OPERATING SYSTEMS

UNIT 1
AN INTRODUCTION TO OPERATING SYSTEMS

Application software performs specific task for the user.
System software operates and controls the computer system and provides a platform to run application software.

An **operating system** is a piece of software that manages all the resources of a computer system,both hardware and software, and provides an environment in which the user can execute his/her programs in a convenient and efficient manner by hiding underlying complexity of the hardware and acting as a resource manager.

Why OS?

1. What if there is no OS?
    a. Bulky and complex app. (Hardware interaction code must be in app's code base)
    b. Resource exploitation by 1 App.
    c. No memory protection.
2. What is an OS made up of?
    a. Collection of system software.

An operating system function -

- Access to the computer hardware.
- interface between the user and the computer hardware
- **Resource management (Aka, Arbitration) (memory, device, file, security, process etc)**
- **Hides the underlying complexity of the hardware. (Aka, Abstraction)**
- facilitates execution of application programs by providing isolation and protection.

User

Application programs
Operating system
Computer hardware

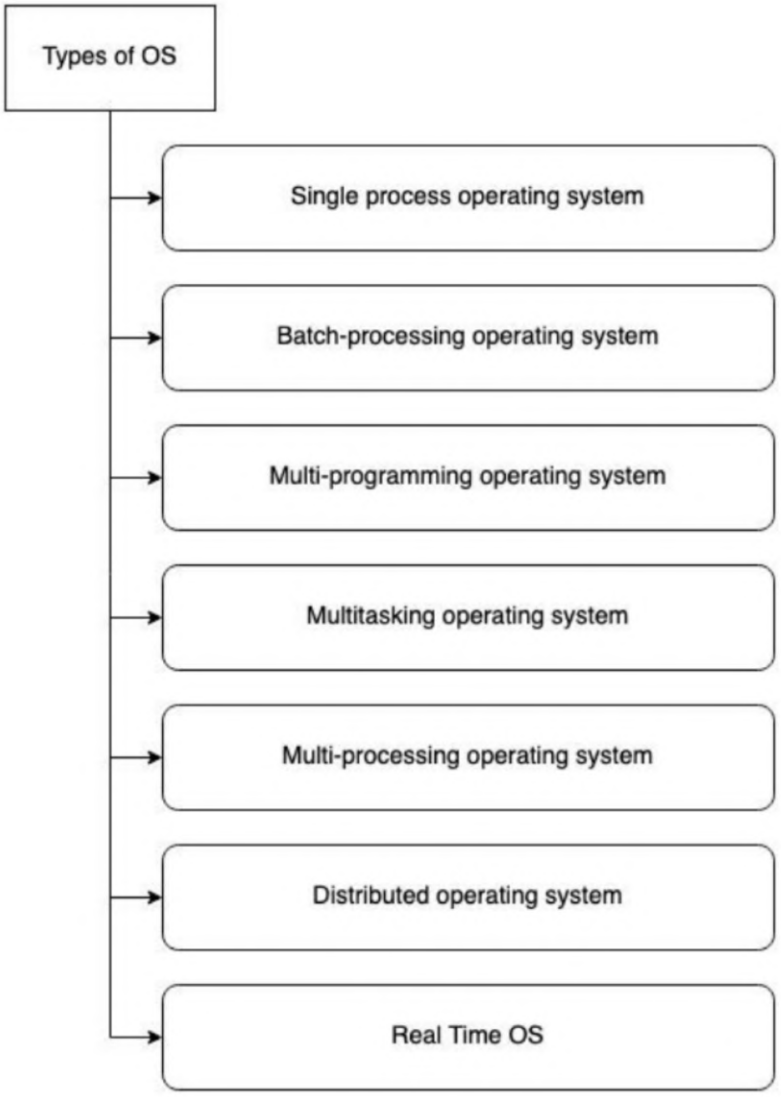
The operating system provides the means for proper use of the resources in the operation of the computer system.

OS goals –

- Maximum CPU utilization
- Less process starvation
- Higher priority job execution

Types of operating systems –

- Single process operating system [MS DOS, 1981]
- Batch-processing operating system [ATLAS, Manchester Univ., late 1950s – early 1960s]
- Multiprogramming operating system [THE, Dijkstra, early 1960s]
- Multitasking operating system [CTSS, MIT, early 1960s]
- Multi-processing operating system [Windows NT]
- Distributed system [LOCUS]
- Real time OS [ATCS]

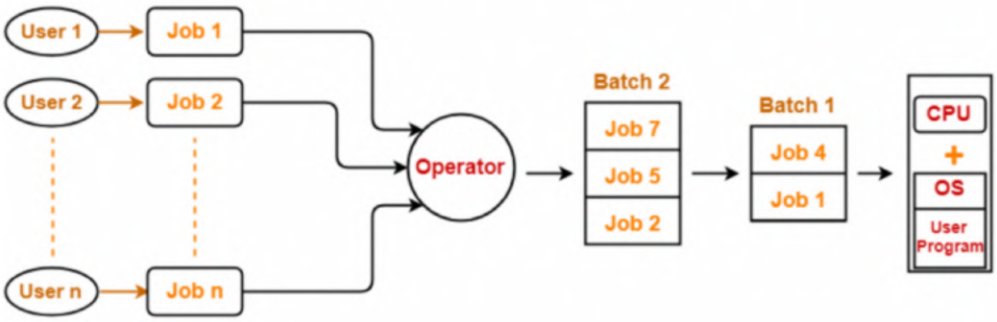


Single process OS, only 1 process executes at a time from the ready queue. [Oldest]

Batch-processing OS,

1. Firstly, user prepares his job using punch cards.
2. Then, he submits the job to the computer operator.
3. Operator collects the jobs from different users and sort the jobs into batches with similar needs.
4. Then, operator submits the batches to the processor one by one.
5. All the jobs of one batch are executed together.

- Priorities cannot be set, if a job comes with some higher priority.
- May lead to starvation. (A batch may take more time to complete)
- CPU may become idle in case of I/O operations.



Multiprogramming increases CPU utilization by keeping multiple jobs (code and data) in the**memory** so that the CPU always has one to execute in case some job gets busy with I/O.

- Single CPU
- Context switching for processes.
- Switch happens when current process goes to wait state.
- CPU idle time reduced.

Multitasking is a logical extension of multiprogramming.

- Single CPU
- Able to run more than one task simultaneously.
- Context switching and time sharing used.
- Increases responsiveness.
- CPU idle time is further reduced.

Multi-processing OS, more than 1 CPU in a single computer.

- Increases reliability, 1 CPU fails, other can work
- Better throughput.
- Lesser process starvation, (if 1 CPU is working on some process, other can be executed on other CPU.

Distributed OS,

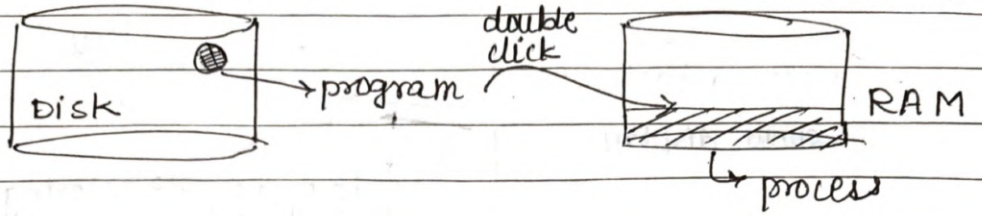
- OS manages many bunches of resources, >=1 CPUs, >=1 memory, >=1 GPUs, etc
- **Loosely connected autonomous,** interconnected computer nodes.
- collection of independent, networked, communicating, and physically separate computational nodes.

RTOS

- **Real time** error free, computations within tight-time boundaries.
- Air Traffic control system, ROBOTS etc.

Chapter - 3

.exe → double click → process (program under execution)
(used to run a program)



NOTE: A computer program is nothing but a set of instruction (smallest unit of execution) that are used to execute particular tasks to get particular results.

Computer program Terminologies :

1.) Source Code - (.cpp file for eg) written in high level language

2.) Machine code - Binary code that computers or machines understand.

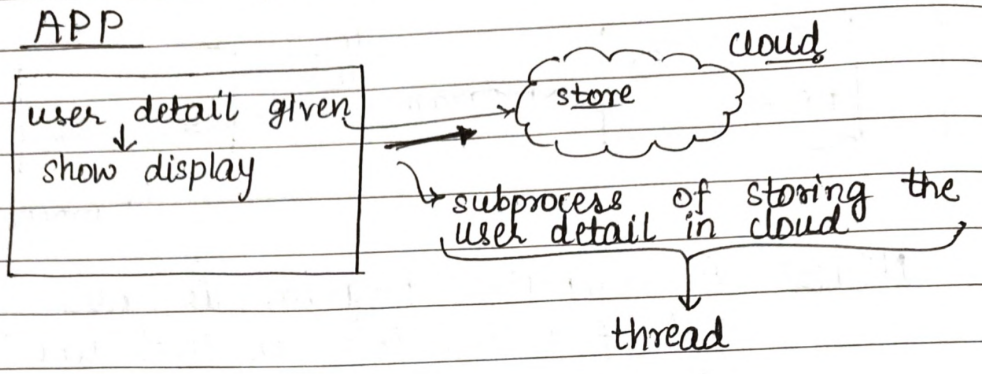
3.) Compiler - translate source code to machine code

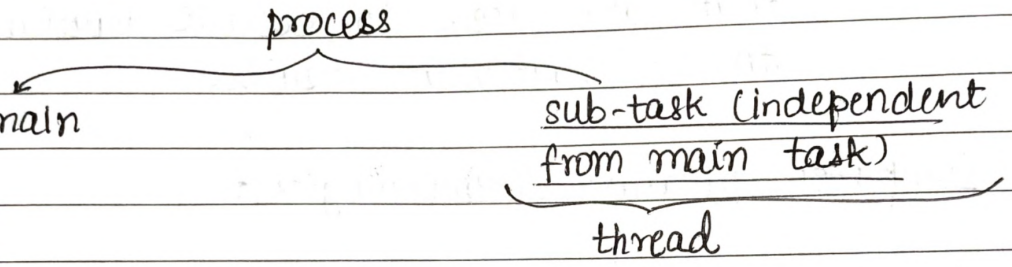
4.) Interpreter - executes the source code line by line by converting it into machine code.

5.) Algorithm - set of instruction to solve a problem.

Thread → light weight process.
→ independently executes
eg: Multiple tabs in a browser.

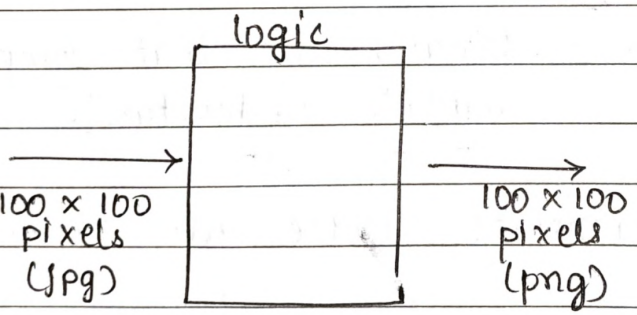
eg: APP





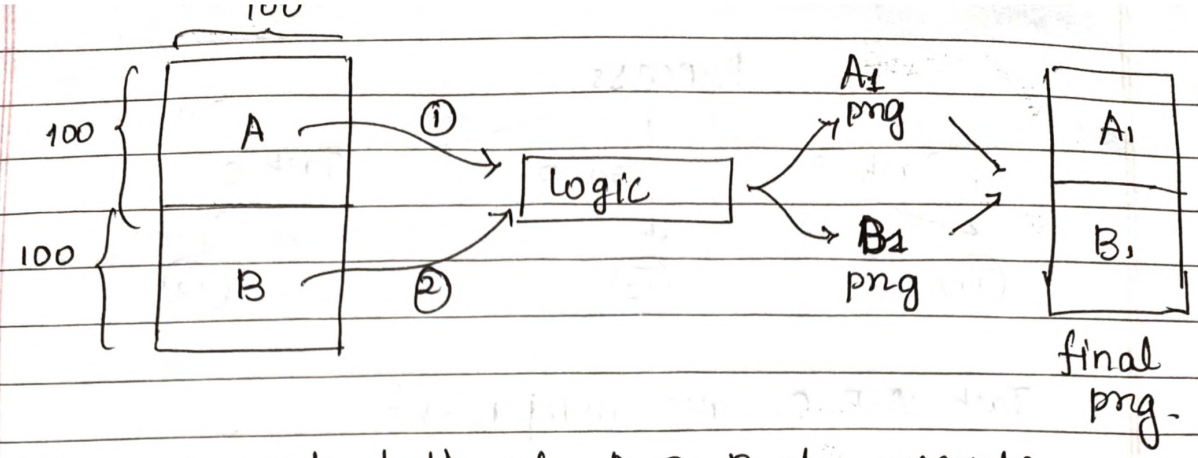
eg:

JPG to PNG convertor

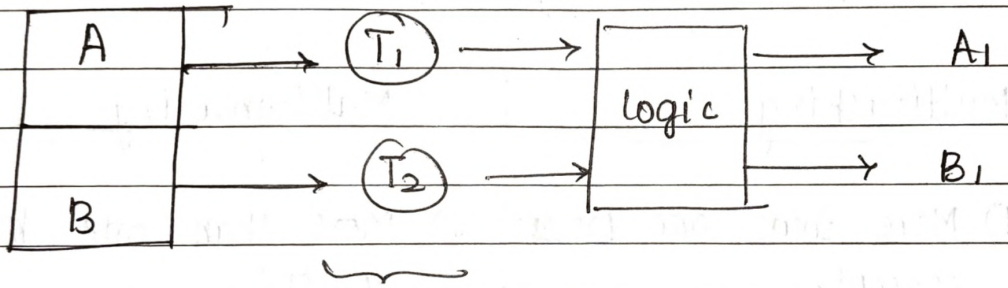


Suppose we have an image file of 100 x 200 pixels.

Here, we can divide the image and do the iteration sequentially and then again concatenate the image.



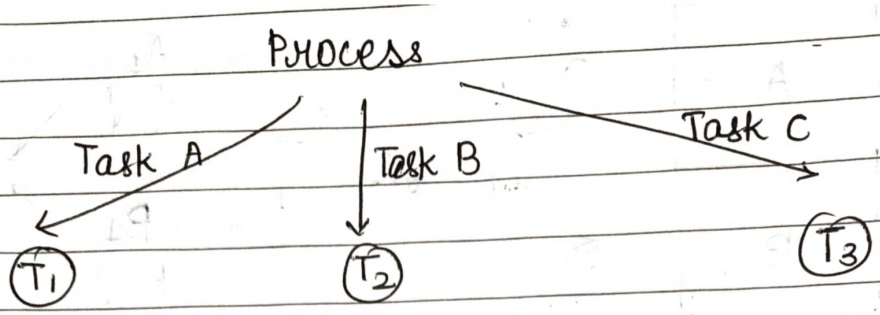
Now, we want both of A & B to execute simultaneous as processing of both A and B are independent. This can be acheived through multithreading.



executed parallely (not sequentially). So, time of execution is reduced by half.

But, here to execute T_1 & T_2 simultaneously we will require 2 CPUs. In case we have only 1 CPU, multithreading has no gain.

→ We need multiprocessing OS (>= 1 CPU)



Task A, B, C are independent.

All threads uses the same memory allocated to the process.

So, here there is no isolation.

Multitasking	Multithreading
1) More than one process concept.	1) More than one thread concept.
2) Isolation & memory protection b/w processes	2) No isolation or memory protection.
3) Process scheduling	3) Thread scheduling
4) No. of CPU - 1	4) CPU >=1 (better to have >1 CPU)
	Threads are scheduled for execution based on their priority.

Date ___ / ___ / ____

no. of cores in a CPU = maximum no. of threads that can be used in a process

eg: Macbook air has only 8 cores.

Difference between Thread context switching and process context switching

Thread context switching	Process context switching
OS saves current state of thread and switches to another thread of same process.	OS saves current state of process & switches to another process by restoring its state.
Doesn't include switching of memory address space.	Includes switching of memo address space.
Fast switching.	Slow switching.
CPU's cache state is preserved.	CPU's cache state is flushed.

Chapter - 4

Components of OS :

1) User space → App are ran here.
2) Kernel → access to underlying hardware.

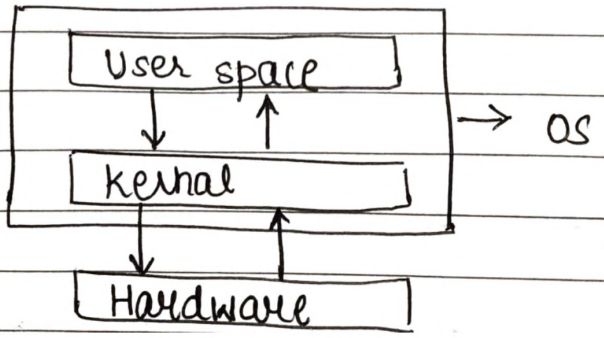
User space (GUI) / CLI
↳ no hardware access
↳ provides a convienient environment to user apps

GUI → graphical user interface.
CLI → command line interface.
(Terminal / Powershell)

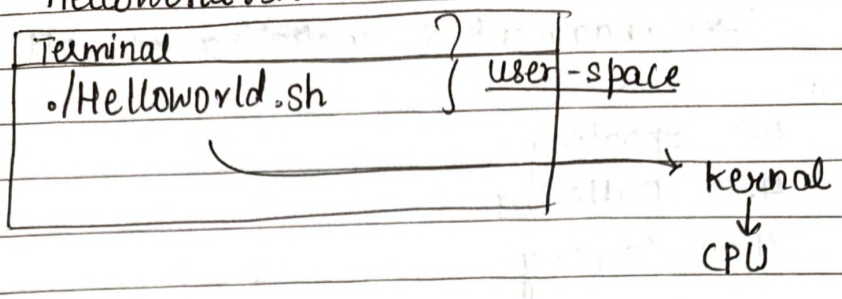
mkdir → system calls → storage managment
↓
Make directory ← File management

User space interacts with kernal

Kernel - Heart of OS
- Interacts with hardware



eg: Helloworld.sh



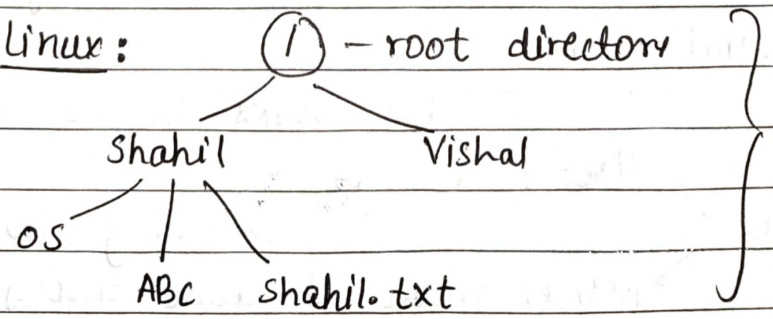
Functions of Kernel :

1) Process management
   - process creation / termination
   - process & threads schedule (context/ time switching
   - process sync
   - process communication

2) Memory management
   - allocation / de-allocate
   - free space management.

