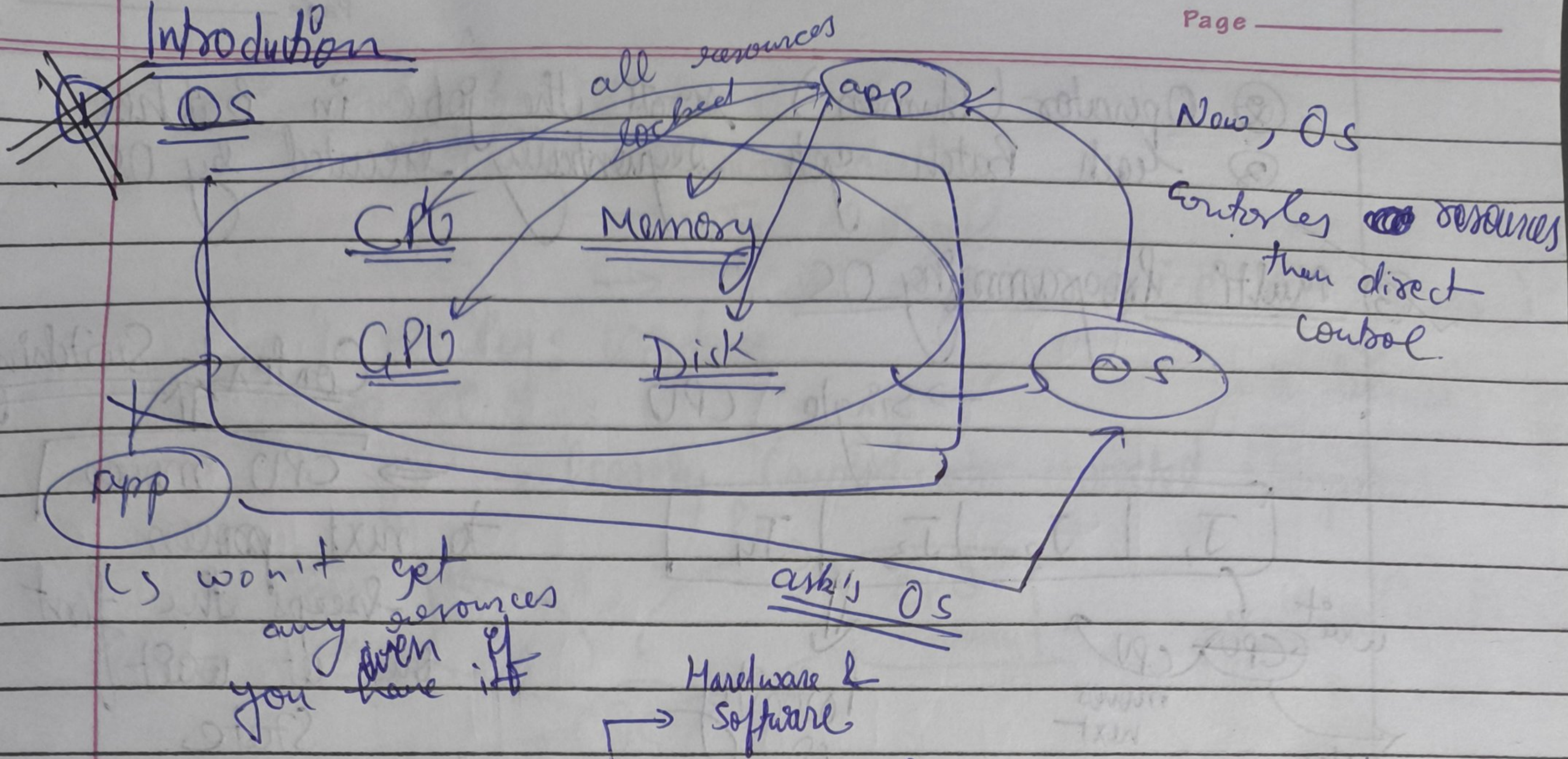


~~Phle~~ Hardware banta hai
uske band OS.

Introduction

OS



OS : ① Interface ② Resource Management & memory management
③ isolation & protection
↳ One app's code won't interfere in another's app code.

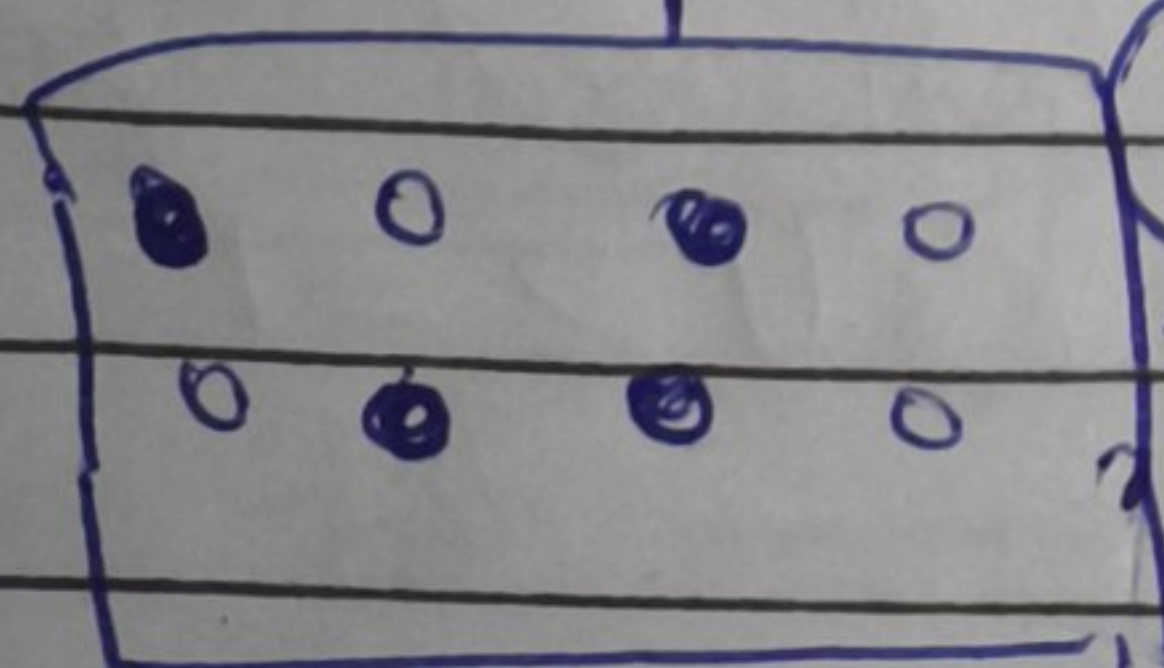
Types of OS

Goal of OS? → ① maximum CPU Utilization
② Process starvation should not occur
③ High priority job execution

Types of OS

① Single process OS ⇒ One job at a time
eg: MS DOS

② Batch processing OS ⇒ ① jobs were submitted as punchcard



punch card

Scan

Digital Information

eg: ATLAS

- ② Operator (Human) sorts the jobs in Batches
 ③ Each Batch gets sequentially executed by OS.

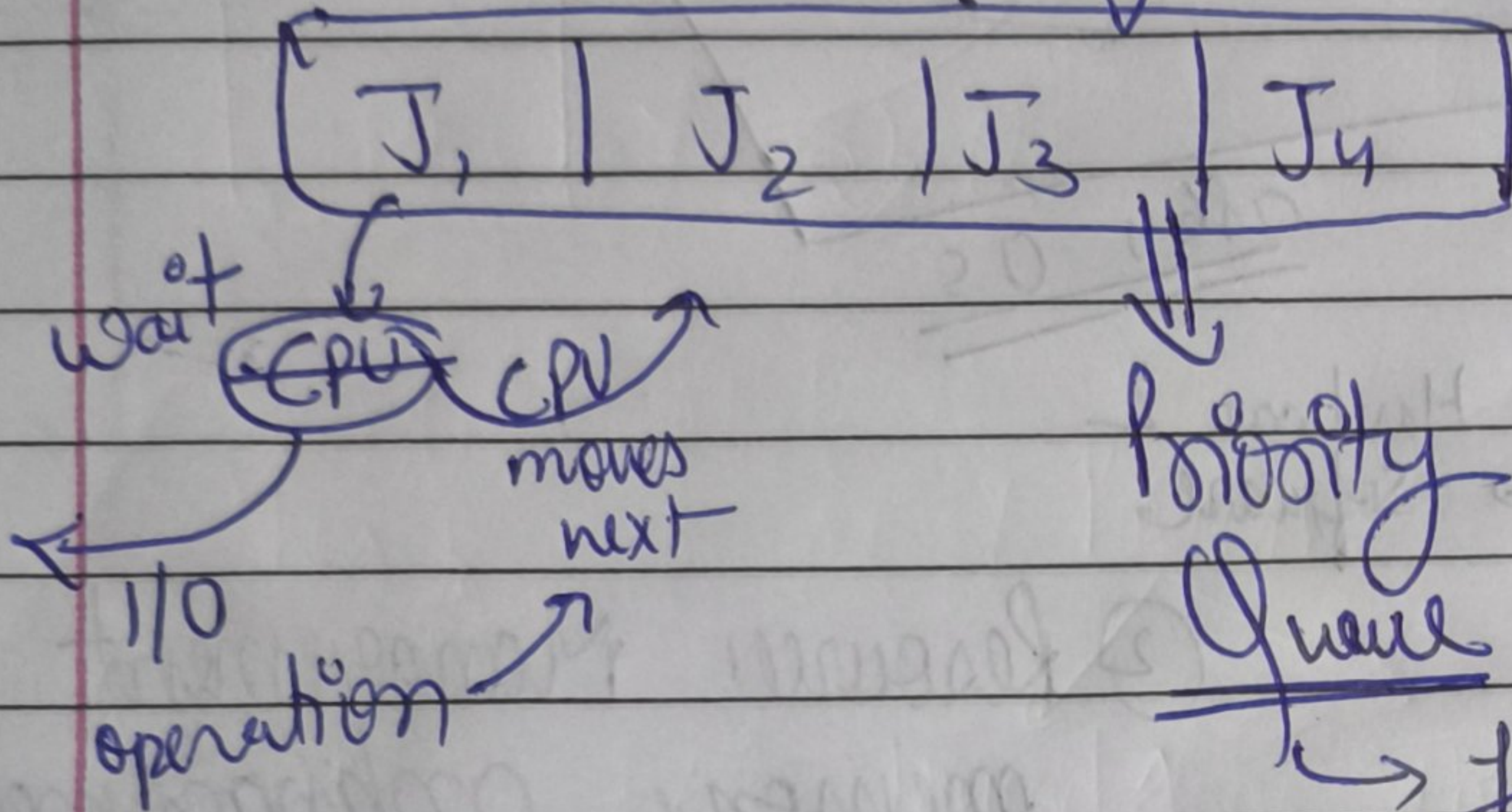
③ Multi Programming OS :

→ Single CPU

Context Switching

⇒ CPU moves

to next process
keeps the first
one at wait
State.



eg: DHE
filled with ready
to executed jobs.

⇒ The context (like current State, Call stack Info, etc.) gets stored in PCB → Process Control Block

④ Multi-Tasking OS

→ Single CPU

→ Context Switching

→ Time sharing

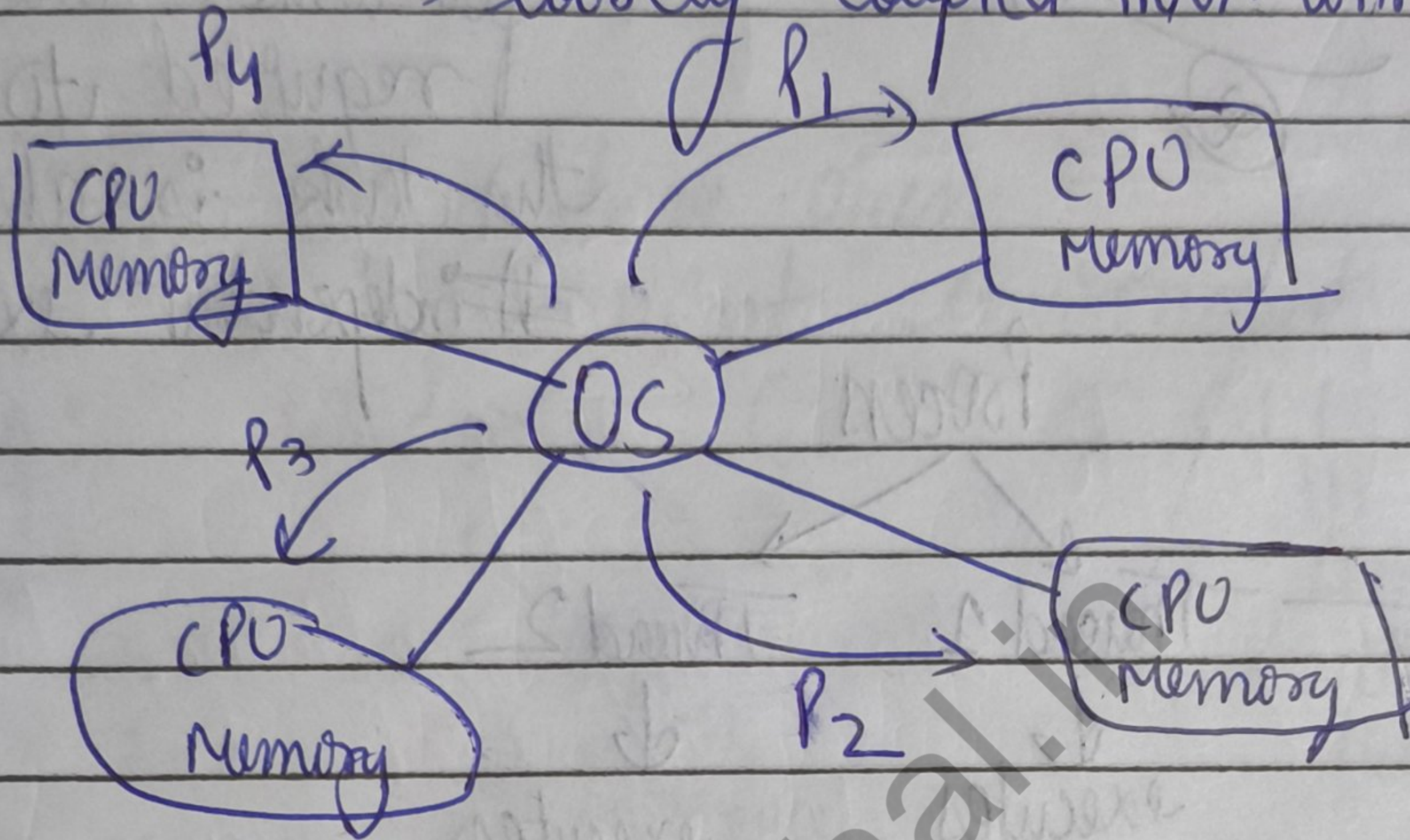
↳ Rather than waiting for a process to go into I/O to switch context, time quantum (maximum time a process can be executed) added, so that starvation does not happen.

⇒ responsiveness of application Res.

eg: CTSS

⑤ Multi processing OS :-
 → Context Switching
 → Time Sharing
 → CPU ≥ 1 (i.e. CPU cores)

⑥ Distributed OS :-
 → loosely coupled inter-connected processors computers
 eg: Windows



P_1, P_2, P_3, P_4

eg: Amalpa Plan 9.

* Multiple CPUs, Memory connected over LAN i.e. on N/w.
 Distributes the job according to power processing

⑦ RTOS (Real time Operating System)

→ Error chance is lowest & Real time execution.
 eg: ATC (Air traffic Control), Industrial application
 nuclear plants

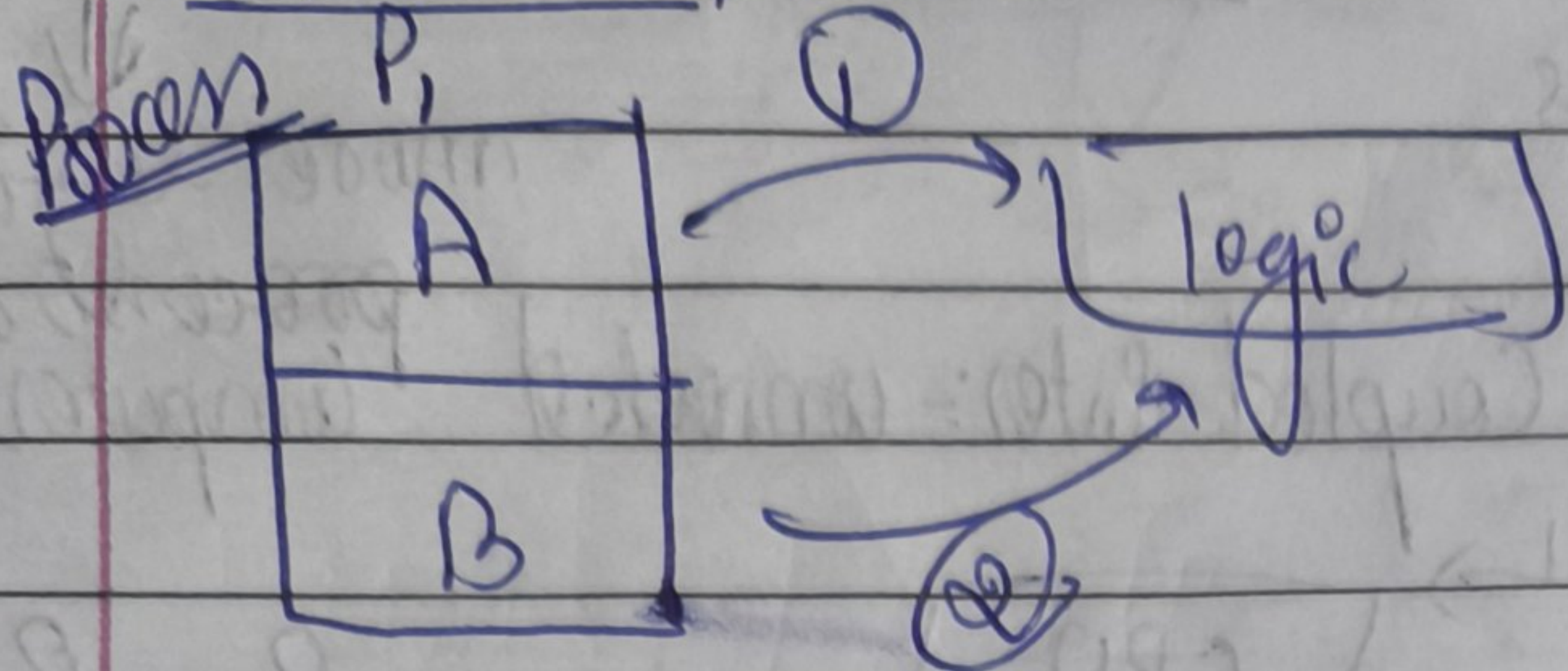
~~Multi-Tasking~~ v/s Multi-Threading

Multi-Threading

exe file → double click → process → A program under execution

Thread $\longrightarrow$ A light weight process
 $\longrightarrow$ independently executes.

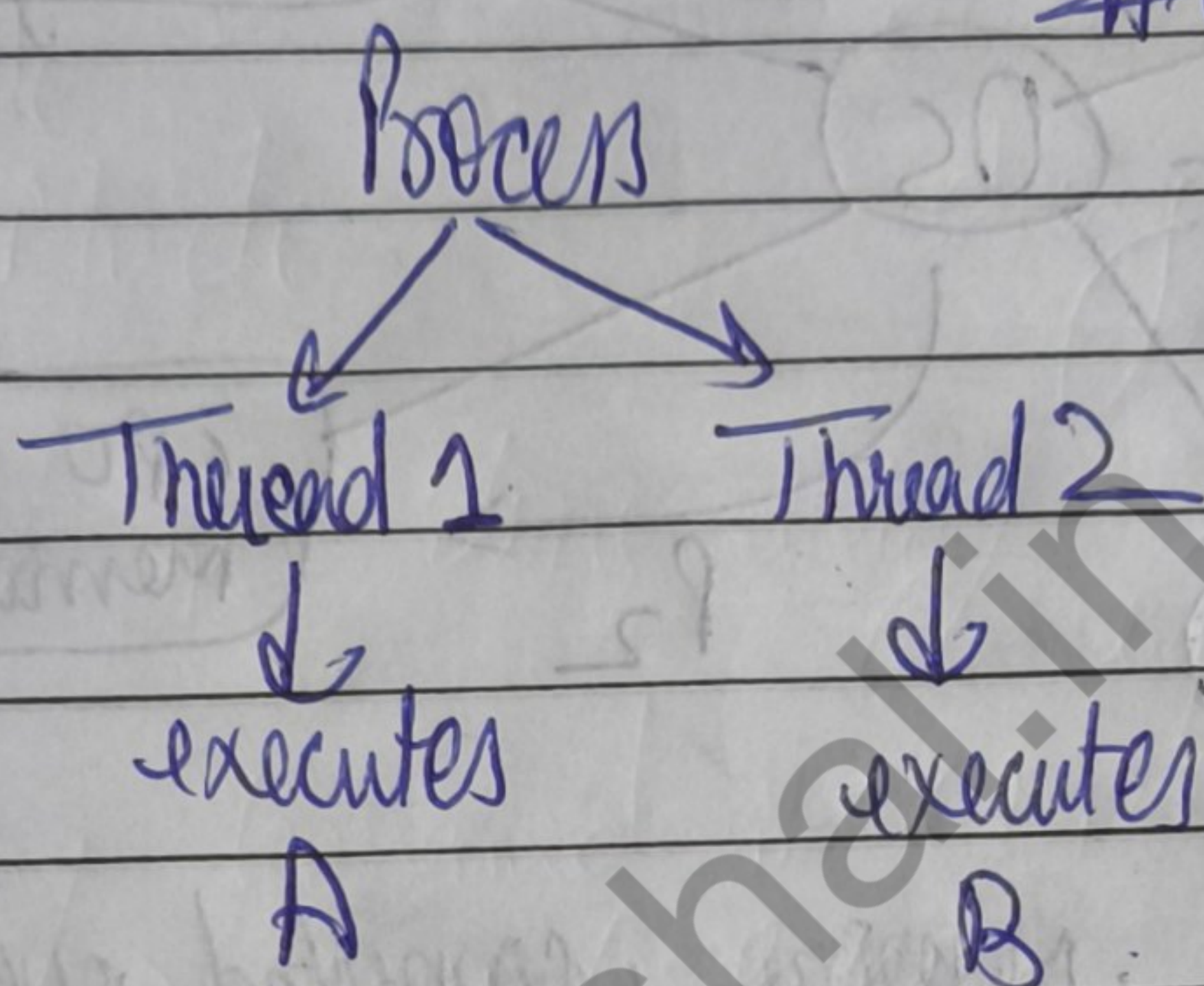
Multi-thread :



Process P_1 has two independent tasks, for which the logic required to execute the task is also independent ~~independent~~ execution.

Then,

Both uses same logic



As, thread requires multiple CPUs $\implies$ only multi-processing supports multi-threading & above

★ No isolation, as a thread is part of the process.
 [6/w threads]

★ Thread spawn is based on their priority.
 Multi-Tasking

- ① More than One process concept
- ② Isolation & Memory protection
- ③ process scheduling
- ④ No. of CPU = 1

Multi-threading

More than One thread concept
 No isolation & memory protection.

Thread scheduling

No. of CPU ≥ 1

Components of OS

① User Space

↓
App run here

② Kernel

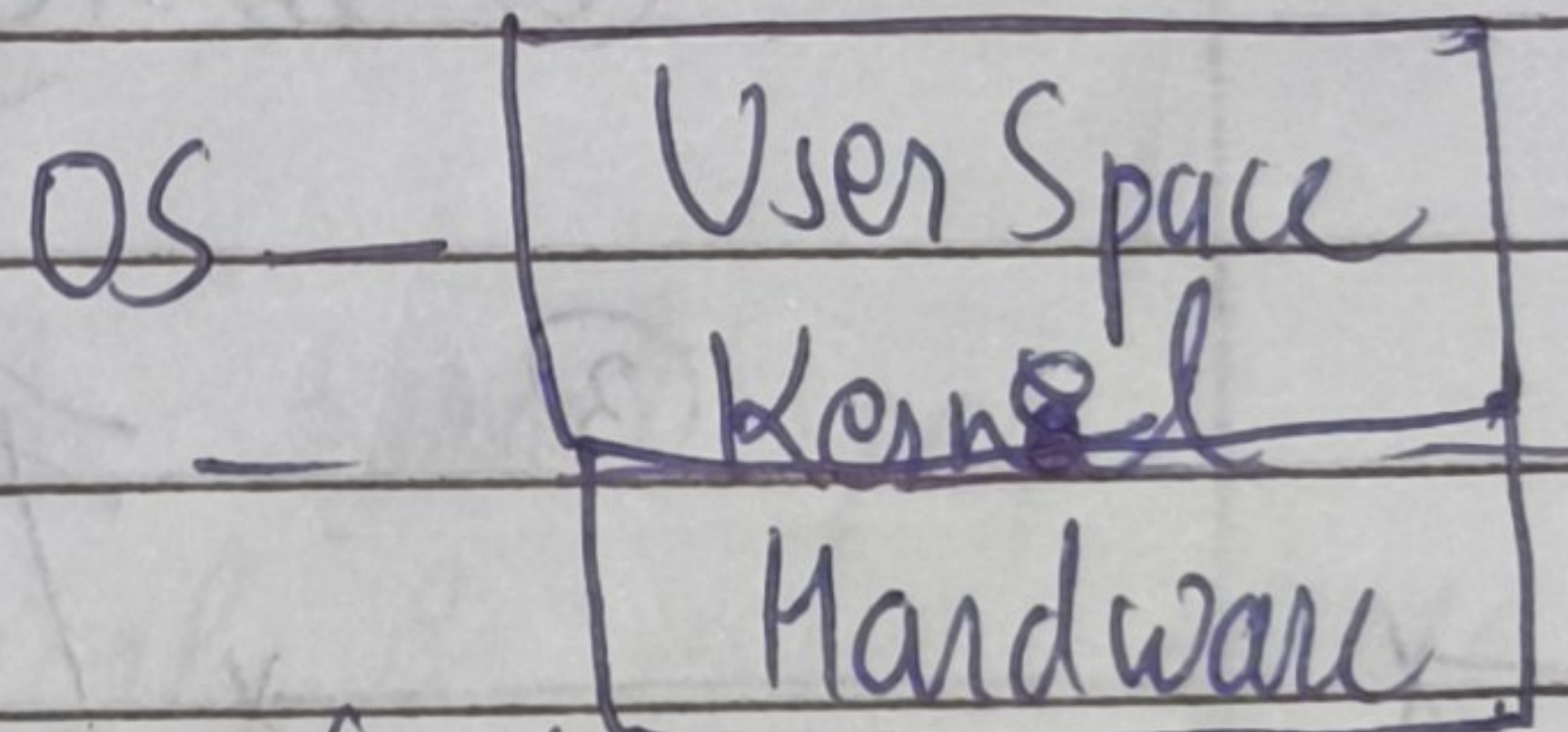
↓
Acum of Hardware

① User Space :

→ No Hardware access
→ convenient environment for user app
eg: Windows GUI & CLI
↓
Graphical User Interface Command line Interface

⇒ User space only interacts with Kernel

② Kernel : Heart of OS
→ interacts with hardware



Functions :

→ Process Management

→ Memory Management

→ Storage / file management

→ Process Creation
→ Termination & Communication
→ Scheduling etc.
→ Context Switching

→ allocation & deallocation
→ free space management

→ file creation
→ directory management

→ follows tree's structure

→ I/O Management

→ Input output (removable) device management

→ Spooling, Buffering, Caching

Types of Kernel

① Monolithic Kernel : Everything is centralised & all the functions are present in the kernel itself because of which it's faster.

Modes of CPU

a) User Mode

b) Kernel Mode

* A CPU switches context based on the task it's doing.

This switching is done using software interrupt.

eg: ① writing command in CLI ^ : user mode
→ pressed enter

② store / file management fn : Kernel Mode

③ feedback : User Mode

Advnt : fast communication as all fn are centralised.

disadvnt : Kernel Bulky

→ less reliable

eg: Linux & Unix, MS DOS

→ one fn breaks, all down.

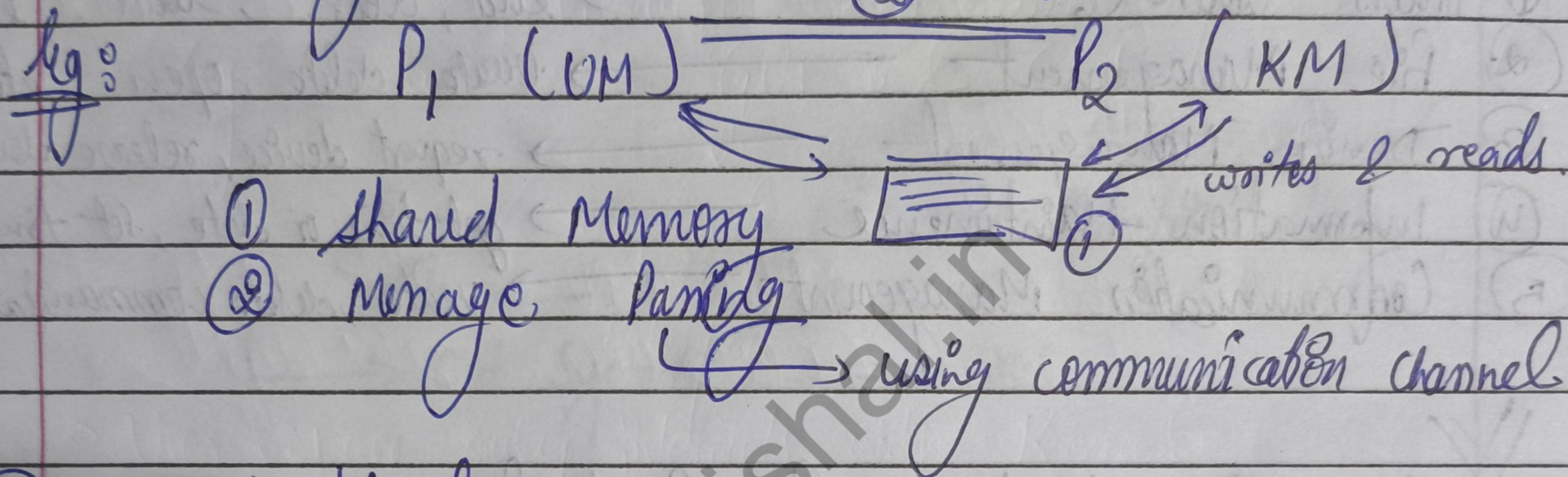
② Micro Kernel : only core functionalities are kept in kernel mode & others are sent to user mode.

Adv: less bulky, more reliable, stable

Disadv: less performance $\Rightarrow$ as more switching happens as more overhead.

eg: LY linux, Symbian OS etc. cost to switch happens.

Ques: How User Mode & Kernel Mode Communication $\uparrow$?
 $\rightarrow$ using IPC: inter process - communication



③ Hybrid Kernel

$\rightarrow$ combined approach

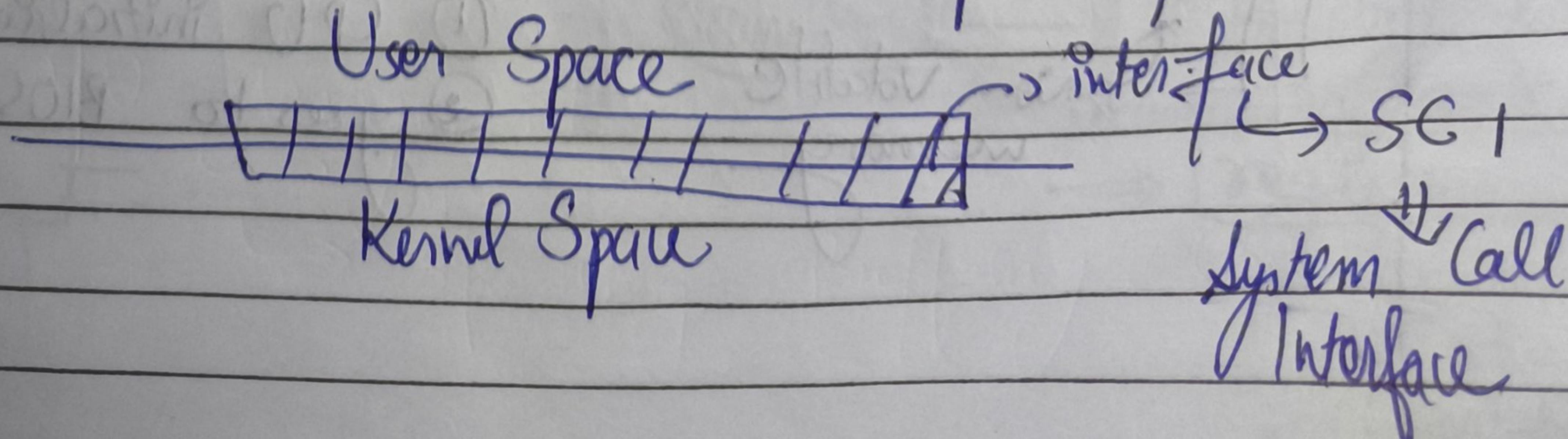
$\rightarrow$ runs critical fn in Kernel Space, for speed & non critical in user space.

eg: MacOS, Windows 7+ ~~Android~~

~~System Calls~~

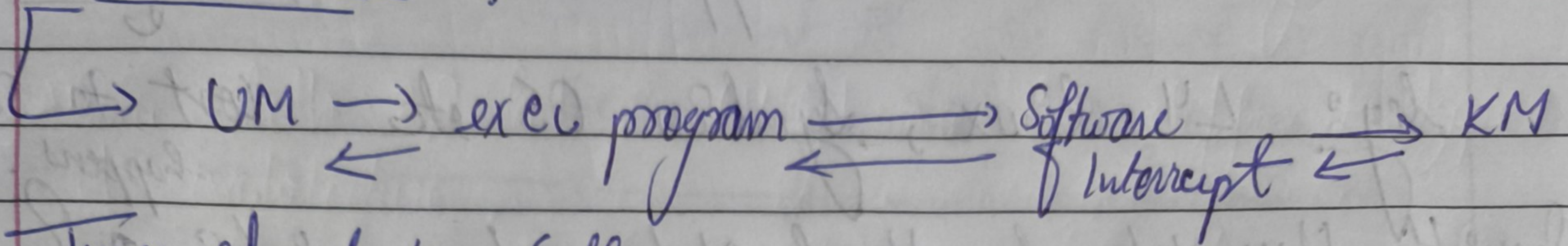
are the only way which

system calls provides a mechanism to applications running on User Mode to access Kernel space / Mode.



Actual Implementation of system calls are implemented using low level C language

UM to KM?



