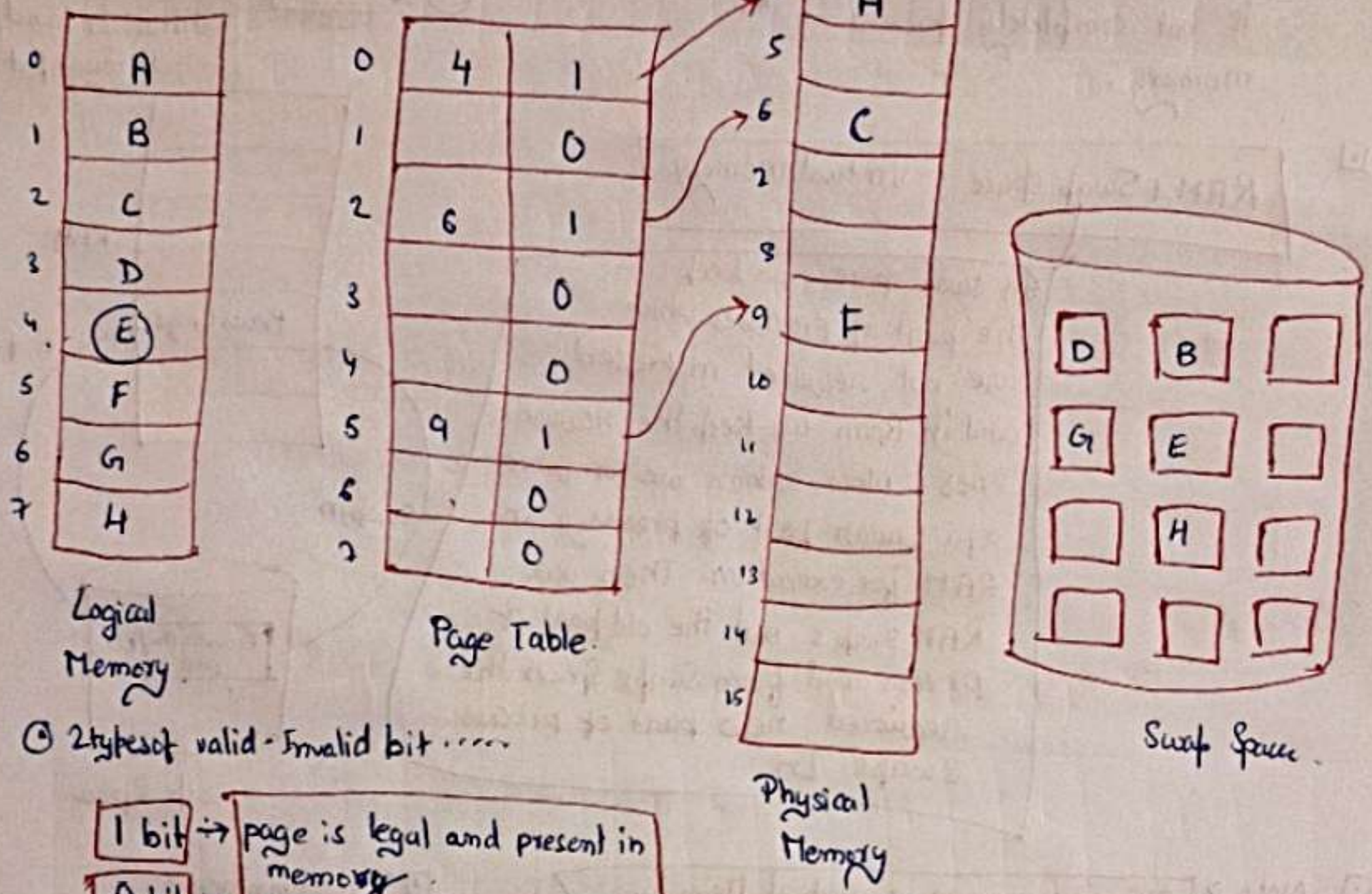
② A new page of a process is only copied to main memory when demand is made, or page fault occurs.

③ We use Lazy swapper to swap in-out our page.

↓

Instead of swapper please use pager as we are here not swapping the entire process but the pages of different parts of a process.

⊙ Working of Demand Paging ⇒



⊙ 2 types of valid - Invalid bit .....

1 bit ⇒	page is legal and present in memory.
0 bit ⇒	page is either not valid or present in in Swap Space, ib page is valid.

Let's take an example to understand this, suppose we want to put E at frame 12 in physical memory Now we will be checking that whether our E is present on Swap Space or not. If it is present we will take E from our Swap Space and Swap in the Physical Memory. Now, we will put the Entry of Frame number in Page Table. and will update the valid-Invalid bit to 0.

Note ⇒

To tes performance of demanding Page we use locality of reference

⊙ Page Replacement Algorithms → **AIM ⇒ (Minimum Page fault).**

⊡ Page fault :→ When our OS basically request for page which is not present in RAM, but present in Disk then, it results in page fault.

⊡ Page fault Service Time (PST) :→ The amount of time required to do the service of bringing required page from disk to RAM by swap in-out is called PST.

① FIFO →
① Easy to Implement.
② Performance not always good.
③ Belady's anamaly present. (LRU and Optimal algorithms, we see that page faults will be reduced with ↑es in number of frames. But, in some cases of FIFO algo it not happens which leads to Belady's anomaly.)
④ Allocate frame to page as it comes into memory by replacing oldest page.