3) File management
   - creation/ deletion of files
   - directory management

Linux:



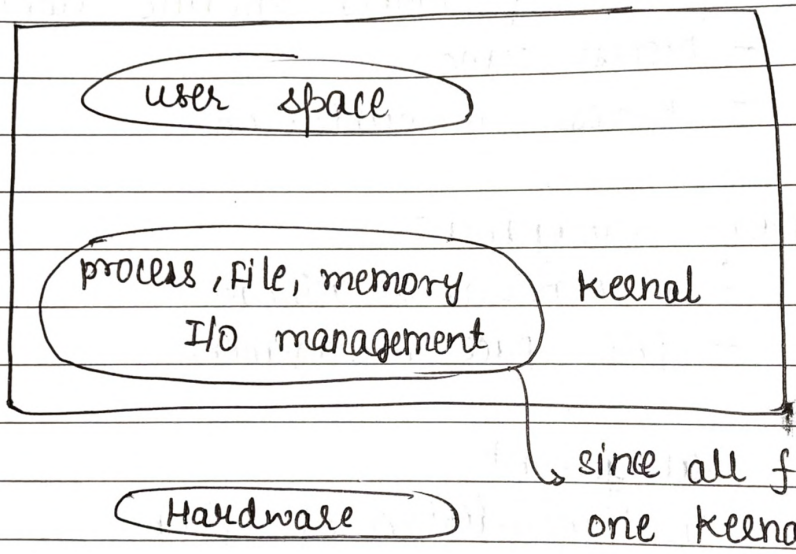
④ I/O management

→ management & controling of all I/O devices

4.1) Spooling
4.2) Buffering
4.3) caching

Types of Kernal

1) Monolythic Kernal (oldest)



since all fns are inside one kernal, switching b/w them are fast & easy.

* user mode
  kernel mode

CPU works in 2 modes

CPU → User mode
CPU → Kernel mode

switching b/w these mode is done through software interrupt.

eg: CLI

mkdir ——— ENTER PRESS → switched to kernal mode to make a directory (file management)

writing command is done in user mode

again switched to user mode

In monolytic kernal, there is fast communicaton b/w components

disadvantage → Bulky as all functions are inside one kernal

all fns are linked { less reliable as if one component gets corrupted, entire kernel becomes useless.

eg: UNIX, LINUX, MS DOS are monolythic

2) Micro kernel :

File / IO management — user space

only core function { Memory management, Process management — kernel space

Now, kernal is less bulky and more reliable/stable

Disadv

less performance } lot of switching b/w user mode & kernel mode

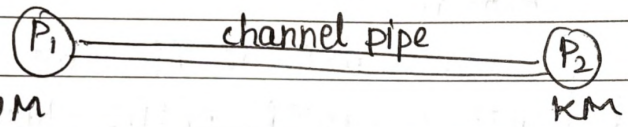
eg: L4 Linux , Symbian OS

Q How user mode & kernel mode communicated bhw each other?

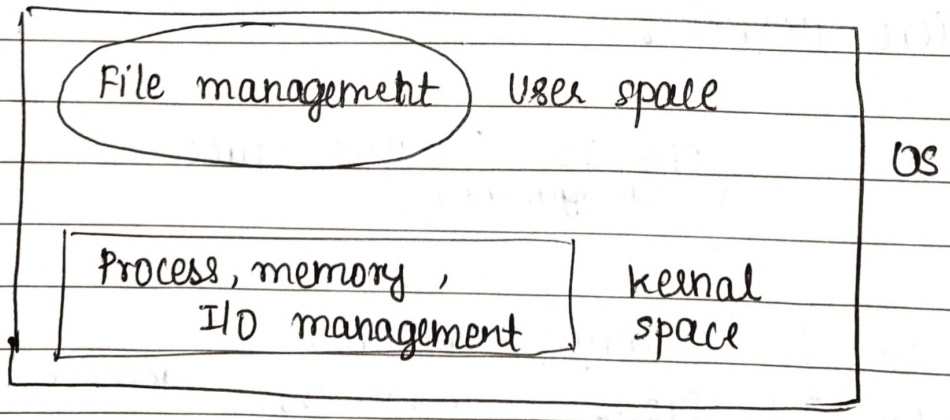
Ans → IPC (inter process communication)

1) Shared memory & through communication is established
2) Message passing

} two ways for IPC

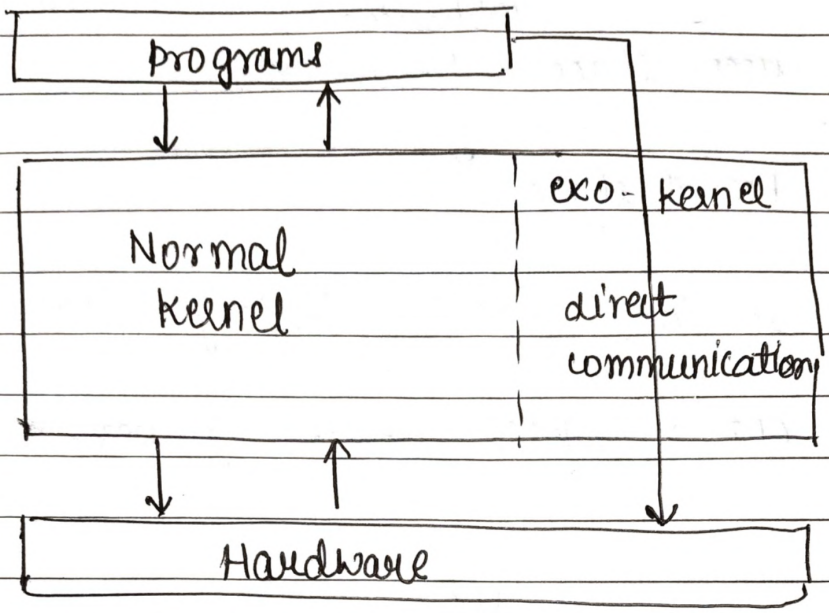


3) Hybrid kernel
- combined approach



eg: MacOS, Windows NT (7+)

4) Nano / Exo Kernel



Exokernels - programs can communicated with hardware much more directly. (significant performance increase)

* Disadv - Complex design & inconsistency.

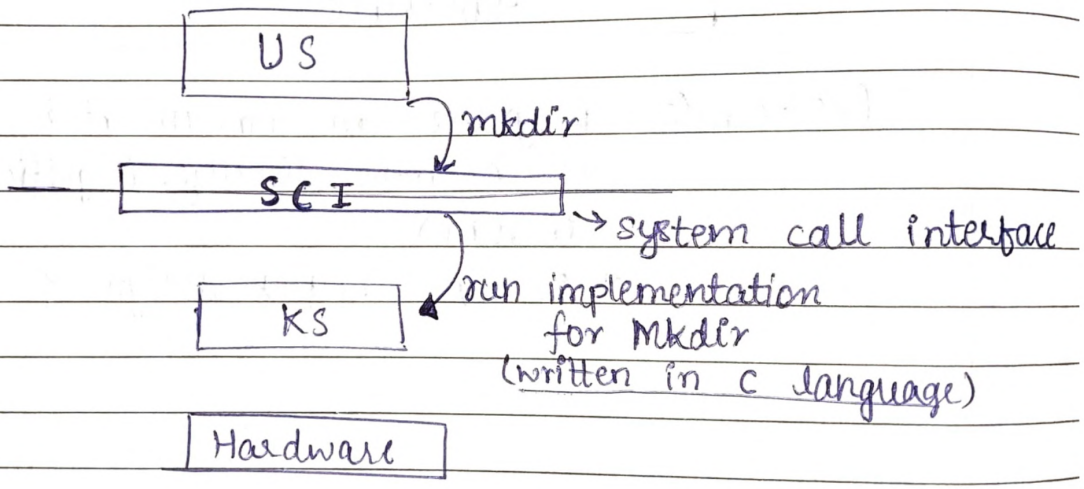
Chapter - 5 : (2:20:20)

System calls : mechanism to interact with kernal space

(application)
user space | system call
kernal space

eg: ~~matlab~~ create directory [movies]

CLI → mkdir movies } user - space



system calls are implement in C language.

eg: Helloworld.exe (double click)

↓

system call

↓

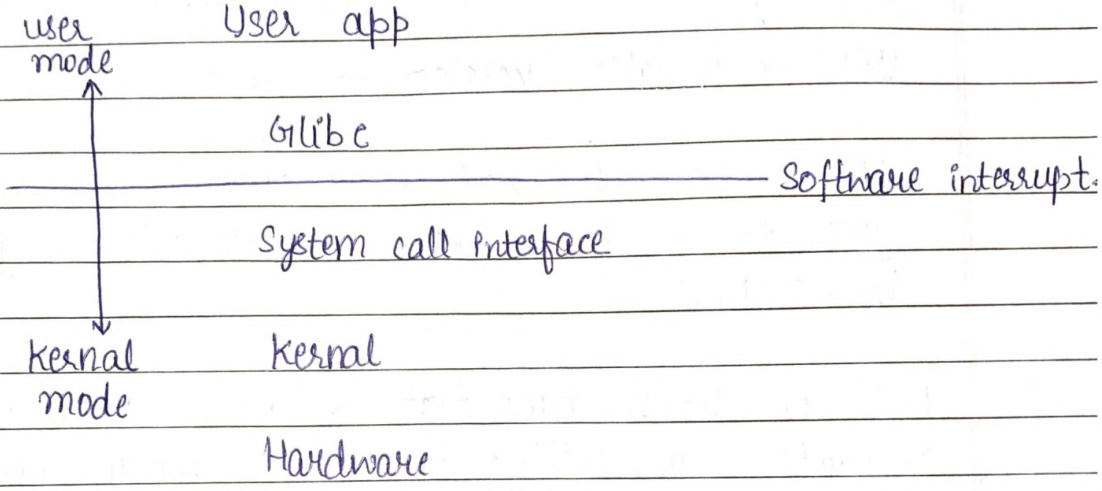
Kernal mode (process management)

↳ create process / allocate memory in RAM...

again switch from kernal mode to user mode.
GUI will show the execution of process.

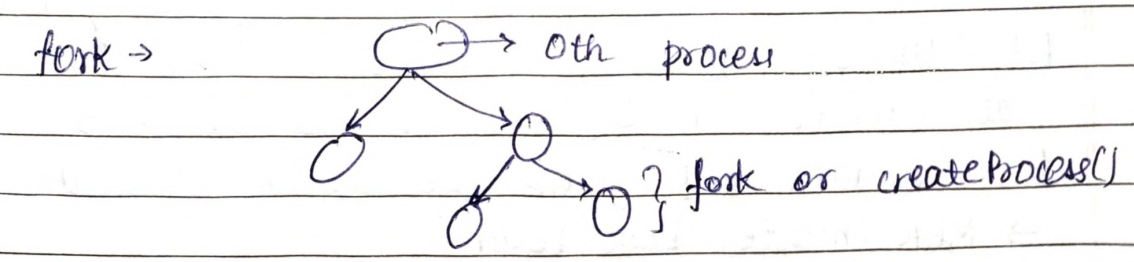
Q. How switching happens b/w user mode & kernal mode?

↳ software interupt.



Types of System calls

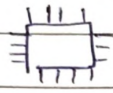
1) Process control eg: fork() / createProcess()
2) File management
3) Device management
4) Information management
5) Communication management



Chapter-6: How Operating System Boots up:

5 step process (very important)

1. Power-ON:
   - Power-supply → Mother board → Hardware

2. CPU loads BIOS or UEFI:
   -  } non volatile chip (BIOS chip)
   - UEFI → updated version of BIOS

   CPU gets initialized then CPU goes to a BIOS chip. It loads the program which is inside BIOS

3. BIOS or UEFI runs test & init Hardware
   - loads some setting from a memory area.
     - cell inside CPU  ← Backed by CMOS Battery
       - used to get correct time in OS.
   - BIOS program loads with setting
     [POST - Power on self test]
     POST is done for each hardware.
     eg: checking if RAM is working or not.

4) BIOS/UEFI hands off to Boot device (stores Boot loader)
   Boot devices - HDD/SSD or CD or USB
   Boot loader → program executes → ON actual OS.
   - MBR (master boot record)

Boot loader is stored in

MBR (master boot Record) — stored at 0th index of Disk — used in BIOS

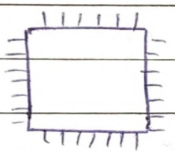
EFI — stores Boot loader in a partition. (used in UEFI)

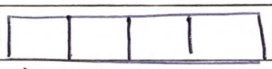
5. Boot loader loads the full OS

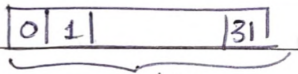
↳ program that launches the actual OS

- Windows - bootmgr.exe
- mac OS - boot.efi
- Linux - GRUB

Chapter - 7 : Difference between 32-bit & 64-bit OS





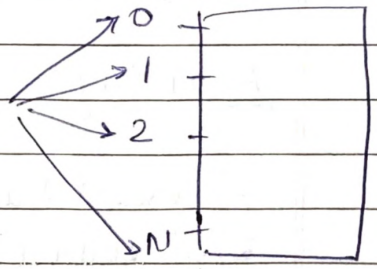
for 32 bit 

4 byte } it can store 4 byte of data

for 64 bit 

8 byte } it can store 8 byte of data

2^{32} addresses can be located in 32 bit CPU



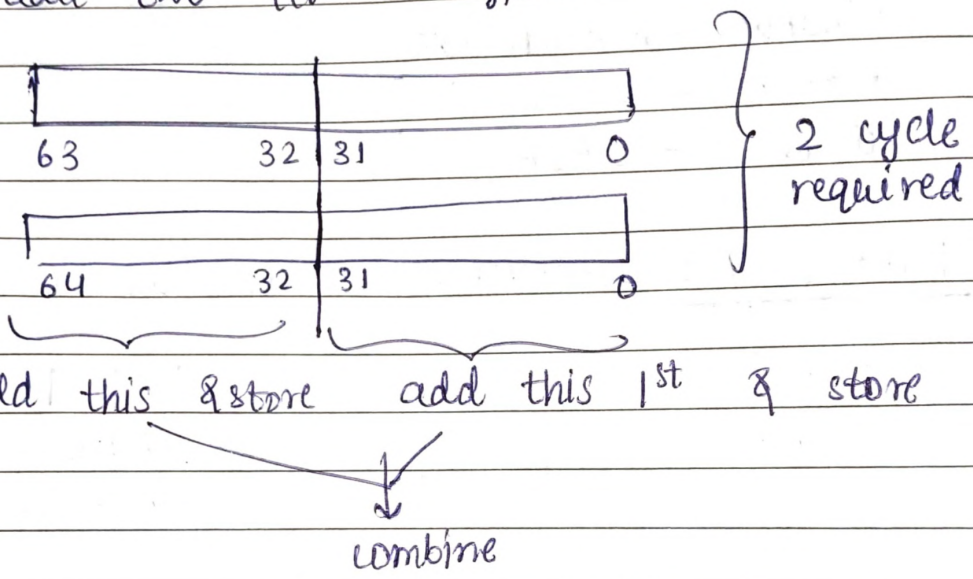
2^{32} addresses = 4 GB addresses

So, 32 bit CPU can store/hold only 4GB RAM.

But for 64 Bit CPU, RAM that it can hold is 2^{64} bits = 17179869184 GB adresses

NOW,
32 bit } store 32 bit at a time

Adding two int can be done but how will it add two 'lli' datatype



Hence, 64 bit can do the same thing in 1 cycle.

∴
performance wise 64-bit > 32-bit.

Advantages of 64 bits :

1) Addressable spaces (more RAM)
2) Resource uses (in 32-bit, we cannot use more than 4 GB ram)
3) Performance - 64 bits can process larger data in less time.

4) Compatibility : 64-bit can run both 64-bit & 32-bit OS

5) Better Graphics performance.

Chapter - 8 : Comparision b/w different STORAGES

Types of storages

CPU ⇅

(32 bit/64 bit) Register
(temporary data) Cache
(RAM) Main Memory
} Primary memory

Electronic Disk
Magnetic Disk
Optical Disk
Magnetic Disk
} Secondary memory (programs/audio/video are stored here)

Comparision

COST : Register > Cache > Main Memory > secondary

Access speed : Register > Cache > Main memory > secondary

Storage size : Secondary > Main > Cache > Register

Volatity : Primary storage fluses off after shutting down.

Chapter-9 : How OS creates a process.

Process - ? (helps in certain task)
Program (.cpp) → compiler → .exe ← program
↳ ready to execute
Program resides inside Disk.

.exe $\xrightarrow{\text{double click}}$ program under execution (called process)

OS creates a process from program. How??

S1: Load the program & static data, to main memory
used for initialization

```
char *name = "Shahil";
```
→ static data

S2: Allocate Runtime stack (allows recursion)
Stack is a part of memory used for local variable, function argument & return value

S3: Allocate Heap
Heap is a part of memory used for dynamic allocation

S4: IO Task
eg: Unix - input reading (handle)
output handle
error handling
eg: fprintf (stderr, "Hang");
↳ error handler.

OS calls main

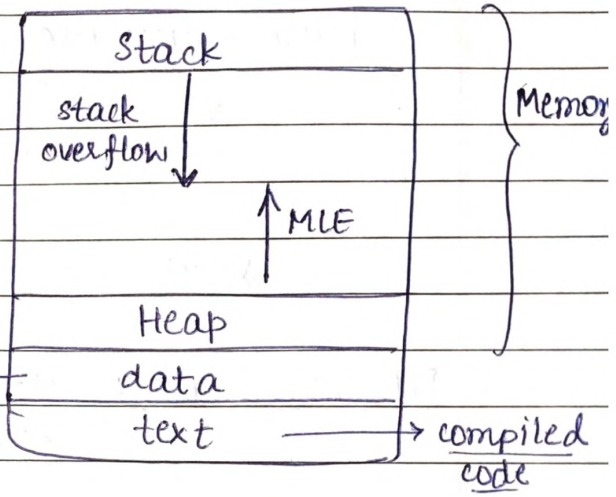
SS: main()
OS handoff control to main(). } process is created.

NOTE: main() {
return 0; } if not successfull execution, 0 will not be returned.
}

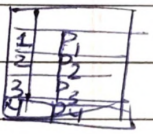
* Architecture of process:

1) stack overflow
↳ write better base case

2) Out of Memory
↳ deallocate unnecessary object.



* Attributes of Process:



P_1 P_2 P_3 P_4 → OS creates process Table.
↳ each entry is called PCB (process control block)

PCB

Process ID	→ unique identifier for a process
Program counter	→ pointer to instruction number
Process state	→ state of process (new, wait, run)
Priority	→ priority of the process
Register	→ stores the state when process is context switched
open file. list	} File/ device descriptors
open Device list	

Date ___/___/_____

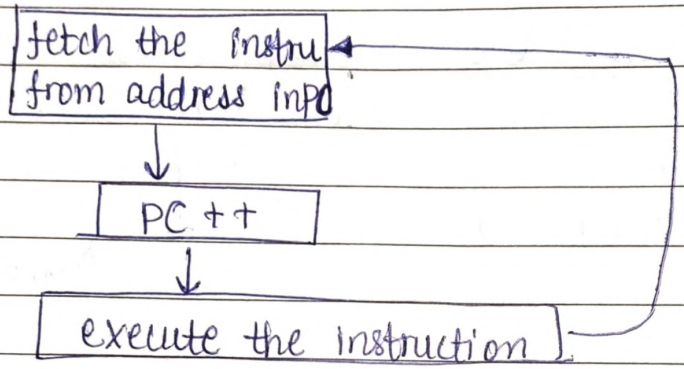
Chapter - IV

Program Counter (PC)

{ 1) _____
2) _____
3) _____ } — after compiling → get converted into instruction

(eg 1 2 ADD)
① Instruction 1
② " 2
③ " 3

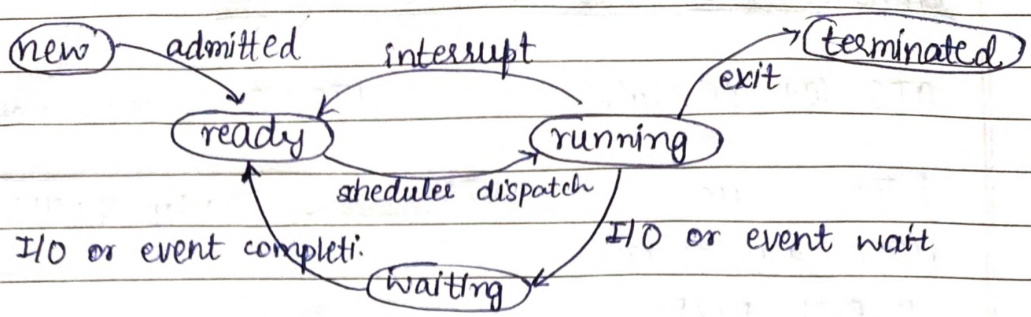
program counter points to the instruction no. which is currently getting executed.