Types of system calls

- ① Process Control eg. end, abort, load etc
- ② File Management create, delete, open etc
- ③ Device Management request device, release device etc
- ④ Information Maintenance get time or date, set time etc
- ⑤ Communication Management create, delete communication etc

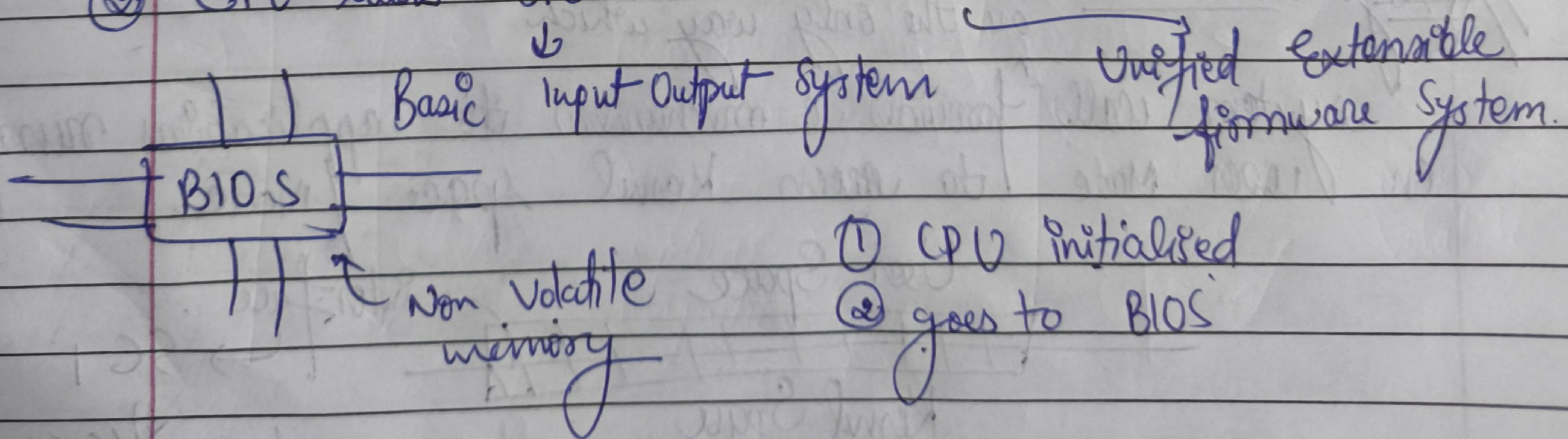
How OS boots up? Boot process

5 steps:

① Power On

On $\rightarrow$ power supply $\rightarrow$ ~~BIOS~~ motherboard

② CPU loads BIOS or UEFI



① CPU initialised

② goes to BIOS

③ BIOS or UEFI runs test & init. Hardware.
3.1) Loads some settings from memory area.

3.2) BIOS loads with settings.
& POST $\Rightarrow$ Power On Self Test.
Backed by CMOS battery

④ BIOS or UEFI; handoff to boot device / boot loader.
Actual OS initialization
disk (HDD or SSD), CD, USB

★ MBR : Master Boot loader.

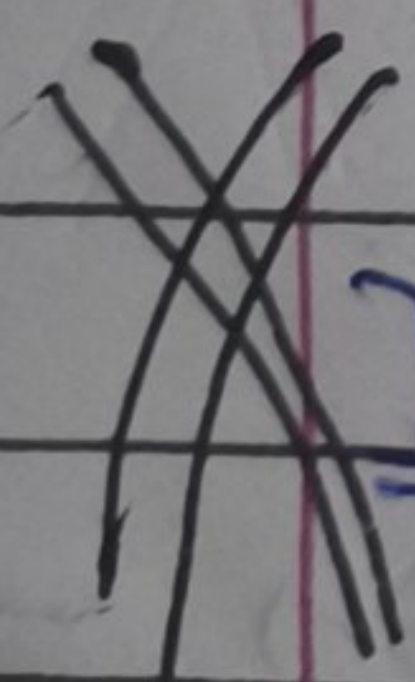
$\hookrightarrow$ Actual Area, where OS is present.
i.e. exact location
disk's $\rightarrow$ 0th index.
 $\rightarrow$ Old System $\rightarrow$ BIOS

★ EFI : Extensible Firmware Interface

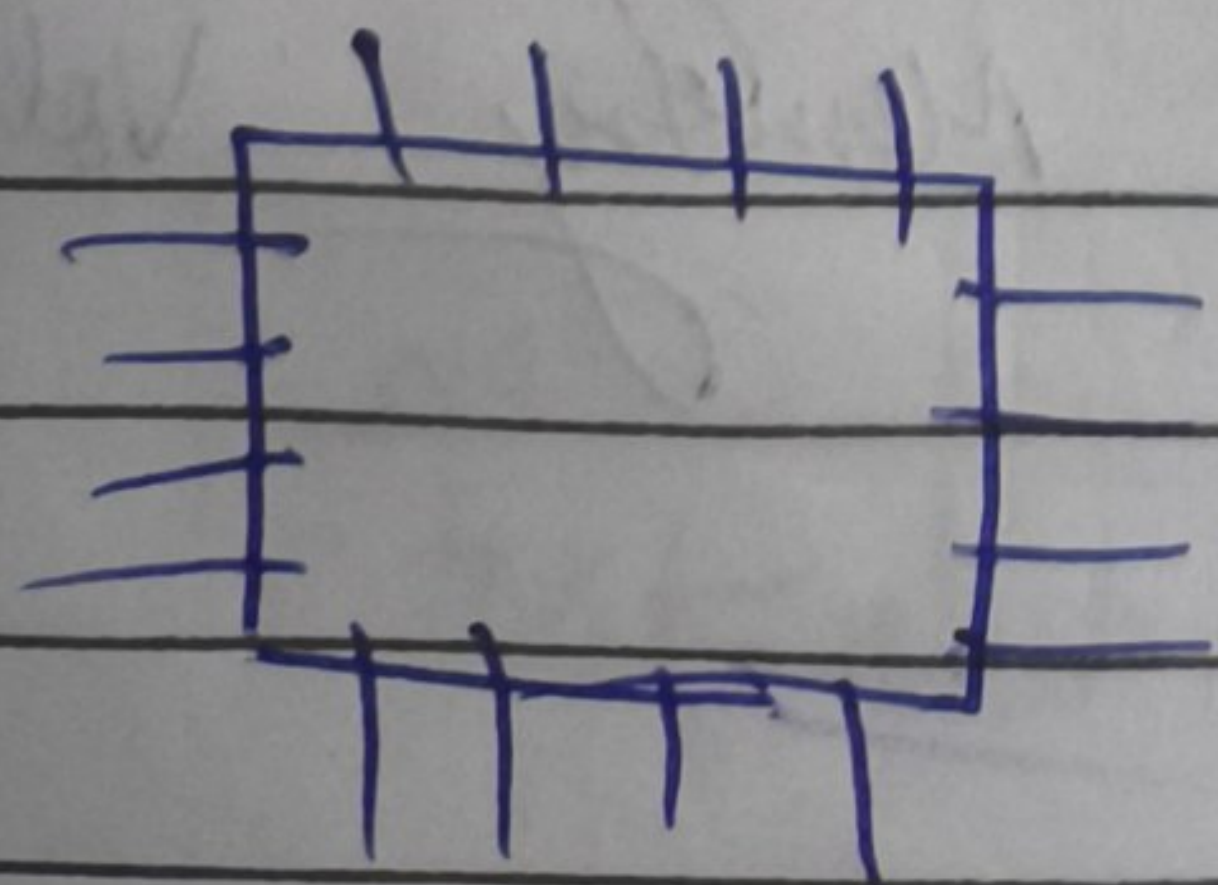
$\hookrightarrow$ Partition in disk system where boot loader is available

⑤ Bootloader loads the full OS

Bootloader : windows $\rightarrow$ bootmgr.exe
macOS $\rightarrow$ bootefi
Linux $\rightarrow$ GRUB



Diff b/w 32 bit & 64 bit OS



CPU

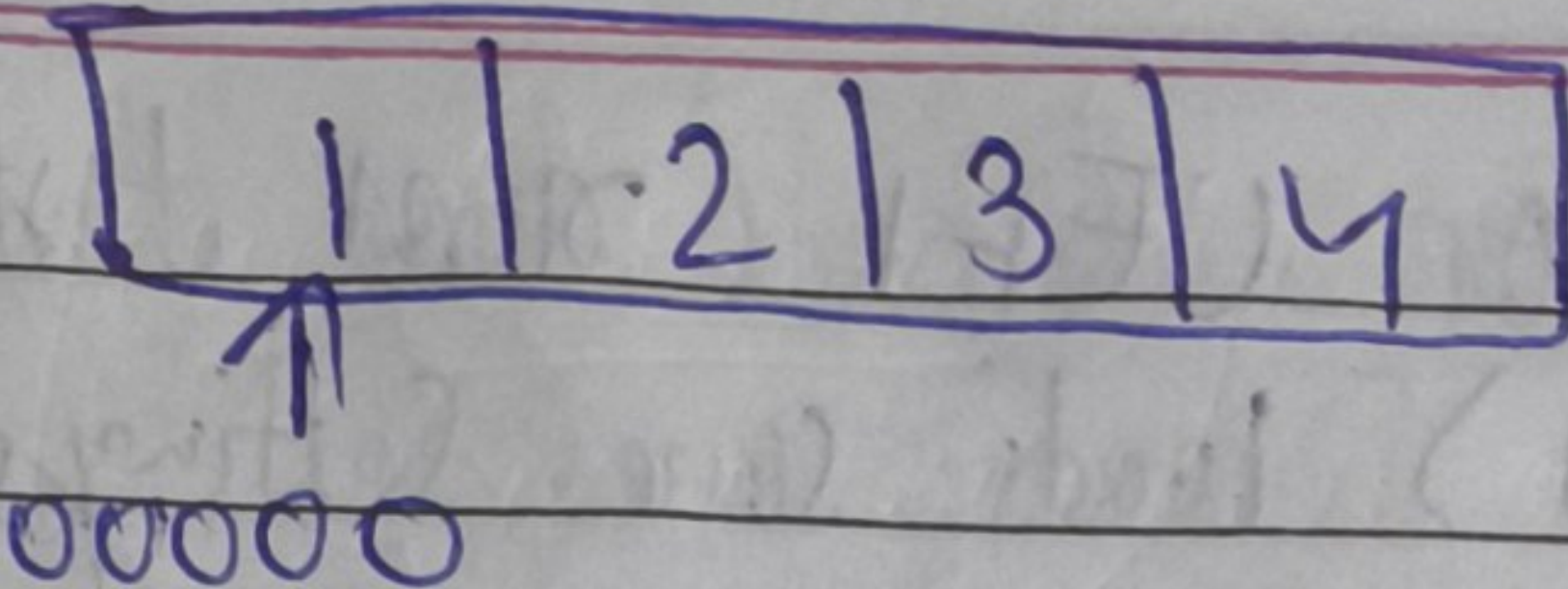
$\rightarrow$ billion's of Transistors

individual data circuit $\rightarrow$ NAND, OR gates etc.

$\rightarrow$ Implementing those circuits we create

$\Rightarrow$ Registers

where actual computation

A 4 bit register $\Rightarrow$ 

A 32 bit hardware needs a 32 bit system.

Hence both 32 bit & 64 bit are two different type of architecture.

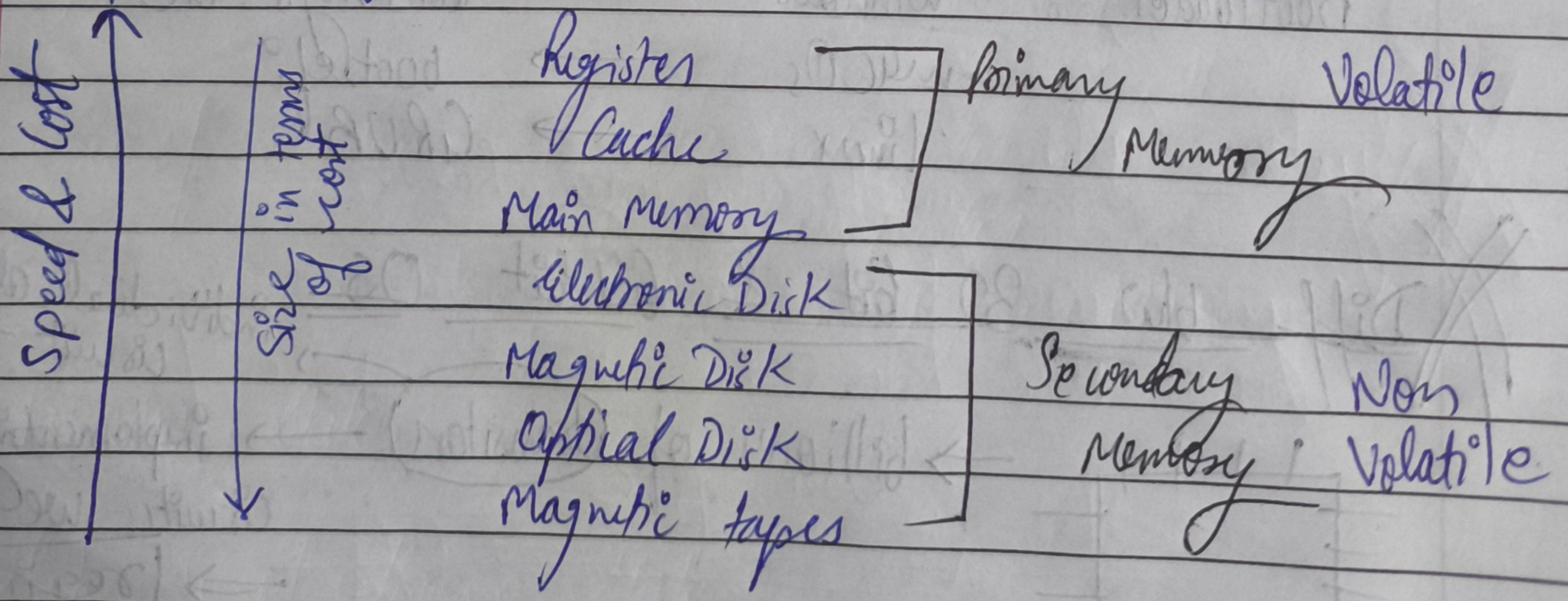
64 bit is now a Global Standard.

why 4GB RAM is limit of 32 bit architecture.

Advantages of 64 bit

- ① Addressable spaces : 32 bit CPU = 2^{32} diff locations
64 bit CPU = 2^{64} diff locations
- ② Better resource allocation
- ③ Performance high
- ④ Compatibility $\rightarrow$ backward Compatible
- ⑤ Better Graphic performance

~~Diff. storage used in Computers~~



Process Creation

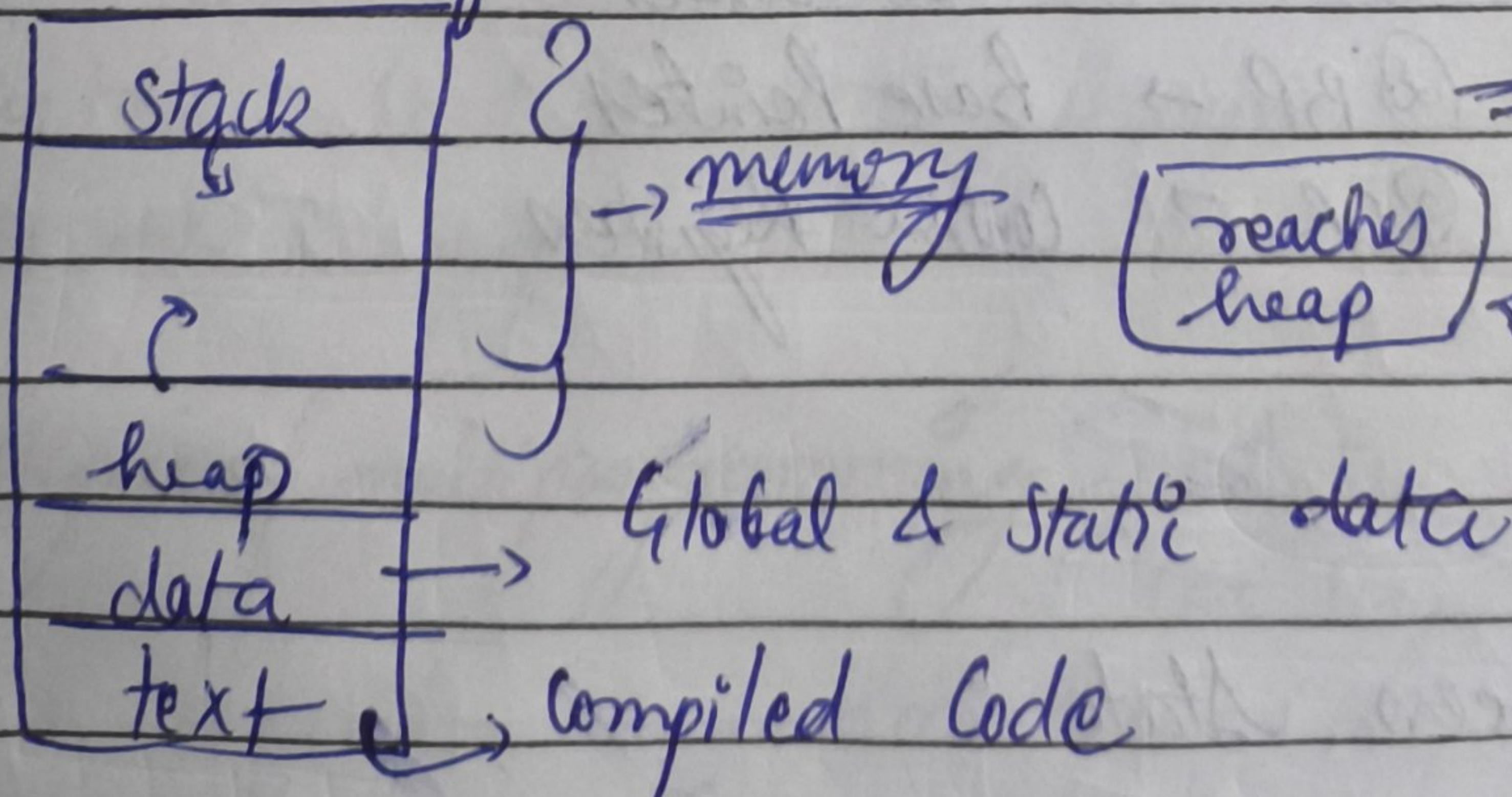
Program : ready to execute files. (stored in storage)
 eg: C++ compiler $\rightarrow$ Compiler $\rightarrow$.exe file.

Process : program under execution (stored in RAM)

How OS creates a process

- ① Load the program & static data in memory (RAM)
 $\rightarrow$ used for initialization
- ② Allocates run-time stack. eg: `char *name = "app";`
 $\rightarrow$ part of memory used for local variables, arguments & return values.
- ③ Allocates heap : for dynamic allocation.
- ④ I/O related tasks $\rightarrow$ Unix descriptors : I/P handler, O/P handler, error handler
- ⑤ OS handoff control to `main()`
 $\rightarrow$ If successful execution $\rightarrow$ `return 0;`
 $\rightarrow$ If for errors or failure $\rightarrow$ `exit(1)`

Architecture of Process



Stack overflow error?

$\Rightarrow$ If stack reaches its limit, OS detects error & sends overflow error.

out of memory or insufficient memory : — same reason as stack, but for heap.

Attribute of process

→ OS manages all processes by maintaining a process table. For each entry of process a process control block (PCB) is defined.

In simple words, OS identifies each process from its PCB only.

PCB design

- | | | |
|--------------------|--|----------------|
| ① Process ID | → unique identifier | upcoming ones. |
| ② Program Counter | → tracks the instruction it is executing & ↑ | |
| ③ Process State | → each process has its own state | |
| ④ Priority | → process priority | |
| ⑤ Registers | → saves the registers | |
| ⑥ open file list | → file descriptors | |
| ⑦ open device list | → device descriptors | |

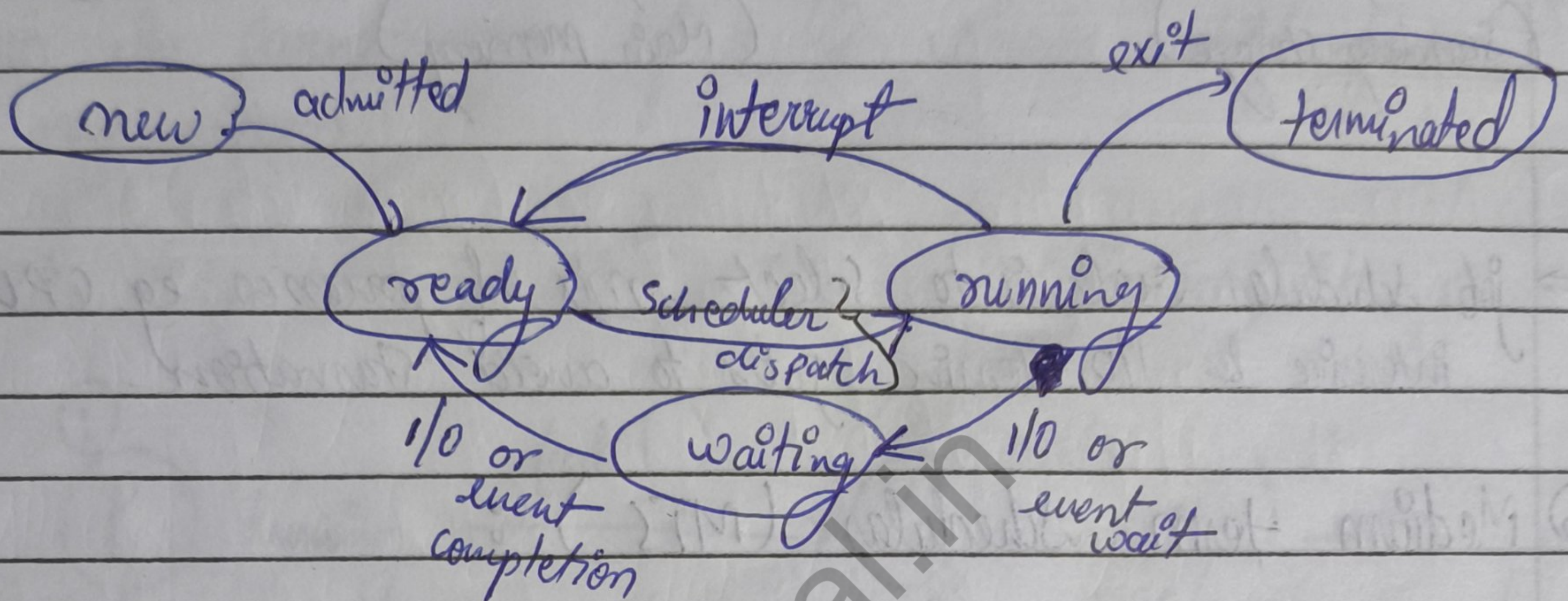
types of registers :

- ① SP → Stack Pointer
- ② BP → Base Pointer
- ③ CR → Control Register

~~Process state~~

→ lifecycle of process state :

- ① New : program getting converted to process
- ② Ready : process is in memory (Ready Queue)
- ③ Running : CPU allocated and getting executed
- ④ Waiting : Waiting for I/O completion
- ⑤ Terminated : process finished



Scheduler dispatch : sends the process in ready queue low freq.

Different types of Queue

long term scheduler (LTs)

or
Job scheduler

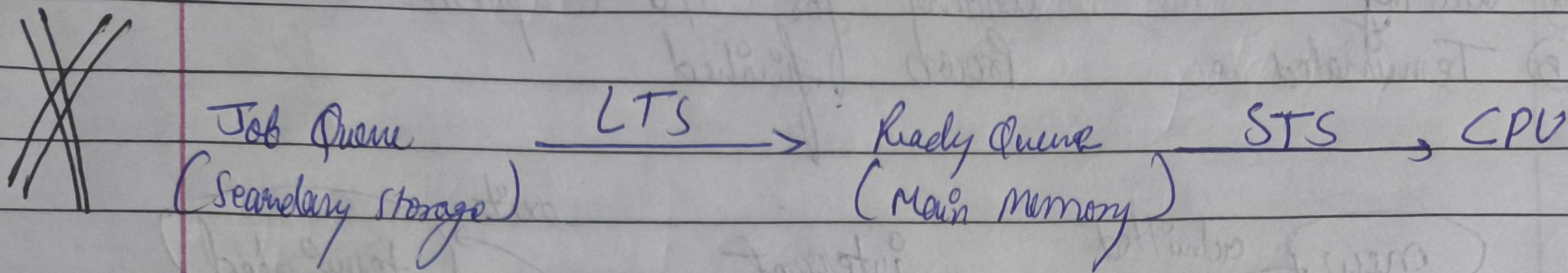
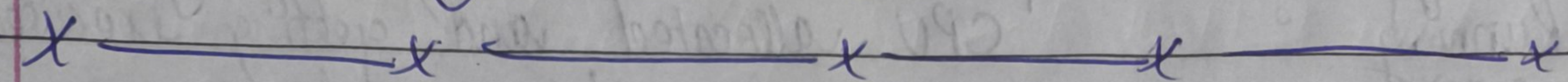
① Job Queue : stores new state process high freq. → sends → ready state (ready queue)

② Ready Queue : CPU scheduler dispatches the ready states to running state. also called short term scheduler (STC)

* Long term & Short term naming is used because of freq. at which they work & respond.

Degree of multiprogramming at one instance how many processes can stay in ready queue. → maintained & handled by job scheduler.

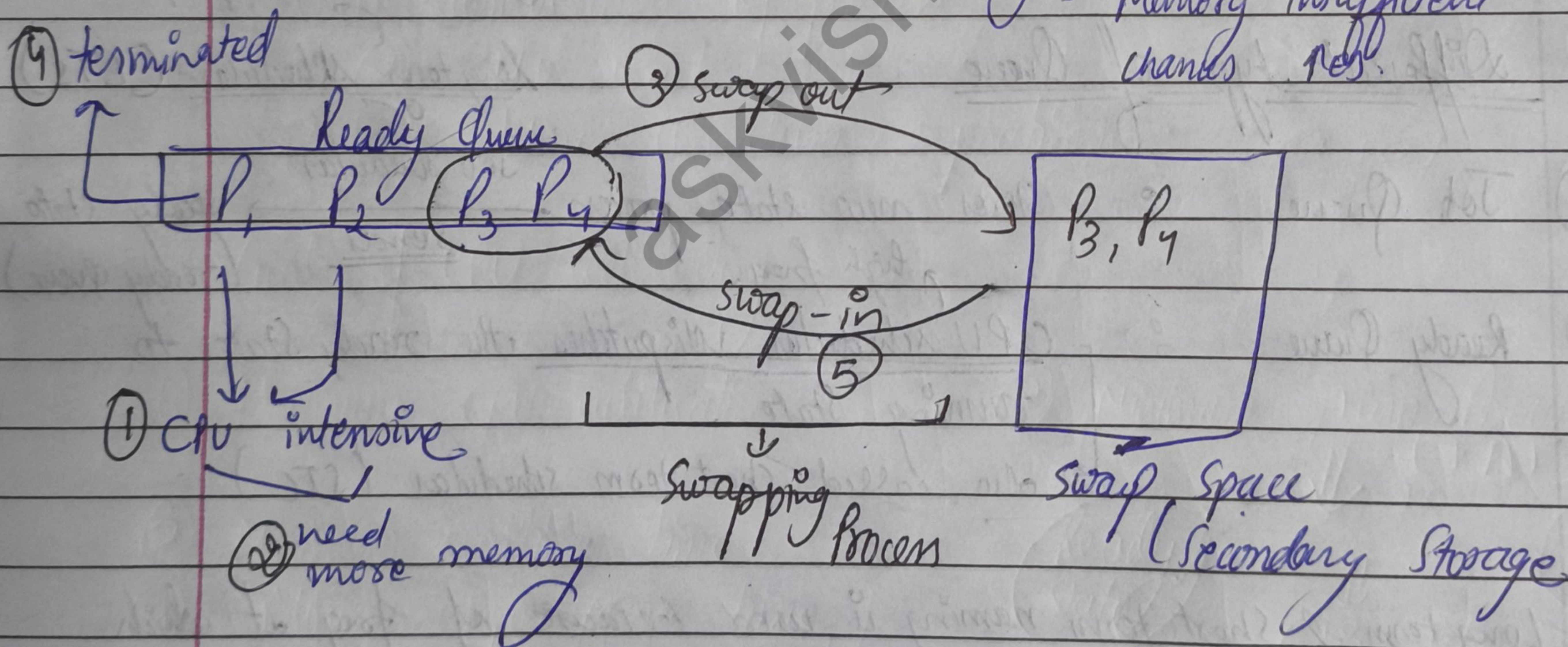
Context Switching / Swapping



⇒ job scheduler role is to select mix of processes eg CPU intensive & I/O intensive jobs to avoid starvation

③ Medium term Scheduler (MTS) :

When degree of multiprogramming is high. Memory insufficient chances are high.



This swap in & swap out is done by MTS.

Context Switching

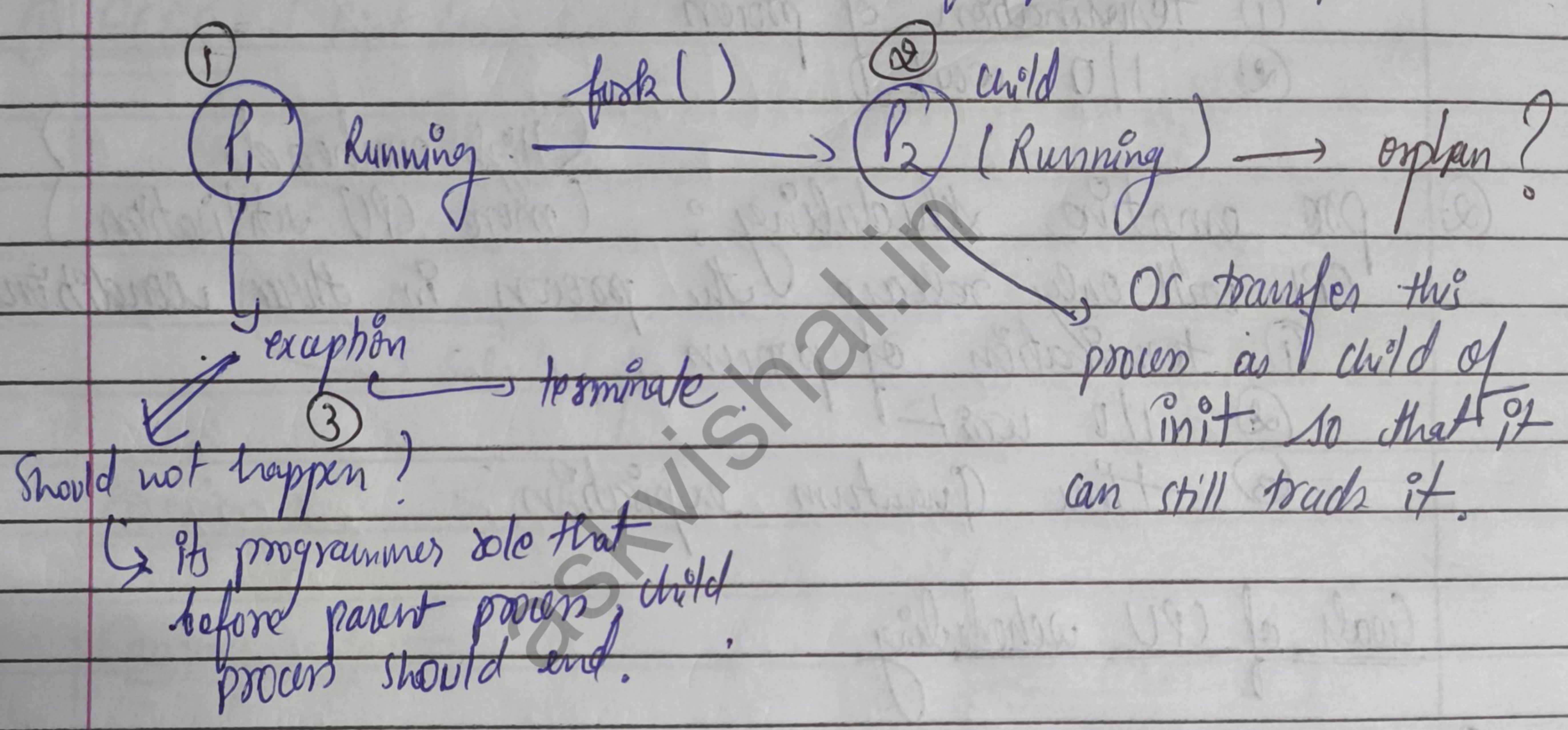
⇒ Saves all the context of process in PCB's registers as state and whenever gets it from PCB's registers when re-executing. All this is done by kernel

context switching is pure overhead as no usefull work is done.

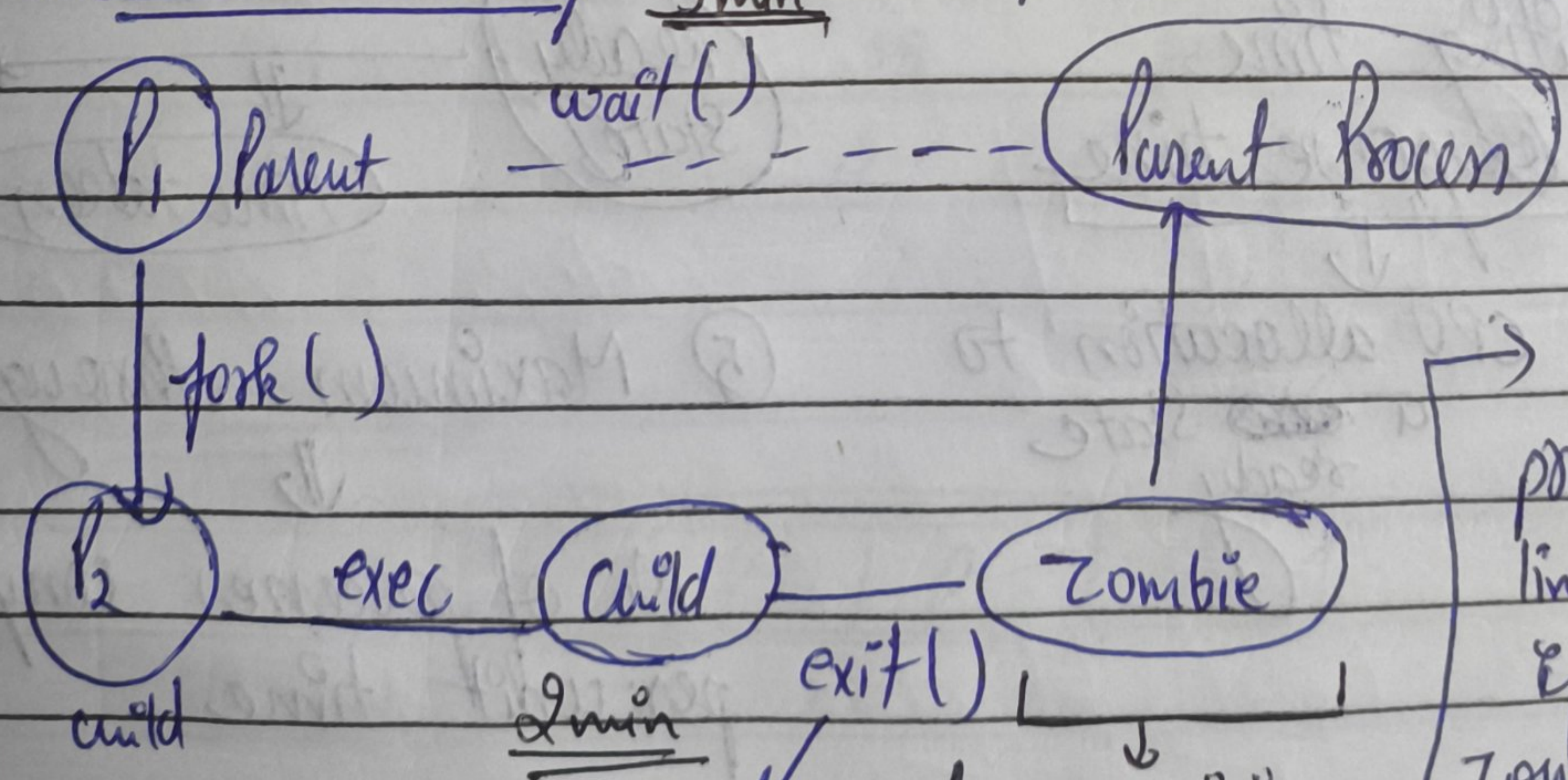
ORPHAN process

* all processes in OS is created from `fork()` command, which gets the parent process `fork` & creates child `fork()` i.e new process.

⇒ In linux : `init` is the first process, its `processId = 1`



④ Zombie process / Defunct process



as `exit` releases all the resources from OS but process entry is kept.

* Because of this process table entry limit will be exhausted if we have too many zombie process.