Example :→ 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1

P_1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
P_2		0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
P_3			1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
	M			M			M	M		M	M			M	M			M		M

Page faults = Total requests - Page found
= 20 - 5 = (15)

② Optical Page Replacement → ① Best Algo, give min Page faults.
② Impossible to implement.
③ Find a page whose reference is there in future much later than others and replaces it.

Example → 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

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

check in R.H.S. → ∴ Page faults = 19 - 10 = (9)

③ LRU (Least Recently Used) :→

① We check for the oldest pages allocated to frame in past and replace it here.

Example → 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 ← Check L-H-S.

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

20 - 8

∴ Page Fault = = (12)

Limitation might be the counting capacity as it has fixed size (Int datatype)

□ LRU Implementation

① Counter :→ By use of Global shared count.

From above ex.

Global = 0

Here Global counter get incremented and gets stored in local count 0, 1, 2, 3, 4, 7, from which we do replacement

0 → (2) (5) (7) (11) (16) (19) →
1 → (3) (14) (17) (20) →
2 → (4) (9) (13) (15)
3 → (6) (10) (12)
4 → (8)
7 → (1) (18) →

② Stack → □ Keep a stack of page number.

□ Whenever page is referenced it is removed from stack and put on top.

□ As enteries might be removed from middle so, doubly linkedlist is used.

⊙ Counting Based page replacement :→ ▣ keep a counter of the no. of references that have been made to each page (Reference counting)

(Not Imp)

2 types

ⓘ LRU (Least Frequently Used)

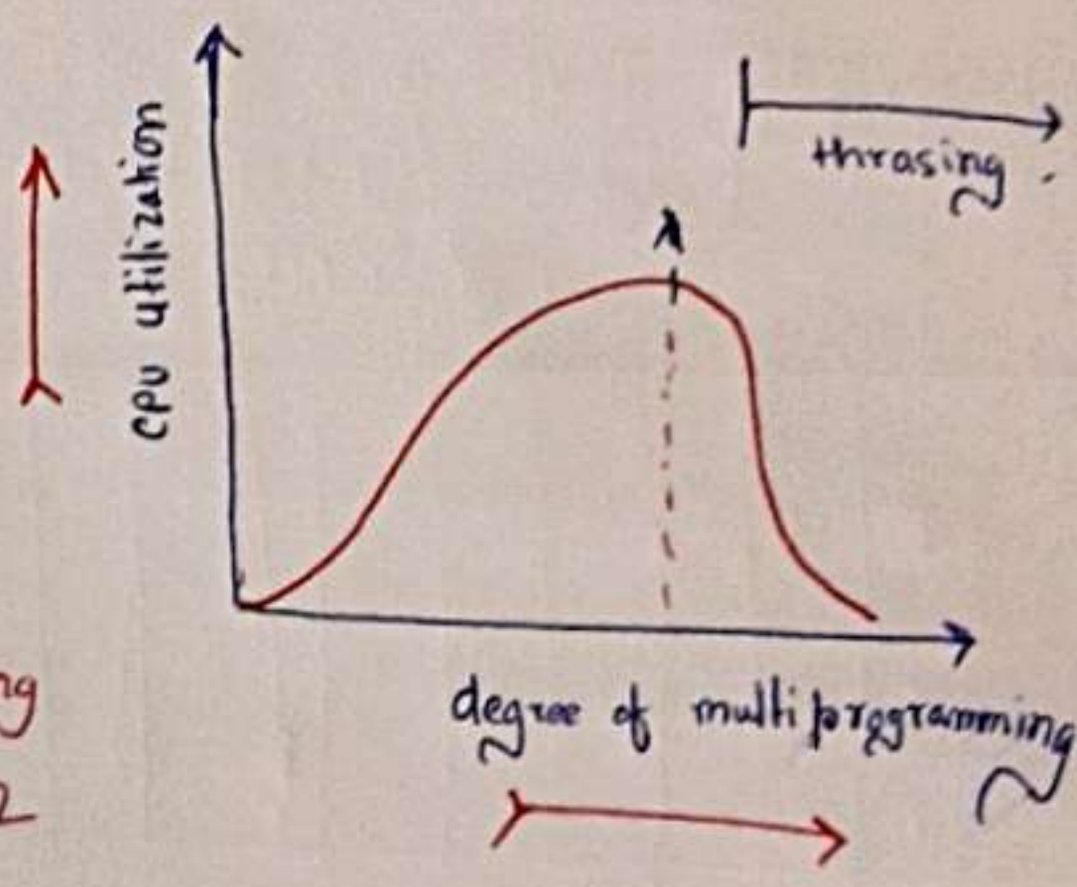
Here we keep pages with large reference count and removes the ones with small reference count.

ⓘⓘ MFU (Most Frequently Used)

Here we keep pages with small reference count and removes the ones with larger reference count.

⊙ Thrashing :→

Here basically the process does not have required number of frames to hold all pages.



▣ Firstly we need to ↑es our CPU utilization. So in order to do that we will ↑es the degree of multiprogramming by dividing process into pages and and keeping 1 or 2 pages of each process in RAM.

▣ By doing this stuff a time comes basically in which the CPU requests for pages which are not present & in main memory (RAM), which Results in a Page fault. Now, in order to solve the page fault (Swap in-out of pages from disk to RAM), it takes a lot of time due to which there CPU utilization ↓es drastically. This is called as Thrasing.

Note →

A System is in thrasing when it spends more time serving page fault than executing process

⊙ How to Remove thrasing?

~~ⓘ~~ ↑es main memory size.

~~ⓘⓘ~~ By reducing degree of multiprogramming by use of Long Term Scheduler.