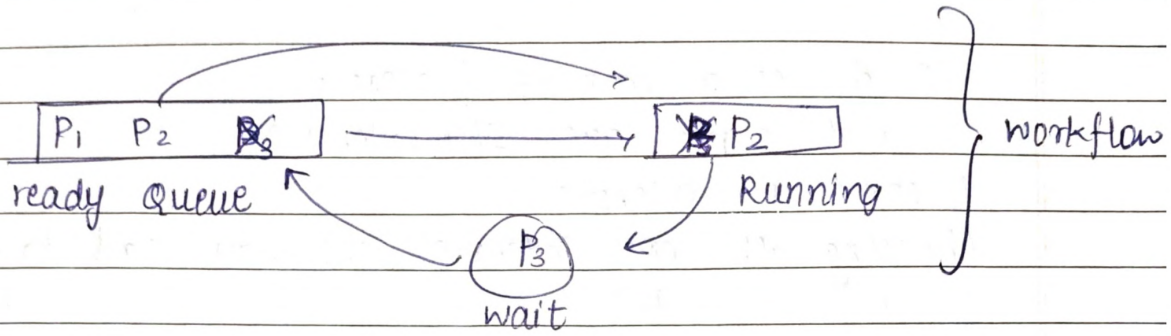


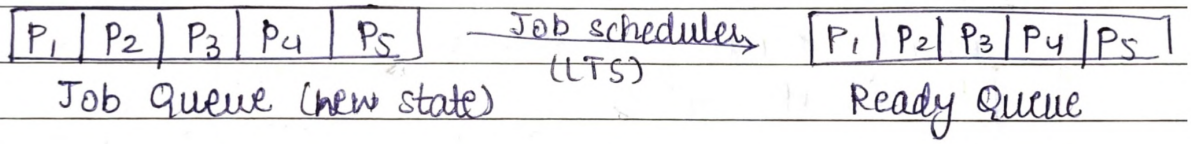
loop continues till program terminates

Process State: Life cycle of process (from generation to termination)

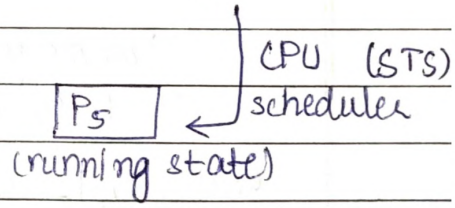
1) New state : When program is getting converted into process, its called new process.
2) Ready state : Process is now is in Memory. (in Ready Queue).
3) Running state : CPU is allocated to process & it is being executed.
4) Waiting state : Waiting for I/O completion.
5) Terminated state : Process execution is now finished.







1. Job scheduler (long term scheduler)
2. CPU scheduler (short term scheduler)



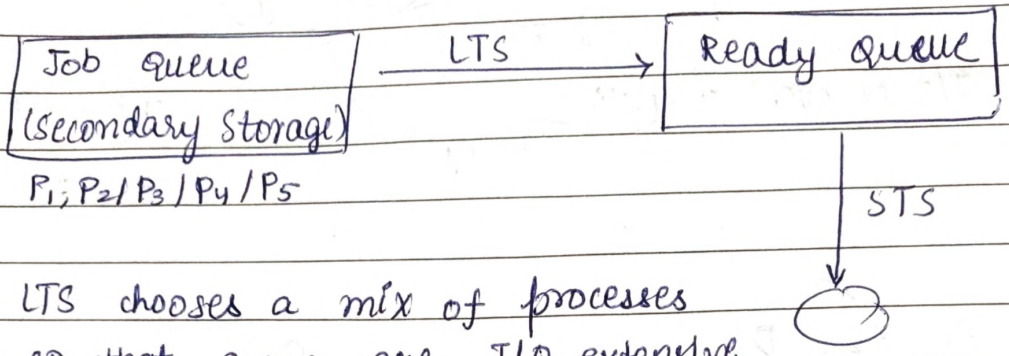
CPU scheduler works at high frequency ⇒ short term scheduler (so that CPU doesn't remains idel). Delay time is very small.

Job scheduler idel time is highter so, long term scheduler

#degree of multiprogramming = #process in Ready Queue so degree of multiprogramming is handled by LTS.

Chapter - 11 : Context Switching

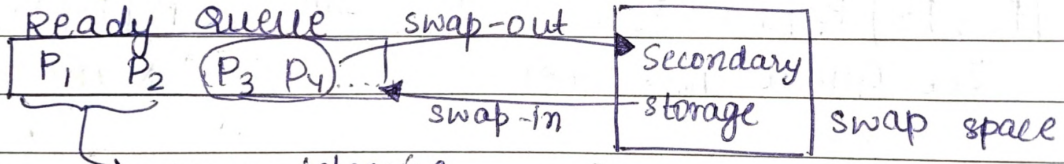
STS (CPU schedul.) LTS (Job scheduler)



LTS chooses a mix of processes so that some are I/O extensive & some CPU extensive

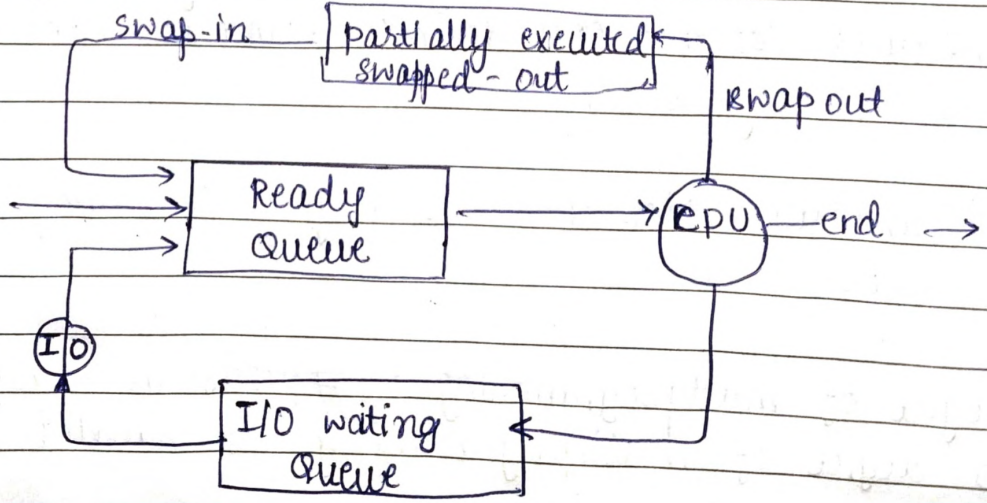
Choosing all CPU intensive jobs can lead to starvatio
" " I/O " " " " " idel CPU.

MTS → Medium Term Scheduler

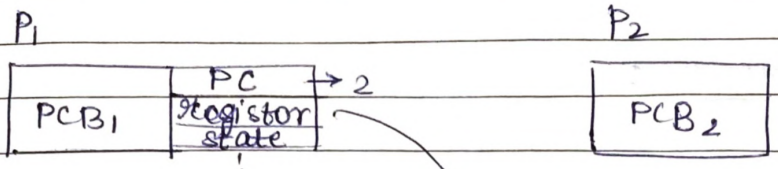


→ memory intensive
so, P3 P4 have less memory } so P3 & P4 are swapped out

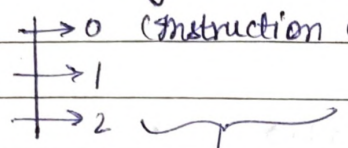
once, P1/P2 is terminated, P3 & P4 are swapped-in again.



Context Switching (Done by Kernal)



Initially P1 is executed → Registors



↓
(stack pointer) SP, CP, ...

↳ upto this P_1 is executed

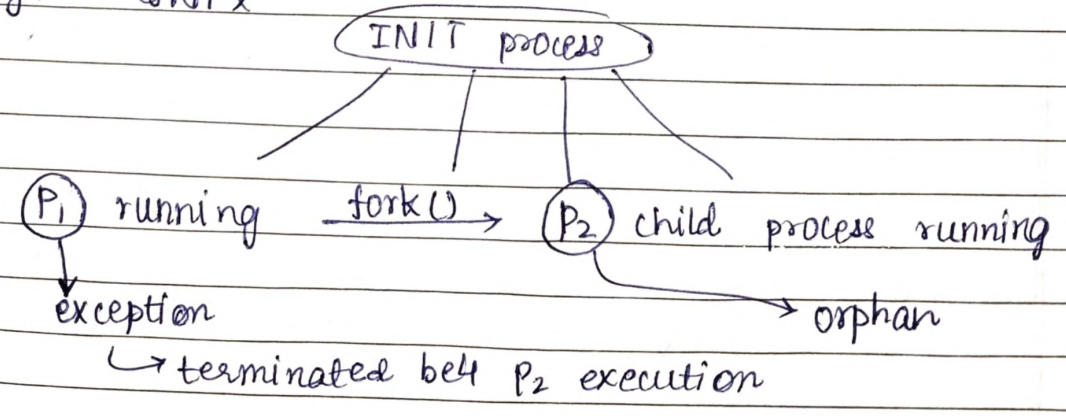
Now, PCB_1 will save the context (PC, Resistor, state) for process 1 and then continue the process 2 by restoring the context of P_2 using PCB2.

Context switching is pure overhead, i.e, during context switching CPU is not performing any task. Speed of context switching depends on speed of RAM (DDR3, DDR4), register, etc.

Orphan Process:

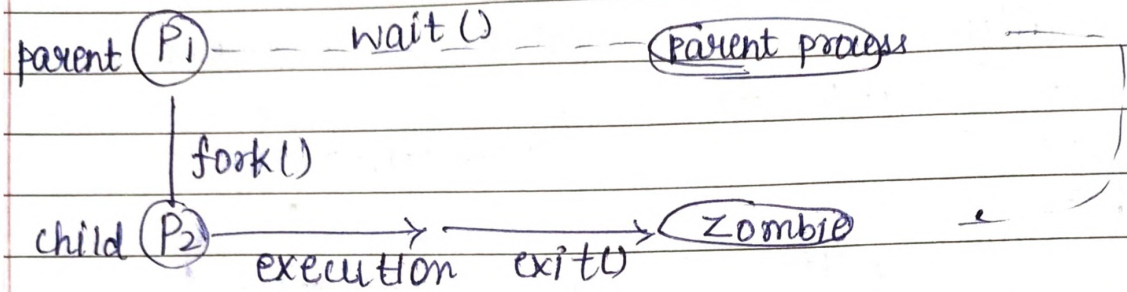
fork() : adds child to the parent process.

eg: UNIX



OS links the P_2 to the INIT (first process of OS)

Zombie Process :



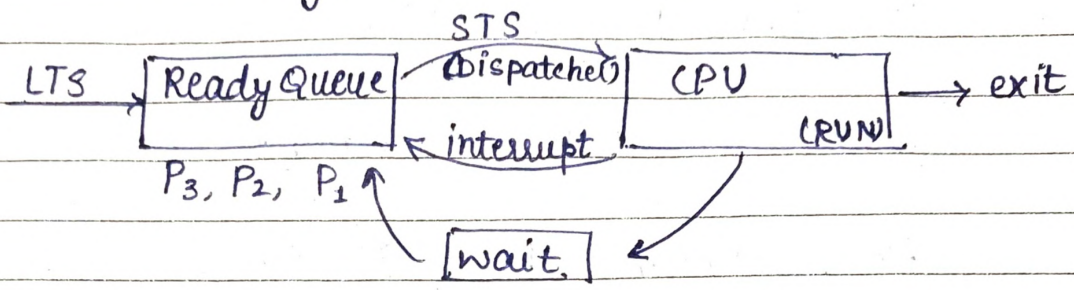
Parent waited for the child process for a longer time (child process terminated bef that). Now, P_2 once the exit() is called, it released all the resources. Now, the process will be kept in the process table untill parent process is waiting.

Once, parent will read the status of child (which has now become zombie process) then only its zombie process is deleted from process table. This is called reaping of zombie process.

Operating System (continued..)

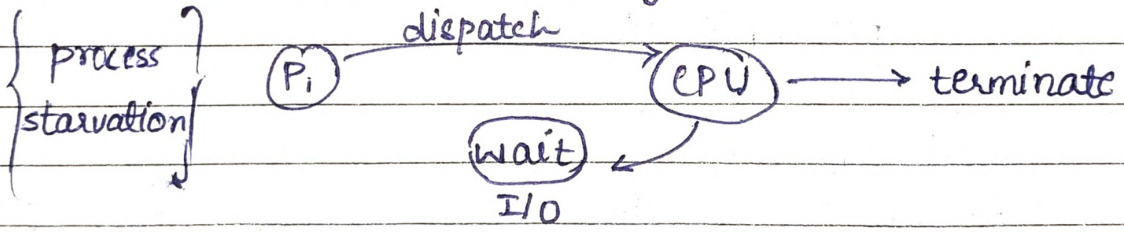
FCFS CPU scheduling algorithm

Process scheduling -



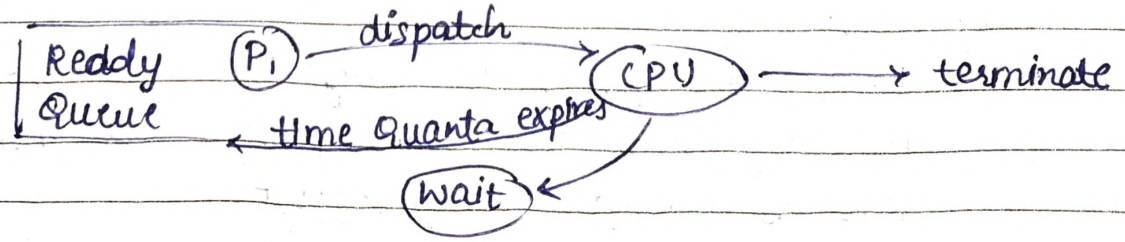
The selection of process from ready Queue & transferring the process to CPU for execution is called process scheduling.

1) Non - preemptive Scheduling :



Process remains in CPU until it's totally executed or it goes into wait state. (No Time sharing) even if it takes a lot of time.

2) Preemptive scheduling :



Process starvation : Non-preemtive > preemtive

CPU utilization : Preemtive > non-preemtive

Overhead : Preemtive > non-preemtive

Goals of CPU scheduling :

i) Maximum CPU utilization.
ii) Minimum turnaround time (time b/w the process to enter the readyQueue & exit).
iii) Minimum wait time. (no process starvation)
iv) Minimum responce time (time b/w the process enter the ReadyQueue & CPU allocation to it)
v) Maximum throughput (no. of processes completed per unit time)

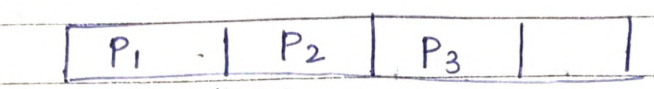
Basic Terms

Arrival Time : Time when process enters a ReadyQueue
Burst Time : time required by process for its execution
Completion Time : time taken till process gets terminated

* FCFS (First come-first serve)

a) whichever process comes first in RQ will be given CPU first.
b) Convoy effect
If heavy job with large BT is scheduled initially then the waiting time is increased alot. This is called convoy effect.

Gantt chart	(arrival)	(Burst)	(completion)	(turn-aro.)	(wait)
Process no.	AT	BT	CT	TAT	WT
1	0	20	20	20	0
2	1	2	22	21	19
3	2	2	24	22	20



convoy effect

average wt time = 13

Now,

Process no.	AT	BT	CT	(CT-AT) TAT	WT
1	0	2	2	2	0
2	1	2	4	3	1
3	2	20	24	22	2

Here, average wt. time = 1

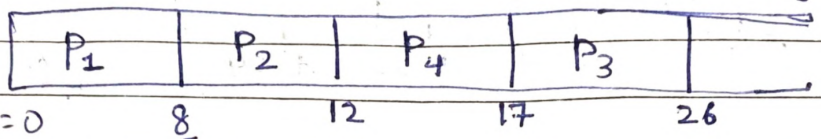
* Shortest Job first (SJF) [Non-preemptive]

Job with smallest Burst time is executed first.

P_1, P_2, P_3 ⟶ BT ?? (estimation). It's an estimation only which is a drawback.

estimated

Process	AT	BT	CT	TAT (TAT − BT)	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	7.75



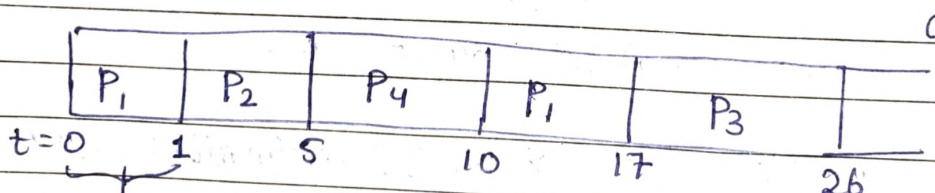
we have P_2/ P_3/ P_4

only P_1 is there & considering non-preemtive } could be a victim of convoy effect

criteria : AT & BT

* Shortest Job first [preemtive]

Process	AT	BT	CT	TAT	WT
P_1	0	~~8~~ 7	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	6.5



now
P_1 burst time is 7

No convey effect here.

SJF is really difficult to implement as estimating Burst time (BT) is quiet difficult.

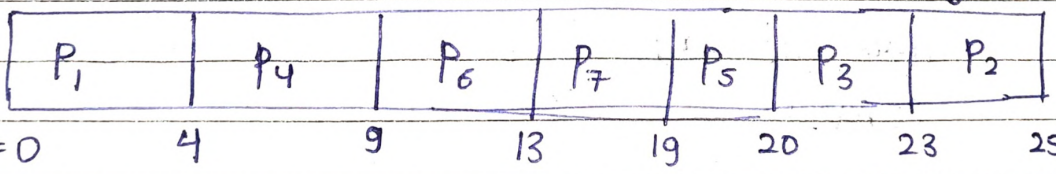
* Priority Scheduling [Non-preemtive]

SJF was also priority Scheduling (Job with lowest BT has higher priority)

We assign priority to each job

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	21	18
4	10	3	5	9	6	1
5	8	4	1	20	16	15
6	12	5	4	13	8	4
7	9	6	6	19	13	7

avg - 9,571 s

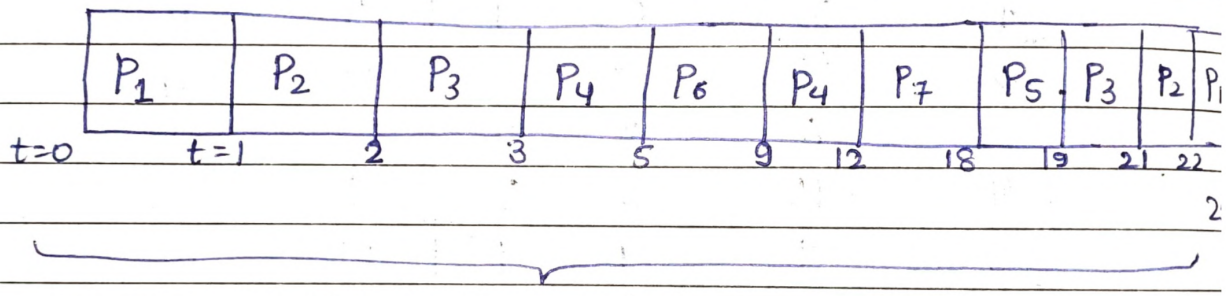


average wait time is high. (convoy effect)

* Priority scheduling [Preemtive]

P	Priority	AT	BT	CT	TAT	WT
1	2	0	~~4~~ 3	25	25	21
2	4	1	~~2~~ 1	22	21	19
3	6	2	~~3~~ 2	21	19	16
4	10	3	~~5~~ 3	12	9	4
5	8	4	1	19	15	14
6	12	5	4	9	4	0
7	9	6	6	18	12	6

avg 11.43



Context switching is done alot, so overhedd is high.