Process Scheduling Algorithms

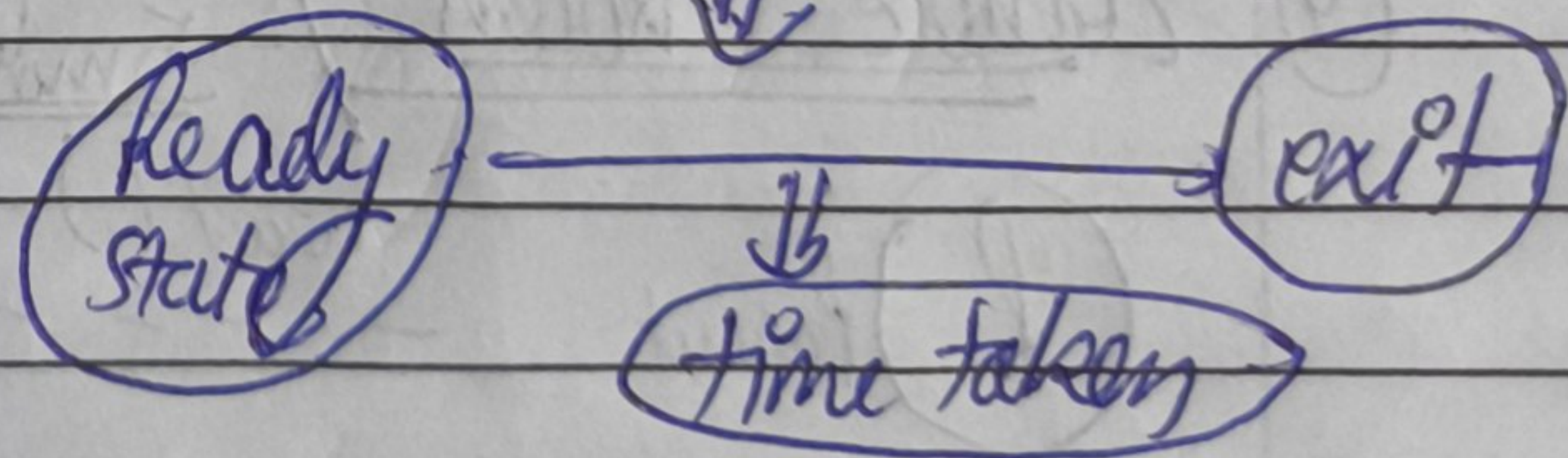
⇒ selection of processes by STS so that it could be sent to CPU.
 ↳ also known as dispatcher.

Types :

- ① Non-pre-emptive scheduling : (more starvation)
 CPU will only release the process in two condition :
 - ① termination of process
 - ② I/O wait
- ② pre-emptive scheduling : (High overhead, more CPU utilisation)
 CPU will only release the process in three conditions:
 - ① termination of process
 - ② I/O wait
 - ③ Time Quantum expiration

Goals of CPU scheduling

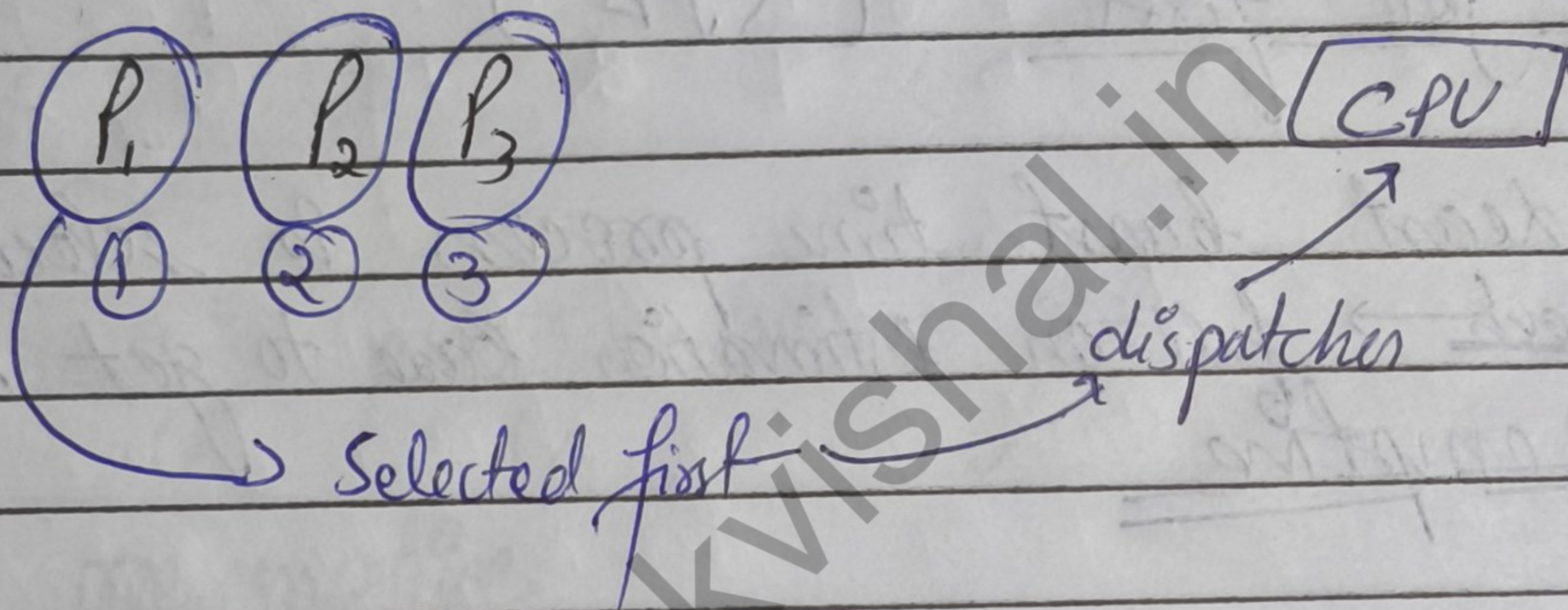
- ① Maximum CPU utilisation
- ② Minimum Turn Around Time (TAT)
- ③ Minimum waiting time
- ④ Minimum response time
 ↓
 CPU allocation to a ready state
- ⑤ Maximum throughput
 ↓
 No. of processes completed per unit time



- a) Arrival time (AT): Time when process arrived at the ready state.
 b) Burst time (BT): Time required by process for its execution.
 c) Completion time (CT): Time taken till process get terminated.
 Also, $TAT \Rightarrow CT - AT$
 d) Wait time (WT): Time process spends waiting for CPU.
 $WT = TAT - BT$

Non-pre-emptive algorithms:

① FCFS (First Come First Serve)

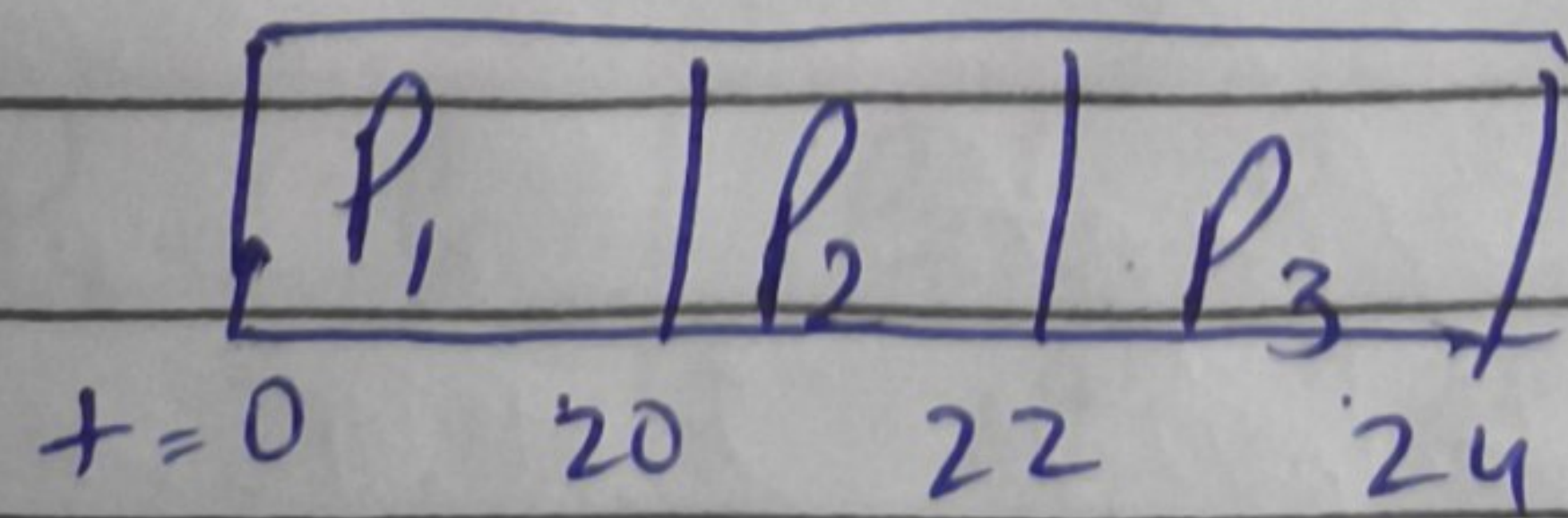


Problem:

Convoy effect

	Proc	AT	BT	CT	$(CT - AT)$ TAT	$(TAT - BT)$ WT
⇒	1	0	20	20	20	0
	2	1	2	22	21	19
	3	2	2	24	22	20
					<u>39</u>	13
					Avg. wt	3

Gantt Chart



II	P _{no}	AT	BT	CT	TAT	WT
	P ₂	0	2	2	2	0
	P ₃	1	2	4	3	1
⇒	P ₁	2	20	24	22	2
wt Avg. $\frac{3}{3} = 1 \text{ unit}$						

P₂ | P₃ | P₁

Convey effect : when a heavy job gets first priority
WT dep. significantly.

~~③ Shortest job first (SJF)~~

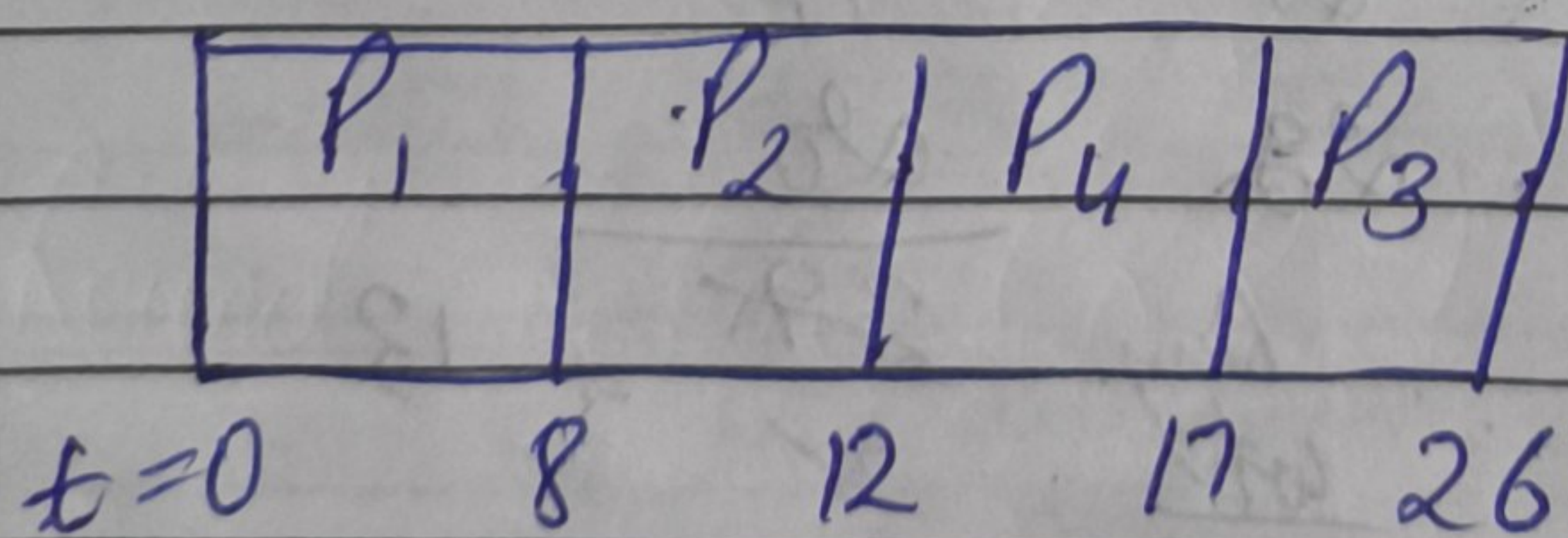
→ least burst time process is selected.

Drawback → full on estimation base to get BT.

a) Non-preemptive

Seq ^o	P	AT	BT	CT	TAT	WT
1	P ₁	0	8	8	8	0
2	P ₂	1	4	12	11	7
3	P ₃	2	9	26	24	15
4	P ₄	3	5	17	14	9

Avg: $\frac{7.75}{1}$

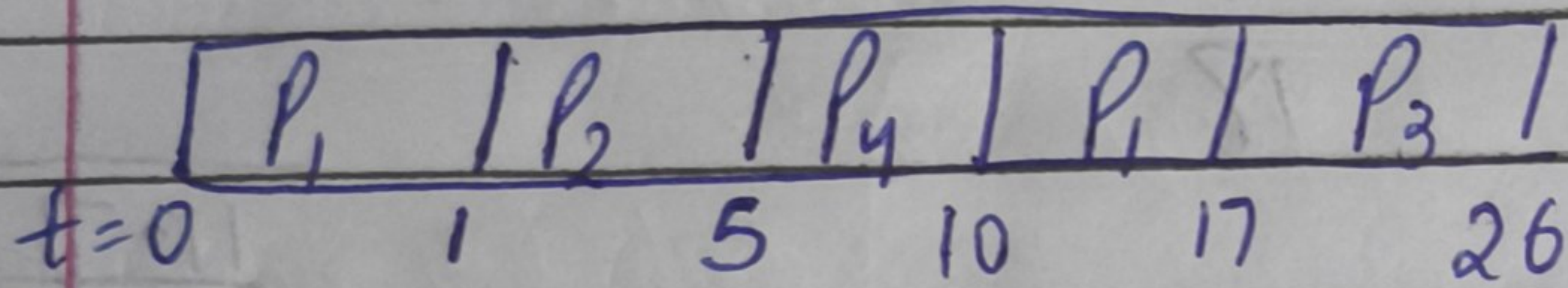


Criteria : AT+BT

Preemptive SJF

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

Avg wt = 6.5



No convoy effect

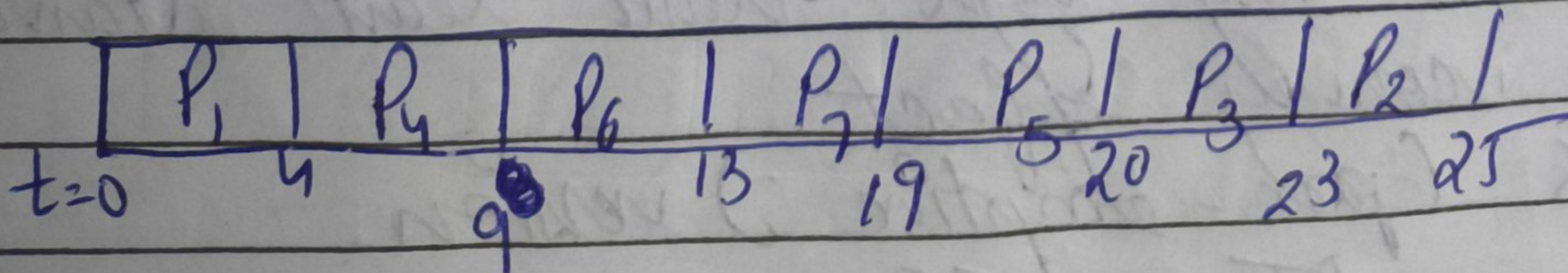
③ Priority Scheduling

→ Assign priority to each job

① Non-preemptive

P	Priority	AT	BT	CT	TAT	WT
1	2	0	4	4	4	0
2	4	1	2	25	24	22
3	6	2	3	23	22	19
4	10	3	5	9	8	1
5	8	4	1	20	16	15
6	12	5	4	13	8	4
7	9	6	6	19	13	7

Avg wt = 9.7



Preemptive Priority Scheduling

Pno	P [*]	AT	B.T	CT	TAT	WT
1	2	0	43	25		
2	4	1	21	22		
3	6	2	32	21		
4	10	3	52	12		
5	8	4	1	19		
✓ 6	12	5	4	9		
7	9	6	9	18		

11.7

	P ₁	P ₂	P ₃	P ₄	P ₆	P ₄	P ₇	P ₅	P ₃	P ₂	P ₁
t=0	1	2	3	5	9	18	18	19	21	1	3

Assumption : 1 sec time Quantum

⇒ Convey effect starts coming if over & over only high priority tasks comes, low priority keeps waiting.

Biggest drawback : Preemptive & Non-preemptive
* indefinite waiting or extreme starvation for lowest jobs.

Solution : Ageing : Gradually inc priority of lowest jobs.

④ Round Robin (RR)

→ Response time is very high. Hence least starvation & no convey effect.

→ It's FCFS (pre-emptive) version.

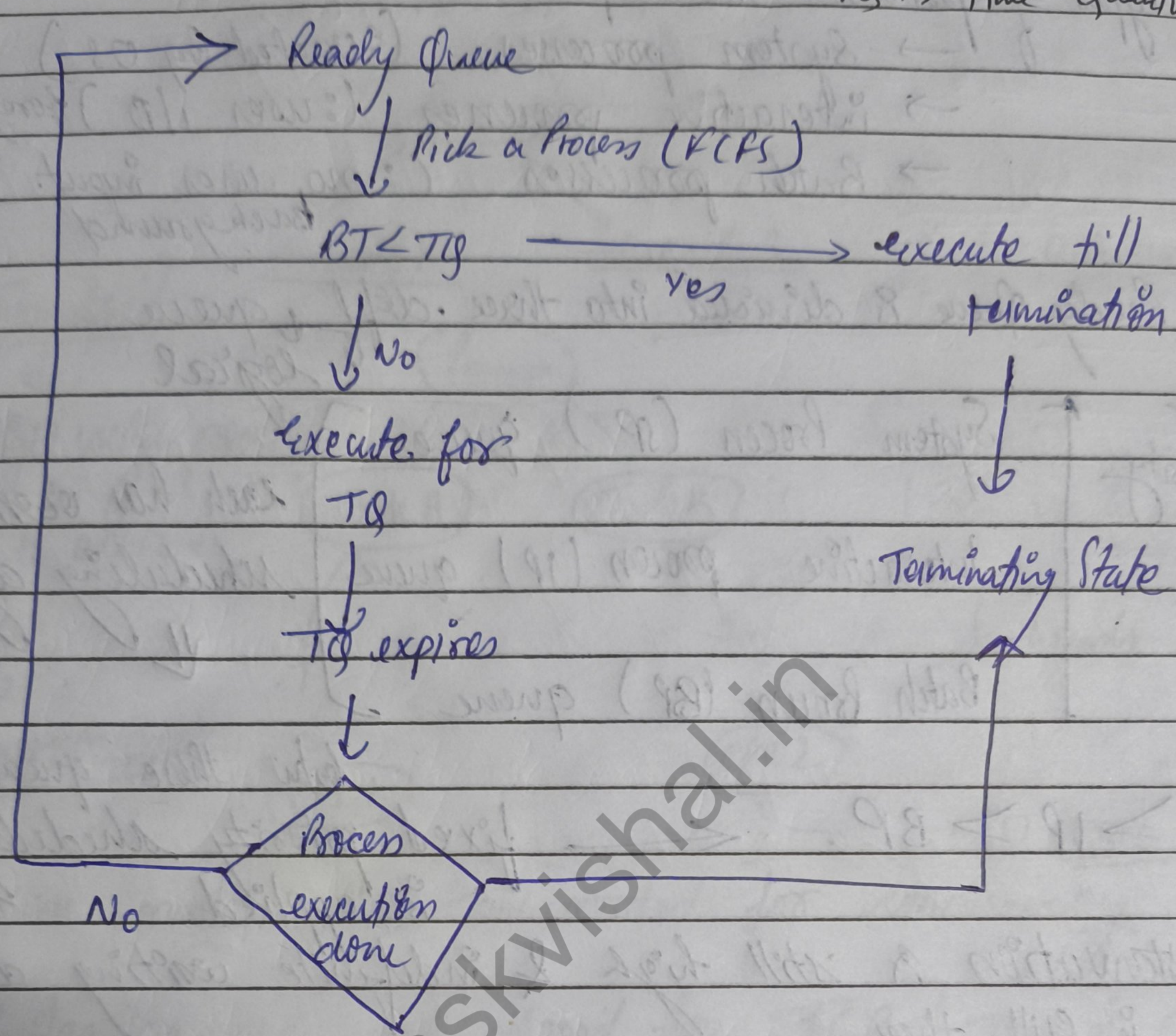
→ Criteria : AT + TQ

→ design : time sharing

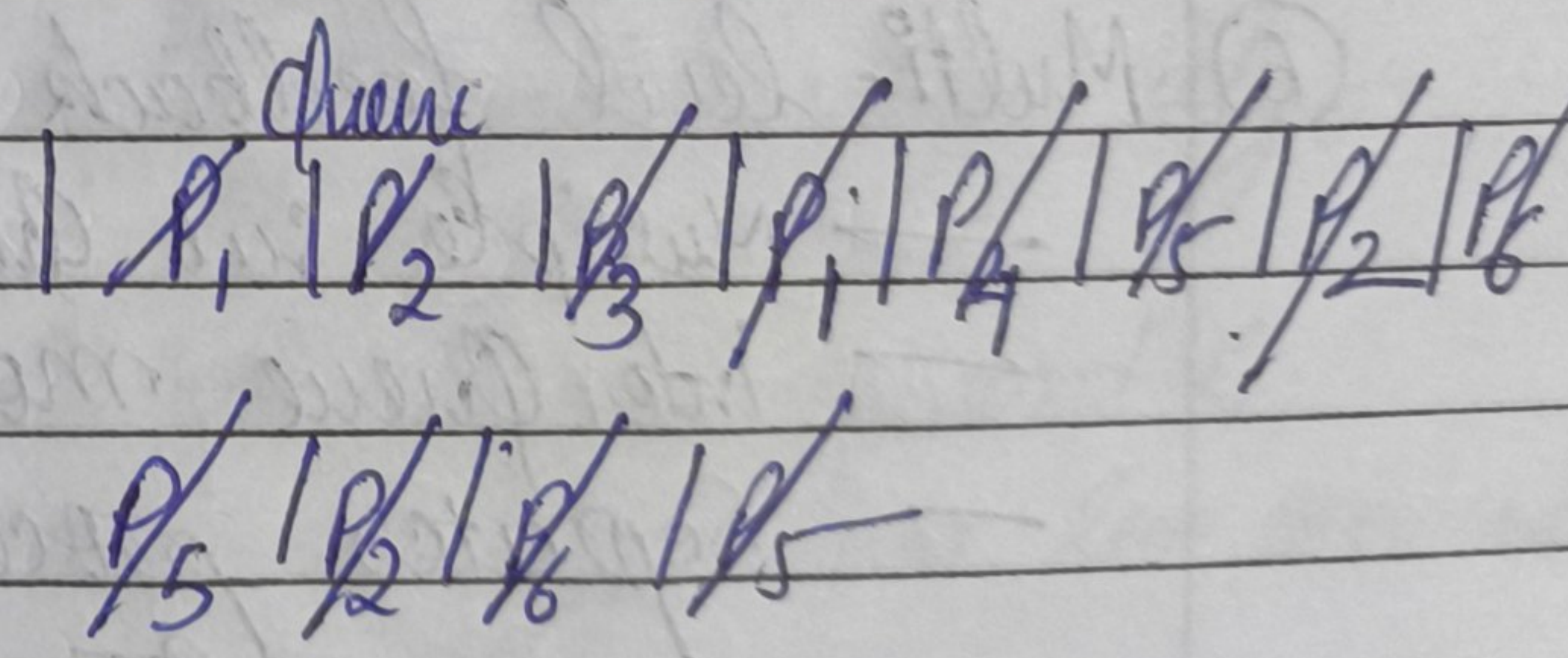
→ easy to implement

hence popular for multitasking OS.

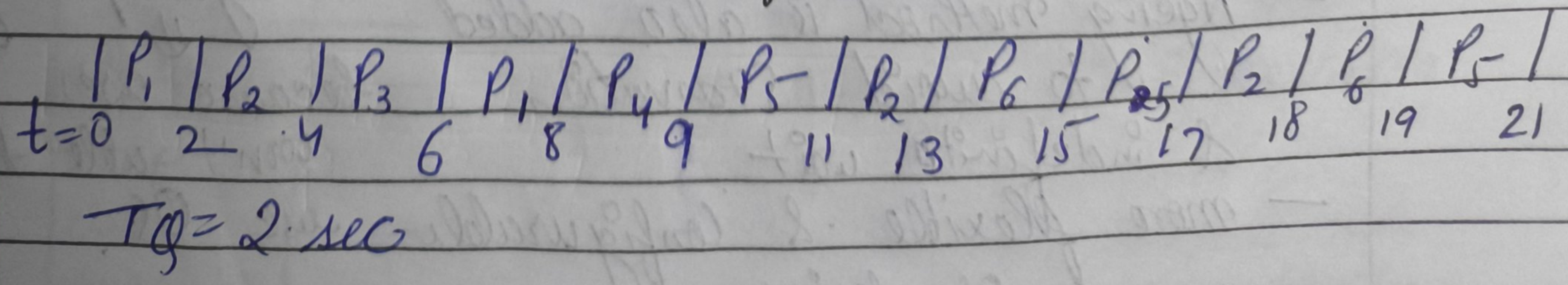
TQ $\Rightarrow$ Time Quantum



P	AT	BT
1	0	12
2	1	8
3	2	20
4	3	
5	4	6
6	6	3



$\Rightarrow$ even more overhead.
if TQ ~~yes~~ yes, overhead $\uparrow$

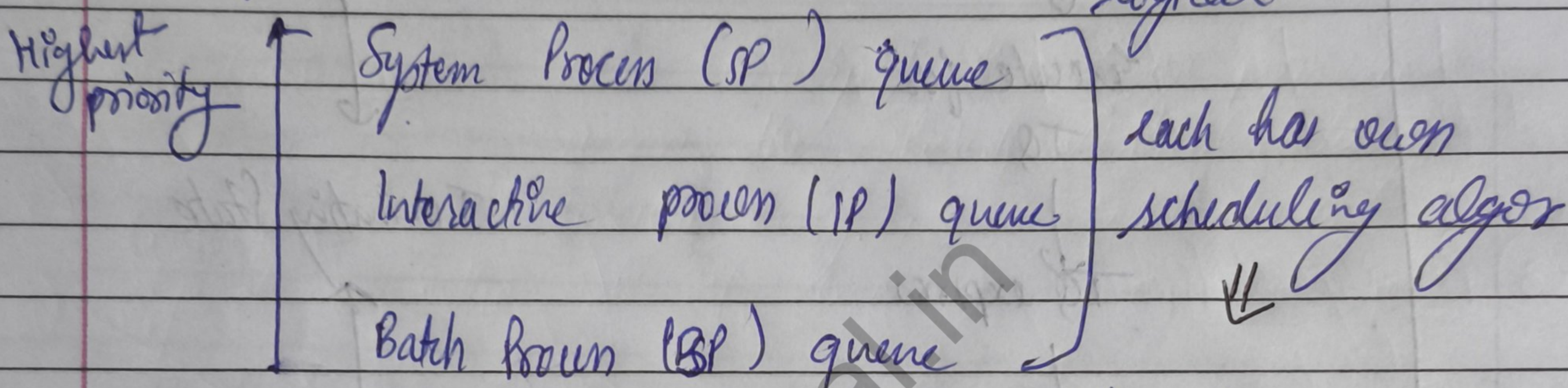


⑤ Multi-level queue scheduling (MLQ)

Types of processes in computer :

- System processes (created by OS)
- Interactive processes (uses I/O) foreground
- Batch processes (no user input) background

⇒ Ready Queue is divided into three diff. queue logical



$SP > IP > BP$ ← b/w these queue's fixed priority scheduling is applied

~ starvation is still high & indefinite waiting also is still there.

⑥ Multi-level feedback Queue Scheduler

- Multiple sub Queue (more queues can also be added)
- Inter Queue movement is allowed.
- Separates processes based on BT
 - BT ↑ → lower queue
 - I/O bound & Interactive process

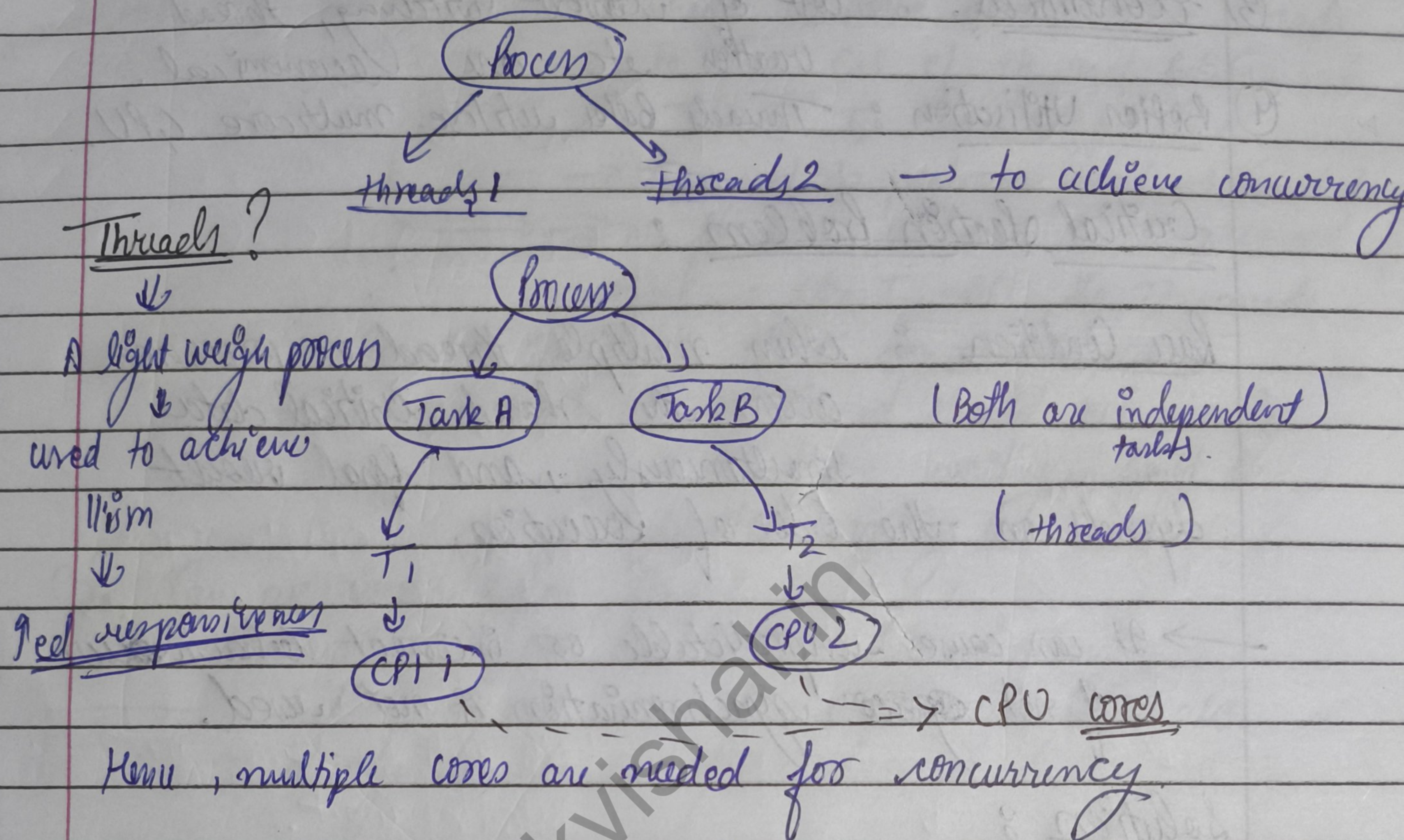
— Ageing method is also added to avoid starvation & indefinite wait

— more flexible & configurable OS design.

higher priority because user won't able to use OS.

Concurrency

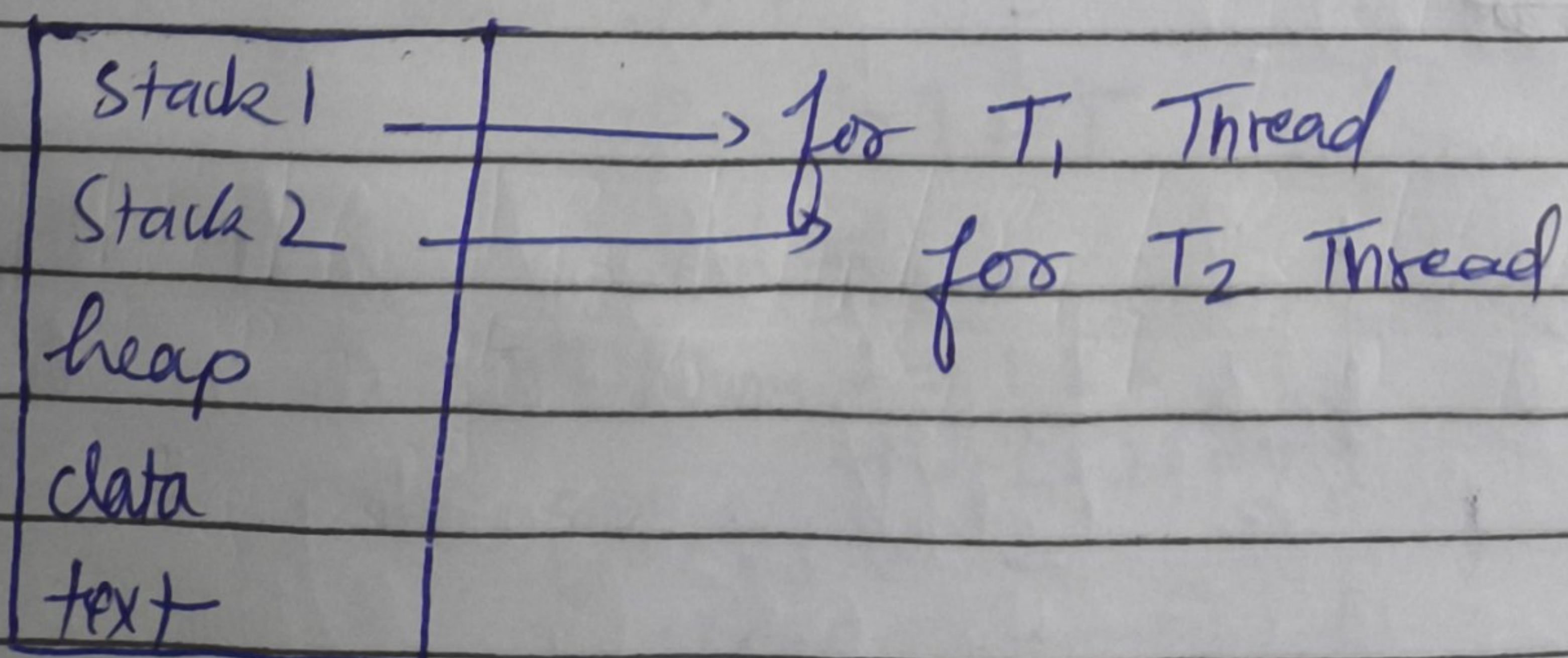
→ Multiple instructions at the same time



Similar as PCB, each thread has TCB (Thread Control Block)

- each thread has its own program counter
- and using TCB & Thread scheduling algo's each thread executes its task.
- Threads have Context Switching and is very fast as no memory switching hence less overhead

Thread Memory Organisation



⇒ Benefits of multi-threading:

① Responsiveness

↓
each thread can execute some task in process hence no blocking of tasks

- ② Resource sharing : share same memory so communication is easy, else we need to use IPC.
- ③ Economical : cost of context switching, thread creation etc are economical.
- ④ Better Utilisation : Threads better utilise multicore GPU.

Critical Section Problem :

Race Condition : when multiple thread or processes access the shared critical data simultaneously, and final result depends on the order of execution.

→ It can cause unpredictable or incorrect behaviour if not ~~proper~~ synchronisation is not used.

Solution :

- ① Atomic operation : the shared resource should be updated in one CPU cycle only.
- ② Mutual exclusion : Only one thread can ^{perform} write operation at a time.
in P. language, locks are provided to achieve the behaviour.

Q: Can we use single flag?

⇒ Solⁿ of critical section should have 3 conditions.

- mandatory
- ① Mutual exclusion
 - ② Progress : should not have blocked other threads to use C.S if it's not being used.
 T_1 & T_2 were using C.S.
 → T_3 need to perform read → should be allowed.
 → No fix order should be there → like T_1 visit, the T_2 needs to visit C.S even if it doesn't need.
 - ③ Bounded wait : A thread should not be in indefinite waiting. should be limited waiting.

So, Can we use single flag?

↳ Mutual exclusion could be achieved easily but progress won't be a flag would always create a fix order.

* Petersen Solⁿ

⇒ Improvement over single flag,

→ flag [2] (array) ⇒ enter C.S, flag[i] = true implies i is ready to enter C.S.
 → turn
 ↳ indicates whose turn is to enter the C.S

Solution :

T_1
 while (1) {
 flag[0] = T
 turn = 1

while (turn == 1 && flag[1] == T)
 {
 C.S
 flag[0] = F
 }

T_2
 while (1) {
 flag[1] = T
 turn = 0
 while (turn == 0 && flag[0] == T)
 {
 C.S
 }
 flag[1] = F
 }

Both available : ME ✓, Progress ✓
 But, Peterson solⁿ is only for two threads,
 more than 2, it fails.

Solution of C.S

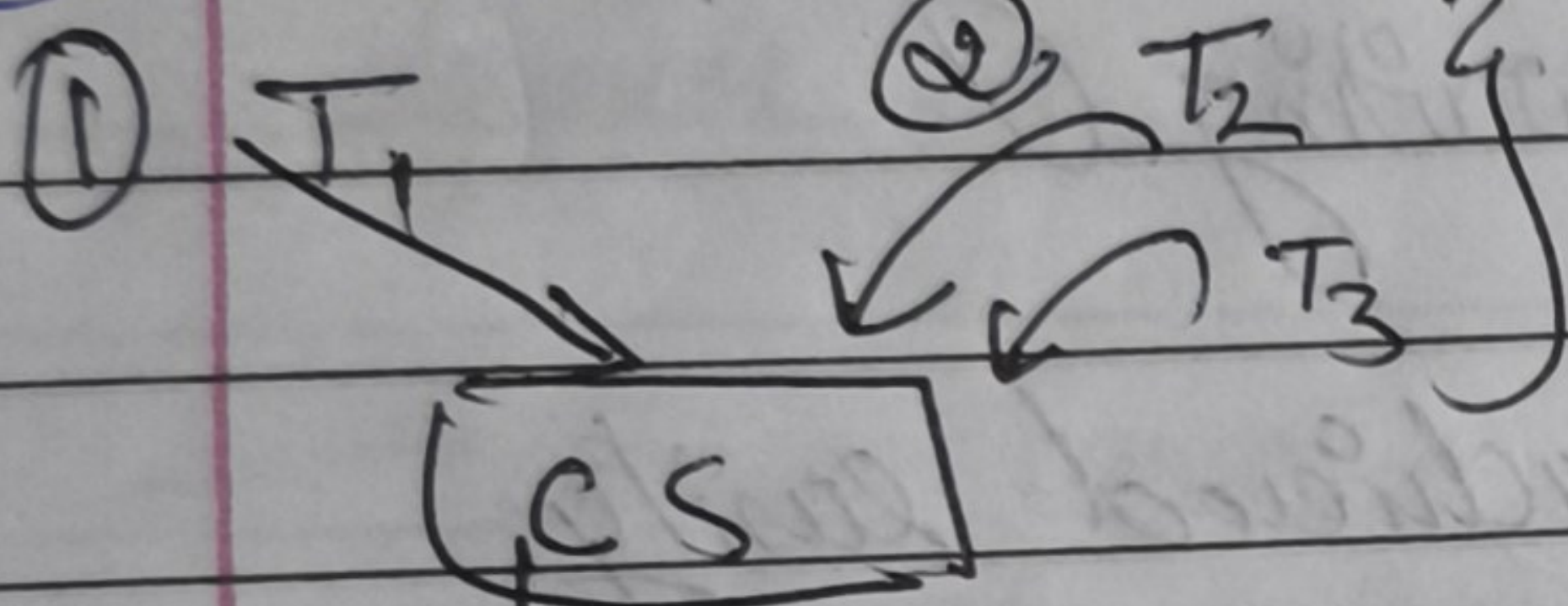
(3) Locks /
 Mutex

Disadvantages
of using locks

Locks } C.S

↓
 unlocks

(i) Contention :



busy wait

consuming CPU's cycle
 but not doing anything.

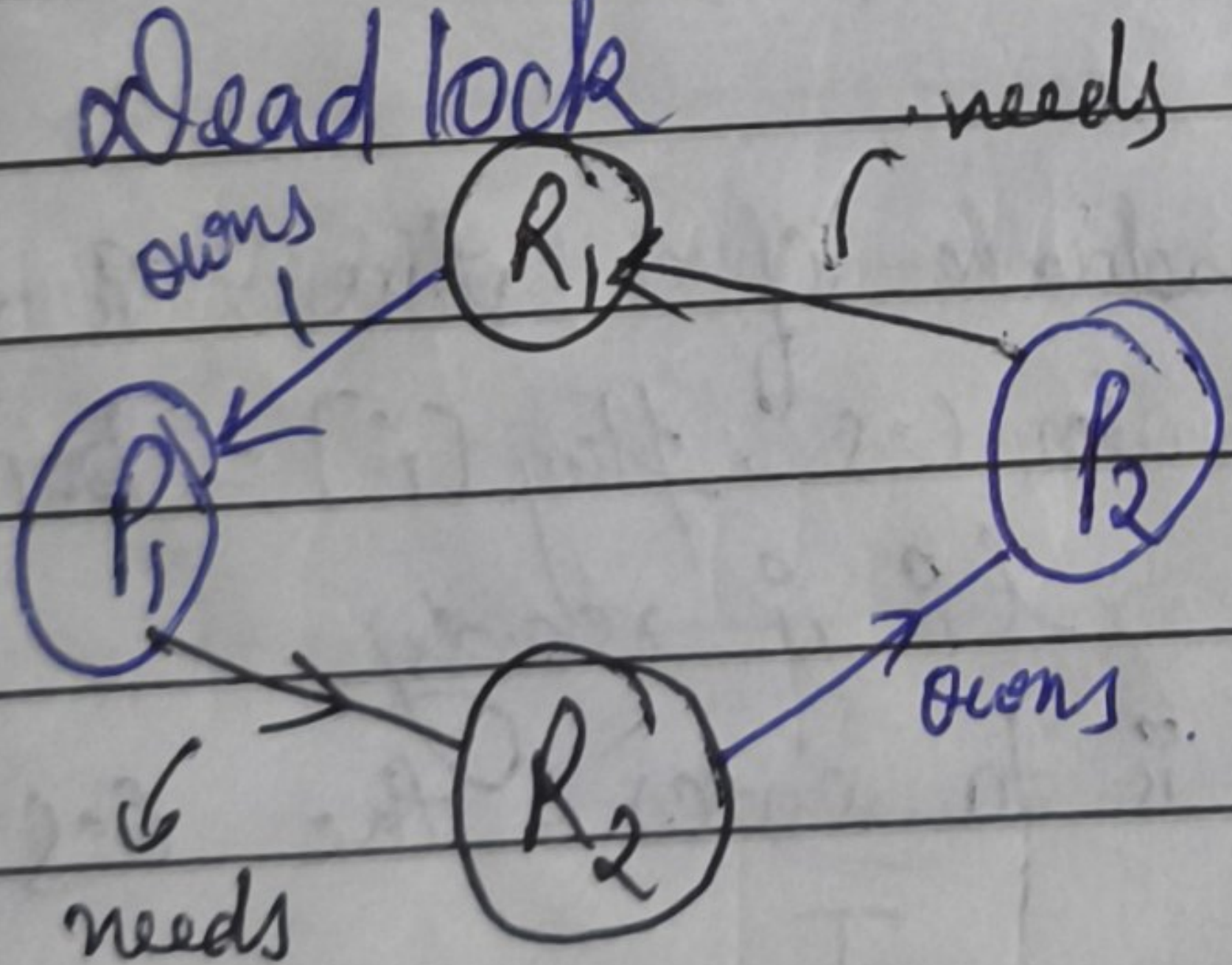
↓ after T₁ dies
 both will be stuck in
 infinite wait.

(2) died without
 unlocking

Solⁿ
 in case of exceptions always unlock C.S

(ii)

Dead lock



infinite lock.

(iii)

Debugging is hard.

(iv)

Starvation

: if low priority thread locks the C.S
 & high priority thread comes but
 is stuck in waiting to
 unlock the C.S by low priority
 thread.

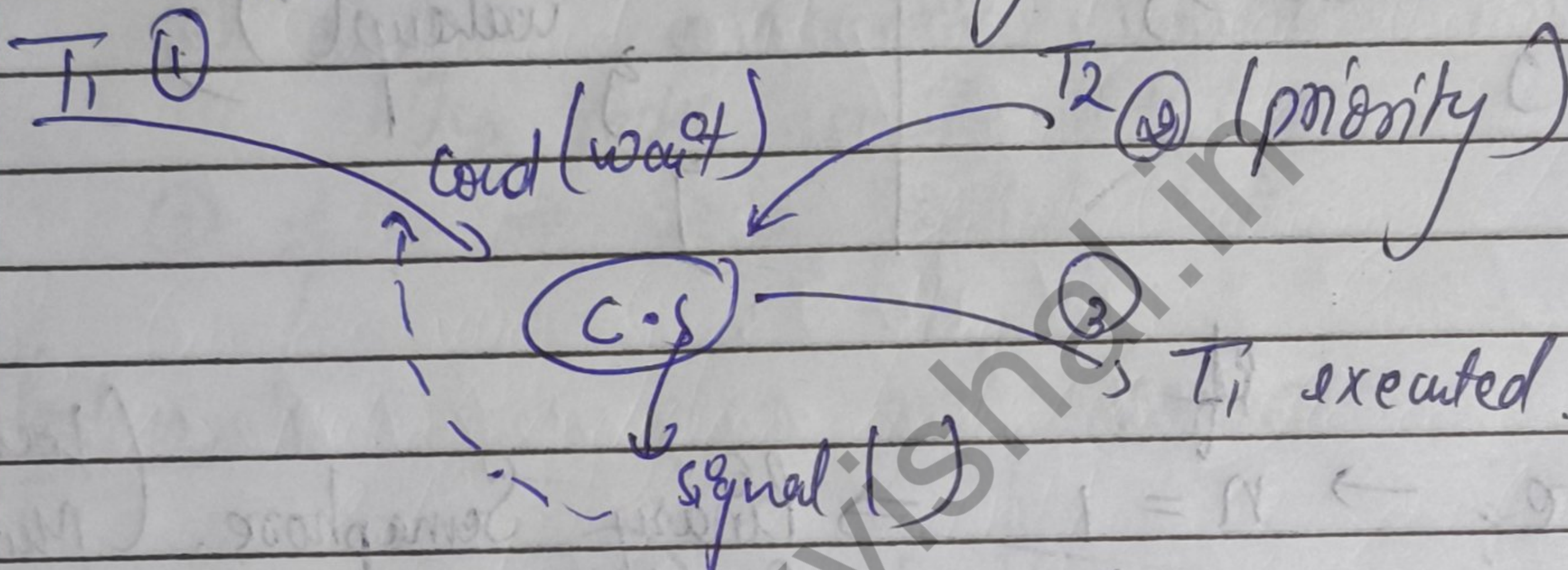
④ ~~Conditional variables~~ $\Rightarrow$ uses locks only but solves contention problem.

Var = cond. (lock)

two states:

① Waiting: CPU executes some other task if Var is in waiting state (CPU cycles are not wasted)
 (wait is like block (No CPU cycle) CPU

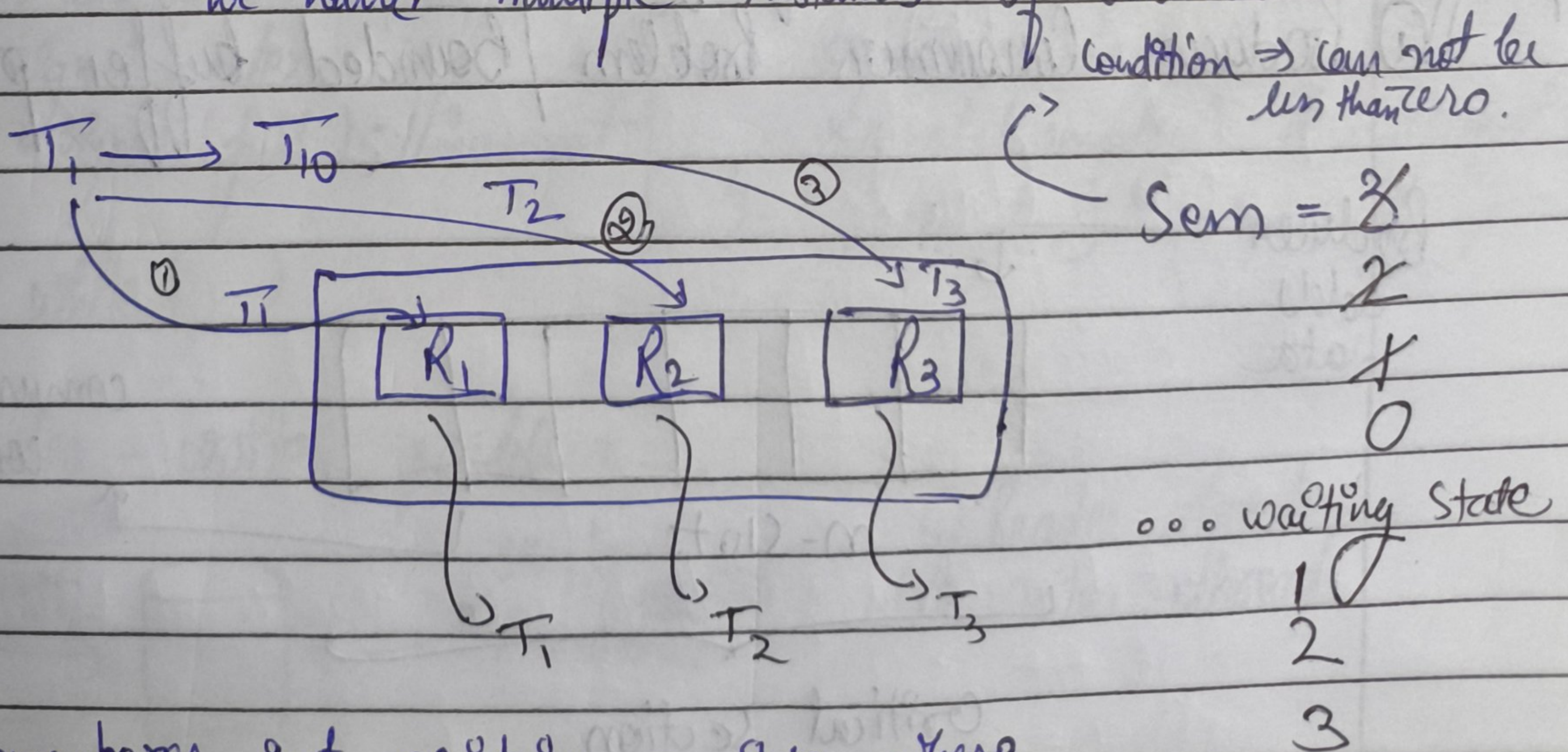
② Signal: After a interrupt, gets signal & start executing the variable



⑤ Semaphores

$\hookrightarrow$ It's an integer value (res when thread exits, dec when thread enters)
 $\hookrightarrow$ It's only applicable when we have multiple instances of resources.

eg:



* Semaphores just maintains a state where no. of thread enters in resource should not be greater as collision could occur.

- It's not necessary that the value of semaphores first reach 3 (In this eg.) then only new thread can enter, anyone can enter or exit as long as value is not less than 0.

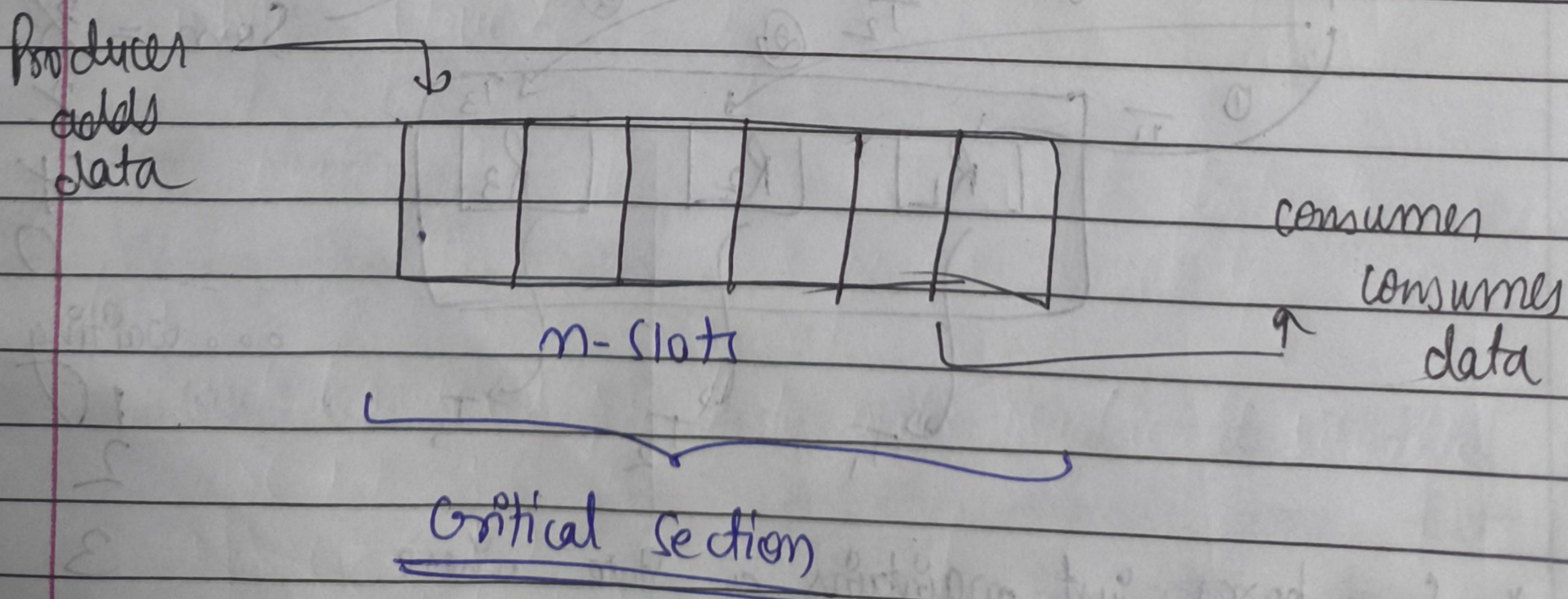
```
wait(S) {
    S → value --;
    if (S → value < 0) {
        add to S → blocklist
        block();
    }
}
```

```
signal(S) {
    S → value ++;
    if (S → value <= 0) {
        remove T from S → blocklist
        wakeup();
    }
}
```

Semaphore $\rightarrow n = 1 \Rightarrow$ Binary Semaphore (Locks/Mutex only)
 $\rightarrow n > 1 \Rightarrow$ Counting Semaphore

Classic Problems of Synchronization:

① Producer Consumer Problem / Bounded buffer problem.



Problem :

- ① sync between producer & consumer
- ② Producer must not add if buffer is already full
- ③ consumer must not pick / remove data when buffer is empty.

Soln : Semaphores

- ① m, mutex (binary Semaphore)
↳ used to acquire lock on buffer
- ② empty, (counting Semaphore)
↳ tracks empty slots
- ③ full $\Rightarrow$ tracks filled slots.

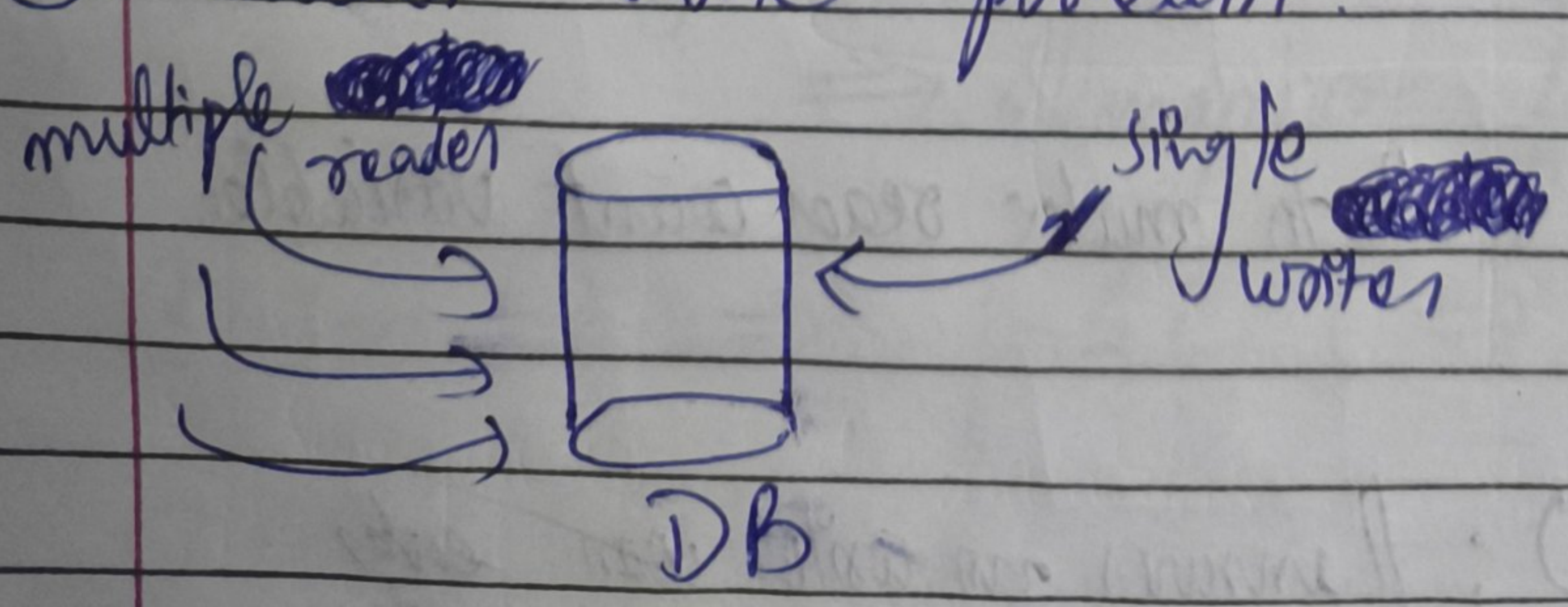
Producer

```
do {  
    // wait until  
    wait(empty); empty > 0  
    then, empty--  
    wait(mutex);  
    // C.S add data to buffer  
    signal(mutex);  
    signal(full); // increment  
} while (1)
```

Consumer

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

② Reader - write problem.



- ① Reader thread
- ② writer thread

→ if > 1 readers $\Rightarrow$ no issue ✓
→ if > 1 writes OR 1 writer & some other thread (R/W) $\Rightarrow$ Race Condition.
 & data is inconsistent.

soln

① mutex $\rightarrow$ Binary Semaphore

$\rightarrow$ to ensure mutual exclusion, when read count (rc) is updated.

* No two threads modify RC at same time.

② wst $\rightarrow$ Binary Semaphore

$\rightarrow$ common for both reader/writer.

③ read count : int {0},

tracks how many readers are reading

writer

do {

wait (wst);

// do write operation

signal (wst);

}

while (true)

~~Reader~~ Reader

do {

wait (mutex); // to mutex read count variable

rc ++;

if (rc == 1)

{ wait (wst); // ensures no writer can enter even if there is one reader.


```

signal(mutex);
// C.S : Reader is reading

```

```

wait(mutex)
rc--; // a reader leaves

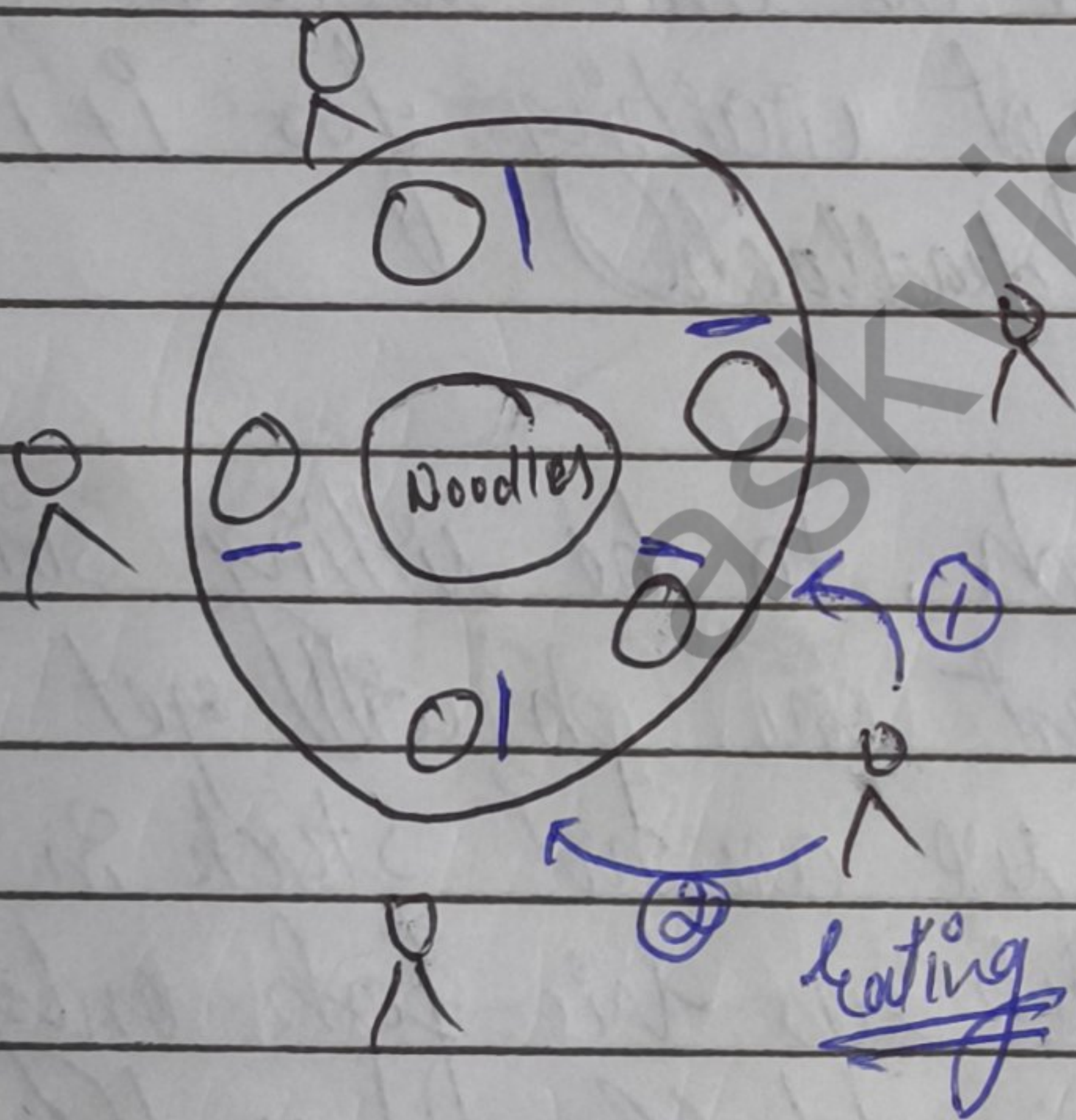
```

```

if (rc == 0) // no reader is left in CS
{
    signal(wait);
    signal(mutex);
}
while(1)

```

② Dining Philosophers Problem :



- ① 5 philosophers
- ② 1 Noodle plate
- ③ 5 forks

Philosopher has two states

- ① Thinking
- ② Eating

* A philosopher need two forks for eating.

how to sync & avoid deadlock.

Solⁿ

- ① Each fork — semaphore (Binary)
 $\Rightarrow$ Semaphore fork[5] {1}

- ② Wait() $\Rightarrow$ fork[i] $\Rightarrow$ ph[i] $\rightarrow$ acquire

- ③ Release() $\Rightarrow$ fork[i] $\Rightarrow$ fork $\rightarrow$ release

make
this also
a
G.S

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

→ This solⁿ ensures adjacent ph. don't eat together
as the $i+1$ fork will be locked.

→ Problem: Deadlock: If all at once locked i fork,
& kept waiting for $i+1$!
(infinite deadlock)

try to
avoid this
solⁿ
methods

a) allow atmost $n-1$ ph, to be sitting simultaneously
→ so atleast one ph will get the
fork & we are not stuck in deadlock.

b) Allow a ph to pick up his fork only if
both forks are available (make it also
a G.S)

c) Odd even Rule:

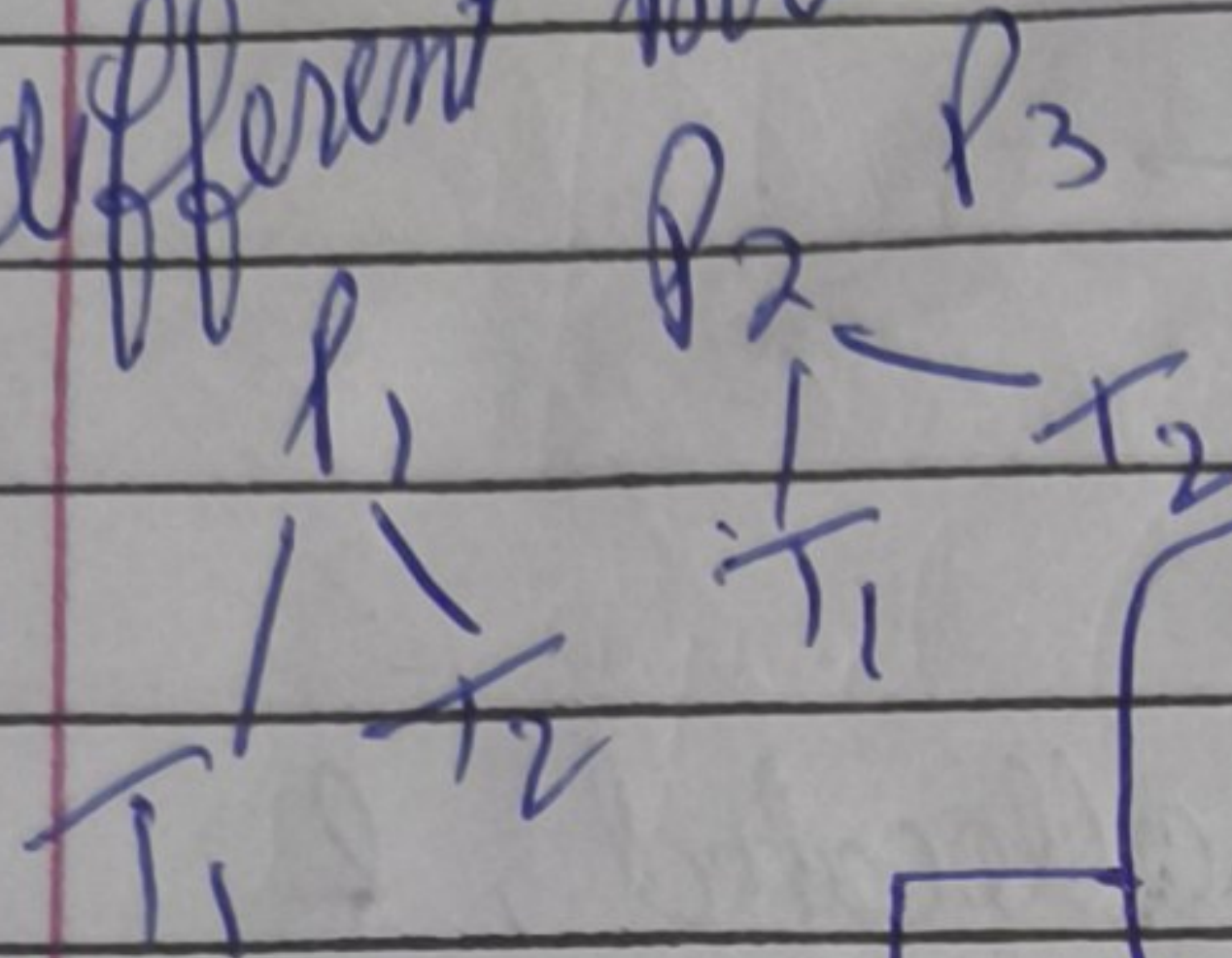
an odd ph → pick left fork then right one.
an even ph → picks right fork then left
one.

hence, only semaphores are not enough for this
problem.

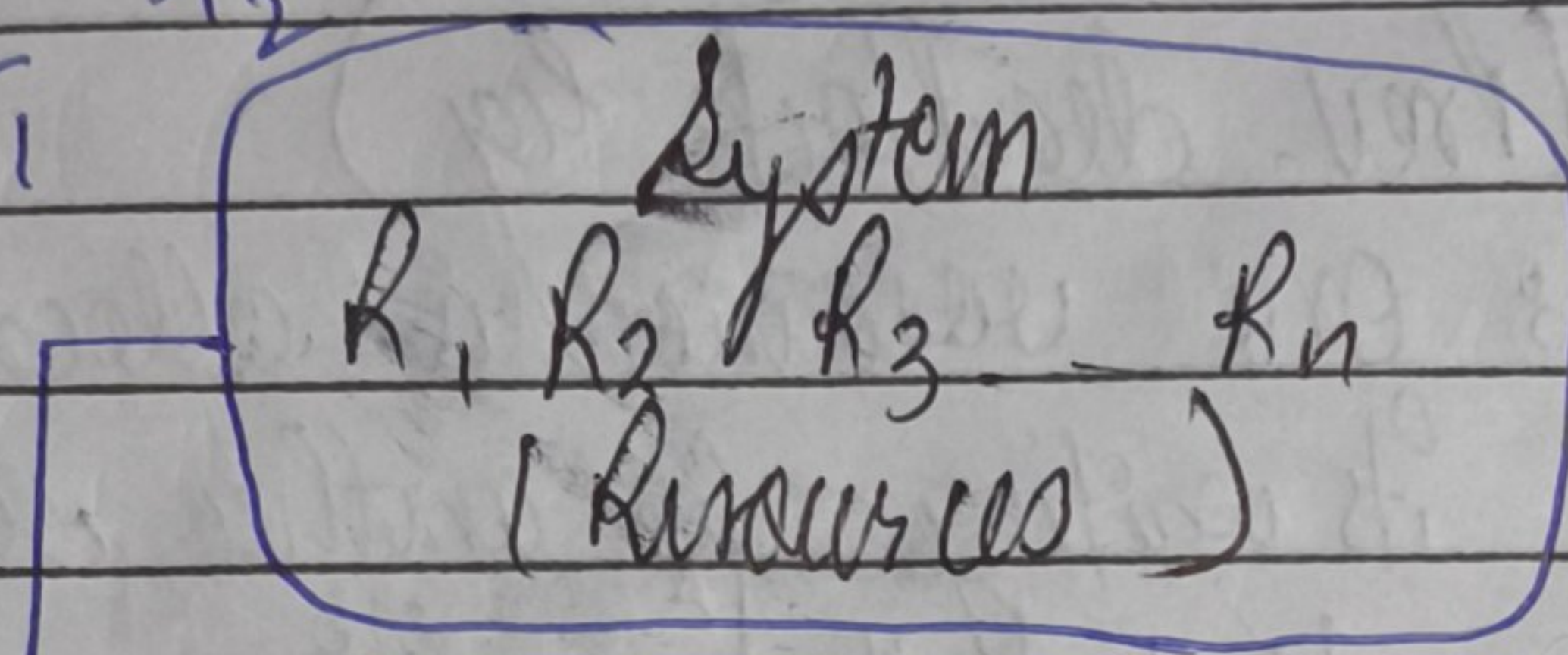
What is deadlock?