Here is also convoy effect (process 2 is stavved)

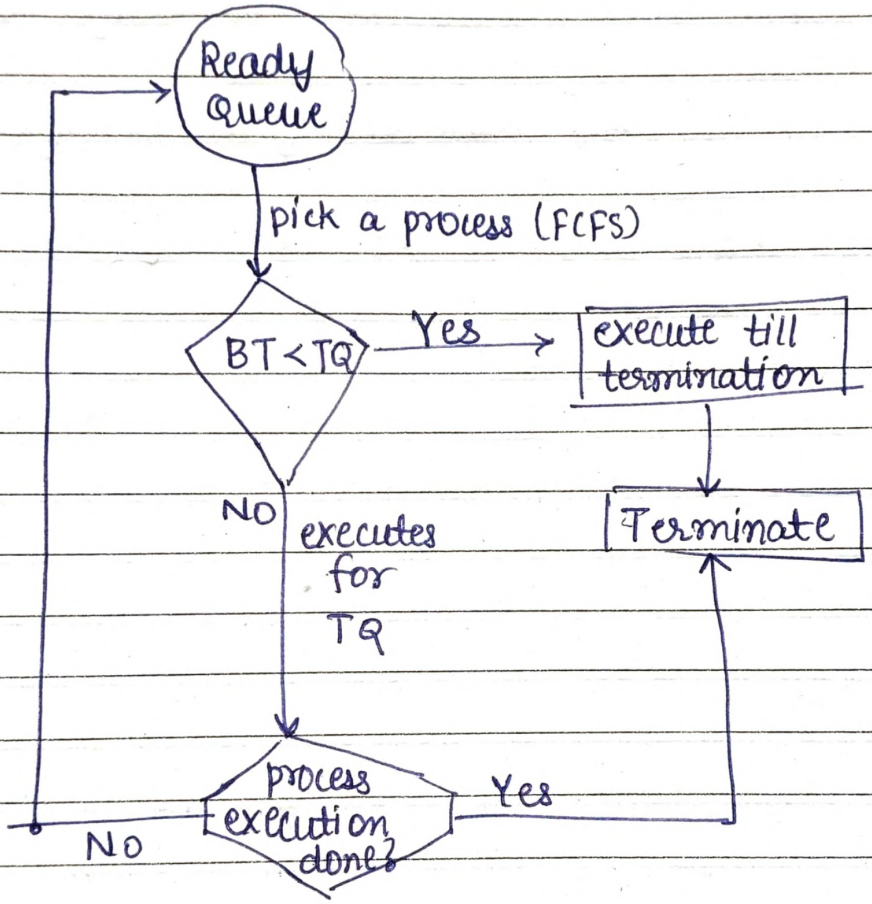
Biggest Drawback: Indefinite waiting or extreme starvation

High priority job will come again & again, will block the low priority jobs

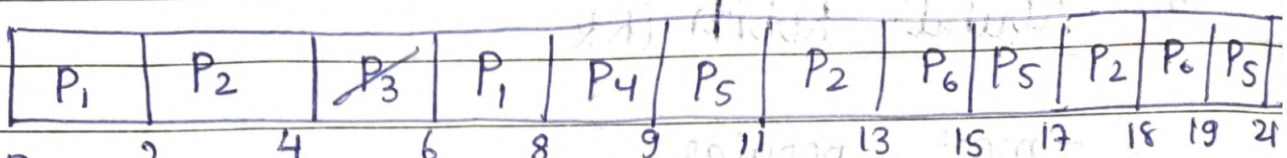
Solution: Ageing → gradually increase the priority of lowestpriority job. eg: every 15 min priority (lowest P. job) ++;

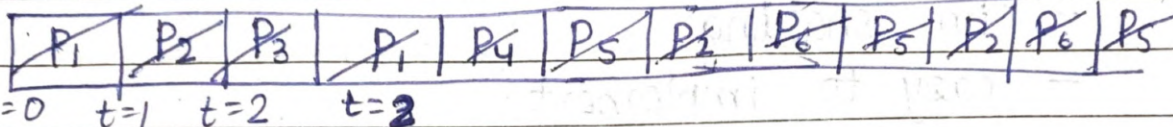
Round - Robith (RR)

- most popular
- FCFS preemtive version } AT + time quantum (TQ)
- Time sharing
- easy to implement



TQ → 2 sec





Queue

Process	AT	BT	CT	TAT	WT
1	0	4→2→0	8	8	4
2	1	5→3→1→0	18	17	
3	2	2→0	6	4	varies
4	3	1→0	9	6	
5	4	6→4→2	21	17	
6	5	3→1	19	14	

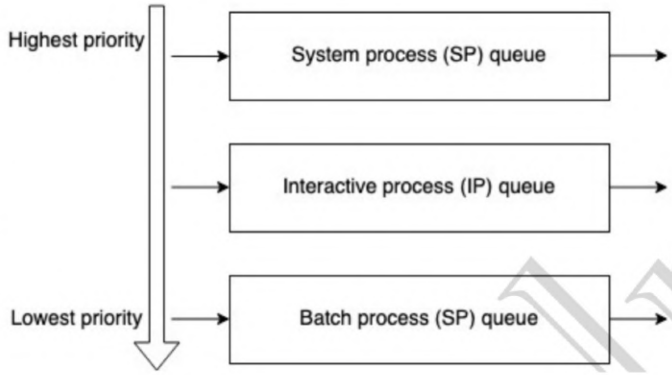
convoy effect is minimized, but overhead is increased.

If we will further reduce TQ then overhead will increase.

Real Life Scheduling Algorithms

1. Multi - level Queue Scheduling (MLQ) } Not used due to convoy effect

a) System process : created by OS
b) Interactive / foreground process : user input req. (I/P)
c) Batch / Background process : Runs in background , no I/P



Multi-level queue scheduling (**MLQ**)

a. Ready queue is divided into multiple queues depending upon priority.
b. A process is permanently assigned to one of the queues (inflexible) based on some property of process, memory, size, process priority or process type.
c. Each queue has its own scheduling algorithm. E.g., SP -> RR, IP -> RR & BP -> FCFS.
d. System process: Created by OS (Highest priority)

Interactive process (Foreground process): Needs user input (I/O).

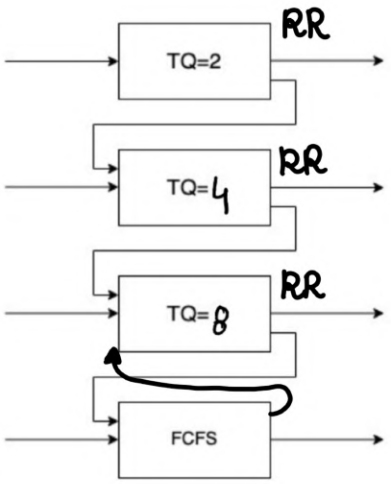
Batch process (Background process): Runs silently, no user input required.

e. Scheduling among different sub-queues is implemented as **fixed priority preemptive** scheduling. E.g., foreground queue has absolute priority over background queue.
f. If an interactive process comes & batch process is currently executing. Then, batch process will be preempted.
g. Problem: Only after completion of all the processes from the top-level ready queue, the further level ready queues will be scheduled.
This came starvation for lower priority process.
h. Convoy effect is present.

Between SP, IP, BP Queue follows pre-emtive priority scheduling.
for Batch process, starvation can occur (i.e, convoy effect ,may be ∞ waiting)

2. Multi-level feedback Queue scheduling

- Multiple sub-queues
- Inter queue movement is allowed
- separate process based on BT
- BT ↑ → lower queue
- I/O bound / interactive process are higher priority
- Ageing method is used for process in lower Queue to prevent convoy effect



} configurable

Design of MLFQ

1) No. of Queue
2) Scheduling Algo. in each Queue
3) Method to upgrade a process to higher queue
4) " " demote a process to lower queue
5) What is starting queue for a process.

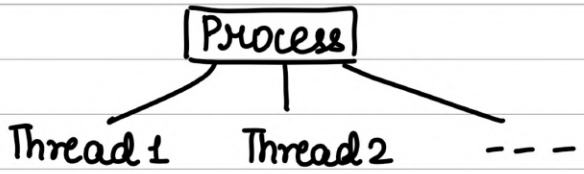
	FCFS	SJF	PSJF	Priority	P-Priority	RR	MLQ	MLFQ
Design	Simple	Complex	Complex	Complex	Complex	Simple	Complex	Complex
Preemption	No	No	Yes	No	Yes	Yes	Yes	Yes
Convoy effect	Yes	Yes	No	Yes	Yes	No	Yes	Yes
Overhead	No	No	Yes	No	Yes	Yes	Yes	Yes

calcuting BT is complex

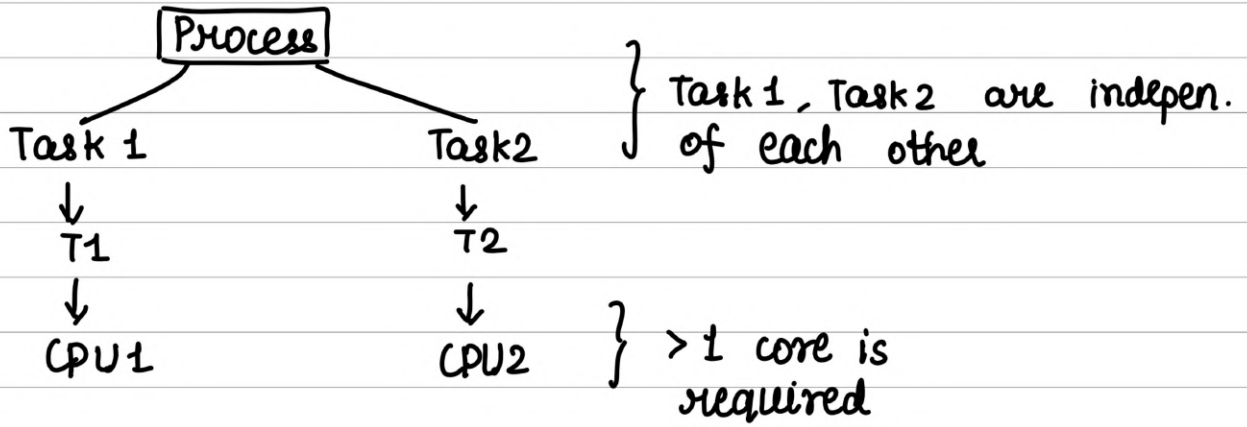
indefinite wait (ageing used)

CONCURRENCY

Ability of OS to execute multiple instructions at the same time.

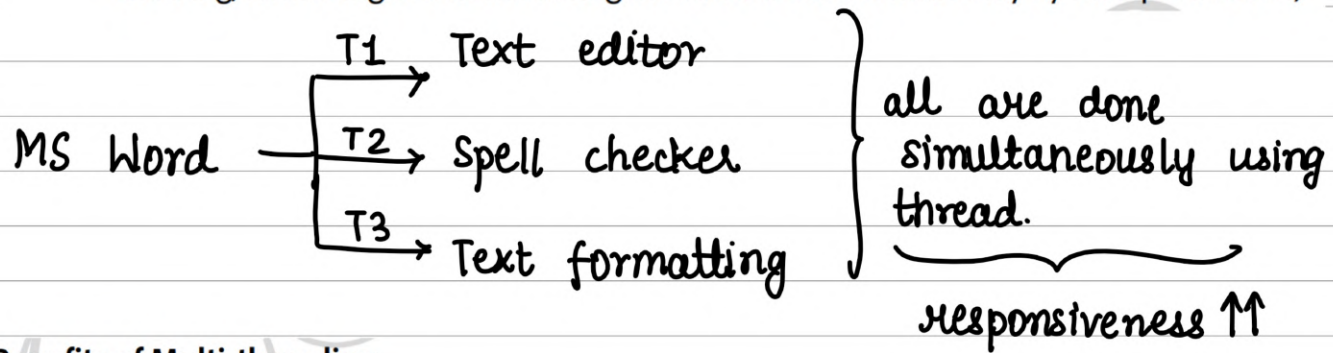


Threads ?? → lightweight process



Thread:

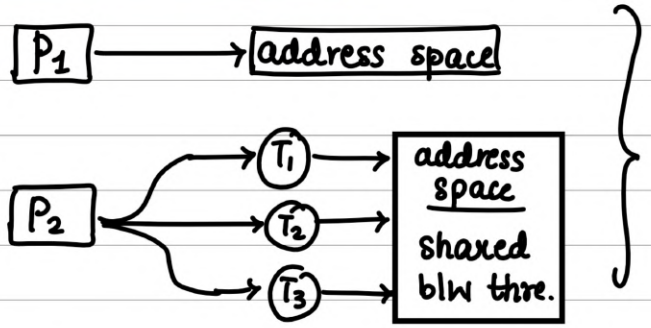
- Single sequence stream within a process.
- An independent path of execution in a process.
- Light-weight process.
- Used to achieve parallelism by dividing a process's tasks which are independent path of execution.
- E.g., Multiple tabs in a browser, text editor (When you are typing in an editor, spell checking, formatting of text and saving the text are done concurrently by multiple threads.)



Benefits of Multi-threading.

- Responsiveness
- Resource sharing: Efficient resource sharing.
- Economy: It is more economical to create and context switch threads.
    1. Also, allocating memory and resources for process creation is costly, so better to divide tasks into threads of same process.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Process and Threads



Thread Scheduling: Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the operating system.

Threads context switching

- OS saves current state of thread & switches to another thread of same process.
- Doesn't includes switching of memory address space. (But Program counter, registers & stack are included.)
- Fast switching as compared to process switching
- CPU's cache state is preserved.

I/O or TQ, based context switching is done here as well

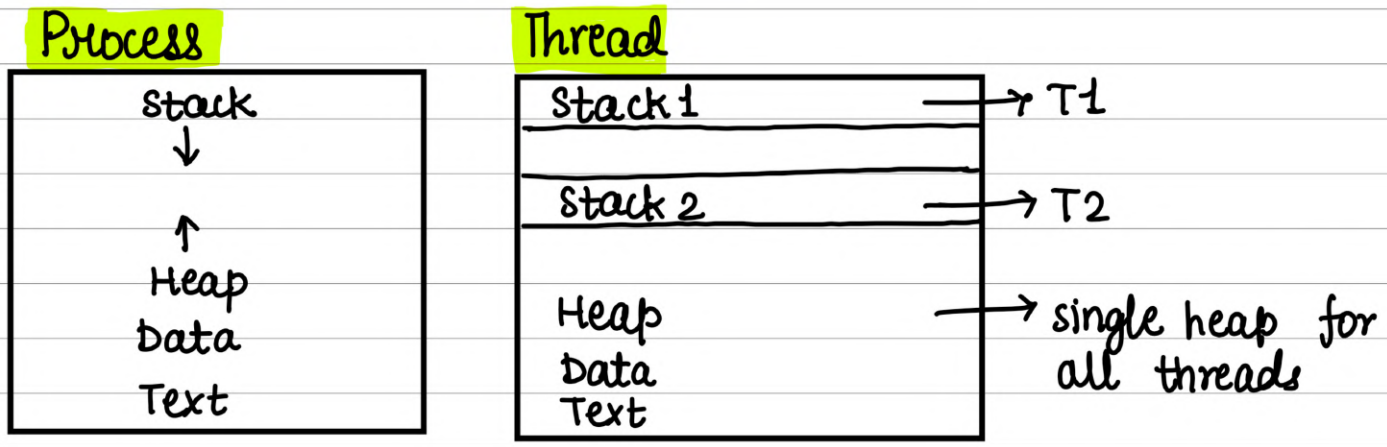
- We have TCB (Thread control block) like PCB for state storage management while performing context switching.

How each thread get access to the CPU?

- Each thread has its own program counter.
- Depending upon the thread scheduling algorithm, OS schedule these threads.
- OS will fetch instructions corresponding to PC of that thread and execute instruction.

Will single CPU system would gain by multi-threading technique?

- Never.
- As two threads have to context switch for that single CPU. *
- This won't give any gain.



```cpp
#include <iostream>
#include <thread> // Correctly include the thread header
#include <unistd.h>

using namespace std;

void taskA()
{
    for(int i = 0; i < 10; i++) {
        sleep(0.1);
        cout << "Task A: " << i << '\n';
        fflush(stdout);
    }
}

void taskB()
{
    for(int i = 0; i < 10; i++) {
        sleep(0.1);
        cout << "Task B: " << i << '\n';
        fflush(stdout);
    }
}

int main()
{
    thread t1(taskA); // Correctly use std::thread
    thread t2(taskB); // Correctly use std::thread
    t1.join(); } wait till thread t1
    t2.join(); } & t2 are executed
    return 0;
}
```

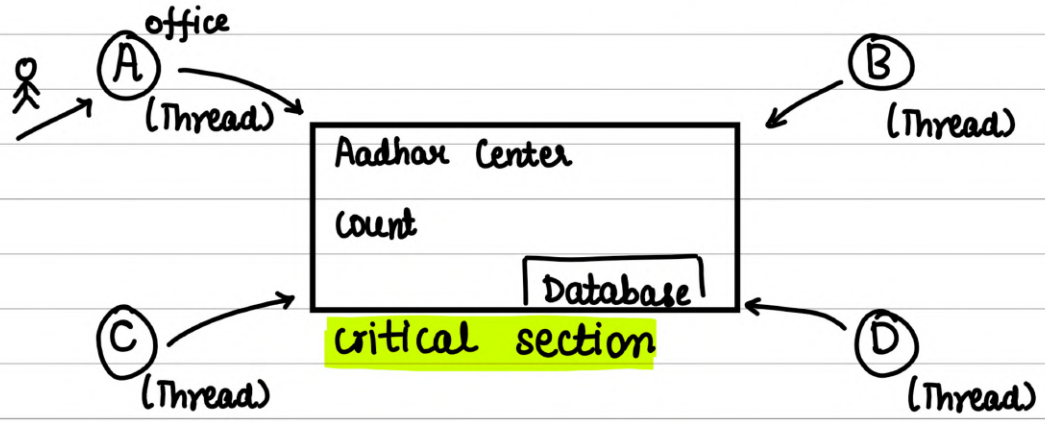
If join statement is missing, the parent process will exit and threads will become zombie.

```
Task B: 0
Task A: 0
Task B: 1
Task A: 1
Task B: 2
Task A: 2
Task B: 3
Task A: 3
Task B: 4
```

} Both are running simultaneously

Critical Section Problem

count++ } Inside CPU { temp = count + 1
count = temp



A count++ → temp = count + 1
& context switched bef count was set to temp
B count++ → temp2 = count + 1 } wrong as count was not update

even after 2 count++ calls, count will be increased by 1.
↳ data inconsistency, this problem is called **race condition**

The critical section refers to the segment of code where processes/threads access shared resources, such as common variables and files, and perform write operations on them. Since processes/threads execute concurrently, any process can be interrupted mid-execution.

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e., both threads are "racing" to access/change the data.

```python
from threading import *
import time

lock = Lock()
count = 0

def task():
    global count
    for i in range(100000000):
        count+=1


if __name__ == '__main__':
    t1 = Thread(target=task)
    t2 = Thread(target=task)

    t1.start()
    t2.start()
    t1.join()
    t2.join()
    print(count)
```

Output won't be consistent. It won't be equal to 200000000

Here 'count' is the critical section.

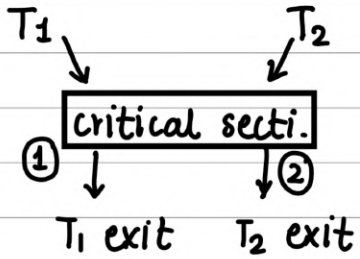
Solution of Race condition

1.) Make it atomic operation

count++ ; temp = count + 1
count = temp
} instead of 2 operations, make it one (atomic)

In C++ → atomic variables are supported.

2.) Mutual Exclusion (Locks / Mutex)



let T_1 exit first and then T_2 starts executing.
(T_1 & T_2 made sequential)

↓

we can use locks for that

```python
from threading import *
import time

lock = Lock()
count = 0

def task():
    lock.acquire()
    global count
    for i in range(1000000):
        count+=1
    lock.release()


if __name__ == '__main__':
    t1 = Thread(target=task)
    t2 = Thread(target=task)

    t1.start()
    t2.start()
    t1.join()
    t2.join()
    print(count)
```

```
PS C:\Users\shahi\Desktop\CDC\OS> python
 -u "c:\Users\shahi\Desktop\CDC\OS\criti
cal_section.py"
2000000
```

T_1 executes & lock the critical section, once it releases it, then only T_2 executes.

Disadvantages:

i. **Contention**: one thread has acquired the lock, other threads will be busy waiting, what if thread that had acquired the lock dies, then all other threads will be in infinite waiting.
ii. **Deadlocks**
iii. Debugging
iv. Starvation of high priority threads.

Q. Can we use single flag?

Solution of C.S problem should have 3 conditions
i) mutual exclusion
ii) Progress: if C.S is free, any thread can access it, there must not exist any order in which threads can access critical section.
iii) Bounded waiting: no indefinite waiting for any thread. (limited waiting time)

Using single flag / turn = 0/1

```
While(1)    // T1
{
    while (turn != 0);
    C.S
    turn = 1
    R.S
}
```

```
While(1)    // T2
{
    while (turn != 1);   } wait till turn becomes 1
    C.S
    turn = 0
    R.S
}
```

Initially let turn = 0; // this solution guarantee mutual exclusion but does not guarentee progress. (fixed order is there)

Improvement of Single-flag solution:

3.) Peterson's solution:

array of flag[2] // flag[n] for n threads
twin

flag[i] = true implies that T_i is ready to enter the C.S

twin → indicates whose twin is to enter the C.S

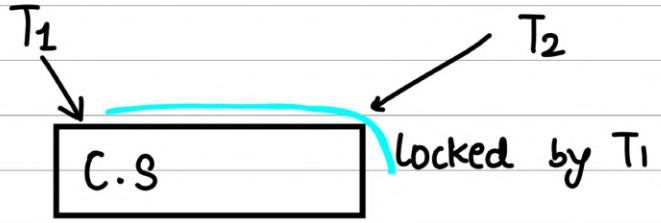
Thread - 1	Thread - 2

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

Both mutual exclusion and progress are guarenteed.

Peterson's solution is made for only 2 threads.

locks disadvantages



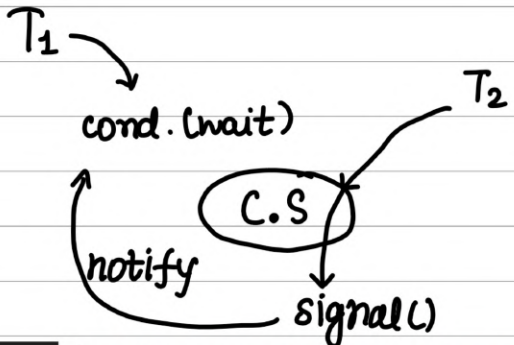
T2 will be busy waiting here, i.e, it will be waiting and at the same time, it will be using CPU cycles.

4.) Conditional variables

var → cond (lock)

wait — while waiting donot use CPU cycle

signal




```python
from threading import *
import time

cond = Condition() #it uses mutex internally
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(0.5)
            print("Signaling condition variable cond", name)
            cond.notify_all()
            print("Notification done", name)
```

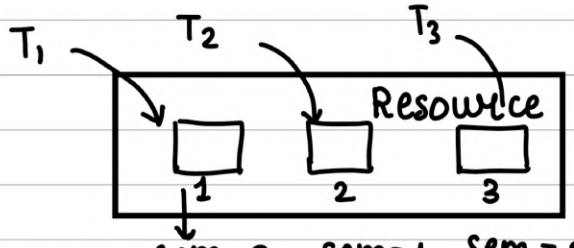
```
Waiting on condition variable cond:  t1
.
.
.
.
.
Signaling condition variable cond t2
Notification done t2
Condition met:  t1
```

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

5.) Semaphores → integer value

resource → multiple instances

Threads that can be executed simultaneously is equal to the #resources



semaphore = 3
(cannot < 0)

} once sem becomes 0, it will wait for threads to terminate.

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

```
signal(S)
{
    S→value ++ ;
    if(S→value >=0)
    {
        remove from S→blockList
        wakeup( )
    }
}
```

semaphore S(2)

T_1 → wait ( ) → S→val = 1 (not < 0 , so, resource allocated)
T_2 → wait ( ) → S→val = 0
T_3 → wait ( ) → S→val = -1 (Block Thread 3)

after completion T_1 → signal ( ) → S→val = 0

T_3 will wake up

```python
from threading import *
import time

sem = Semaphore(3)

def task(name):
    sem.acquire()
    for i in range(5):
        print("{} working\n".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',))
    t4 = Thread(target=task, args=('Thread-4',))
    t5 = Thread(target=task, args=('Thread-5',))

    t1.start()
    t2.start()
    t3.start()
    t4.start()
    t5.start()

    t1.join()
    t2.join()
    t3.join()
    t4.join()
    t5.join()
```

Output:

Thread-1 working

Thread-2 working

Thread-3 working

Thread-3 working

Thread-1 working

Thread-2 working

Thread-3 working
Thread-2 working
Thread-1 working

Thread-3 working
Thread-1 working
Thread-1 working

Thread-2 working

Thread-1 working

Thread-2 working

Thread-3 working

Thread-5 working
Thread-4 working

Thread-4 working
Thread-5 working

Thread-5 working
Thread-4 working

Thread-4 working
Thread-5 working

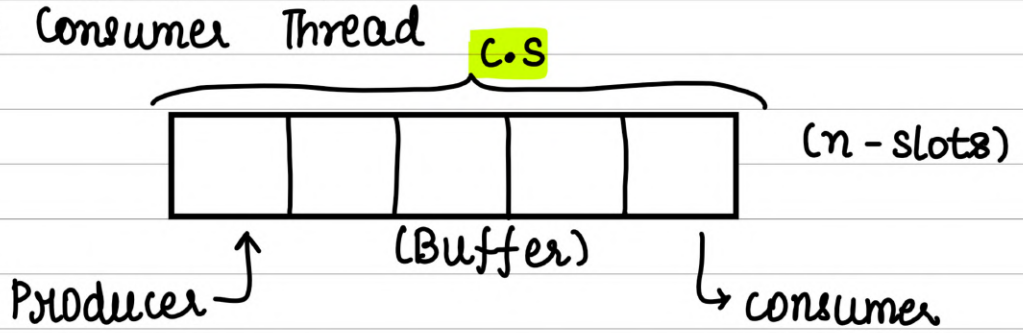
Thread-5 working
Thread-4 working

atonce
3 threads are
executed

Semaphore with $n = 1$ } Binary semaphore (Lock/Mutex)
Semaphore with $n > 1$ } counting semaphore

Producer Consumer Problem
(Bounded Buffer problem)

1) Producer Thread
2) Consumer Thread



Problem : CS (Buffer) → we want sync blw Producer & Consumer
If Buffer is full, producer must not insert
If " " empty, consumer must not remove

Solution: Semaphore

1) m, mutex → Binary semaphore, used to aquire lock on buffer

2) empty → a counting sem (initial value n) track empty slots

3) fill → traks filled slots (initial value 0)

Producer

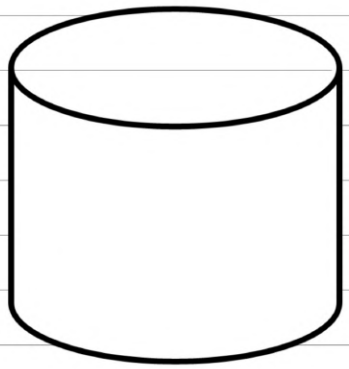
```
do
  { wait(empty); // wait untill
                 empty becomes > 0,
                 then empty→val--
    wait (mutex);
    // CS add data to buffer
    signal (mutex);
    signal (full); // increment
                   full → value
  } while (1)
```

Consumer

```
do
  { wait (full) // wait untill
                full becomes > 0,
                then full→value--;
    wait (mutex)
    // remove data from Buffer
    signal (mutex)
    signal (empty); // increment
                    empty →val
  } while (1)
```

Reader - Writer Problem

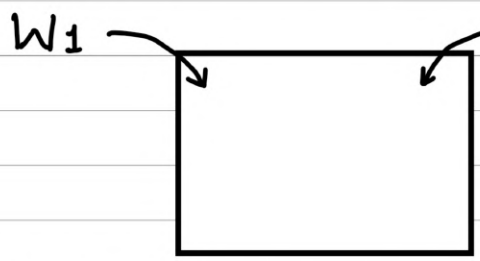
1) Reader Thread
2) Writer Thread



Database

if >1 Readers are reading then no issues

if >1 writers / writer & some other thread (R/W), parallels → Race condition (Data inconsis.)



reader will read old version of data which we donot want.

Solution: (using semaphore)

1. Mutex : used to ensure mutual exclusion, when read count (rc) is updated // Binary semaphore

2. wrt → Binary semaphore
   - common for both reader & writer

3. rc (read-count) → integer (initially 0)

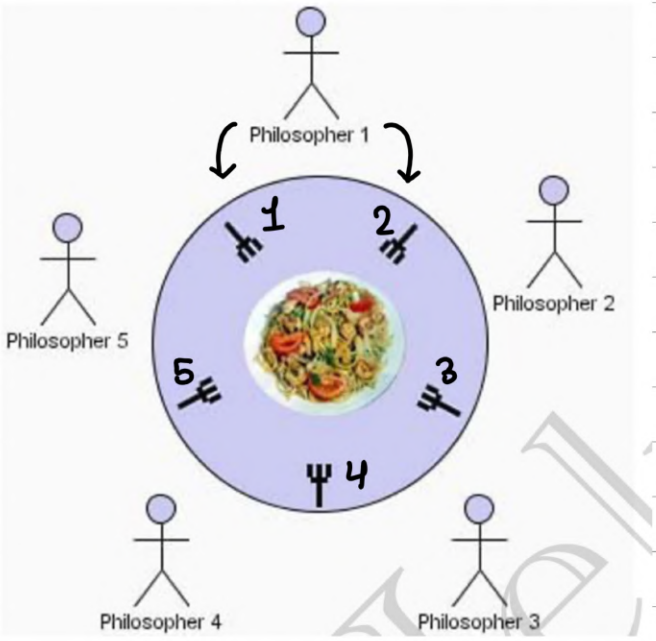
Writer sol^n:

```
do {
    wait (wrt); // wait if eith
                writer or reader is
                present
    // do write operation
    signal (wrt);
}
while (true)
```

Reader sol^n:

```
do {
    wait (mutex); rc++;
    if (rc == 1)
        wait (wrt) // ensure
            no writer can enter
    signal (mutex);
    // CS : Reader is reading
    wait (mutex); rc--;
    if (rc == 0) // no reader
        signal (wrt) // writer in
    signal (mutex)
} while (1)
```

Dinning philosopher



5 philosophers
noodle
5 forks

Philosopher
think eat

each philosopher requires 2 forks to eat.

Solution :

1) each fork - binary semaphore , fork[5] {1}

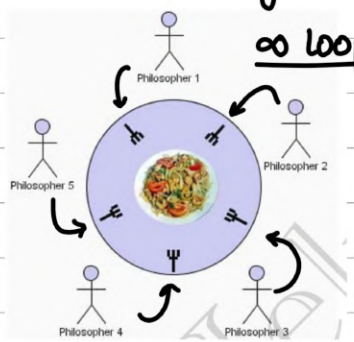
wait() → fork[i] → acquire
release() → fork[i] → released

sol^n:

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

    signal (fork[i])
    signal (fork[(i+1)%5]);
    //think

} while (1)
```

there is a deadlock here, if all philospher picks one fork simultaneously.

∞ loop

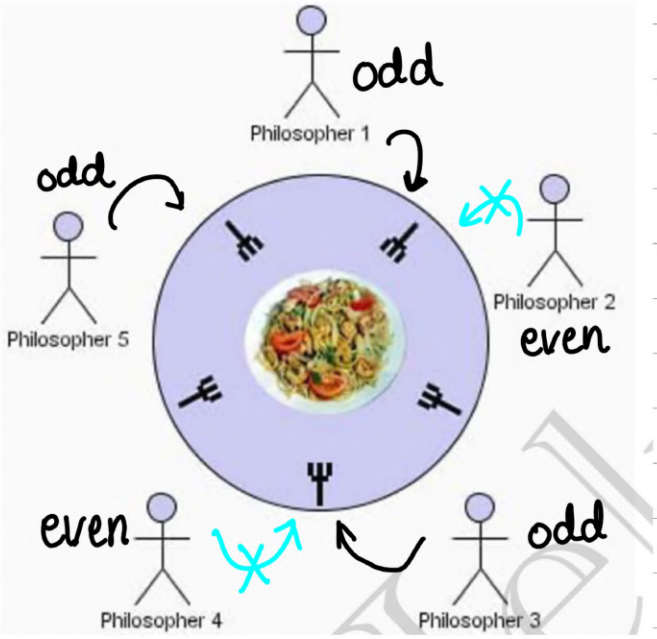


We must use **some methods to avoid Deadlock and make the solution work**

a. Allow at most 4 ph. To be sitting simultaneously.

b. Allow a ph. To pick up his fork only if both forks are available and to do this, he must pick them up in a critical section (atomically).

c. **Odd-even rule.**
an odd ph. Picks up first his left fork and then his right fork, whereas an even ph. Picks up his right fork then his left fork.



Hence, only semaphores are not enough to solve this problem.
We must add some enhancement rules to make deadlock free solution.

Deadlocks

System

Resources

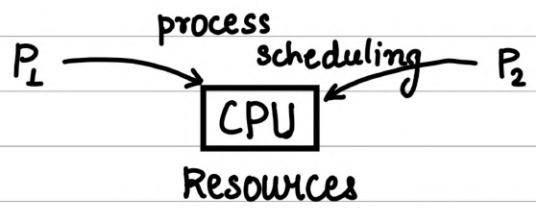
$R_1, R_2, R_3, R_4, -----$

Memory space, CPU, files, locks, I/O devices, etc.

different processes

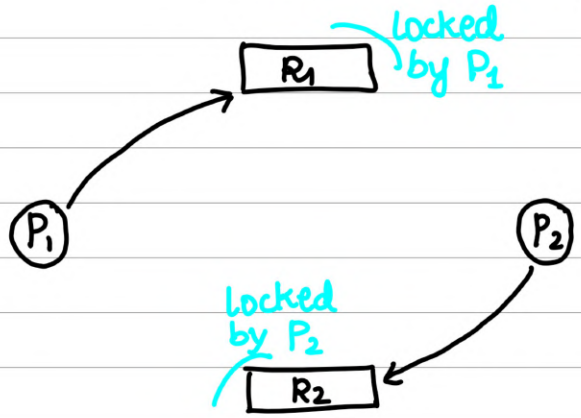
P_1 P_2 P_3 (process)

T_1 T_2 -- --- (Thread)



We have finite resources and multiple processes (or multiple threads)

We want to utilize our resources so that there will be no blockage.



P_1/P_2 both needs R_1 and R_2 to complete their task

Suppose P_1 locks R_1 & P_2 locks R_2. Now, neither of P_1/P_2 will terminate resulting if infinite waiting time.

- Process requests a **resource (R)**, if R is not available (taken by other process), process enters in a waiting state. Sometimes that waiting process is never able to change its state because the resource, it has requested is busy (forever), called **DEADLOCK (DL)**
- Two or more processes are waiting on some resource's availability, which will never be available as it is also busy with some other process. The Processes are said to be in **Deadlock.**
- DL is a bug present in the process/thread synchronization method.
- In DL, processes never finish executing, and the system resources are tied up, preventing other jobs from starting.

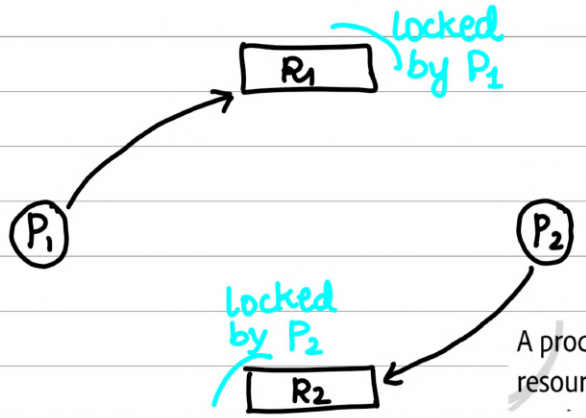
How a process / thread utilize a Resource ?
Ans:
1) Request 2) Lock 3) Use 4) Release

Necessary condition for DL

1) Mutual exclusion

Only 1 process at a time can use the resource, if another process requests that resource, the requesting process must wait until the resource has been released.

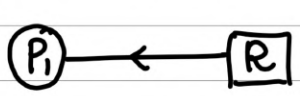
2) Hold and wait



P_1 holds R_1 and is waiting for R_2 to get released

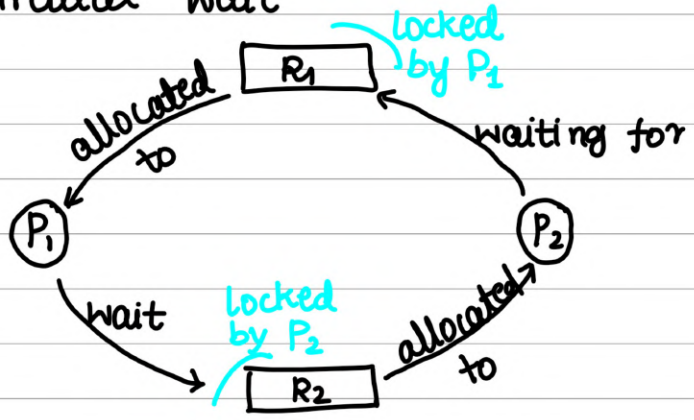
A process must be holding at least one resource & waiting to acquire additional resources that are currently being held by other processes.

3) No preemption



} P_1 will released R only when P_1 is terminated

4) Circular wait



P_1 is waiting for resource which is locked by P_2
and
P_2 is waiting for resource which is locked by P_1

A set {P0, P1, ... ,Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, and so on.

Resource Allocation Graph (RAG)

1) Vertex → (1) Process vertex (P)
(2) Resource vertex [R]

2) Edges → Assign edge
Request edge

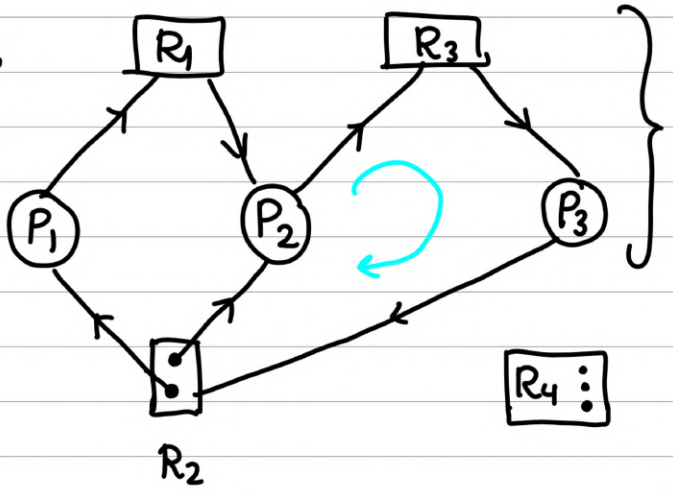
3)

CPU (4 cores) → 4 CPUs → multiple instances of same resource

representation

used for system Representation

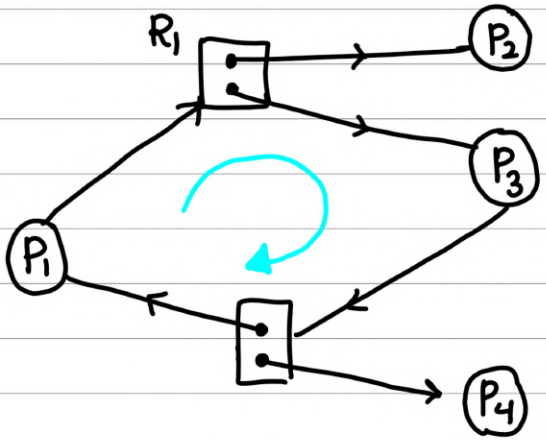
eg:



Dead-lock is present

If cycle is present, there may be presence of deadlock
If cycle is not " , then no deadlock.

eg:



cycle is present but after some time P_2, P_4 will release the resource R_1, R_2

So, no deadlock here

Methods to Handle Deadlocks

a. Use a protocol to **prevent** or **avoid** deadlocks, ensuring that the system will never enter a deadlocked state.
b. Allow the system to enter a deadlocked state, **detect it, and recover.**
c. Ignore the problem altogether and pretend that deadlocks never occur in system. (Ostrich algorithm) aka, **Deadlock ignorance.**
   ↳ leave it to application programmer.

a) Deadlock Prevention

Deadlock Prevention: by ensuring at least one of the necessary conditions cannot hold.

a. **Mutual exclusion**
   i. Use locks (mutual exclusion) only for non-sharable resource.
   ii. Sharable resources like Read-Only files can be accessed by multiple processes/threads.
   iii. However, we can't prevent DLs by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.
b. **Hold & Wait**
   i. To ensure H&W condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it doesn't hold any other resource.
   ii. Protocol (A) can be, each process has to request and be allocated all its resources before its execution.
   iii. Protocol (B) can be, allow a process to request resources only when it has none. It can request any additional resources after it must have released all the resources that it is currently allocated.

} before holding ensure all req. resources are free.

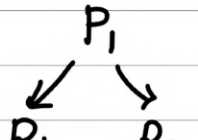
c. **No preemption**
   i. If a process is holding some resources and request another resource that cannot be immediately allocated to it, then all the resources the process is currently holding are preempted. The process will restart only when it can regain its old resources, as well as the new one that it is requesting. (Live Lock may occur).
   ii. If a process requests some resources, we first check whether they are available. If yes, we allocate them. If not, we check whether they are allocated to some other process that is waiting for additional resources. If so, preempt the desired resource from waiting process and allocate them to the requesting process.
d. **Circular wait**
   i. To ensure that this condition never holds is to impose a proper ordering of resource allocation.
   ii. P1 and P2 both require R1 and R1, locking on these resources should be like, both try to lock R1 then R2. By this way which ever process first locks R1 will get R2.

} philosopher problem

Mutual exclusion prevention

↳ only use it at non-sharable resources (critical section)
eg: for Read-only file – sharable resource (no use of locks)

Live Lock



} 2 locks are acquired simult. This may lead to locks collision on kernal side. Add a sleep blw.

Deadlock Avoidance

current state → known to us

1) no. of processes
2) need of resources of each process
3) currenty allocated amount of R. to each process
4) Max amount of each resources

we want to schedule processes & allocate resources in such a manner that our system remains in safe state (free from deadlock)

a. Schedule process and its resources allocation in such a way that the DL never occur.
b. Safe state: A state is safe if the system can allocate resources to each process (up to its max.) in some order and still avoid DL.
   A system is in safe state only if there exists a safe sequence.
c. In an Unsafe state, the operating system cannot prevent processes from requesting resources in such a way that any deadlock occurs. It is not necessary that all unsafe states are deadlocks; an unsafe state may lead to a deadlock.
d. The main key of the deadlock avoidance method is whenever the request is made for resources then the request must only be approved only in the case if the resulting state is a safe state.
e. In a case, if the system is unable to fulfill the request of all processes, then the state of the system is called unsafe.
f. Scheduling algorithm using which DL can be avoided by finding safe state. (Banker Algorithm)

Banker Algorithm

10 A, 5 B, 7C are available in total

Process	allocated			max need			available			Remaining need		
	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~~
P_2	2	0	0	3	2	2	5	3	2	~~1~~	~~2~~	~~2~~
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~~
+	7	2	5				10	5	7			

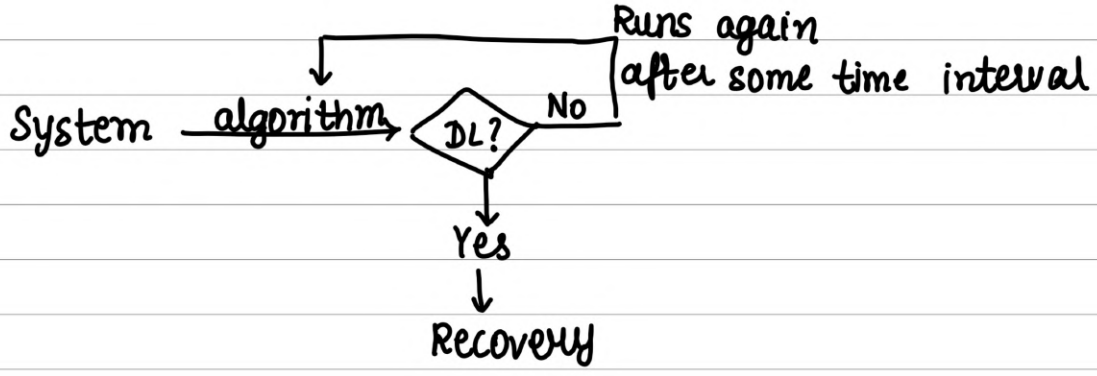
available = TotalResources − TotalAllocated = {10-7, 5-2, 7-5}

(3,3,2) cannot be allocated to P_1 (as its requirement are high) but P_2 only wants 1A, 2B, 2C. Allocated it to B and then making B terminate. so, now B will release all its resources.
After P_2 completion available is updated to
(3+2, 3+0, 2+0) ≡ (5, 3, 2)

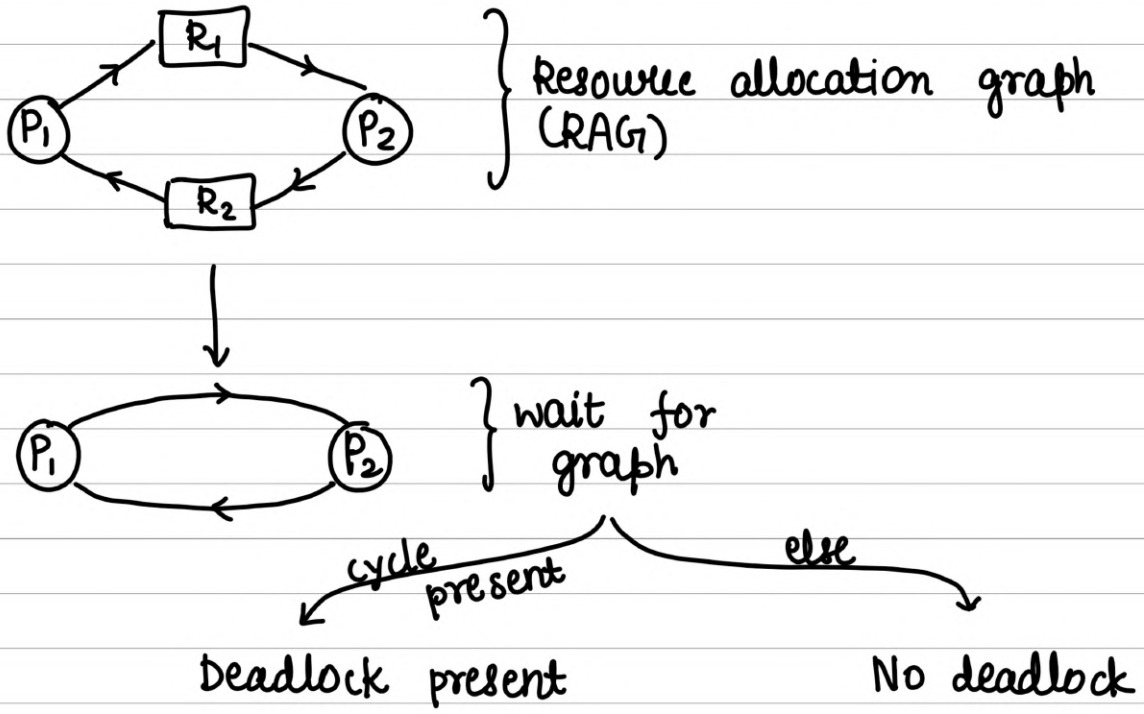
Next, we can schedule P_4, then P_5, then P_1 the finally P_3.
Since, all were able to get scheduled, so we reached a safe state (i.e, no dead lock).

Deadlock detection

Deadlock Detection: Systems haven't implemented deadlock-prevention or a deadlock avoidance technique, then they may employ DL detection then, recovery technique.



- **Single instance of each resource**



a. Single Instance of Each resource type **(wait-for graph method)**
   i. A deadlock exists in the system if and only if there is a cycle in the wait-for graph. In order to detect the deadlock, the system needs to maintain the wait-for graph and periodically system invokes an algorithm that searches for the cycle in the wait-for graph.

b. Multiple instances for each resource type
   i. Banker Algorithm

→ safe sequence available → No DL
→ safe sequence unavailable → Deadlock

Recovery from Deadlock

a. Process termination
   i. Abort all DL processes
   ii. Abort one process at a time until DL cycle is eliminated.

↗ abort one with low priority

b. Resource preemption
   i. To eliminate DL, we successively preempt some resources from processes and give these resources to other processes until DL cycle is broken.

Print in Order

Suppose we have a class:

```
public class Foo {
  public void first() { print("first"); }
  public void second() { print("second"); }
  public void third() { print("third"); }
}
```

The same instance of `Foo` will be passed to three different threads. Thread A will call `first()`, thread B will call `second()`, and thread C will call `third()`. Design a mechanism and modify the program to ensure that `second()` is executed after `first()`, and `third()` is executed after `second()`.

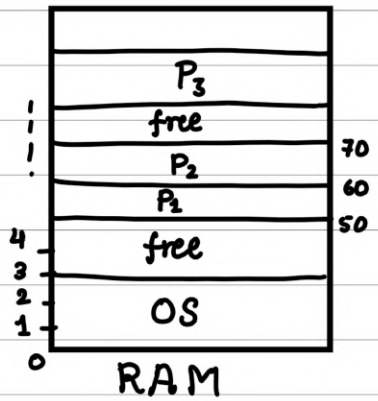
Note:

We do not know how the threads will be scheduled in the operating system, even though the numbers in the input seem to imply the ordering. The input format you see is mainly to ensure our tests' comprehensiveness.

```cpp
class Foo {
    std::mutex m;
    std::condition_variable cv;
    int turn;
public:
    Foo() {
        turn = 0;
    }
    void first(function<void()> printFirst) {

        // printFirst() outputs "first". Do not change or remove this line.
        printFirst();
        turn = 1;
        cv.notify_all();

    }

    void second(function<void()> printSecond) {
        std::unique_lock<std::mutex> lock(m);
        while(turn!=1){
            cv.wait(lock);
        }
        // printSecond() outputs "second". Do not change or remove this line.
        printSecond();
        turn = 2;
        cv.notify_all();
    }