→ deadlock is sort of a bug which comes when we use locks (semaphores) for synchronisation.

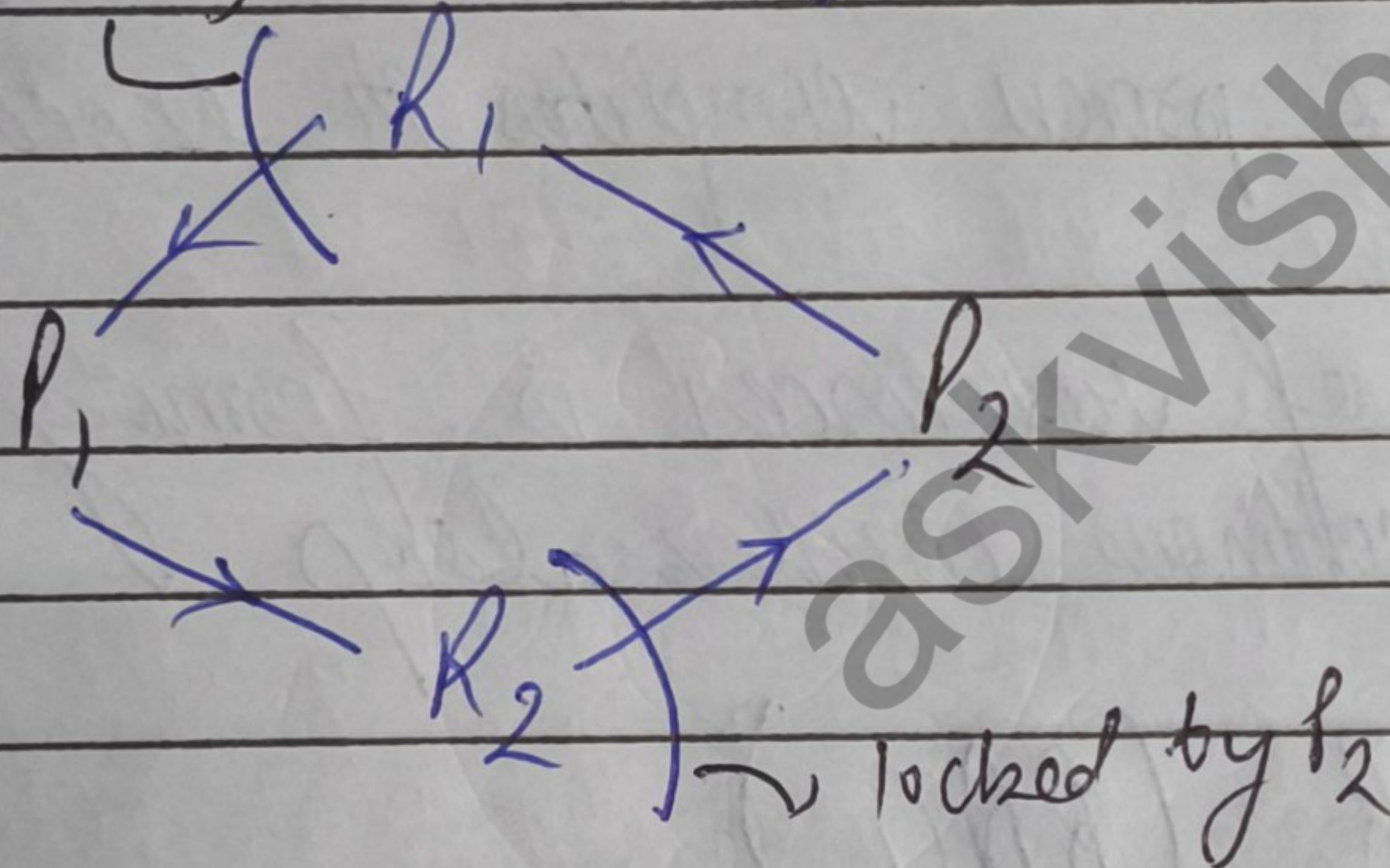
different processes



→ finite no. of resources
→ Multiple processes.



locked by P₁



for complete execution
→ P₁ ⇒ R₁ & R₂
→ P₂ ⇒ R₂ & R₁

→ Now system is in deadlock

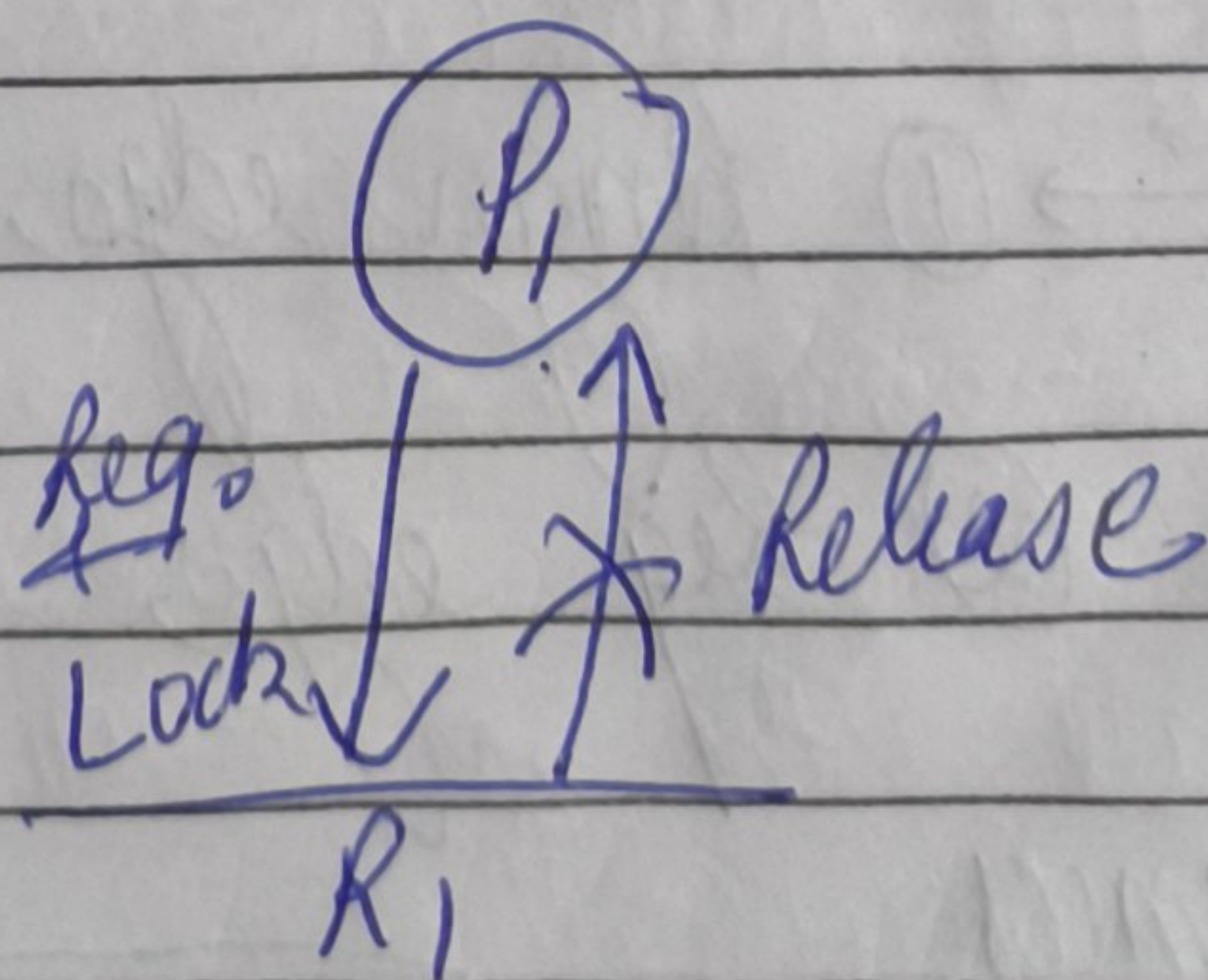
→ No progress → infinite wait.

how a process/thread utilize a resource?

① Request

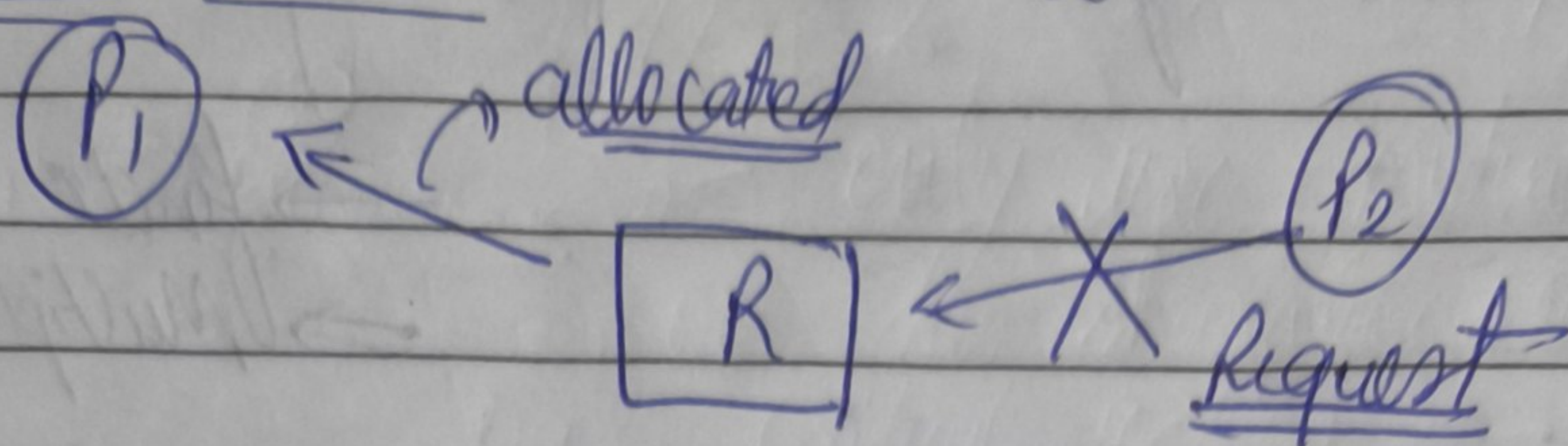
② Use

③ Release



Necessary condition for deadlock [all should be true for deadlock]

① Mutual exclusion : 1 Resource $\xrightarrow{\text{allocated}}$ 1 thread/process



② Hold & wait : (Prev. deadlock eq)
for a process : one resource is allocated, & it's waiting for another resource to start its execution.

③ No pre-emption : Allocated resource won't be released till the process completes its execution.

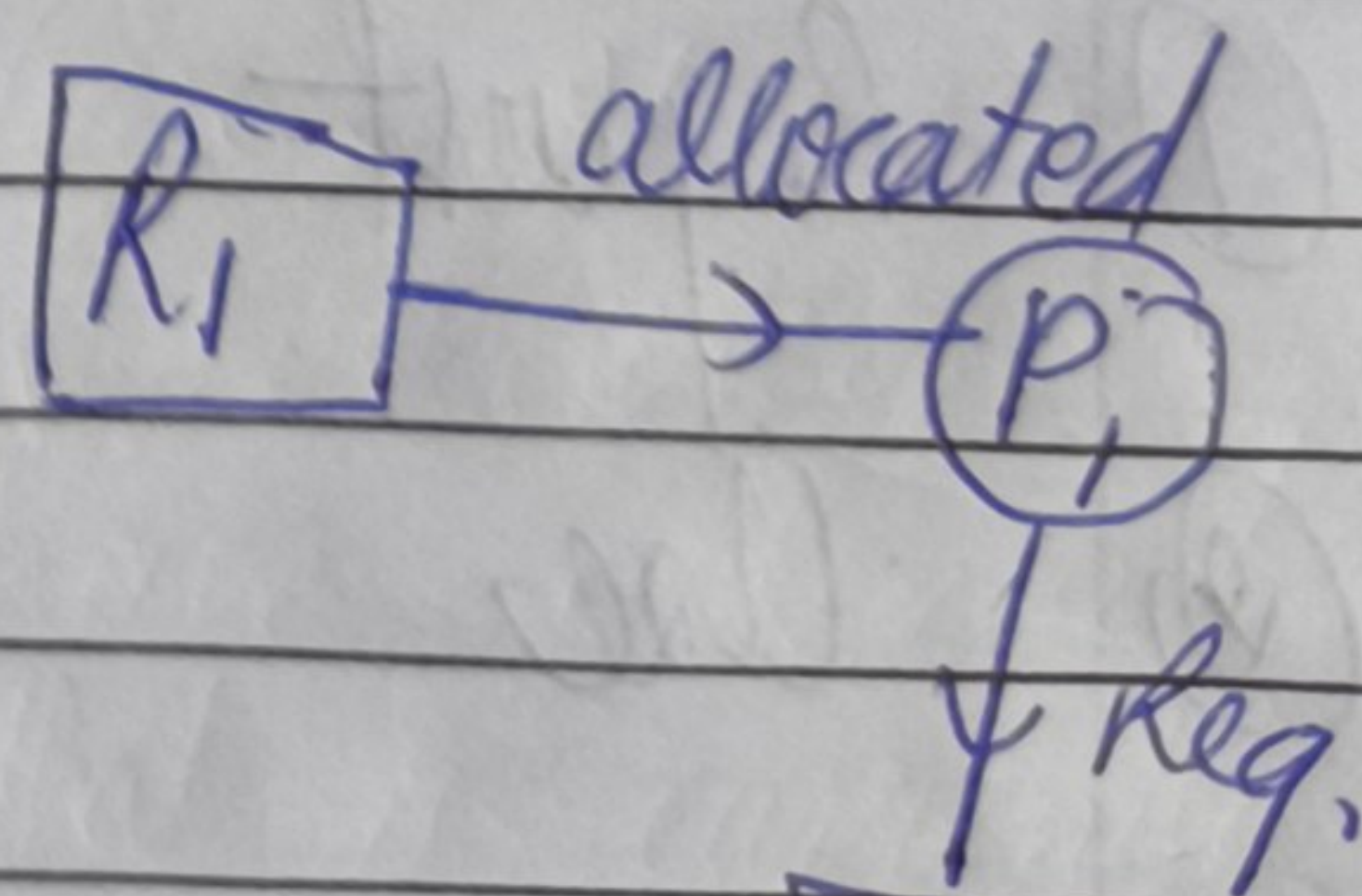
④ Circular wait : waiting of resources is forming a continuous circular loop.

Resource allocation Graph (RAG)

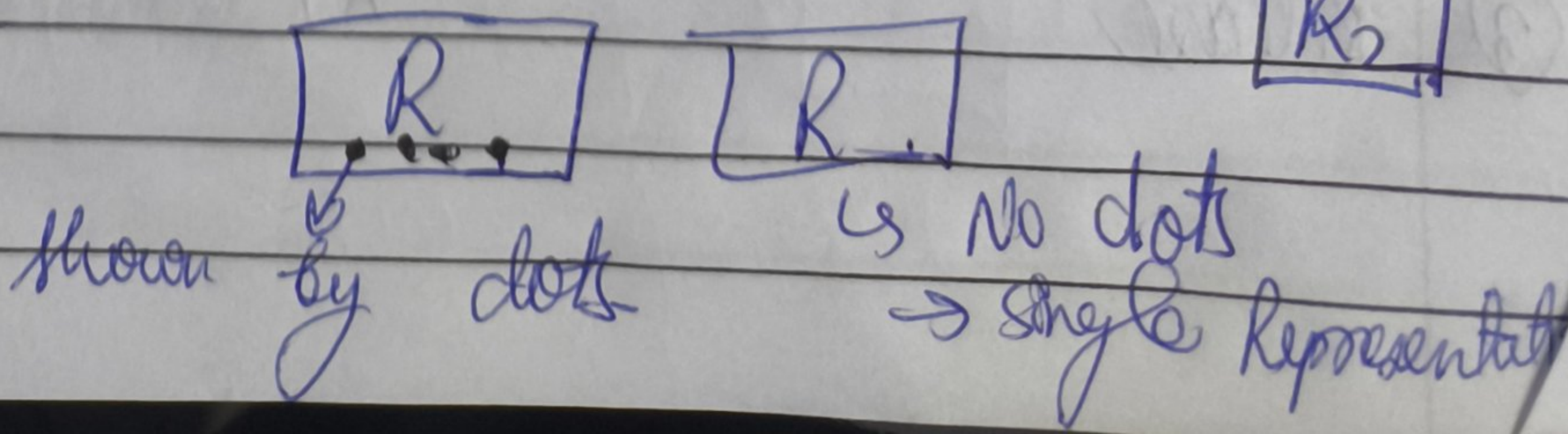
① Vertex $\longrightarrow$ ① process vertex P

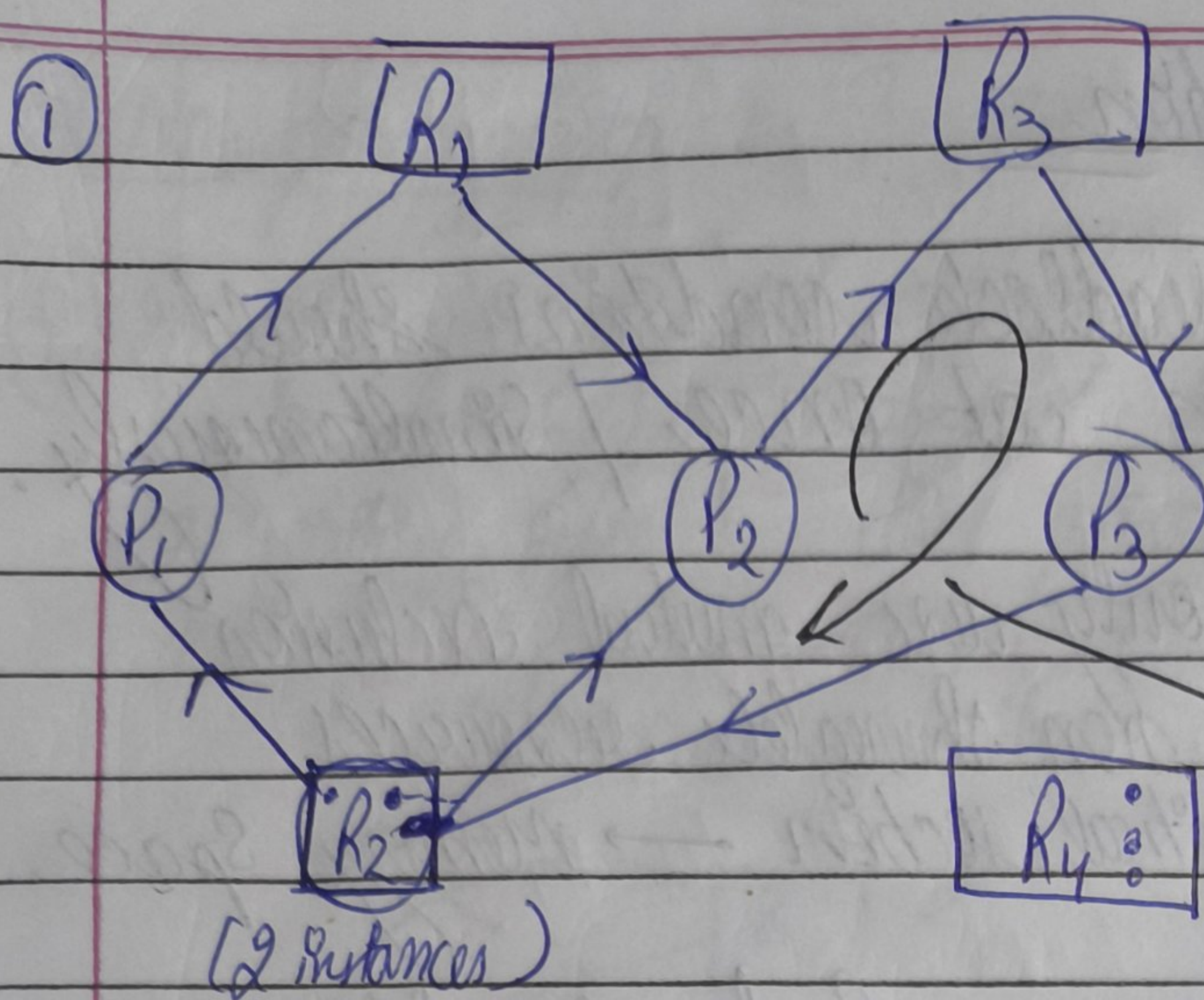
② Resource vertex R

② edge $\longrightarrow$ ① Assign edge
② Req. edge



③ Multiple Instances





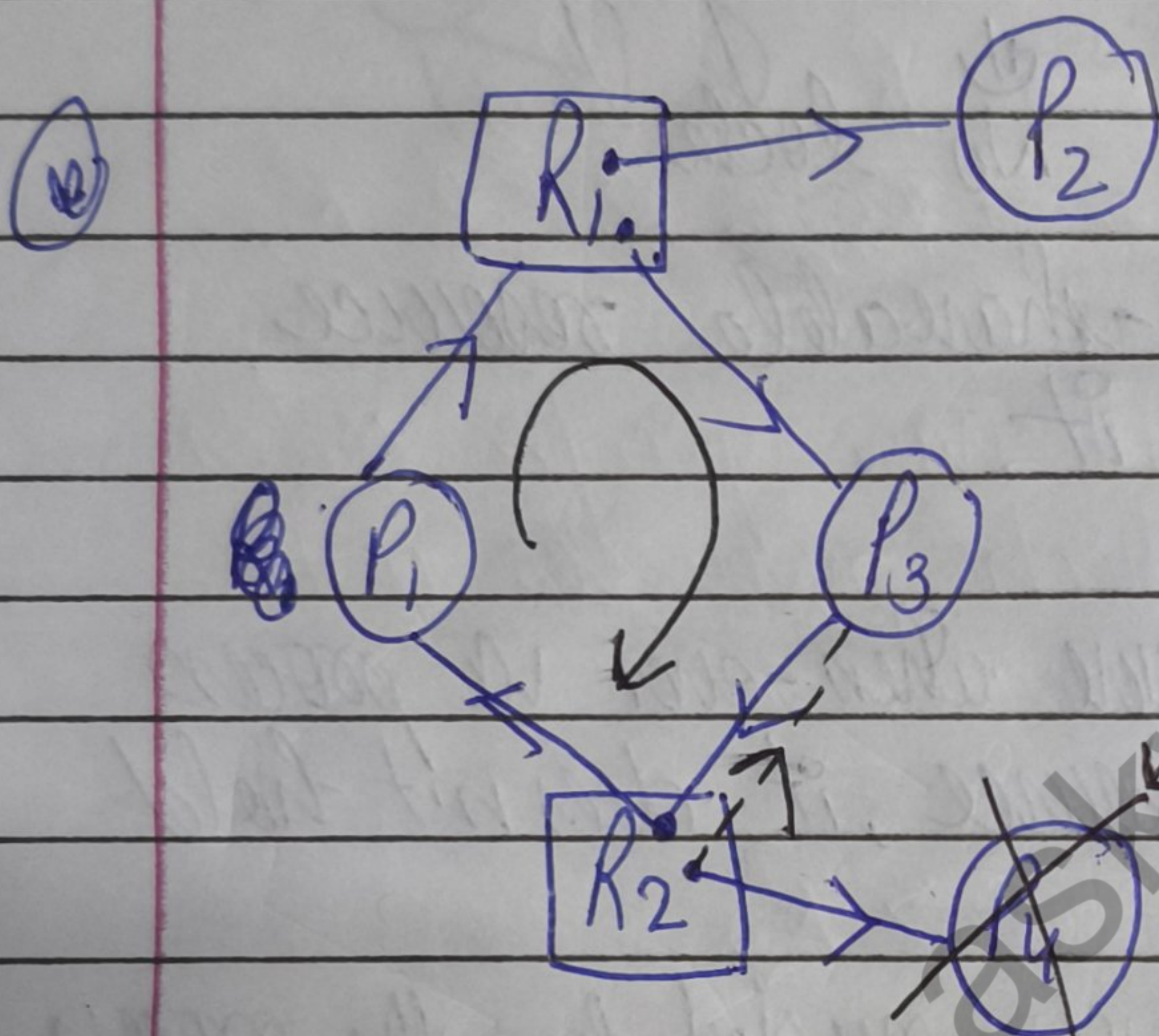
⇒ In RAG:

* If No loop ⇒ No Deadlock

* If loop ⇒ may be deadlock

DL is Present

(check all for condition)



loop ⇒ But No dead lock
⇒ So, may be deadlock.

As, after T time, P4 will release R2, which could be allocated to P3.

Methods to handle deadlock:

① Prevent / Avoid dead lock.

② Allow system to go in deadlock
→ detect → Recover.

③ Ostich algorithm : Deadlock Ignorance

↓
Assuming application program would handle it.

① Dead-lock Prevention

→ Preventing all four deadlock conditions should not happen at once / simultaneously.

① Mutual exclusion :- only use mutual exclusion on non shareable resources.
eg: Critical section → memory space.

eg: Shareable resource :- Read only files.
↓
No locks.

But we can not deny non-shareable resource hence we can just optimise it.

② Hold & wait :- we make sure when-ever a process request a resource, it does not hold any other.

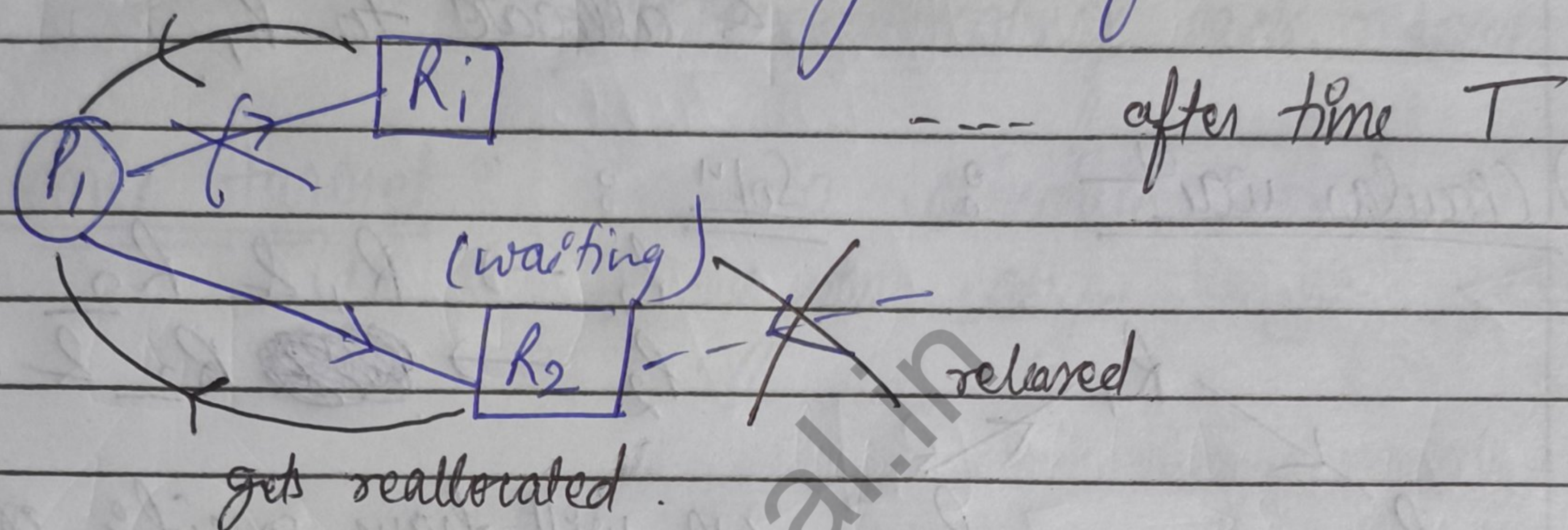
Protocol (A) :- all the resources needed by the process needs to be allocated before its execution.

drawback :- A process will hold resource which it's not even using at the moment.

Protocol (B) :- allow a process to request a resource only when it has none. It can request any additional resources after it must have released all the resources that is currently allocated.

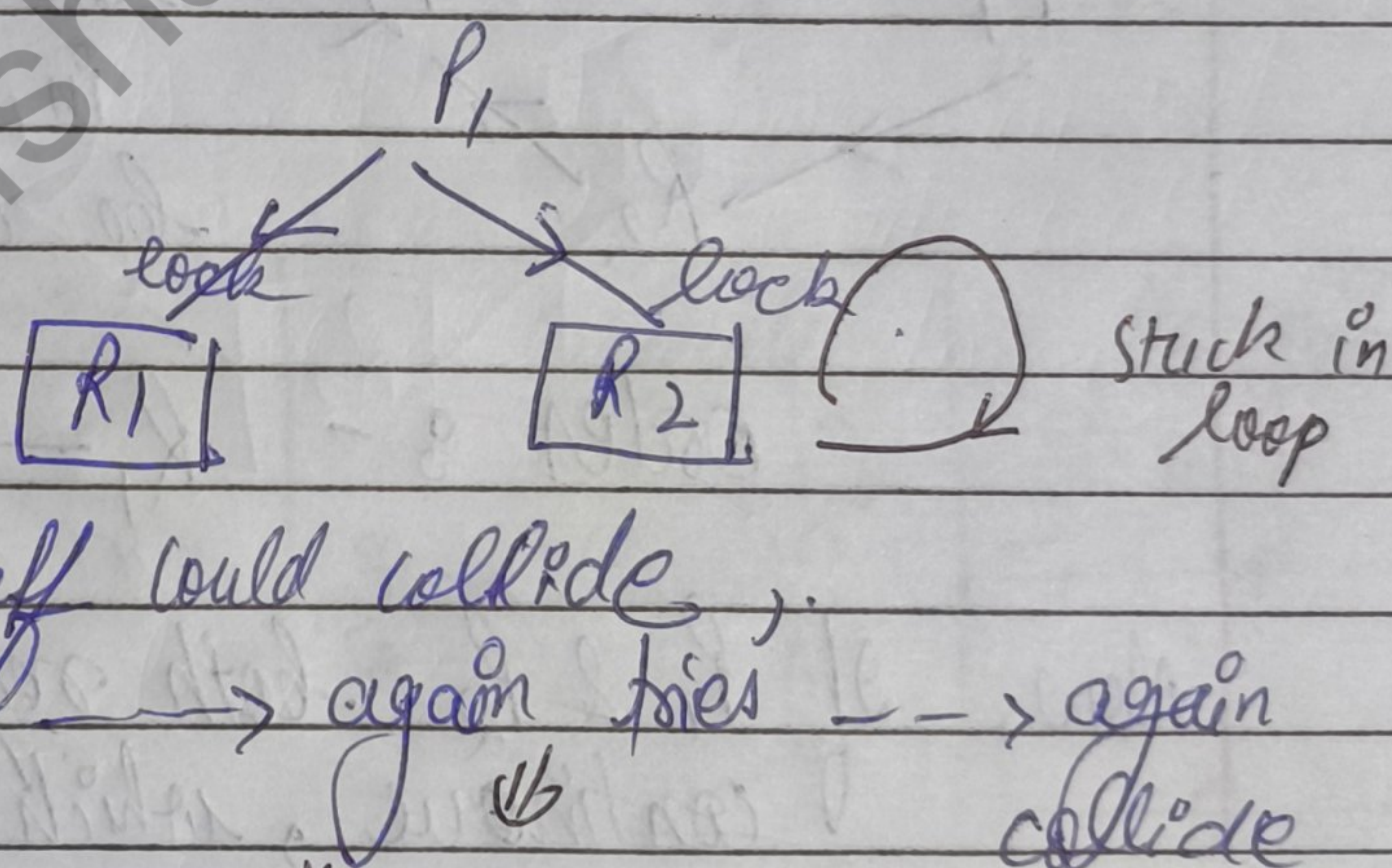
③ No-preemption :

- (i) If a process P_i is requesting another resource that can not be immediately allocated (rather than keeping the process in waiting state) the process will preempt all the holding resources, and ask request for ~~the~~ resources when they are free.