    void third(function<void()> printThird) {
        std::unique_lock<std::mutex> lock(m);
        while(turn!=2){
            cv.wait(lock);
        }
        // printThird() outputs "third". Do not change or remove this line.
        printThird();
    }
};
```

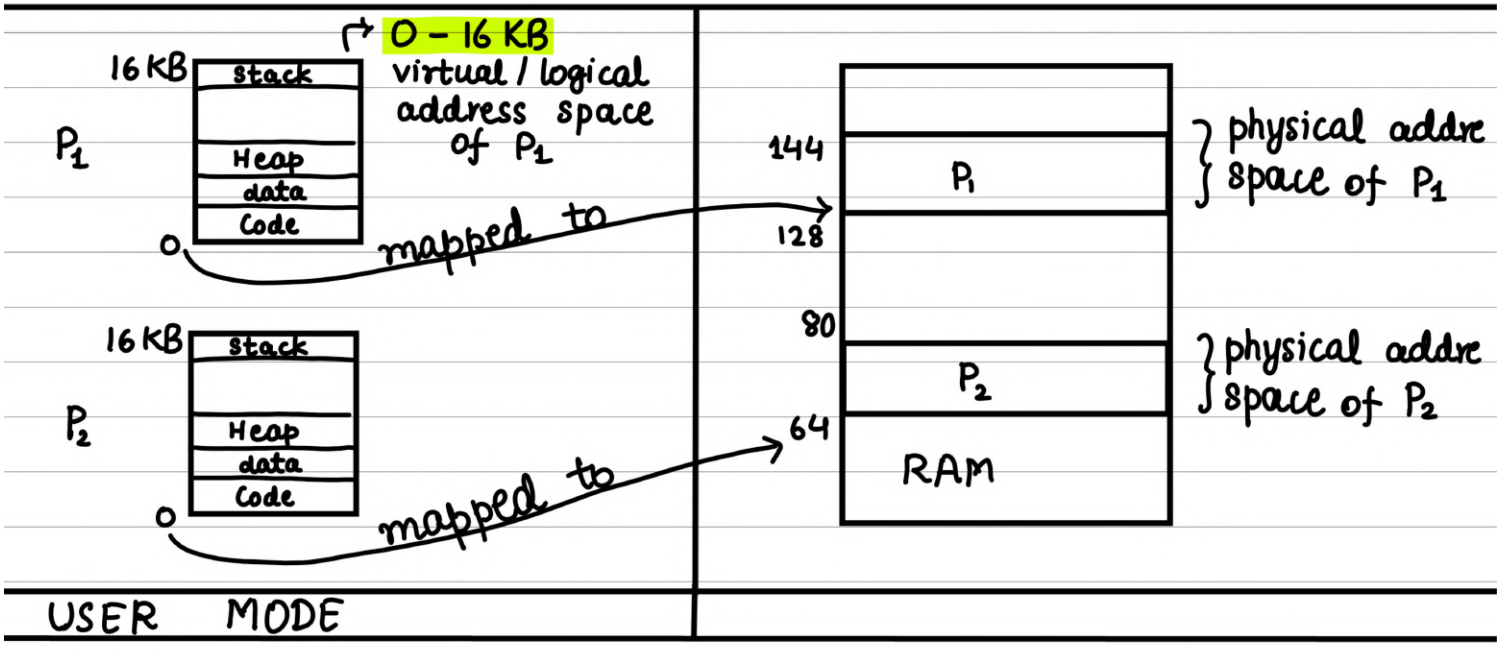
Multi-programming

↳ many processes → RAM



} isolation is affected

P_1 → program add +5 → data
60 + 5 → 65 } isolation lost



logical address space of 0 - 16 KB is mapped to Physical address space in RAM by OS

P_1 → logical address is 0 - 16 KB
Physical address has base 128 and offset 16

Similary,
P_2 → logical address is 0 - 16 KB
Ph sical address has base 64 and offset 16

if P_2 tries to access address outside the range (B, B+Offset) then O.S

a. Logical Address

i. An address generated by the **CPU**.
ii. The logical address is basically the address of an instruction or data used by a process.
iii. User can access logical address of the process.
iv. User has indirect access to the physical address through logical address.
v. Logical address does not exist physically. Hence, aka, **Virtual address.**
vi. The set of all logical addresses that are generated by any program is referred to as Logical Address Space.
vii. **Range: 0 to *max*.**

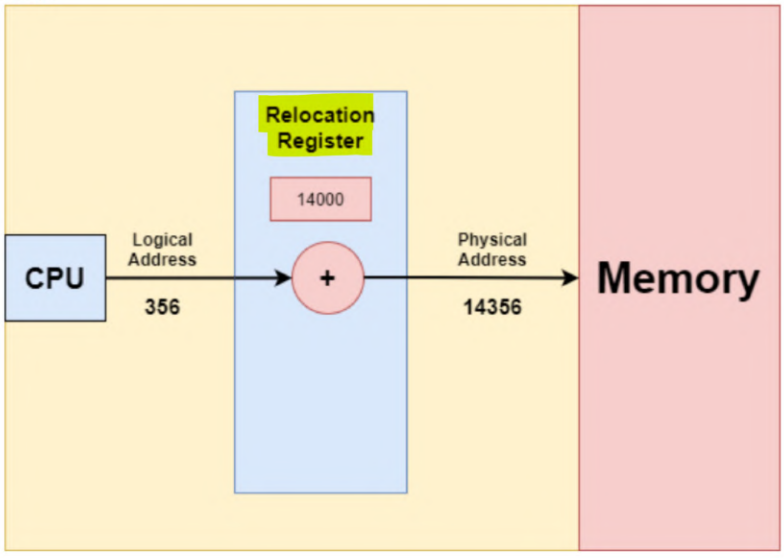
b. Physical Address

i. An address loaded into the memory-address register of the physical memory.
ii. User can never access the physical address of the Program.
iii. The physical address is in the memory unit. It's a location in the main memory physically.
iv. A physical address can be accessed by a user indirectly but not directly.
v. The set of all physical addresses corresponding to the Logical addresses is commonly known as Physical Address Space.
vi. It is computed by the **Memory Management Unit (MMU).**
vii. **Range: (R + 0) to (R + max), for a base value R.**

c. **The runtime mapping from virtual to physical address is done by a hardware device called the memory-management unit (MMU).**

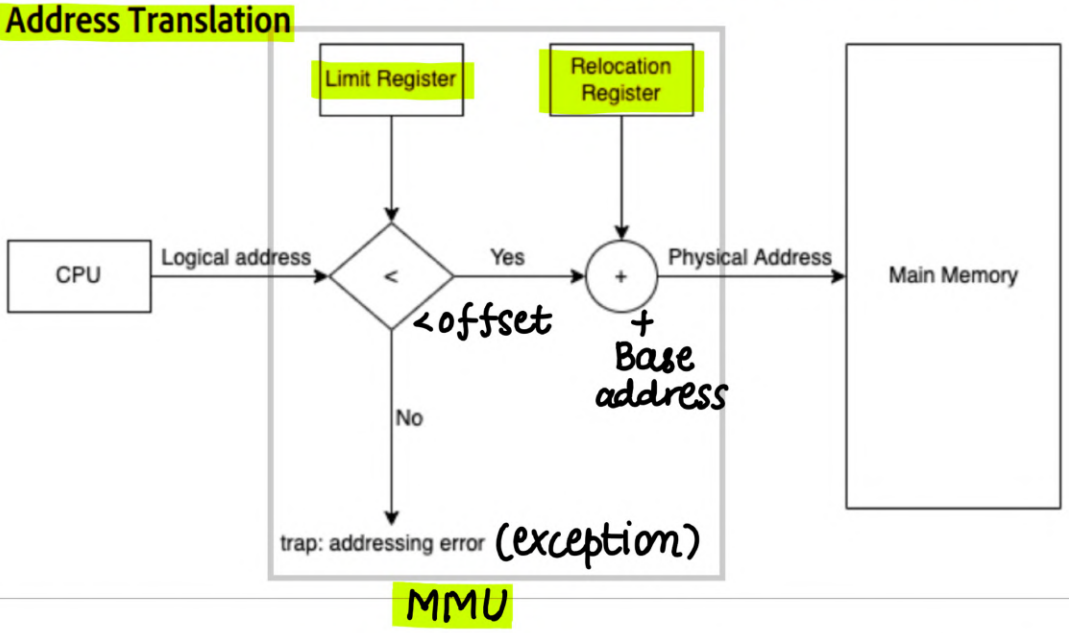
d. The user's program mainly generates the logical address, and the user thinks that the program is running in this logical address, but the program mainly needs physical memory in order to complete its execution.

e.



How OS manages the isolation and protect? **(Memory Mapping and Protection)**

a. OS provides this Virtual Address Space (VAS) concept.
b. To separate memory space, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
c. The relocation register contains value of smallest physical address (Base address [R]); the limit register contains the range of logical addresses (e.g., relocation = 100040 & limit = 74600).
d. Each logical address must be less than the limit register.
e. MMU maps the logical address dynamically by adding the value in the relocation register.
f. When CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Since every address generated by the CPU (Logical address) is checked against these registers, we can protect both OS and other users' programs and data from being modified by running process.
g. Any attempt by a program executing in user mode to access the OS memory or other uses' memory results in a trap in the OS, which treat the attempt as a fatal error.
h. **Address Translation**



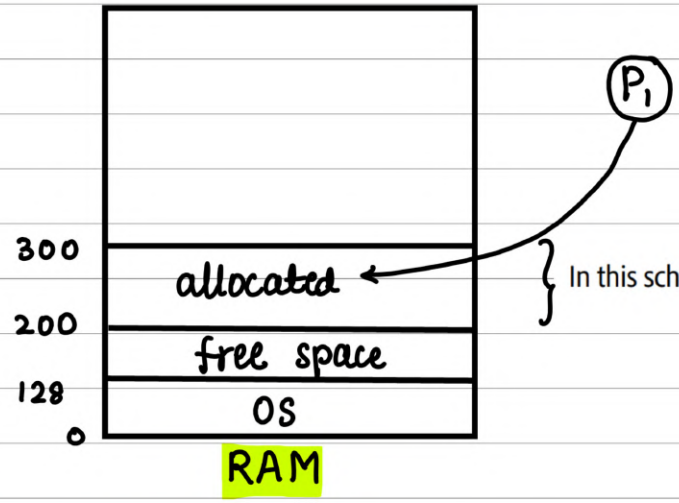
Dispatcher loads the Base (Relocation register) and offset (Limit register) during the context switch

Allocation methods in physical memory

Contiguous allocation

non-Contiguous allocation

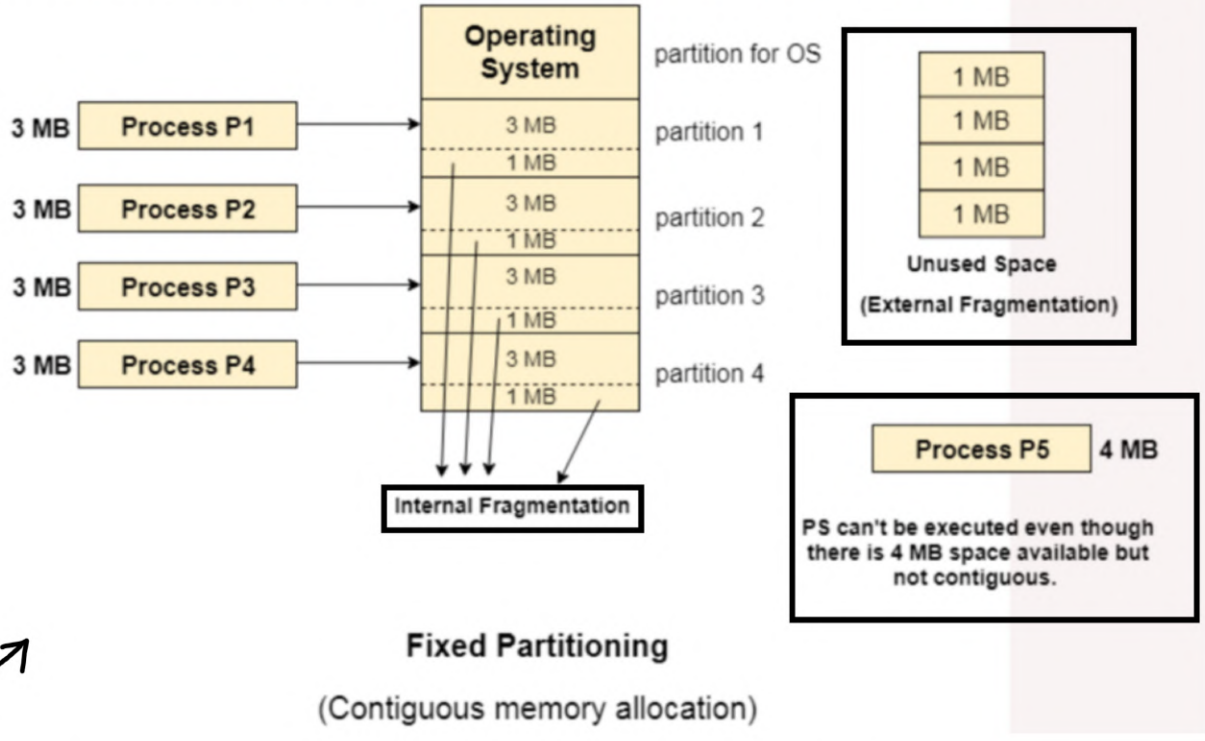
* Contiguous Allocation → fixed partition
  → dynamic partition



In this scheme, each process is contained in a single contiguous block of memory.

Fixed Partitioning

The main memory is divided into partitions of equal or different sizes.



Fixed Partitioning

(Contiguous memory allocation)

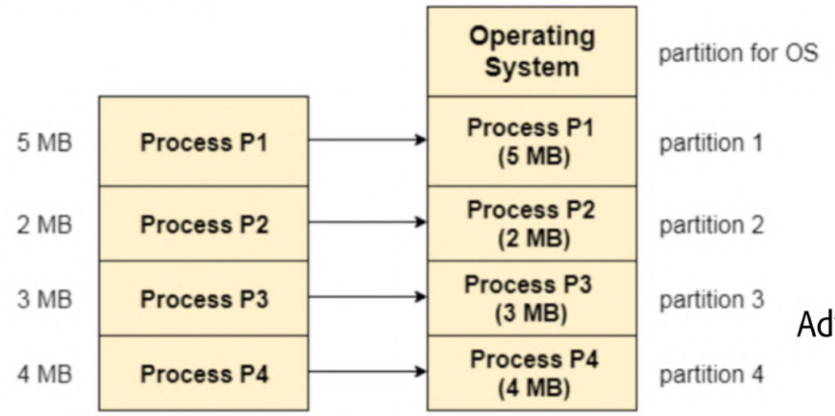
Here memory is not utilized efficiently.

Limitations:

1. **Internal Fragmentation:** if the size of the process is lesser then the total size of the partition then some size of the partition gets wasted and remain unused. This is wastage of the memory and called internal fragmentation.
2. **External Fragmentation:** The total unused space of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form.
3. Limitation on process size: If the process size is larger than the size of maximum sized partition then that process cannot be loaded into the memory. Therefore, a limitation can be imposed on the process size that is it cannot be larger than the size of the largest partition.
4. Low degree of multi-programming: In fixed partitioning, the degree of multiprogramming is fixed and very less because the size of the partition cannot be varied according to the size of processes.

2. Dynamic allocation

In this technique, the partition size is not declared initially. It is declared at the time of process loading.



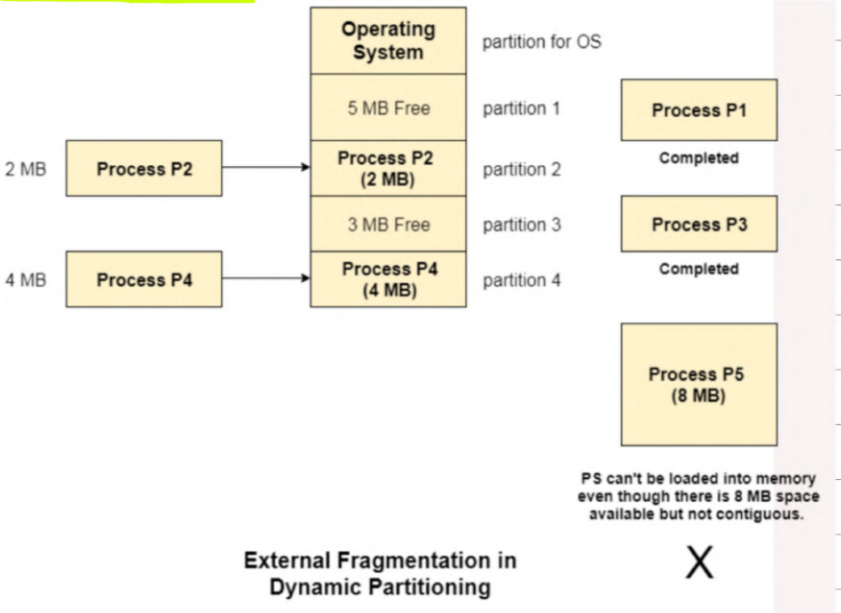
Dynamic Partitioning

(Process Size = Partition Size)

Advantages over fixed partitioning

1. No internal fragmentation
2. No limit on size of process
3. Better degree of multi-programming

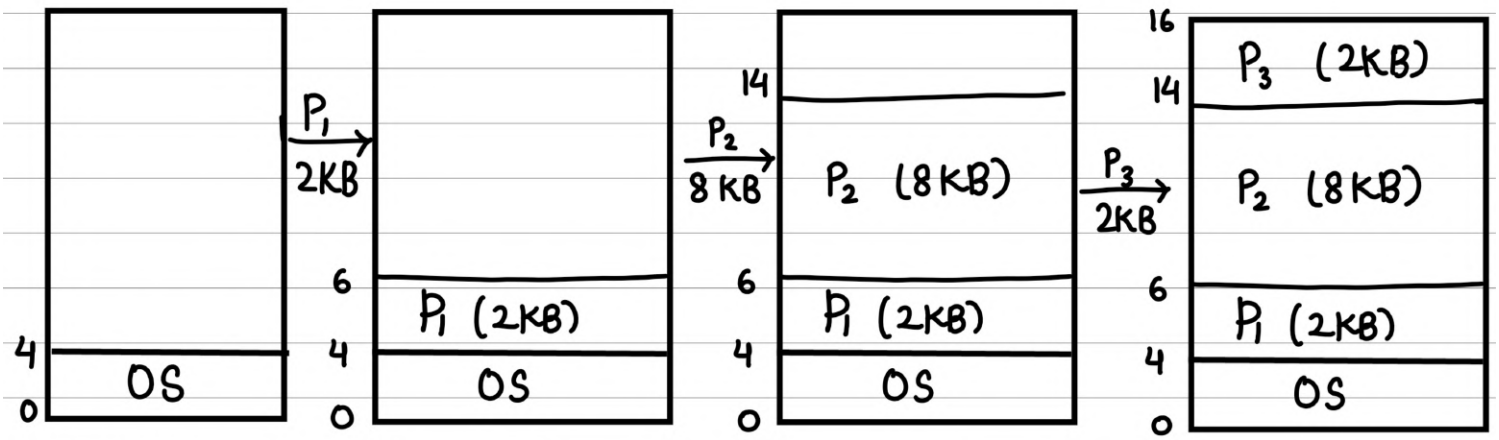
External fragmentation



External Fragmentation in Dynamic Partitioning

} disadvantage

Free Space management



FL : 4KB end of RAM → 12 KB free → NULL

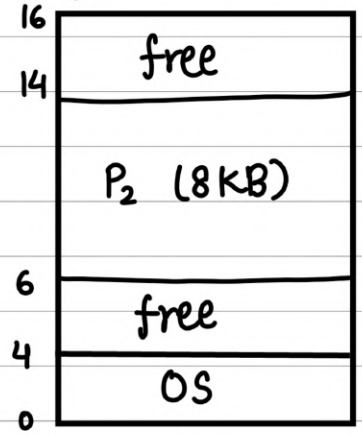
10 KB free → NULL

2KB free → NULL

NULL

Free holes in the memory are represented by a free list (Linked-List data structure)

after P_1 and P_3 exit



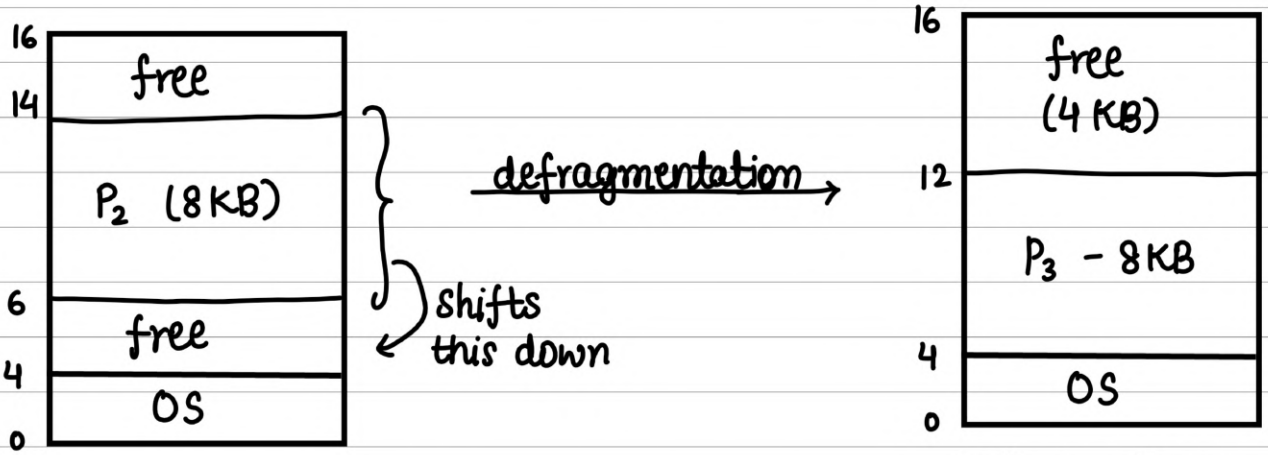
external fragmentation

OS keeps a free list (similar to linked list)

FL : 2KB → 2KB → null

Problem : P_4 → 3KB (Even when 4KB is free, but since memory is externally fragmented we cannot allocate 3KB)

Defragmentation / Compaction



Defrag - RAM

After defragmentation, Base (relocation register), offset (Limit) remains the same
So, user program is not affected as it continues to operate in logical address space.

NOTE: Defragmentation is a time consuming process which adds up to overhead, so, efficiency decreases.

How to satisfy a request of a of n size from a list of free holes?

a. Various algorithms which are implemented by the Operating System in order to find out the holes in the linked list and allocate them to the processes.

b. First Fit
   i. Allocate the first hole that is big enough.
   ii. Simple and easy to implement.
   iii. Fast/Less time complexity

c. Next Fit
   i. Enhancement on First fit but starts search always from last allocated hole.
   ii. Same advantages of First Fit.

d. Best Fit
   i. Allocate smallest hole that is big enough.
   ii. Lesser internal fragmentation.