Live lock Problem

→ when P_1 tries to lock multiple resources

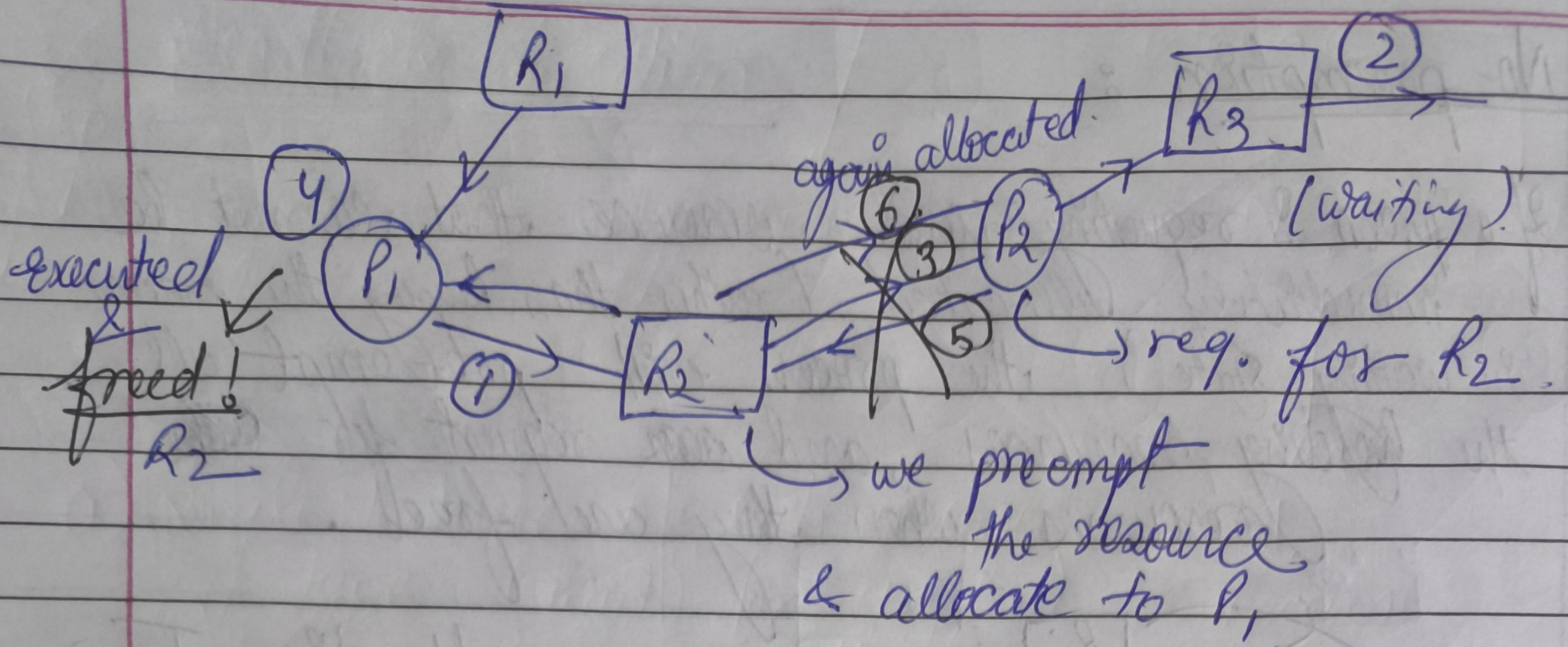


at same time, locks P_1 self could collide,

— after time t — → again tries — → again collide

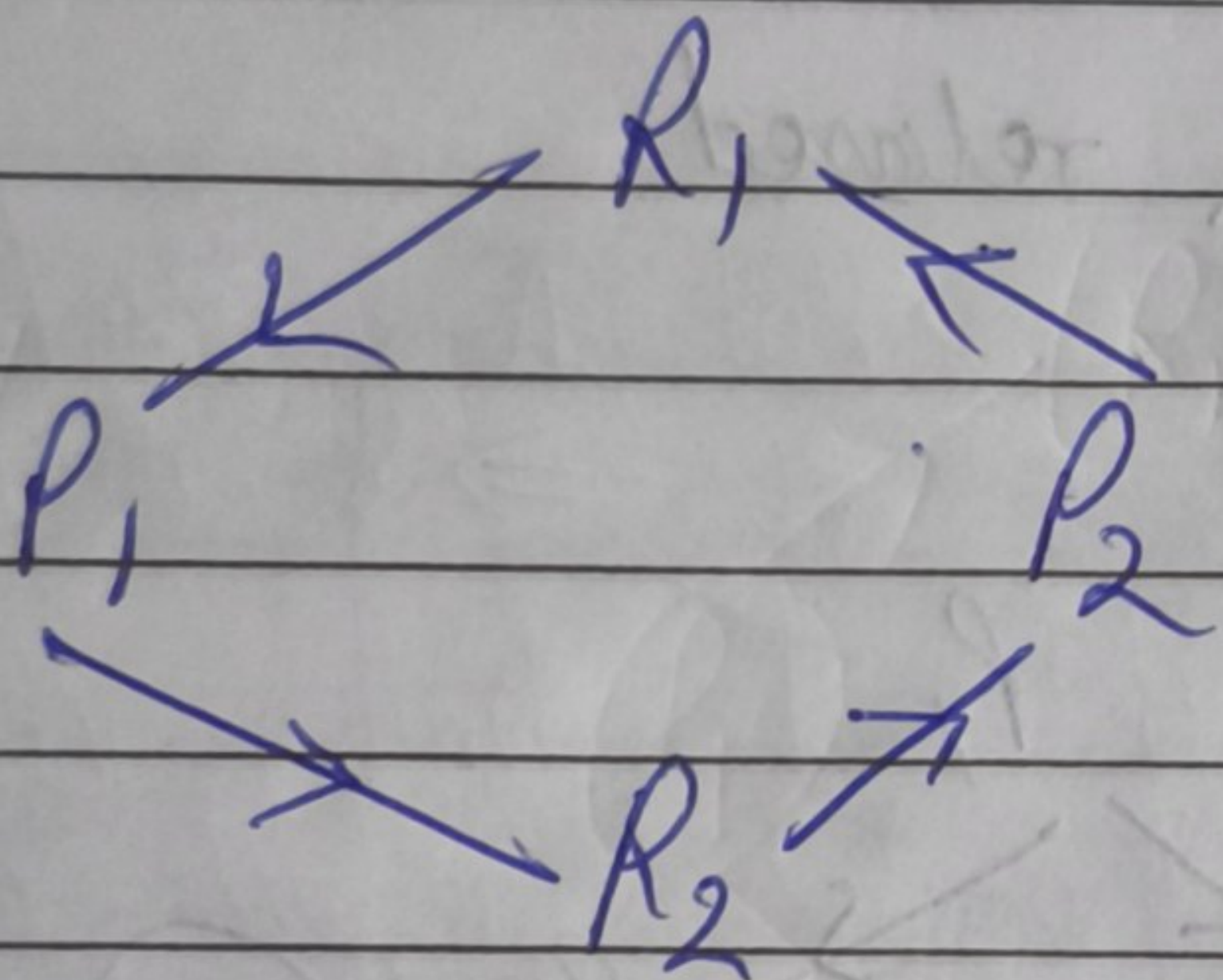
Solⁿ → add a difference of some time b/w multiple lock req

- (ii) If a process is requesting a ~~resource~~ resource, we first check whether they are available or not, if yes → allocate but if not, we check whether they are allocated to some other process that is waiting for additional resources allocation. If so, we preempt the desired resources from waiting process and allocate them to the requesting process.



④ Circular wait

Solⁿ:

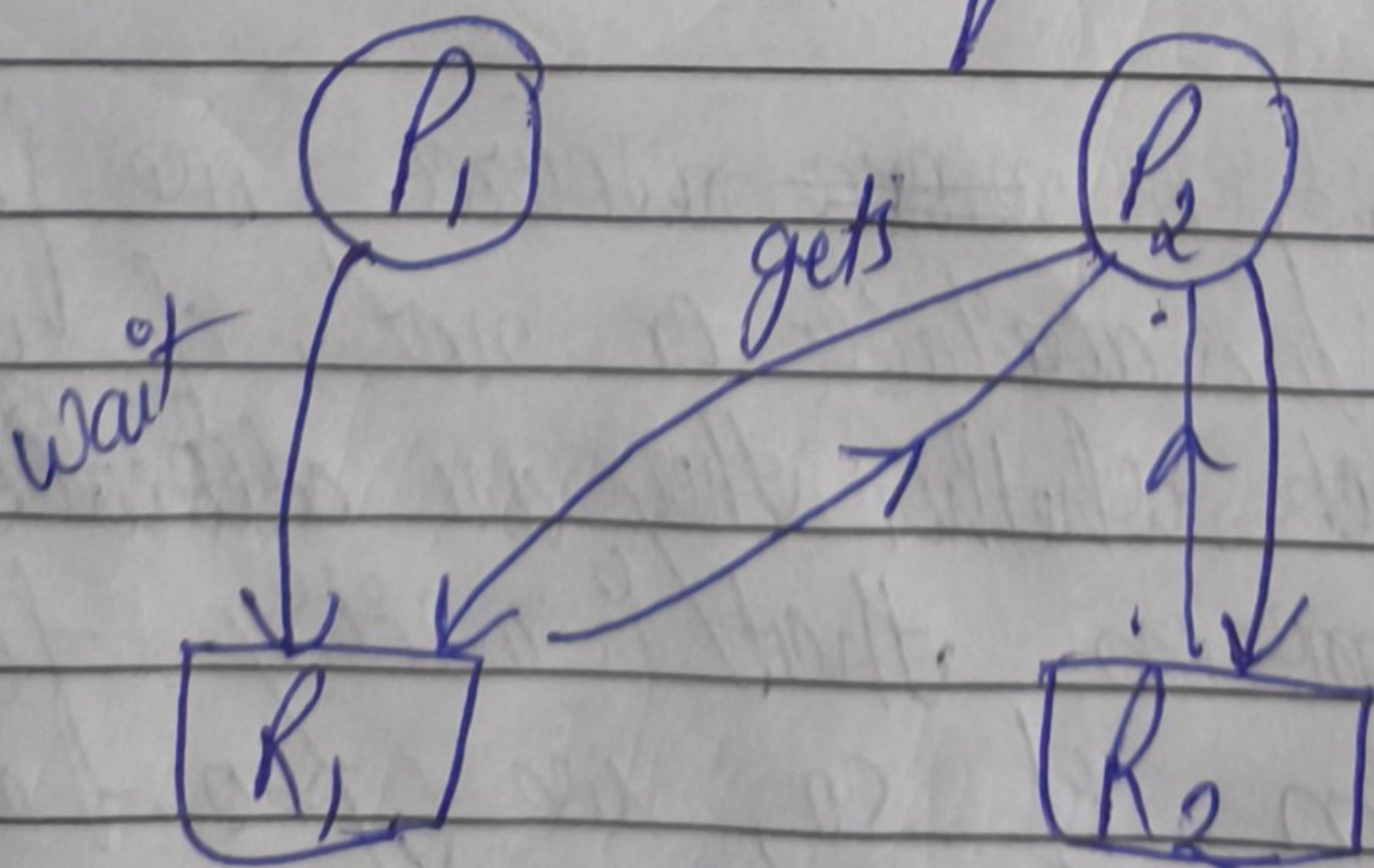


$P_1 \rightarrow R_1 \& R_2$
 $P_2 \rightarrow \text{R2} \& R_1$

we will have specific order in which resources will be allocated

order :- $R_1 \rightarrow R_2$

Now, if P_1 & P_2 both req. for R_1 , which gets R_1 , continue, which does not, gets in waiting state for R_1 & will never move further.



~~Deadlock avoidance:~~

→ things we know in deadlock avoidance:
 → i.e. Kernel

① Current State

- ① no. of Processes
- ② need of Resources of each Process
- ③ currently allocated amount of Resource
- ④ Max amount of each resource.

Avoidance target: how to schedule Process & resource so system stays in safe state

→ always remain deadlock free state.

Banker's algorithm

P	allocated			max need			available			Remaining		
	A	B	C	A	B	C	A	B	C	A	B	C
P ₁	0	1	0	7	5	3	3	3	2	7	4	3
							2	0	0			
P ₂	2	0	0	3	2	2	5	3	2	1	2	2
							2	1	1			
P ₃	3	0	2	9	0	2	7	4	3	6	0	0
P ₄	2	1	1	4	2	2	7	4	5	2	1	1
P ₅	0	0	2	5	3	3	7	5	5	5	3	1
Total:	7	2	5				10	5	7			

Total already

// (Safe Seq)

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

P₂ → P₄ → P₃ → P₁ → P₅

Deadlock detection

1st Step:

System $\Rightarrow$ algorithm

DL?

No

Yes

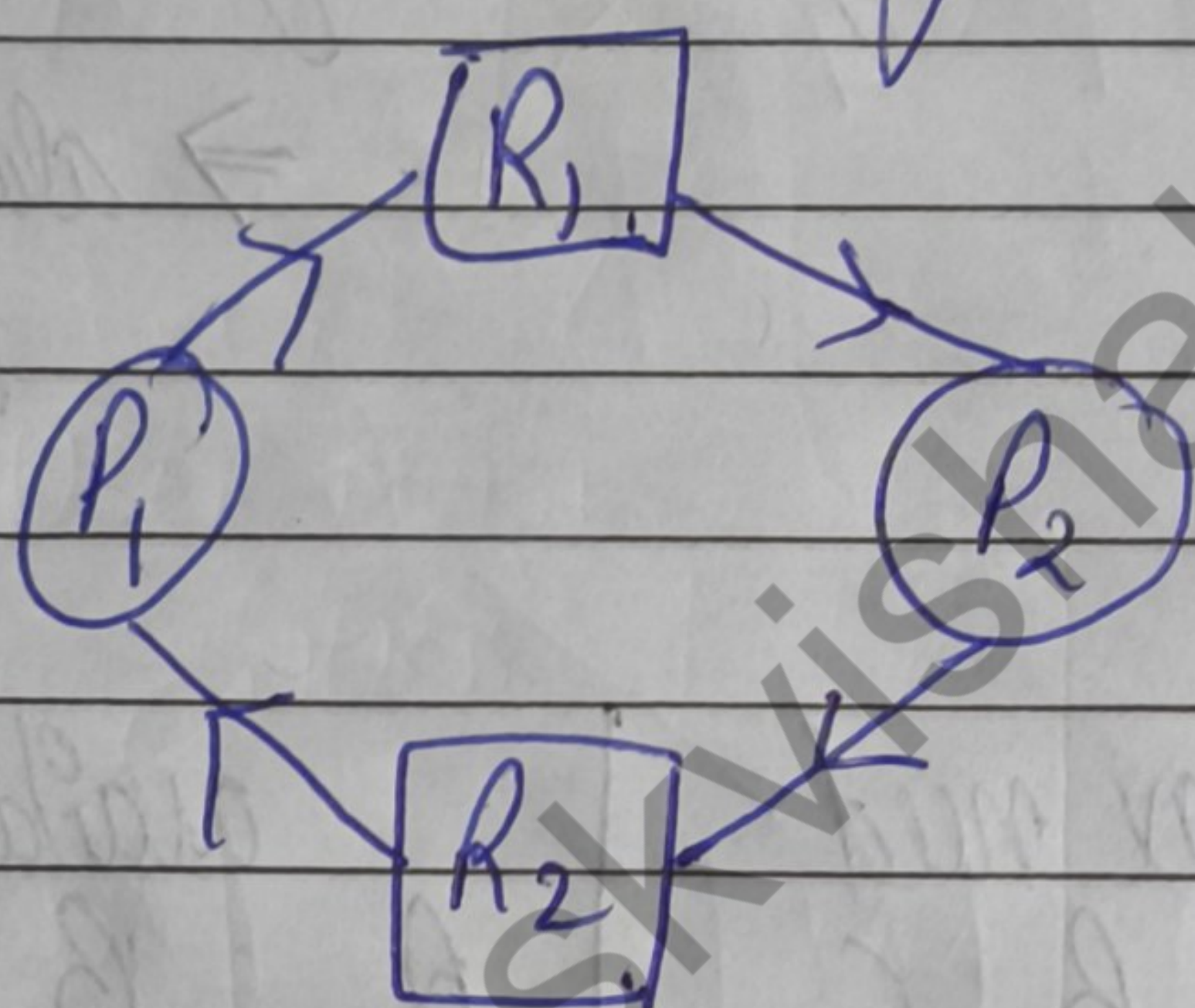
Recovers

Run after some
interval

a) Single instance of each resource
type (wait-for graph)
method

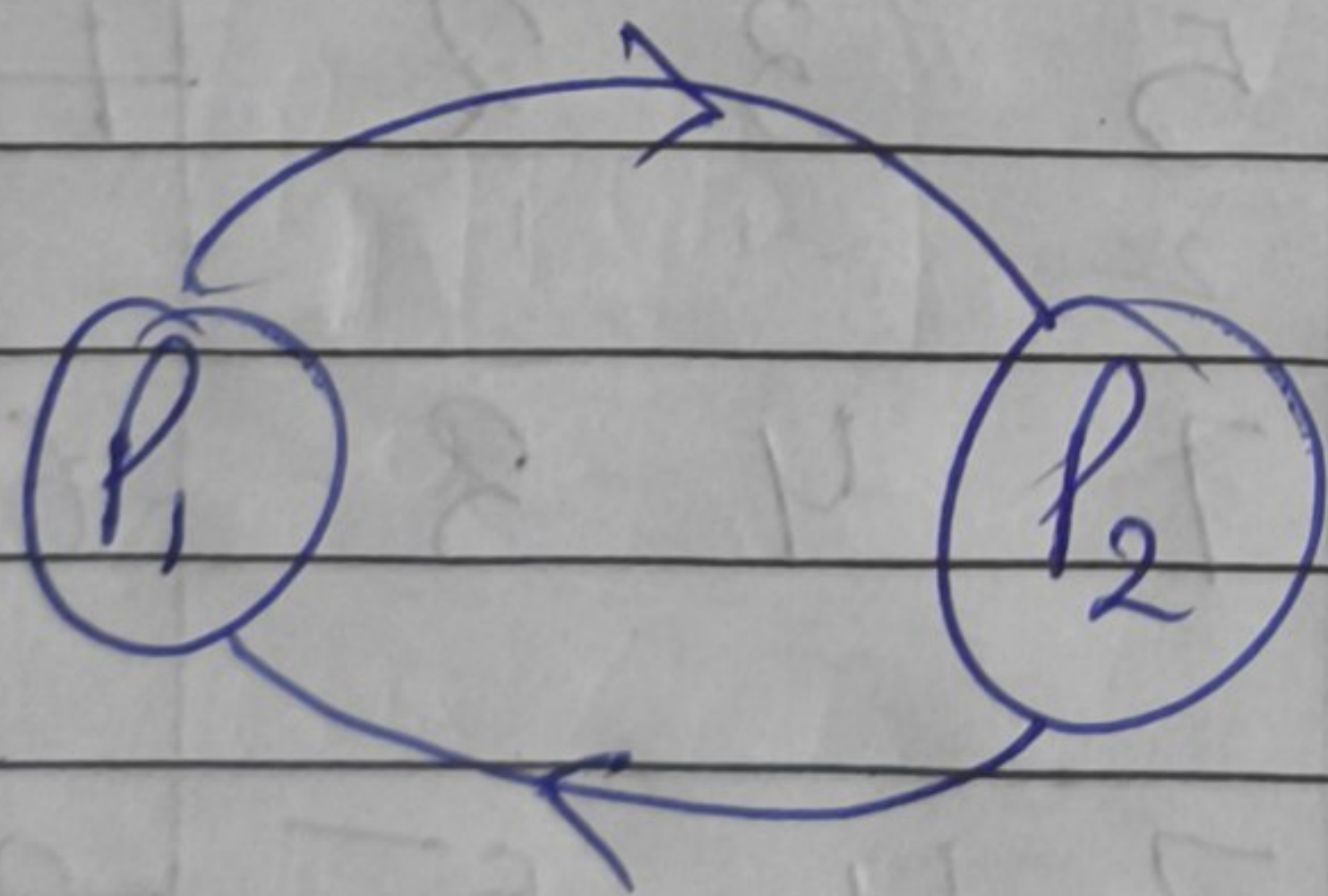
$\Rightarrow$ Each resource has only one instance.

for RAG



A wait for Graph

: After creating wait-for Graph
with the help of RAG.



If loop $\rightarrow$ DE ✓
If no loop $\rightarrow$ No DE

b) Multiple instance :

using Banker Algorithm

$\rightarrow$ If safe sequence $\rightarrow$ No DE

$\rightarrow$ If no safe sequence $\rightarrow$ DE ✓

Recovery from Deadlock :

2nd Step :

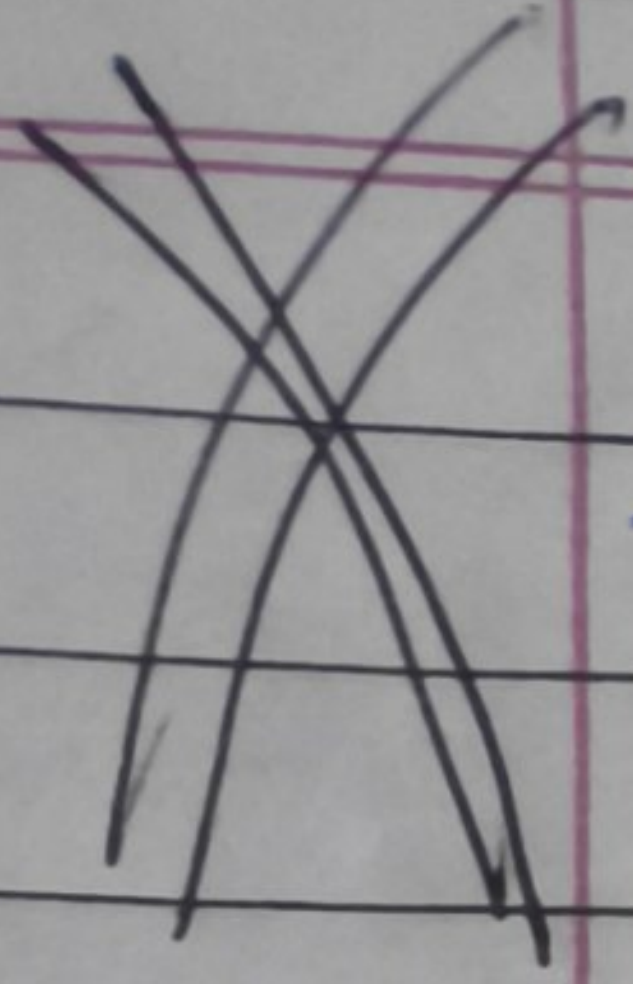
a) Process termination :

- (i) abort all DL processes.
- (ii) abort one process at a time till DL cycle is eliminated

→ i.e based on priority...

b) Resource Preemption :

→ to eliminate DL, we successively preempt some resources from processes & give these resources to other processes until DL cycle is broken.



Memory Management

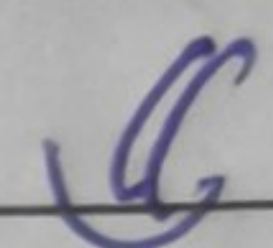
Multi-programming $\rightarrow$ many processes $\rightarrow$ In RAM

90	
80	P ₃
70	free
60	P ₂
50	P ₁
40	free
	OS

P₁, P₂, P₃

P₁ $\rightarrow$ 50 - 60

but tries to access $\rightarrow$ 65th loc.



Security breach

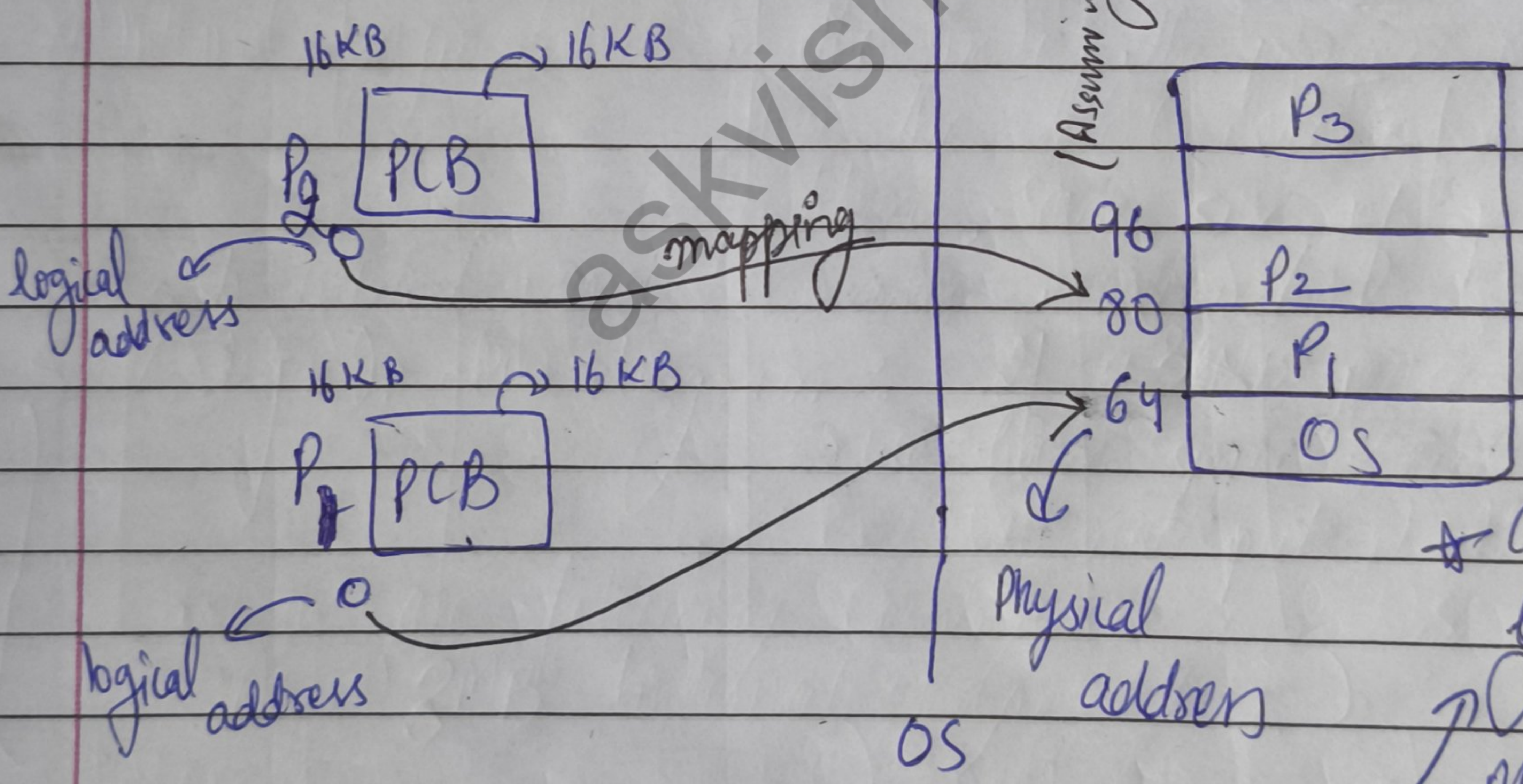
& isolation & protection is not kept

RAM

How OS solves it?

USER MODE

KERNEL MODE



$\rightarrow$ Maps logical address to physical address

Stores $\rightarrow$ range $\rightarrow$ $R+0$ to $R+max$
(R = base value)

P₁ $\rightarrow$ Base at RAM (Physical address)
B $\rightarrow$ 64

offset (size) $\rightarrow$ 16KB

Computed by
Memory Management Unit (MMU)

Hardware device

Similar for P_2 : base $\rightarrow 80$
offset $\rightarrow 16$

Assuming P_1 tries to access memory which is not allocated to it : $0 + \text{Random Value}$
 $0 + 18$
 $\Rightarrow 18^{\text{th}}$ location

OS checks : $64 + 18$: 82^{th}
(Base address)
checks range using base & offset

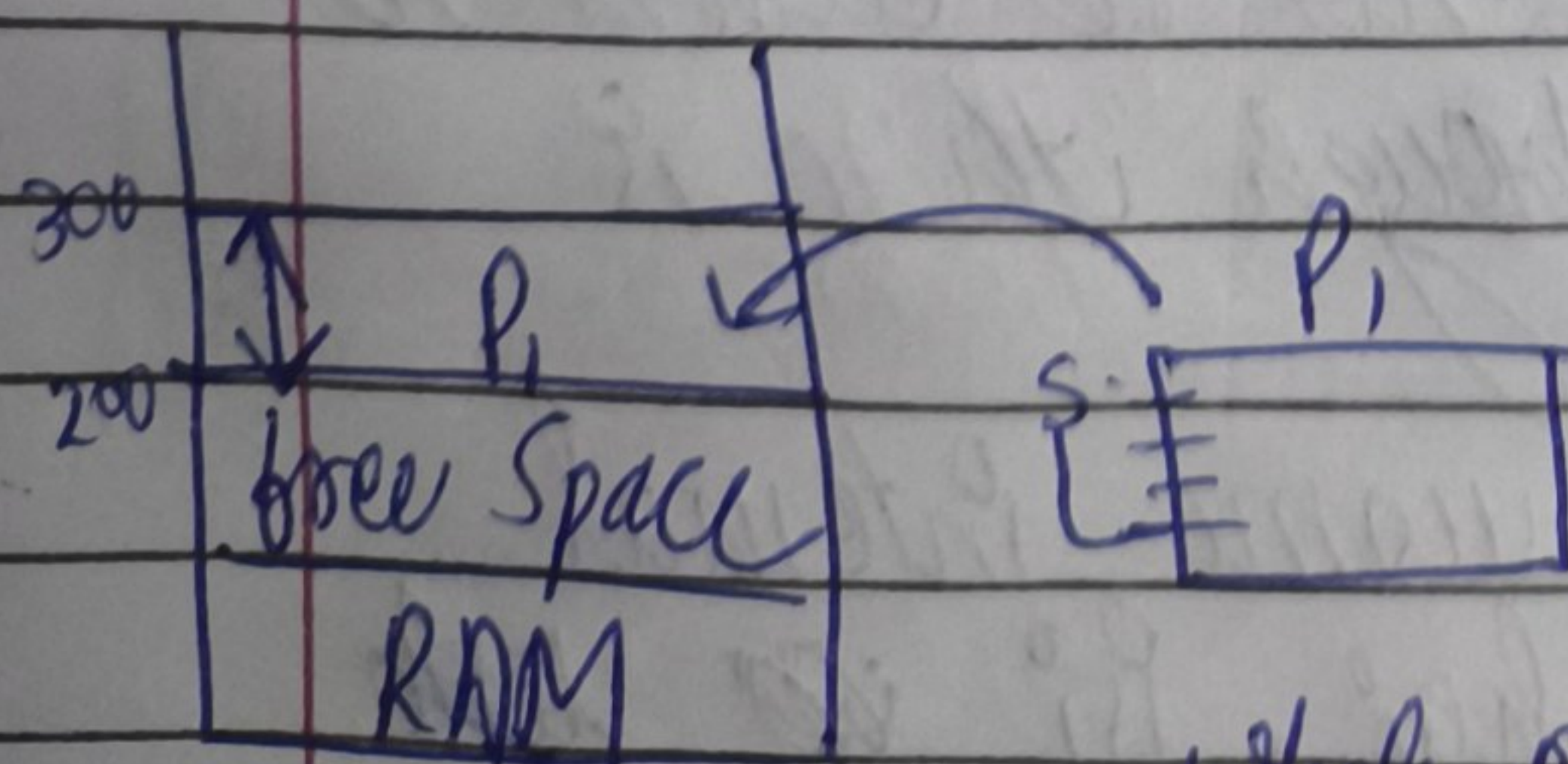
illegal address

* User programs mainly ~~thinks~~ generates logical address. & thinks program is running in this logical address but the program mainly needs main memory.

* OS provides this Virtual Address Space Concept.

Allocation Methods on Physical Memory

Contiguous Memory Allocation



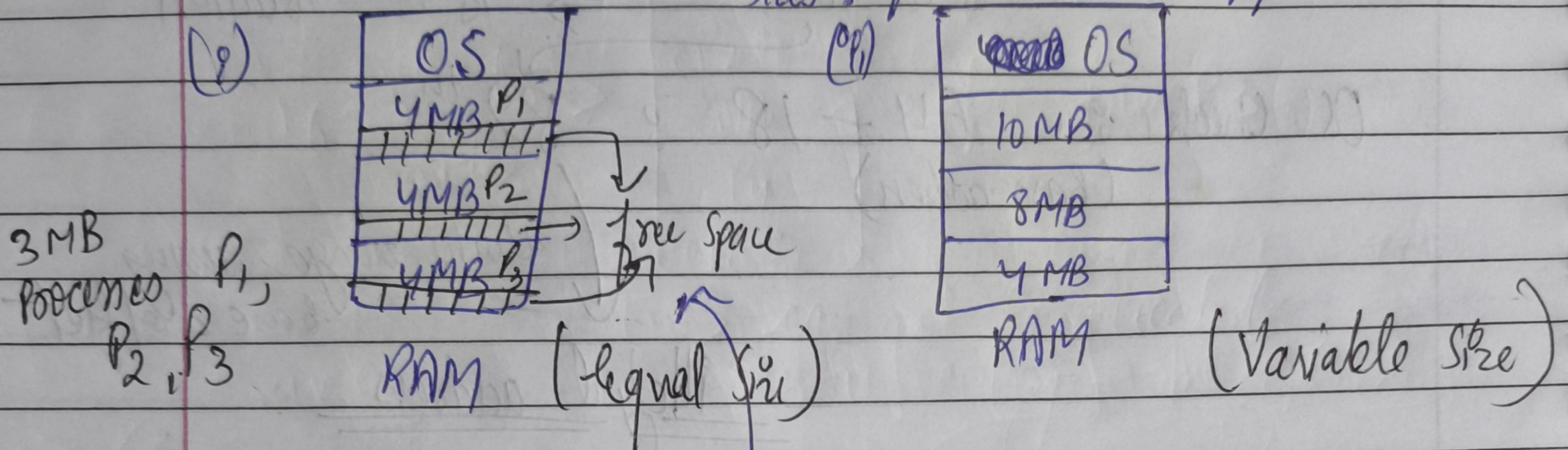
whole process is contained in a single contiguous block

Non Contiguous memory allocation

Paging (P.T.O)

Contiguous : each process is contained in a single contiguous block.

a) fixed partitioning : main memory is divided into partition of equal or different sizes.



Limitations :

a) Internal fragmentation : If the size of the process is lesser than the total size of the partition, some size of partition gets wasted and remains unused.
 eg: In above diagram : If a 3MB process comes.
 → we have space, but we can not put it there

as we are following contiguous allocation.
 → As → 3MB internal fragmentation

b) External fragmentation : the total unused space of various partition can not be used to load the process even though there is space available.

→ Same eg. : ⇒ Kyunki usme internal fragmentation hai, is ko varte external bhi ho gayi.
 ⇒ that collective space is external fragmentation.

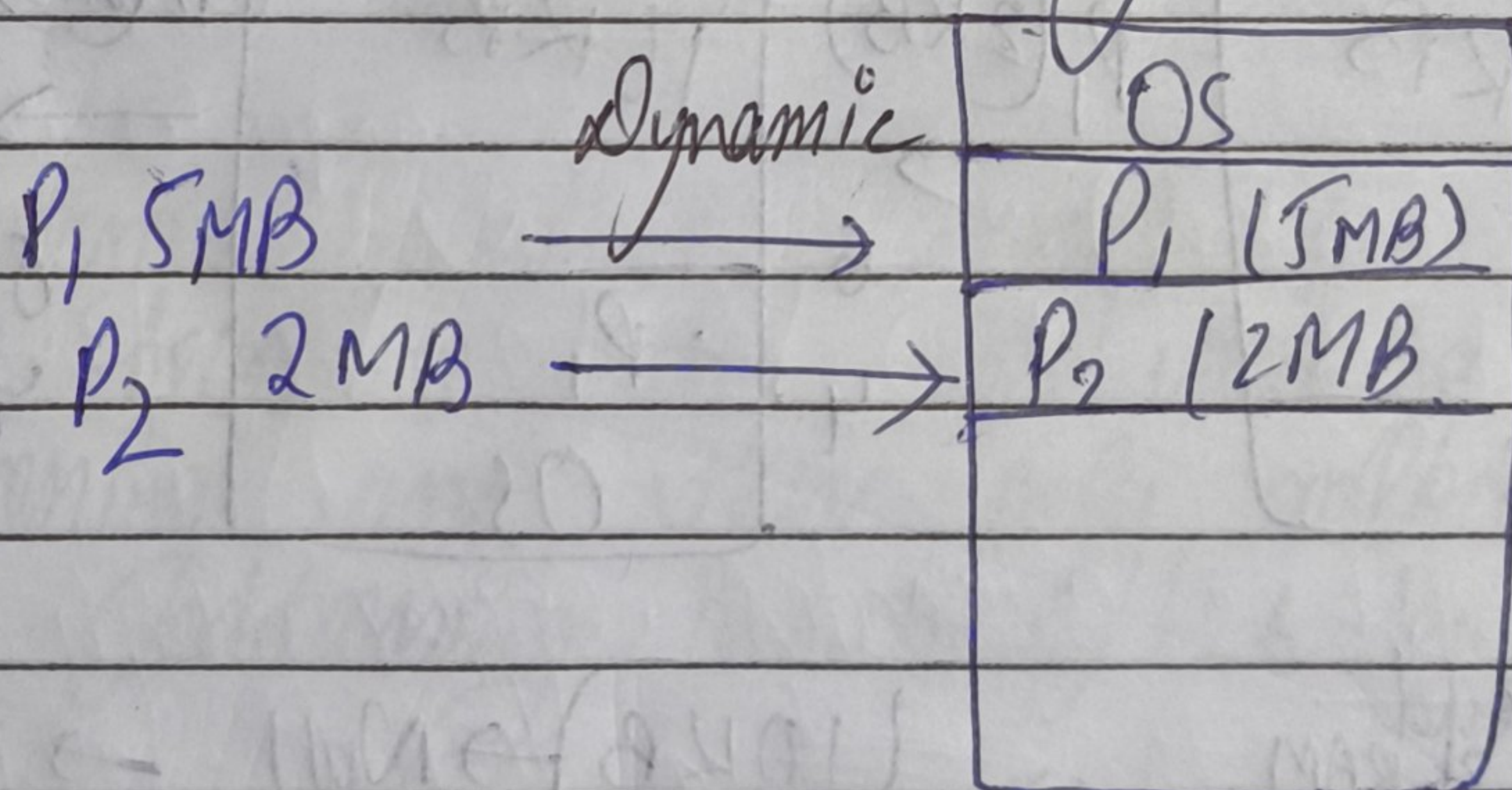
c) Limitation on process size : If the process size is larger than the size of maxi-
-num. partition, then that process can not be loaded into the memory.

as ~~the~~ fixed size also has variable partition.

eg: If a 5MB process comes, we can never allocate it even if we have contiguous space available because memory partition size is 4MB.

d) low degree of multiprogramming : In fixed partition, DOM is fixed and very less because size of partition can not be varied according to size of process.

e) dynamic partitioning : the partition size is not declared initially, it is declared at time of process loading.

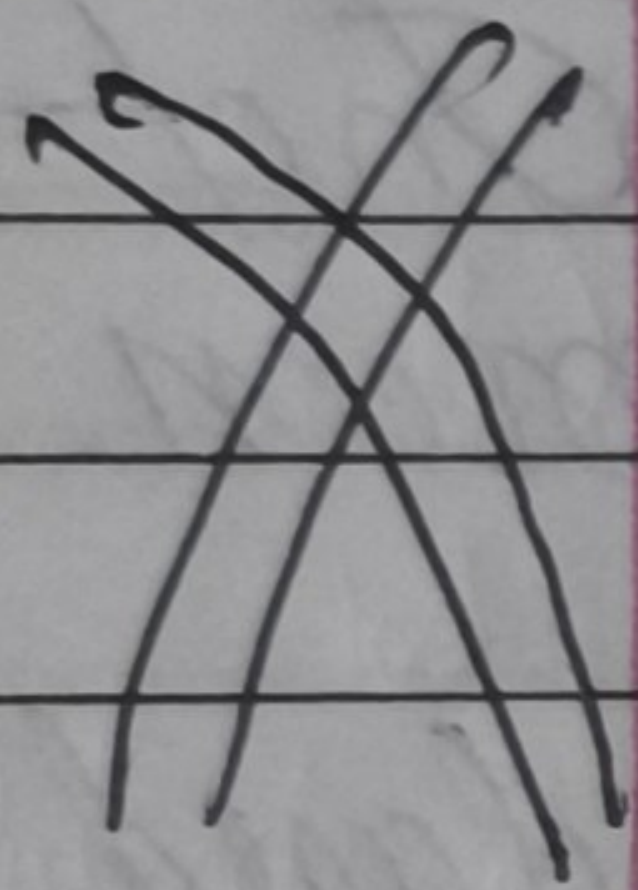
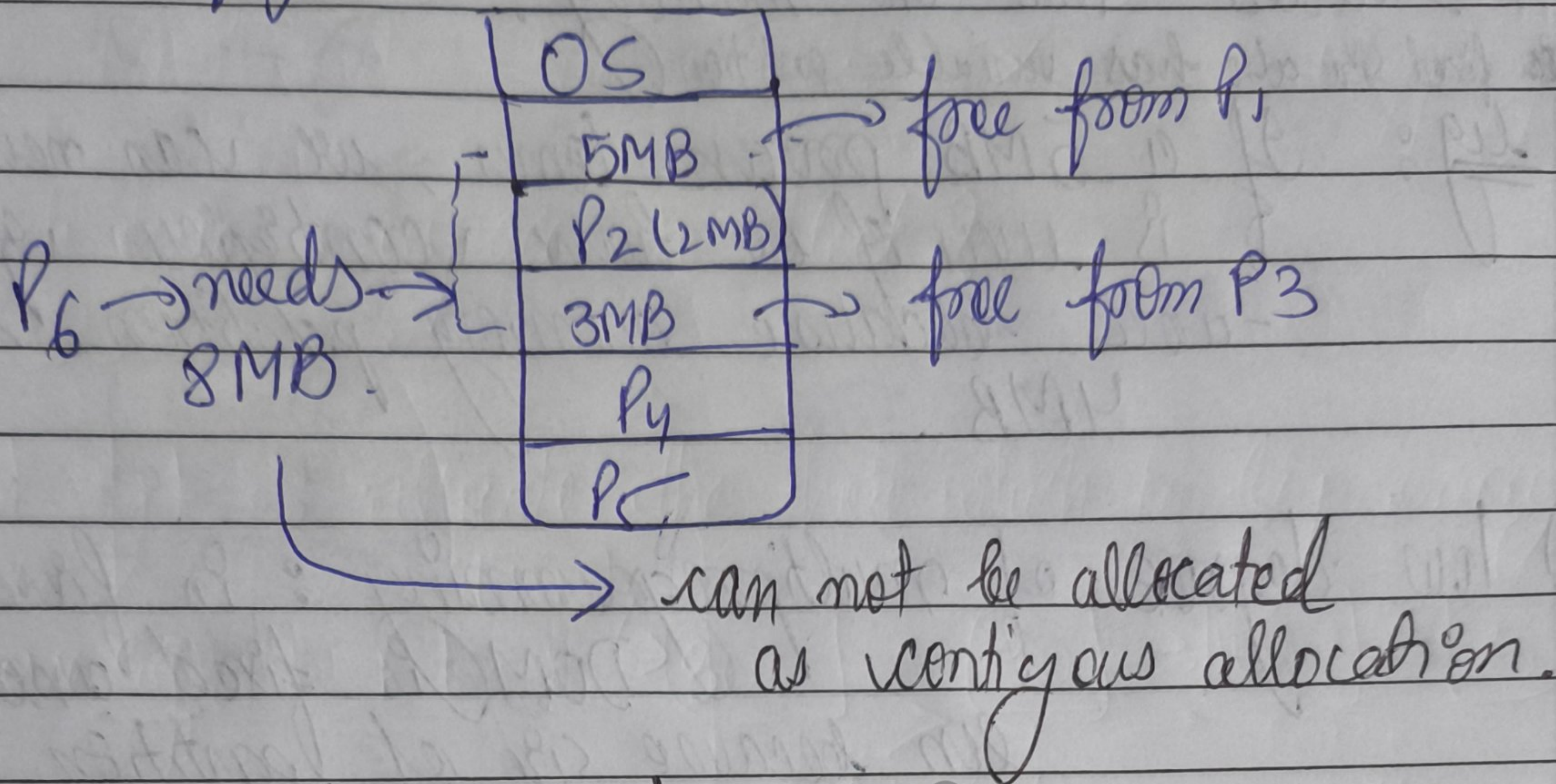


Advantage :

- (i) No internal fragmentation
- (ii) No limit on size of process
- (iii) Better degree of multiprogramming.

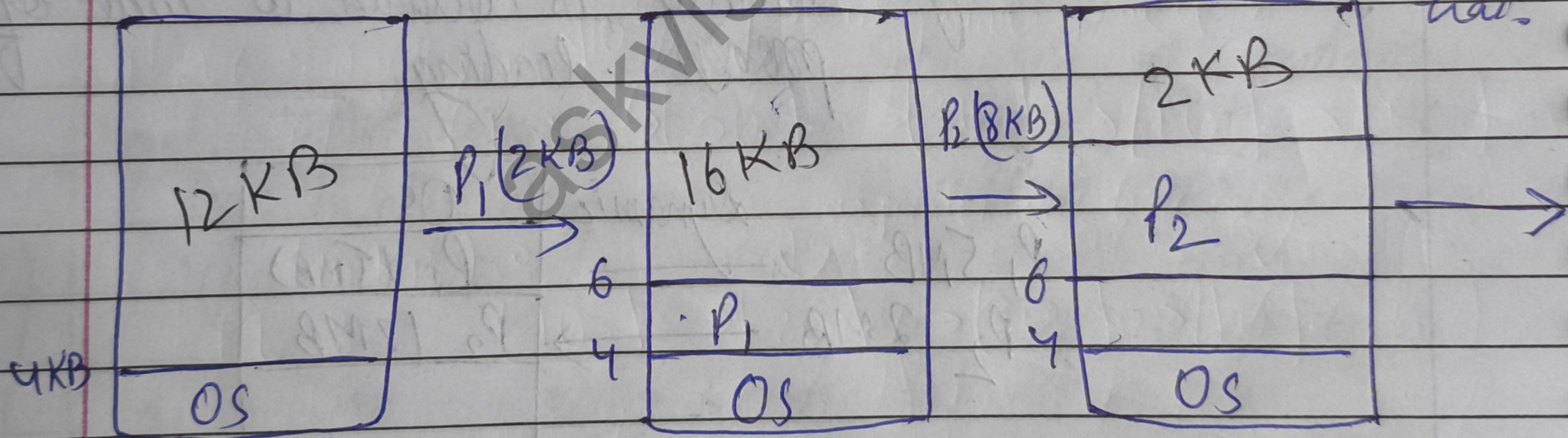
Limitation :

a) external fragmentation :

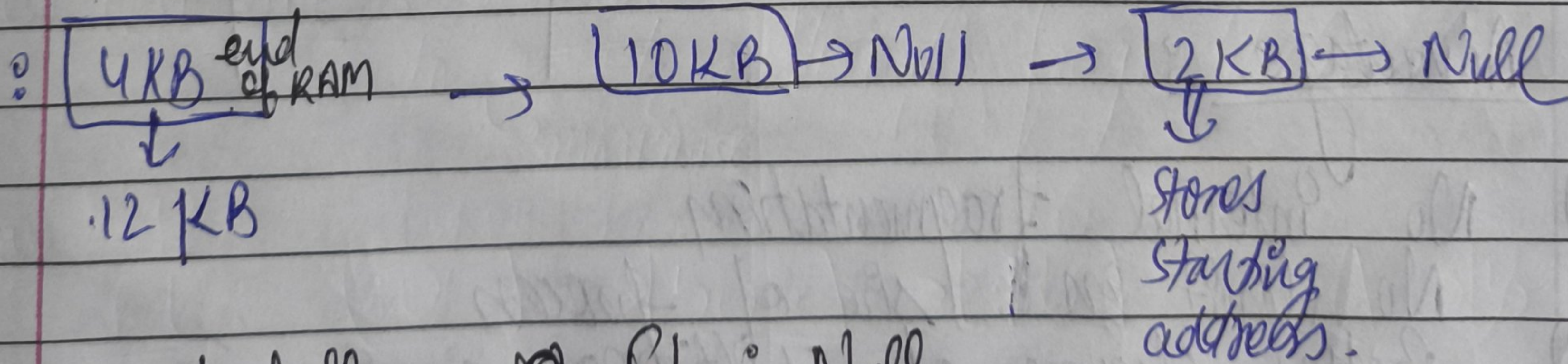


Free Space Management

OS $\rightarrow$ free space $\rightarrow$ free list $\rightarrow$ ko free list ke through manage karta hai.
Linked List.

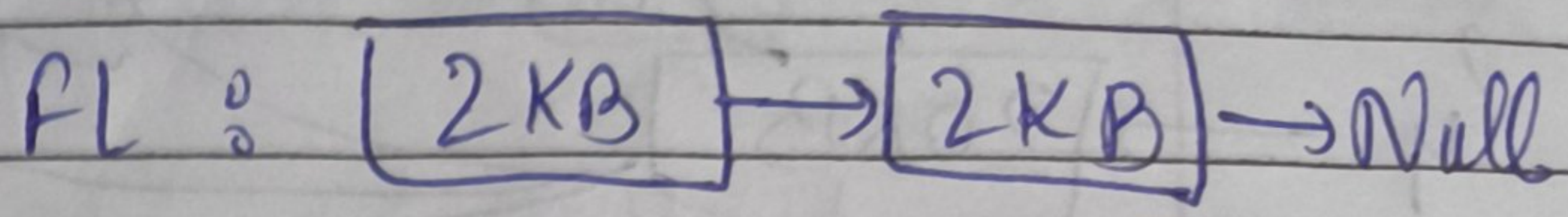
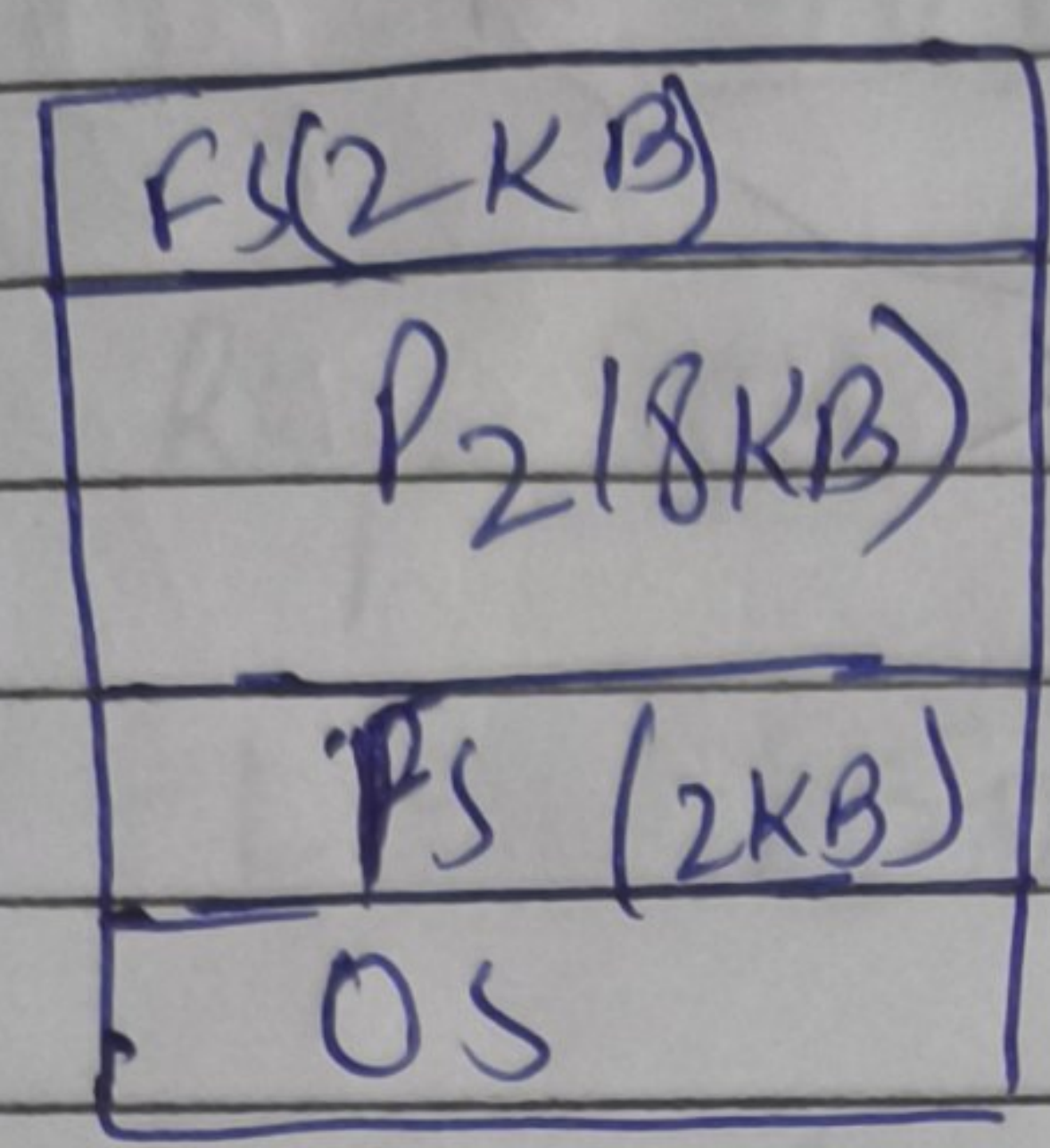


initial
FL



if full, FL : Null

After P₁ exits:

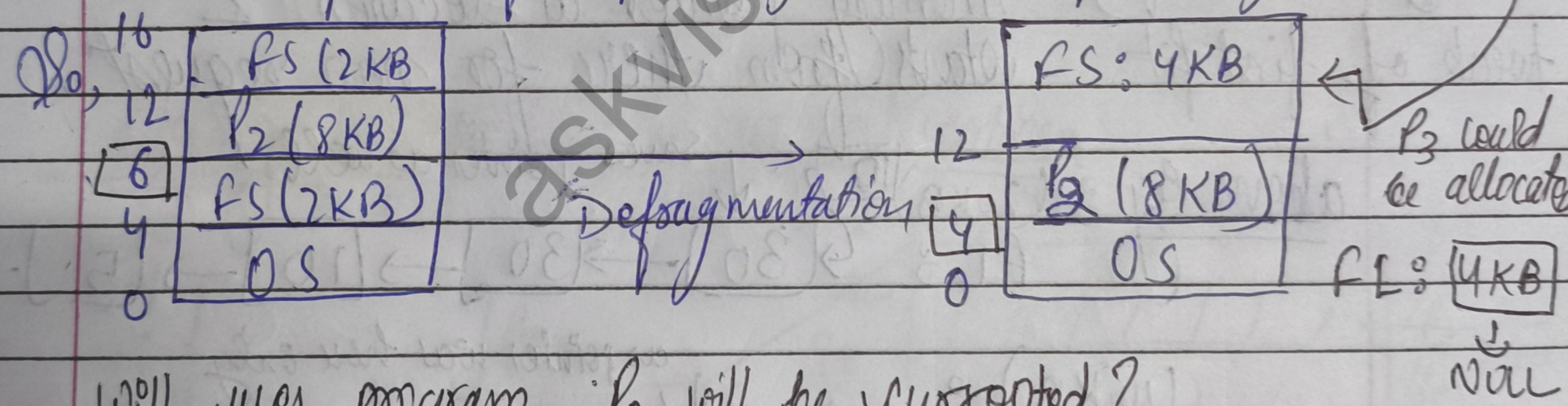


(two ~~nodes~~ nodes as two free space)

Problem : P₃ = 3 KB (space avail, but Req. Rejected)

Solⁿ : Defragmentation / Compaction

∴ Separates free space together & occupied together.



Will user program P₂ will be corrupted?

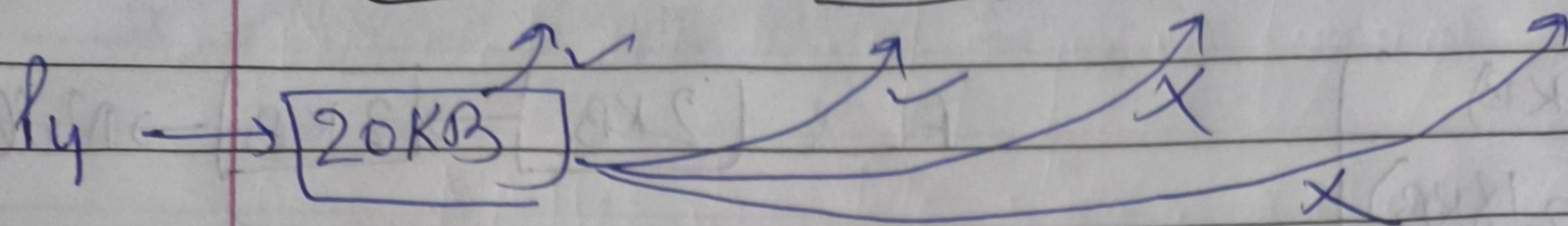
→ No, because user was working in logical addressing and there only relocation of addressing is done (6 ko 4 kaar diya)

When defragmentation happens?

→ When the no. of fragmented space exceeds a particular limit. & it is itself a time consuming process. (overhead ↑)

How to specify a req. of a m size from a list of free holes?

Free list: $\boxed{10} \rightarrow \boxed{50} \rightarrow \boxed{30} \rightarrow \boxed{10} \rightarrow \boxed{5} \rightarrow \text{Null}$



which ~~one~~ freespace to select?

Algorithms:

a) First fit: allocate the hole which is big enough.

fast & simple

above eg: FL: $\boxed{10} \rightarrow \boxed{30} \rightarrow \boxed{30} \rightarrow \boxed{10} \rightarrow \boxed{5} \rightarrow X$

b) Next fit: same as first fit but saves the memory pointed at location & starts from there for new request.

tweak of first fit

saves pointer as well.

above eg: (20KB) FL: $\boxed{10} \rightarrow \boxed{30} \rightarrow \boxed{30} \rightarrow \boxed{10} \rightarrow \boxed{5} \rightarrow X$

as pointer was here only

(10KB) FL: $\boxed{10} \rightarrow \boxed{20} \rightarrow \boxed{30} \rightarrow \boxed{10} \rightarrow \boxed{5} \rightarrow X$

c) best fit: allocates space which is just big enough to ~~avoid~~ internal fragmentation.

minimise

above eg:

FL: $\boxed{10} \rightarrow \boxed{50} \rightarrow \boxed{10} \rightarrow \boxed{10} \rightarrow \boxed{5} \rightarrow X$

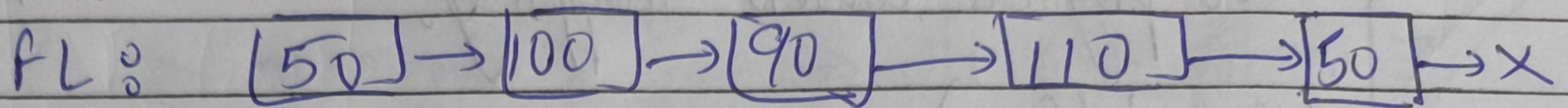
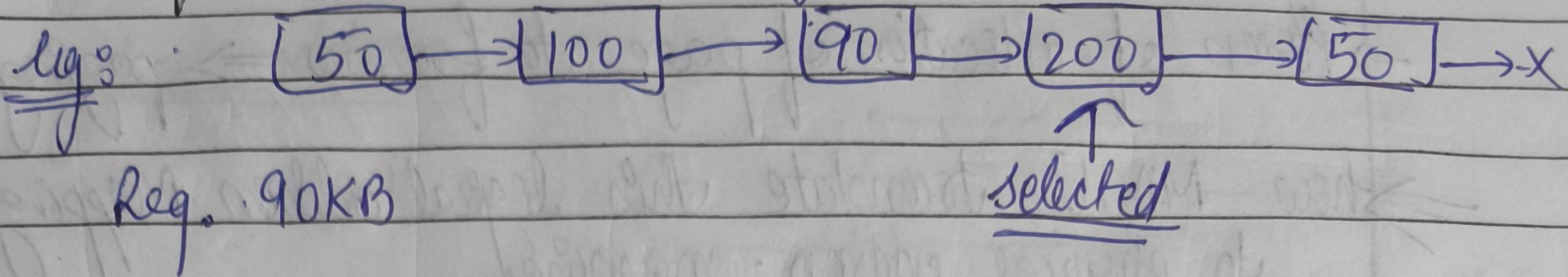
Adv: lesser internal fragmentation, slow & major external fragmentation.

Disadv:

Lesser external fragmentation than best fit.

→ Slow

④ Worst fit : allocate to largest hole.

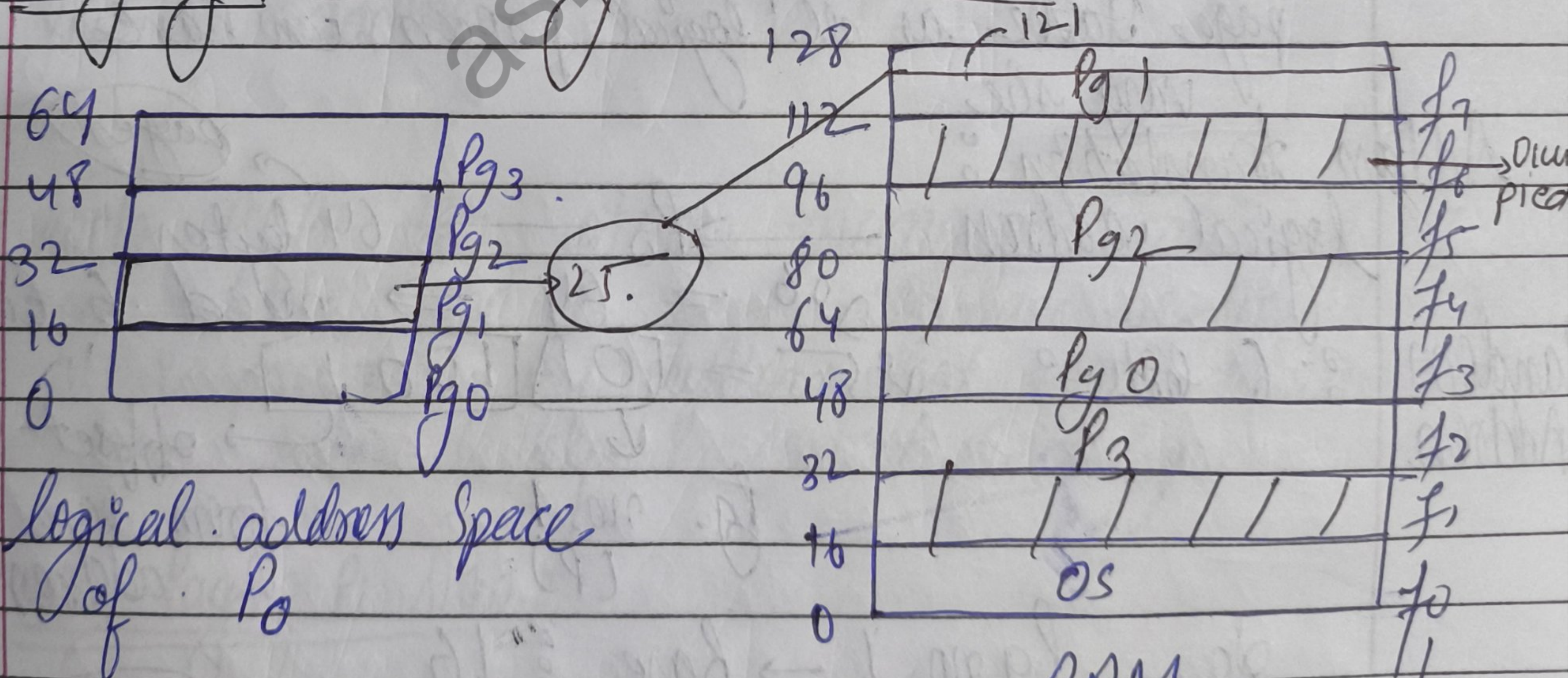


⇒ Each free list Node has :

- ① start-address
- ② size
- ③ next

* The OS keeps track of largest, smallest, best fit free hole depends on implementation and picks the node in single traversal only so T.C of all algo : $O(n)$

Paging / Non Contiguous allocation



16 Kb → size division boxes

Page : logical address fixed size division

frame : physical " " " "

So, Page Size == frame Size always

→ Page size depends on system architecture.

→ Process of allocation of page to frames $\Rightarrow$ paging.

→ how MMU translate this logical address Mapping to physical address mapping.

(Uses table)

logical page no.		Physical frame numbers
00	0	3 $\rightarrow$ 011
01	1	7 $\rightarrow$ 111
10	2	5 $\rightarrow$ 101
11	3	2 $\rightarrow$ 010

→ In memory, each process ~~has~~ has its own page table as all logical processes can have vary size.

Address translation:

logical address

$\xrightarrow{P_0} 64$ bytes

$2^6 = 64 \Rightarrow$ need 6 bits.

Random Address

$\therefore$ 6 bits $\therefore$

$2^5 = \boxed{011001}$

lg. no. 1
(P)

offset (d)
from base address

so, lg no. 1 $\rightarrow$ Base : 16

$\Rightarrow 16 + 9 = \boxed{25}$

logical address : $[p | d]$

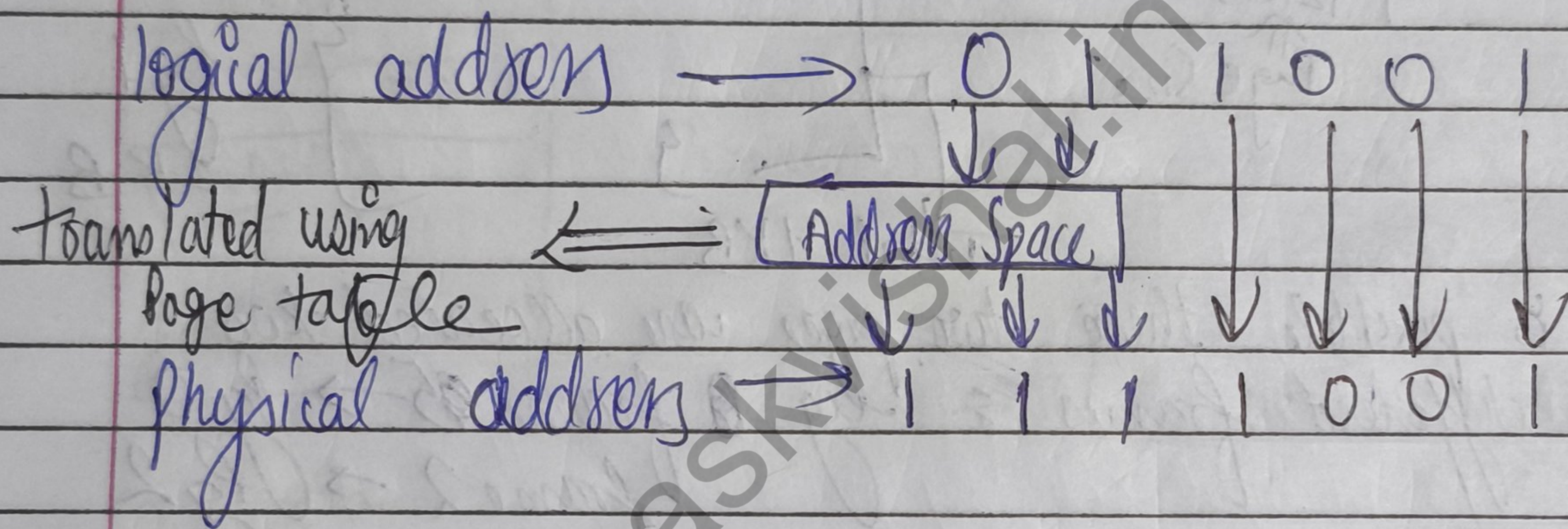
physical address :

121 $\Rightarrow$ $\boxed{111} \boxed{1001} \rightarrow d$

$2^3 = 8 \Rightarrow$ represented 3 bit $\Rightarrow$ '111' is frame 7

$\Downarrow$
No. of frames.

So, overall



$\Rightarrow$ each PCB ~~the~~ stores memory location of its page table and a page table base register (PTBR) is present in the system that points to current page table (Helps in context switching)

Why Paging is Slow:

$\rightarrow$ It uses too many memory references, multiple processes $\rightarrow$ diff page table $\rightarrow$ more context switching $\rightarrow$ more reference etc. Res

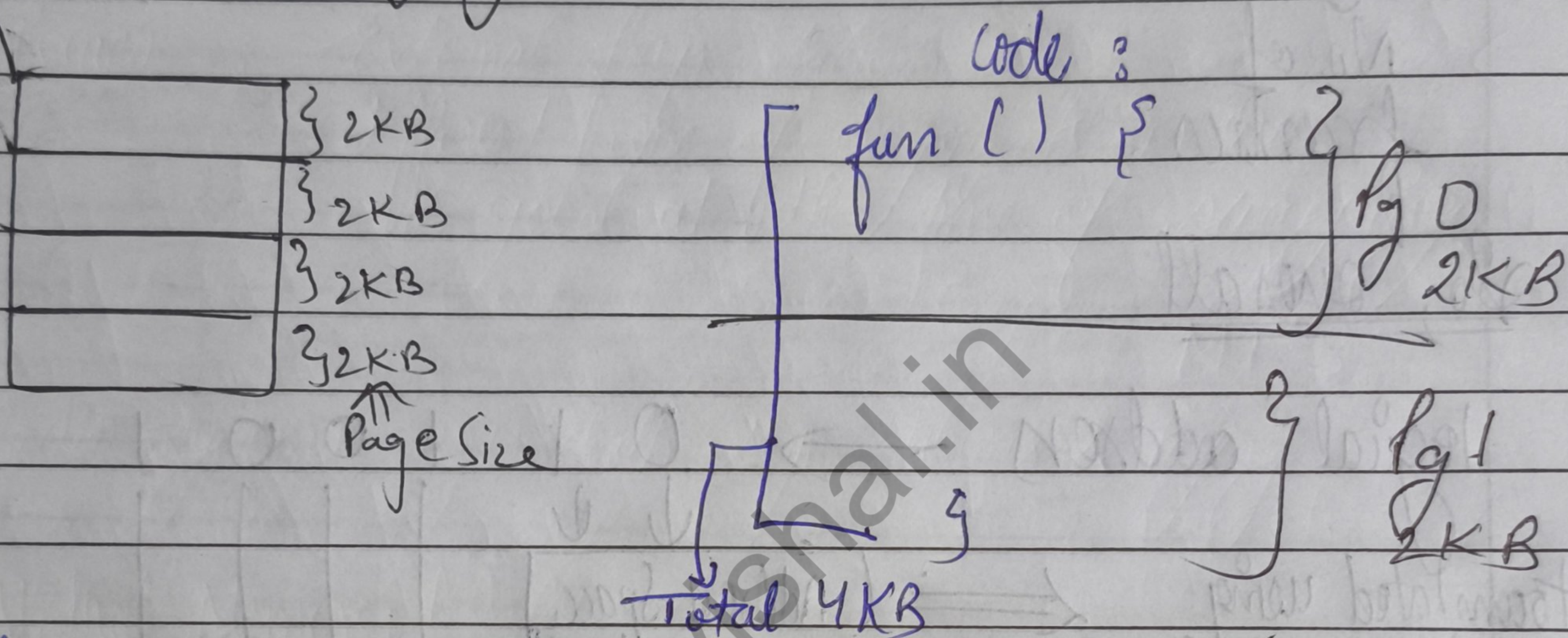
Not of overhead

$\hookrightarrow$ we use caching to reduce overhead.

Translation look ahead buffer (TLB)

- a) A hardware support to speed up paging
- b) It a hardware cache, high speed memory
- c) TLB has key & value.

Problem in Paging

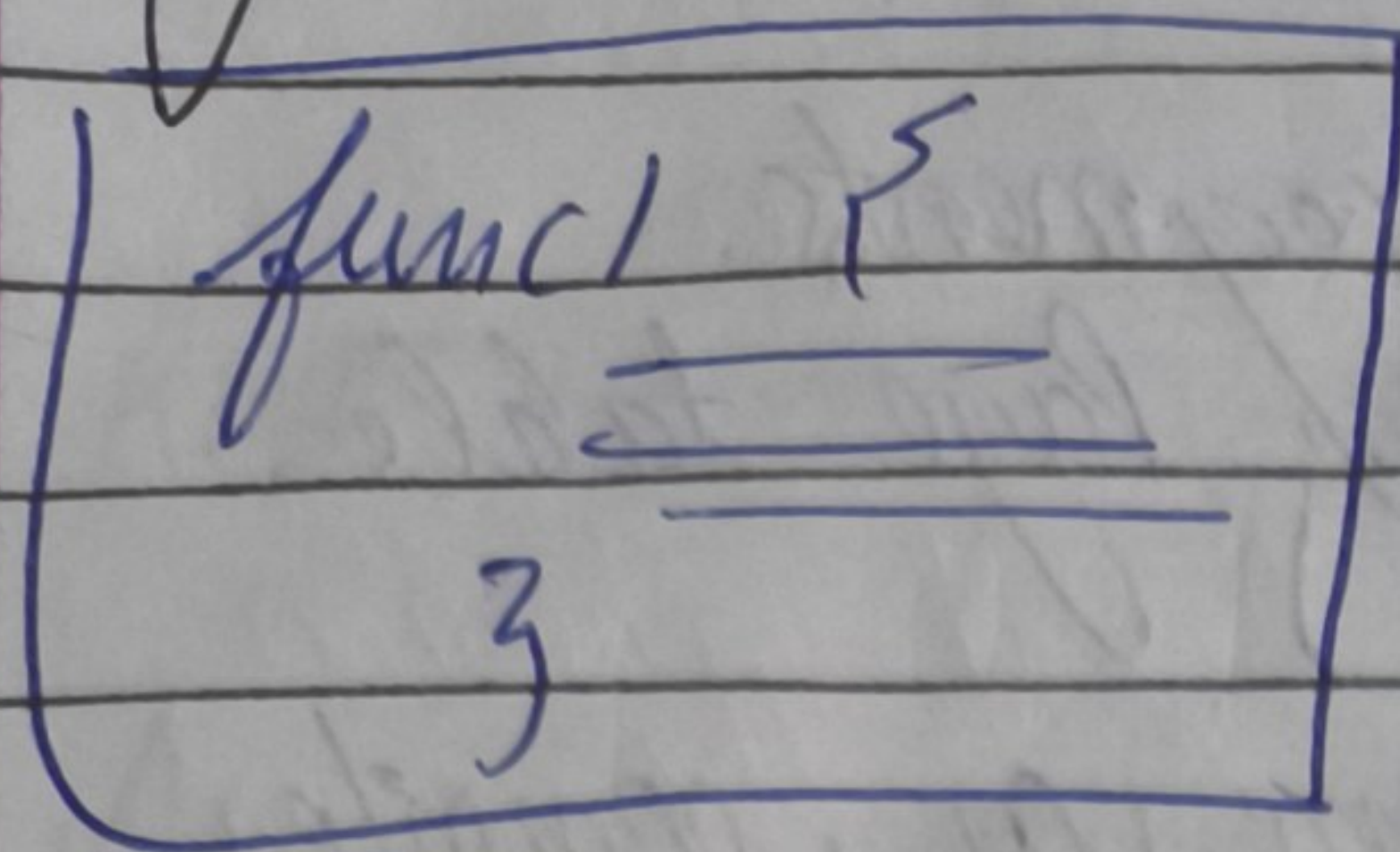


- * And is possible these two pages are allocated two different frames: eg frame 5 $\rightarrow$ Pg 0
frame 2 $\rightarrow$ Pg 2

So, when processor is executing instructions, after Pg 0 ends the MMU switches the frame & processor continues but this has overhead as now processor needs the reference of frame 5 as well because it's the same instruction. So, overhead. And, also inside the same frame execution is faster but here execution also becomes slower.

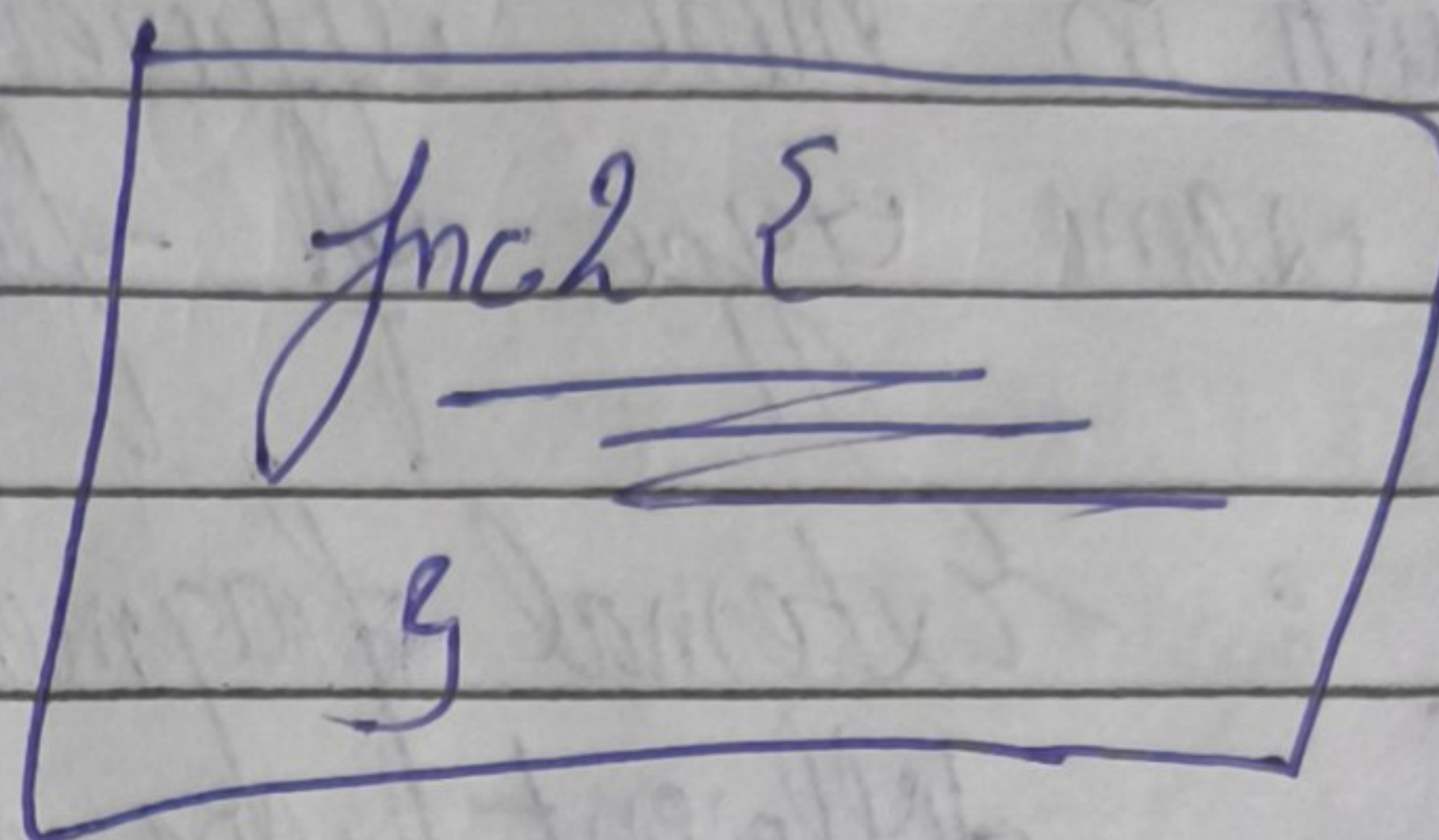
This issue is solved by segmentation.

Segmentation



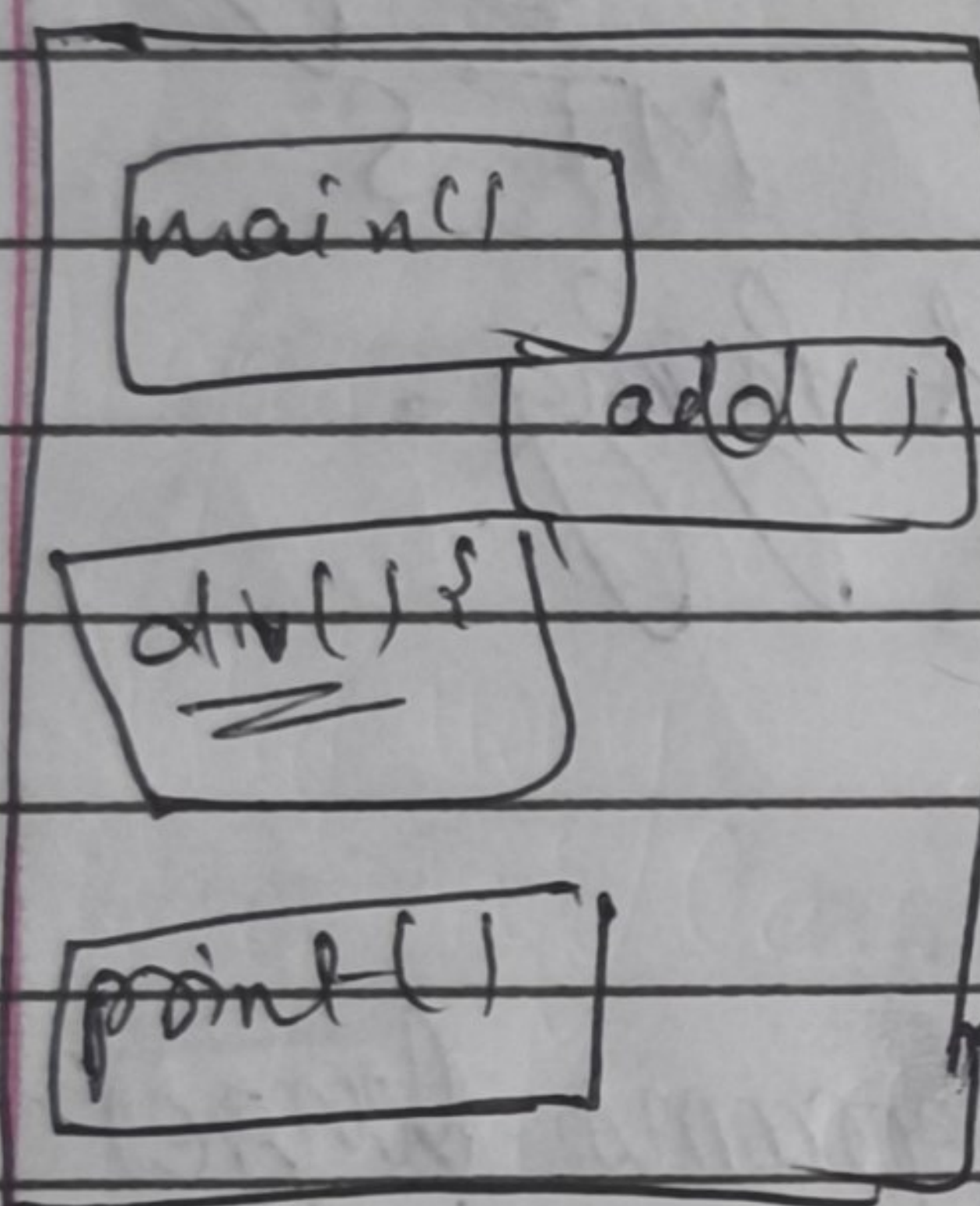
→ each fn is different segment of variable size
user view

Variable partition of logical address space



Segment

decided by Compiler.



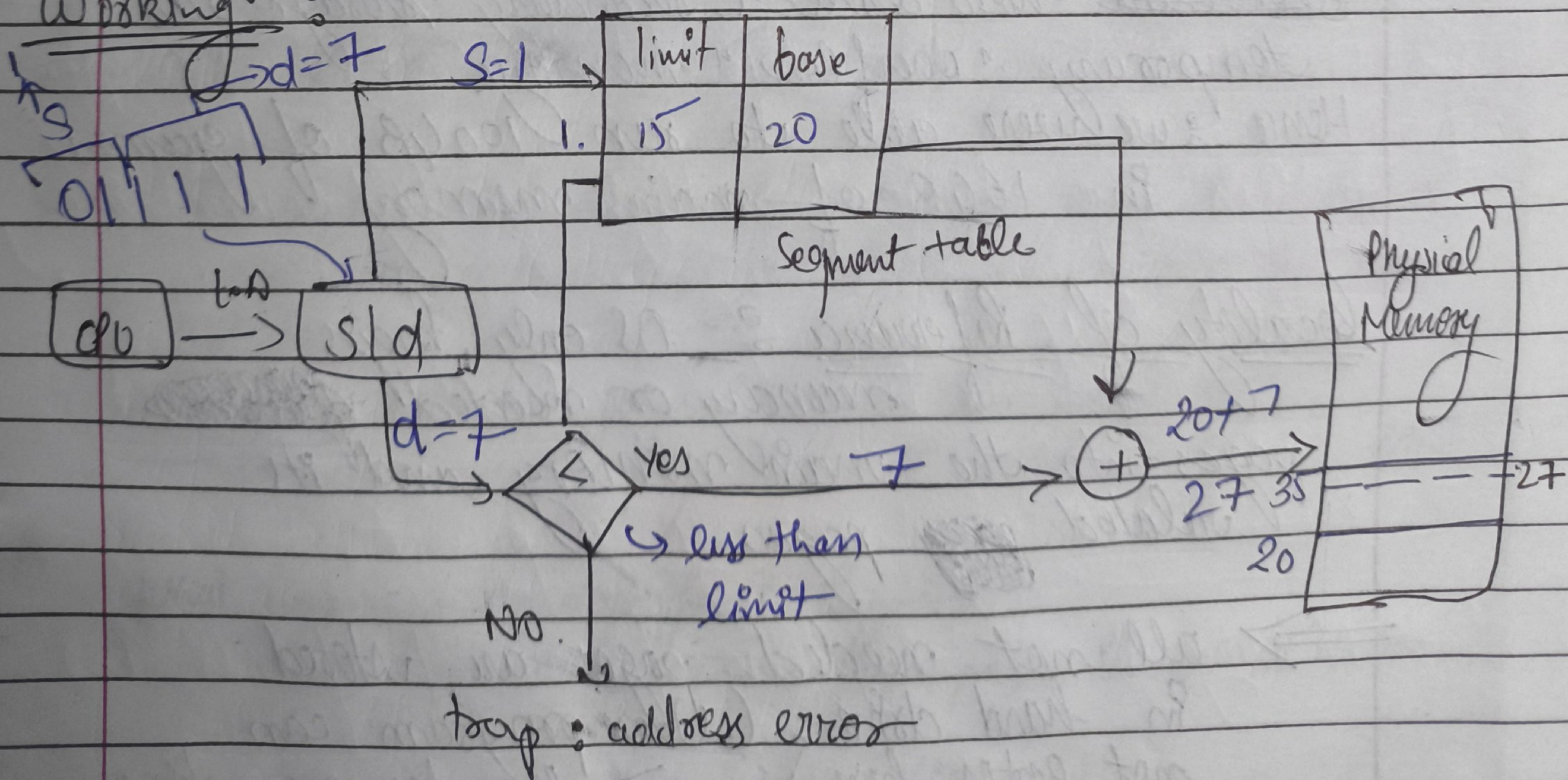
MMU

Translation $\Rightarrow$ logical address $\rightarrow$ physical address

L.A $\Rightarrow$ [Seg. no. | offset]

program

Working:



- * Segmentation also has external fragmentation but no internal fragmentation!
- * Contiguous allocation of each segments.
- * Size of segment table is less than page table (as less entries)
- * It results in more efficient system as compiler keeps same type of functions together.

Issues : External fragmentation
 Different size of segments is not good at the time of swapping
 MT S

Morden systems uses both segmentations & paging.

Virtual Memory Management

VMM allows a computer to run programs larger than its physical RAM by using a portion of the ~~hard drive~~ hard drive (secondary storage) as temporary "overflow" memory.

Hence : we were able to run 100GB of games in 16GB of main memory.

Locality of Reference : OS only keeps memory or needed ~~pages~~ pages into the main memory and its related ~~pages~~ pages.

⇒ all not needed pages are stored in hard drive (User program can not enter here) ⇒ for temporary time only but can be permanent.

⇒ This temp. storage is called Swap Space.

RAM + Swap Space = Virtual Memory.

Demand Paging:

It is used in VMM, when based on demand of the page (could be because of interrupt or pagefault). Wrapping of pages is done from swap space to RAM. ⇒ done by lazy swapper.

Which frame to be selected to be swapped with needed frame ⇒ Page Replacement Algorithms.

Note: Swapper term is technically incorrect, as it is used when the whole ~~process~~ ^{pages} ~~pages~~ ^{pages} is swapped, for pages. Pts ~~pages~~ ^{pages}.

⇒ ~~_____~~

bit

⇒ The valid-invalid scheme in the page table is used to distinguish between pages that are in memory & that are on the disk.

→ 1 means page is legal & in memory.

→ 0 means page is not valid or it's in disk.

What happens when page fault happens?

Req. page is not in RAM.

* Refer Notes. (It's already discussed)

Pure demand paging : All the pages are first kept in swap storage & fetched on page fault. (i.e. on demand)
(no page in RAM)

⇒ Problem a lot of page fault during starting of application, hence locality of reference is preferred

Advnt of VMM : Degree of multiprogramming ↑
user can run larger application with less physical memory.

Disadvnt. System can be slower (swapping takes time)
Thrashing may occur.

~~Page Replacement Algorithms~~ : term page hit : page found

(1) FIFO (First in First out)

→ allocate frames to the pages as it comes into the memory by replacing the oldest page.

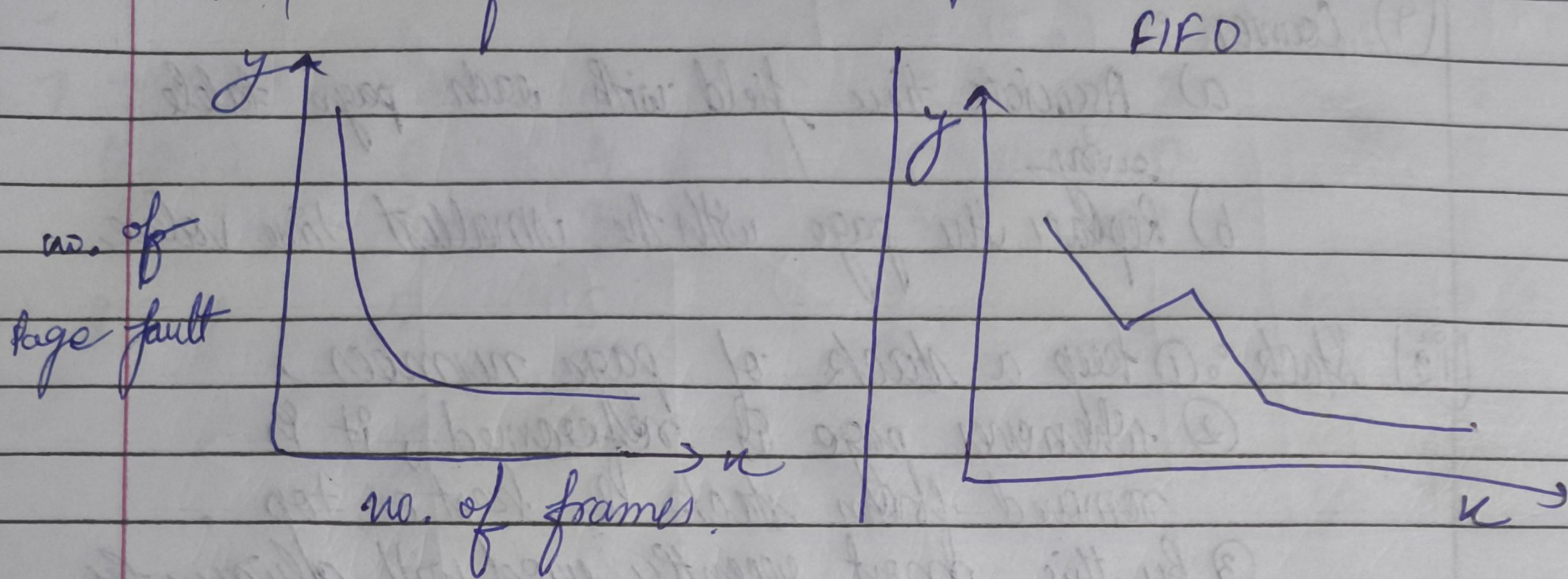
(Page fault Service time : Referral to the overhead of swapping)
⇒ High Page fault → High Page fault Service time

Advnt : easy to implement
performance is not always good (Variable)

Disadvnt : Belady's anomaly is present.

⇒ Normally in all algorithms if no. of frames ↑

page fault decreases but belady found it's not always true for FIFO.



② Optimal page replacement

→ Best algorithm $\Rightarrow$ lowest page fault rate.

→ Page fault minimum

→ Almost impossible to implement

OS require future knowledge

Working: find if a page that is never referenced in future. if such page exists replace this page with new page.

if no such page exists, find the page ~~with~~ that is referenced farthest in the future & replace it
(Similar to SJF)

③ LRU (least recently used)

→ approximation based on past.

→ So we can replace a page that has not been used for the longest period.

how to implement :

(9) Counter :

- Associate time field with each page table entry.
- Replace the page with the smallest time value.

(12) Stack :

- keep a stack of page numbers,
- whenever page is referenced, it is removed from stack & kept on top.
- by this, most recently used will always be on top & least recently used on bottom.
- As entries may be removed from the middle of the stack, doubly LL can be used.

kg of each algorithm :

(9) FIFO

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	4	4	4	0		0	0				7	7	7
	0	0	0		3	3	3	2	2	2		1	1				1	0	0
		1	1		1	0	0	0	3	3		3	2				2	2	1
1	2	3	4		5	6	7	8	9	10		11	12				13	14	15

Page fault : 15

OPR

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					0		
1	1			3		3		3		1							1		
1	2	3	4		5		6		7		8						9		

Page fault = 9

LRU

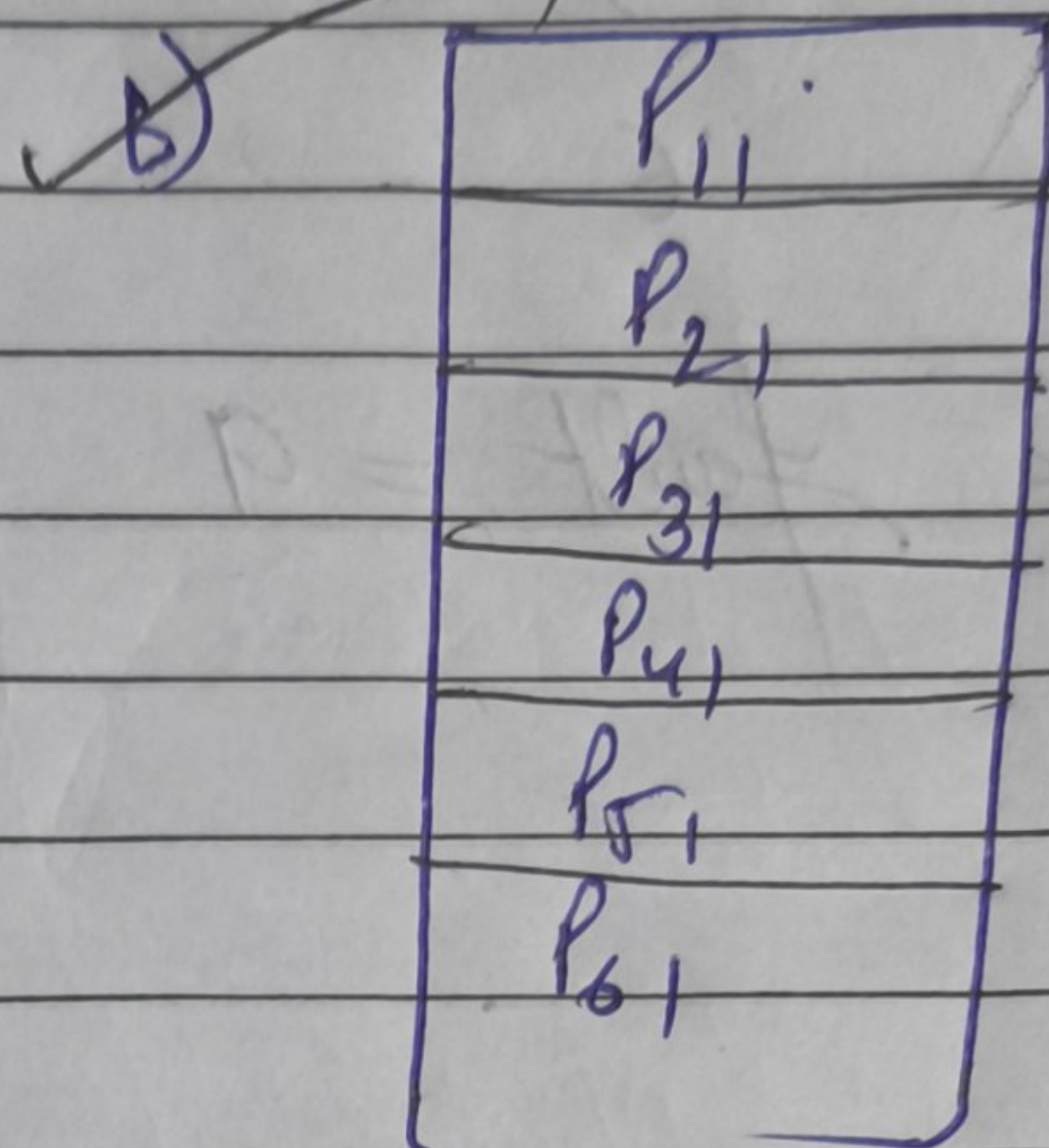
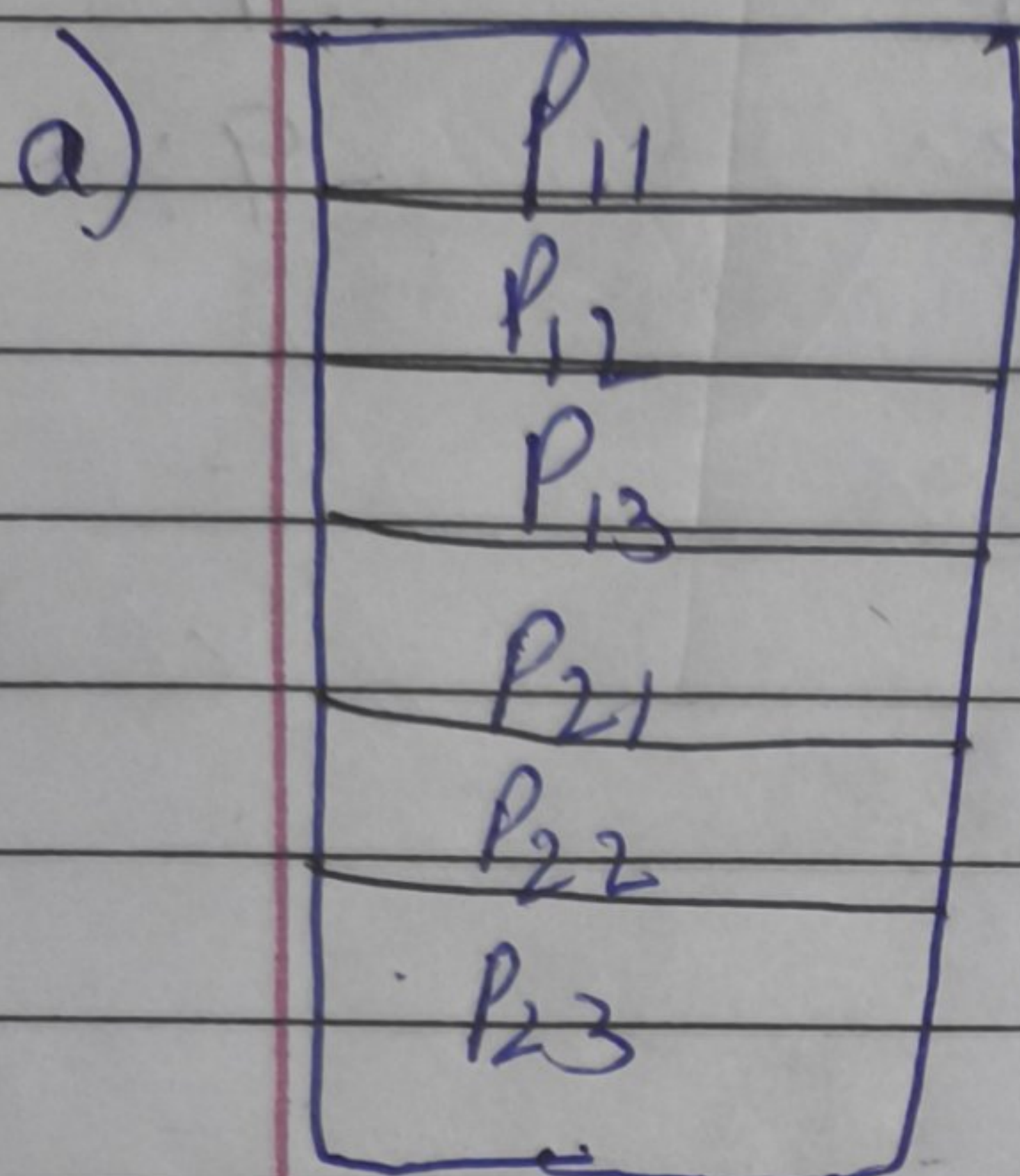
7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		4	4	4	0		1		1					
0	0	0		0		0	0	3	3		3		0				0		
1	1			3		3	2	2	2		2		2				7		
1	2	3	4		5		6	7	8	9		10		11			12		

Page fault = 12

Thrashing

P_{11} means P_1 has page 1.

which one has more no. of page faults?



RAM

$P_1 \rightarrow 3$ pages
 $P_2 \rightarrow 3$ pages
 DOP = $\downarrow$

RAM

$P_1 - P_6 \rightarrow$ each has one page.
 Degree of Multiprogramming $\uparrow$

Reason: b) P_{11} executes the next moment page fault for P_{12} and so on $\dots \rightarrow$ CPU will be more busy

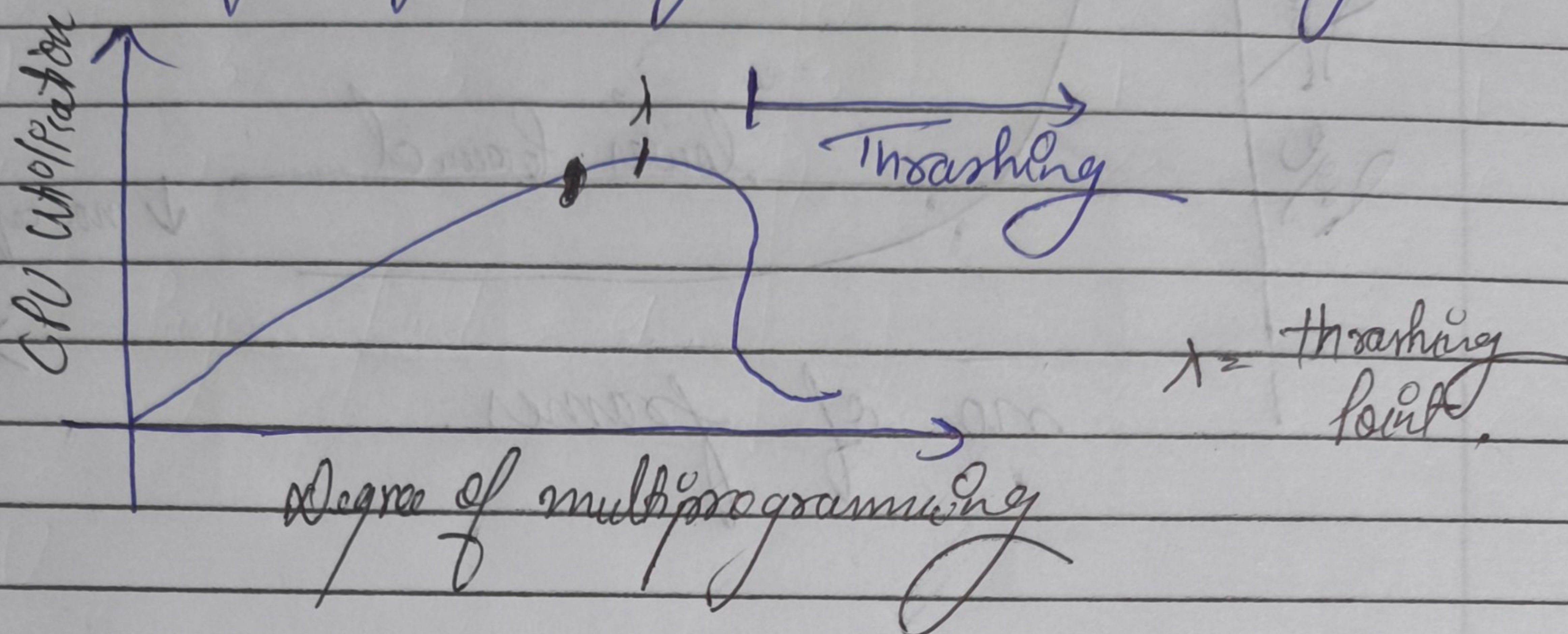
a) P_{11} executes, P_{12} then P_{13} (enough time & less page fault).

Thrashing : If a process doesn't have no. of frames it needs to support page activity in use, it will quickly page fault, it must replace some pages. However since all pages are in active use, it must replace (swap) a page that will be needed right away.

$\rightarrow$ Consequently, it quickly faults again & again.

again, replacing pages that it must bring back immediately $\Rightarrow$ very high overhead.

$\Rightarrow$ This high page activity is called thrashing.



Technique to handle thrashing:

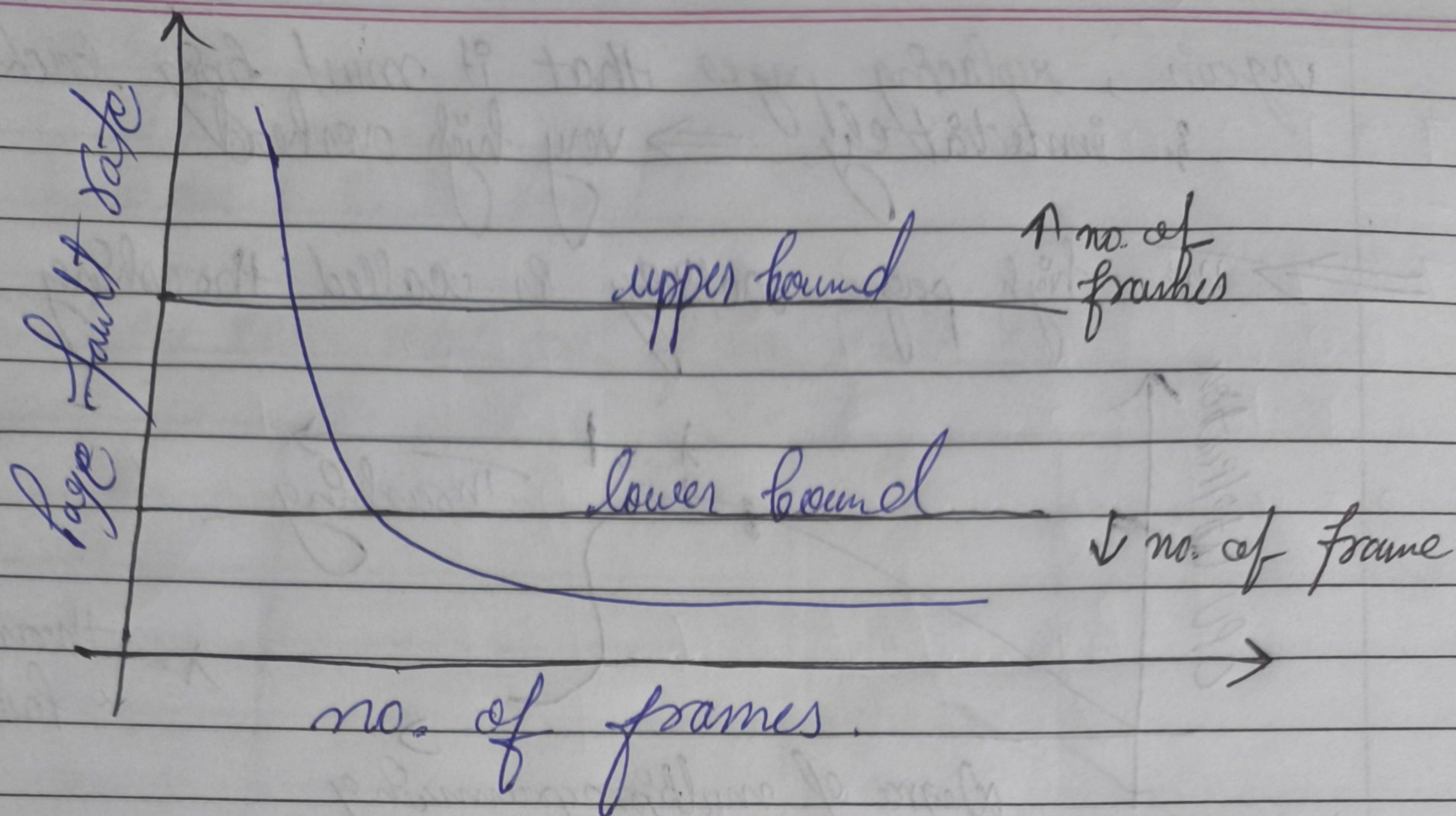
(i) Working Set Model : (locality model)

$\Rightarrow$ It states; if we allocate enough frames to a process to accommodate its current locality, it will only fault whenever it moves to different locality. But if the allocated frames are lesser than size of current locality, the process is bound to thrash.

(ii) Page Fault frequency :

$\Rightarrow$ Thrashing has a high page-fault rate, $\rightarrow$ to control it, when fault freq. is high (thrashing) needs more pages but when fault freq. is too low $\rightarrow$ we must have added too many pages into the frame.

Do
is low



Remaining topics of OS

→ file System Management

→ Storage Organisation

... if you are fully dependend on these notes
I do a bit research about these topics,
but they are no too much asked in
interviews.

Vishal
5/1/26