   iii. May create many small holes and cause major external fragmentation.
   iv. Slow, as required to iterate whole free holes list.

e. Worst Fit
   i. Allocate the largest hole that is big enough.
   ii. Slow, as required to iterate whole free holes list.
   iii. Leaves larger holes that may accommodate other processes.

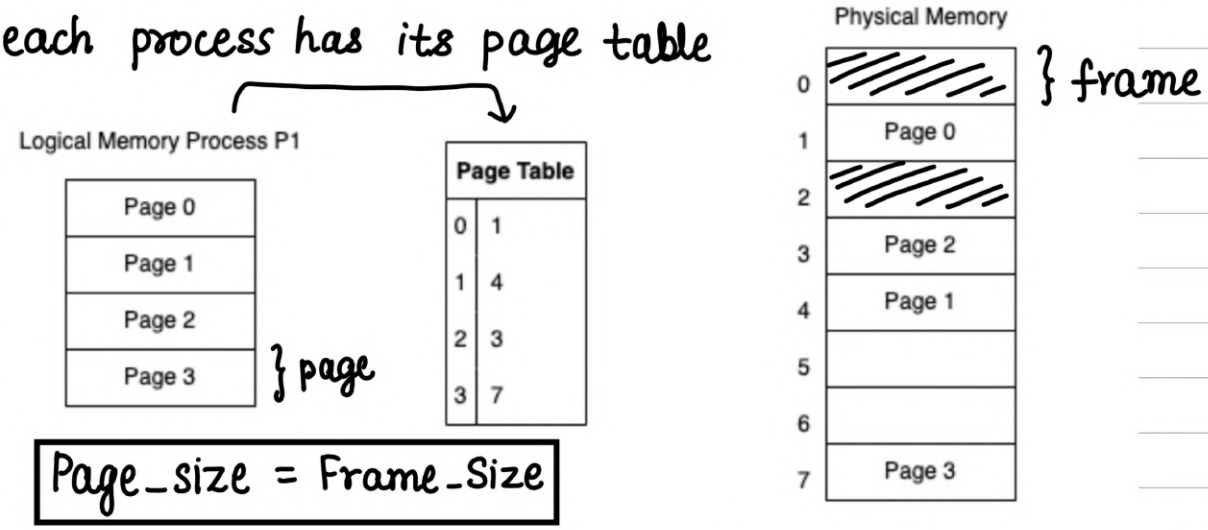
Comparisions

First fit → fast (worst case - O(N)), Simple
Next fit → tweak of first bit, fast, Simple
Best fit → least internal fragmentation, slow, major external fragmentation
Worst fit → major internal fragmentation, slow, lesser external fragmentation

Paging | Non-contiguous memory Allocation

Paging model of logical and physical memory

each process has its page table



Page_size = Frame_Size

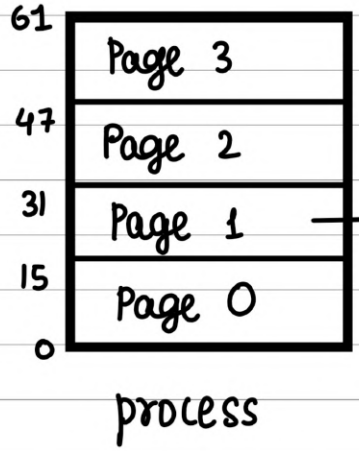
without defragmentation, we are able to allocated memory to a process

Paging

a. **Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous.**
b. It avoids external fragmentation and the need of compaction.
c. Idea is to divide the physical memory into fixed-sized blocks called **Frames**, along with divide logical memory into blocks of same size called **Pages**. (# Page size = Frame size)
d. **Page size** is usually determined by the processor architecture. Traditionally, pages in a system had uniform size, such as 4,096 bytes. However, processor designs often allow two or more, sometimes simultaneous, page sizes due to its benefits.
e. **Page Table**
    i. A Data structure stores which page is mapped to which frame.
    ii. **The page table contains the base address of each page in the physical memory.**
f. Every address generated by CPU (logical address) is divided into two parts: a page number (p) and a page offset (d). The p is used as an index into a page table to get base address the corresponding frame in physical memory.

g. Page table is stored in main memory at the time of process creation and its base address is stored in process control block (**PCB**).

h. A page table base register (**PTBR**) is present in the system that points to the current page table. Changing page tables requires only this one register, at the time of context-switching.



page no.(p)

$25 = (011001)_2 = 16 + 9$

offset (d)

from 16 page no. 1 starts
for page 1, base is 16 and offset is 1

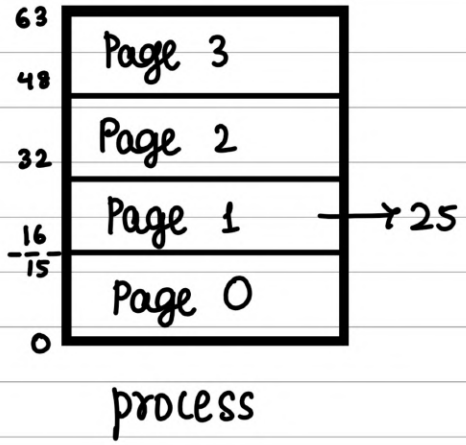
logical page no.	
00	0
01	1
10	2
11	3

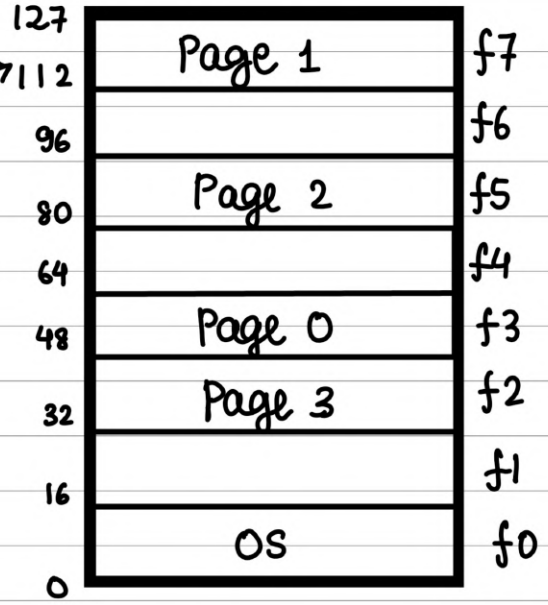
page no./base

011001 ≡ p | d

offset

Address Translation





RAM

for 25, offset is 9
so, physical address value is 112+9 = 121

$121 \equiv (1111001)_2$ | offset is same as logical address space

f d

We want to find a mechanism to map 01 (p) to 111 (f)

Logical address	01	1001
Physical address	111	1001

this mapping is found using page table

How Paging avoids external fragmentation?

a. Non-contiguous allocation of the pages of the process is allowed in the random free frames of the physical memory.

Why paging is slow and how do we make it fast?

a. There are too many memory references to access the desired location in physical memory.

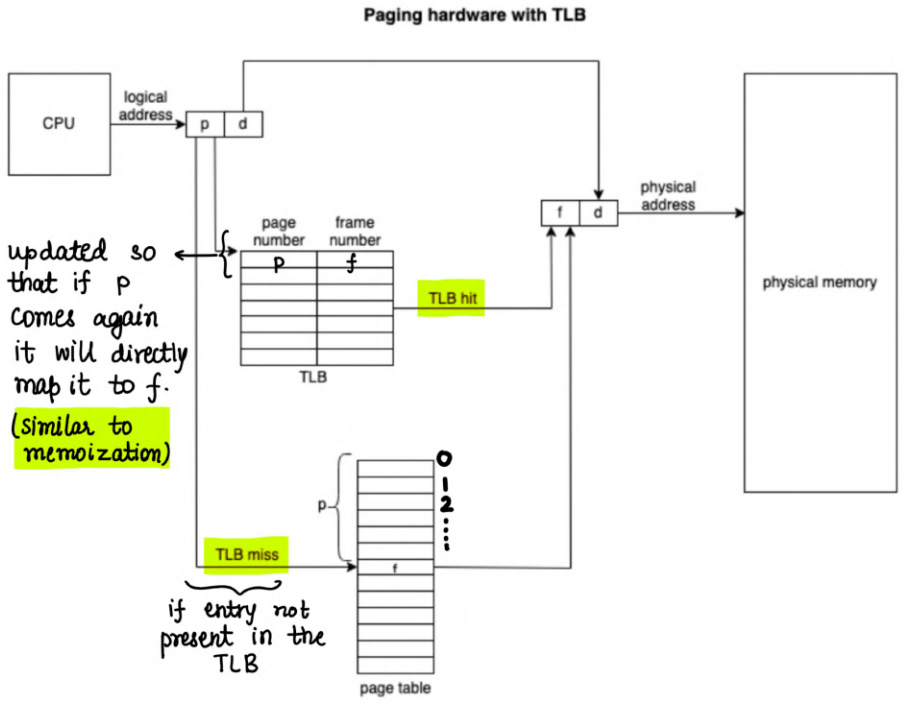
page table used to map logical address to physical address. This adds up to overhead

How to make paging fast ?

Ans: Translation look-aside buffer (TLB)

Translation Look-aside buffer (TLB)

a. A Hardware support to speed-up paging process.
b. It's a hardware cache, high speed memory.
c. TBL has key and value.



- Page table is stores in main memory & because of this when the memory references is made the translation is slow.
- When we are retrieving physical address using page table, after getting frame address corresponding to the page number, we put an entry of the into the TLB. So that next time, we can get the values from TLB directly without referencing actual page table. Hence, make paging process faster.

How TLB is updated during context - switching ?

$P_0 \longrightarrow$ Pno. 10 fno. 100

context switch

$P_1 \longrightarrow$ Pno. ~~10~~ fno. ~~100~~ (delete this)

} either flush the TLB as it can interfere in memory protection and isolation.

But context switching frequency is very high, so flusing it again & again will make TLP slow. So, the purpose of TLP is not fullfilled.

Better Solution

Add unique identifier that will identify unique process
This unique identifier is known as Address space identifiers. (ASID)

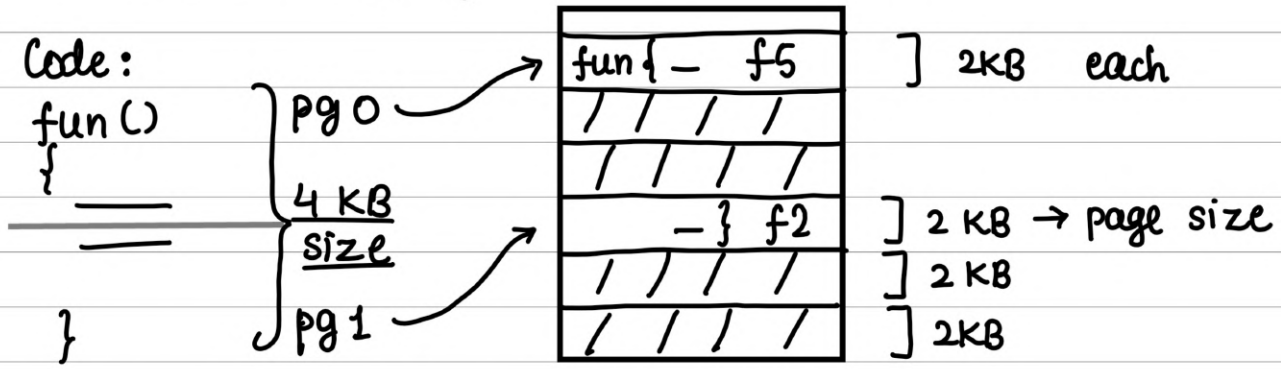
	ASID	Page no.	f no.
$P_0 \rightarrow$	0	10	100
$P_1 \rightarrow$	1	10	11

for different process, page table is different but TLB is same.

TLB hit, TLB contains the mapping for the requested logical address.
Address space identifier (ASIDs) is stored in each entry of TLB. ASID uniquely identifies each process and is used to **provide address space protection and allow to TLB to contain entries for several different processes.** When TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently executing process matches the ASID associated with virtual page. If it doesn't match, the attempt is treated as TLB miss.

Segmentation

Problem in paging

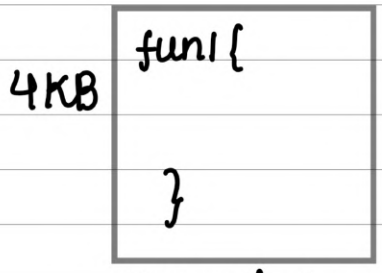


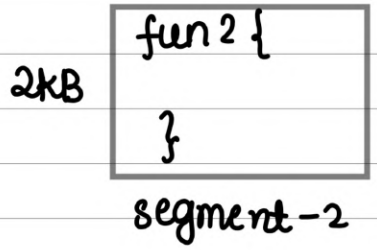
↳ OS will divide it into 2 pages

Now, we will have to switch frame to run the whole code. This makes the process inefficient.

Segmentation is memory management technique that supports the **user view of memory.**

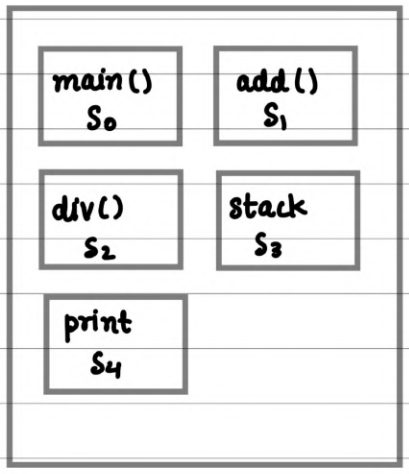
↳ address space is partitioned into different sizes



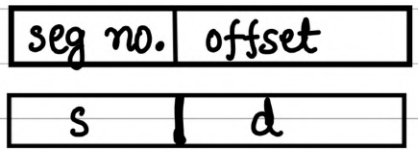


→ different segments
→ varying sizes
→ user view

variable partitioning of address space



segment size is different



Each segment has **segment number and offset,** defining a segment.
<segment-number, offset> {s,d}

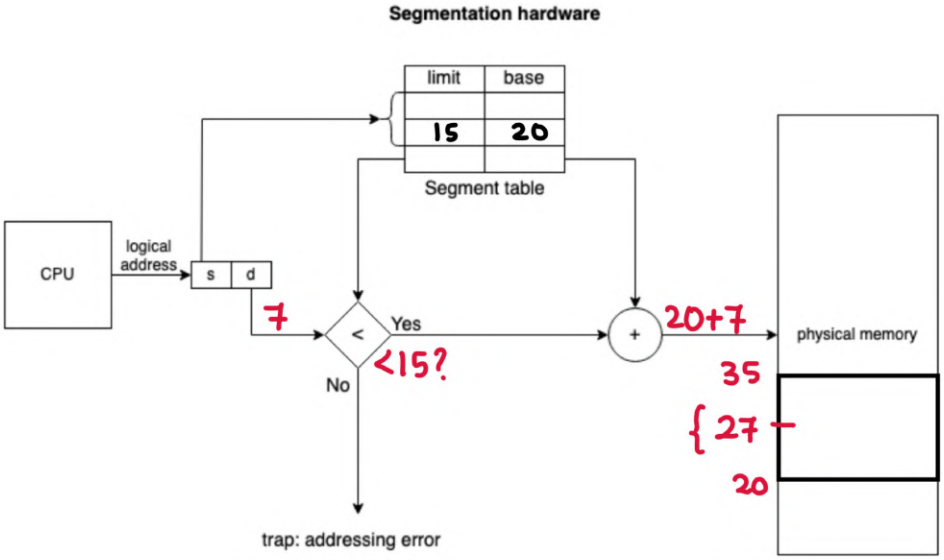
MMU → memory management unit (hardware that maps logical address to physical address).

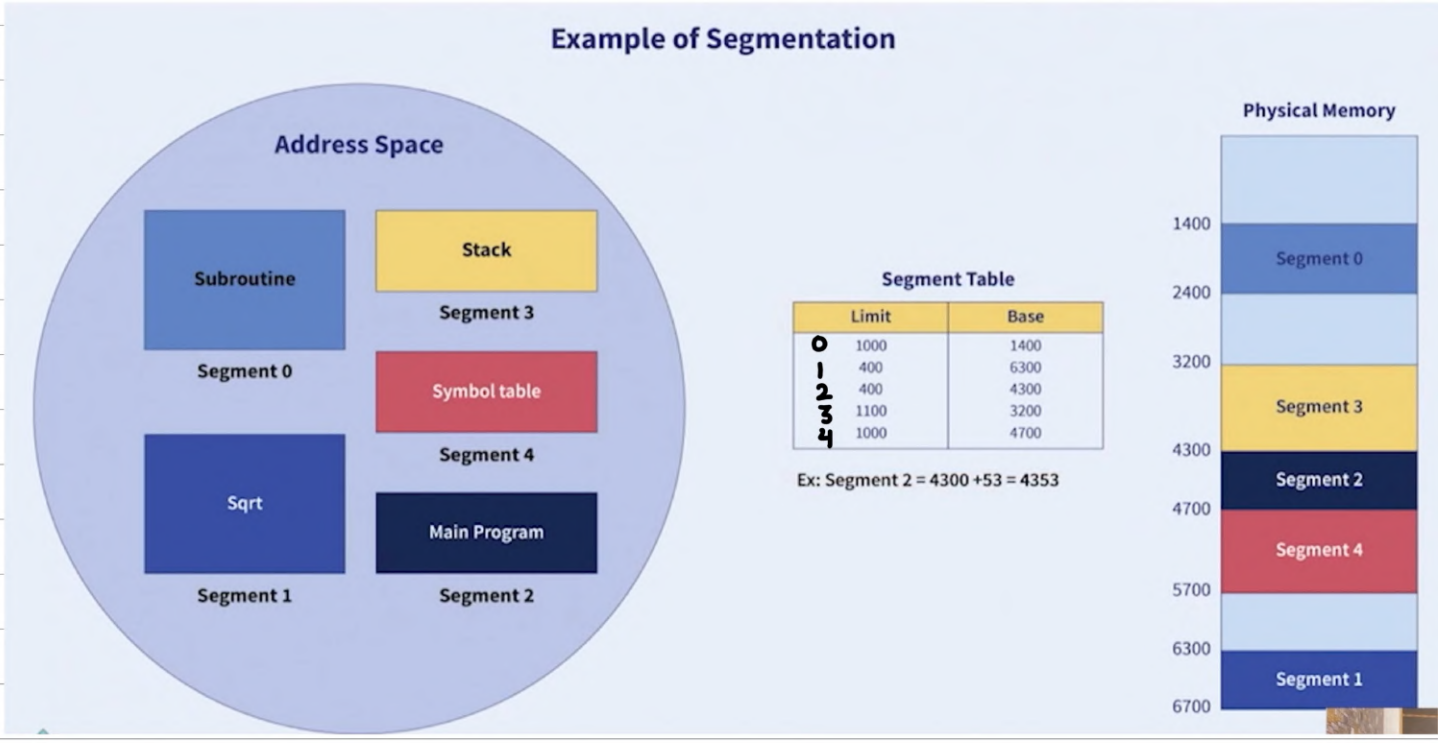
How MMU does translation here?

01 111

s d

1 7





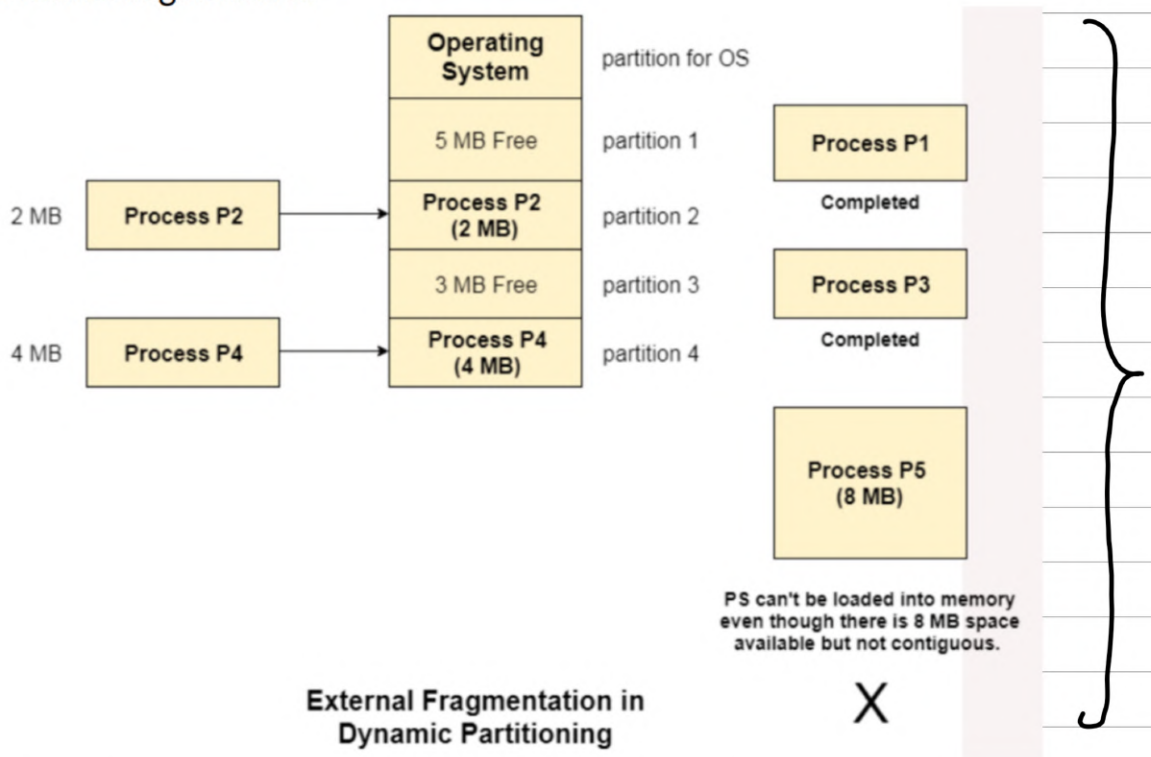
It is better to have segmentation which divides the process into the segments. Each segment contains the same type of functions such as the main function can be included in one segment and the library functions can be included in the other segment.

Advantages:

a. No internal fragmentation.
b. One segment has a contiguous allocation, hence efficient working within segment.
c. The size of segment table is generally less than the size of page table.
d. It results in a more efficient system because the compiler keeps the same type of functions in one segment.

Disadvantage

External fragmentation



As segment cannot be broken further, we need to allocate it in one contiguous memory.

Disadvantages:

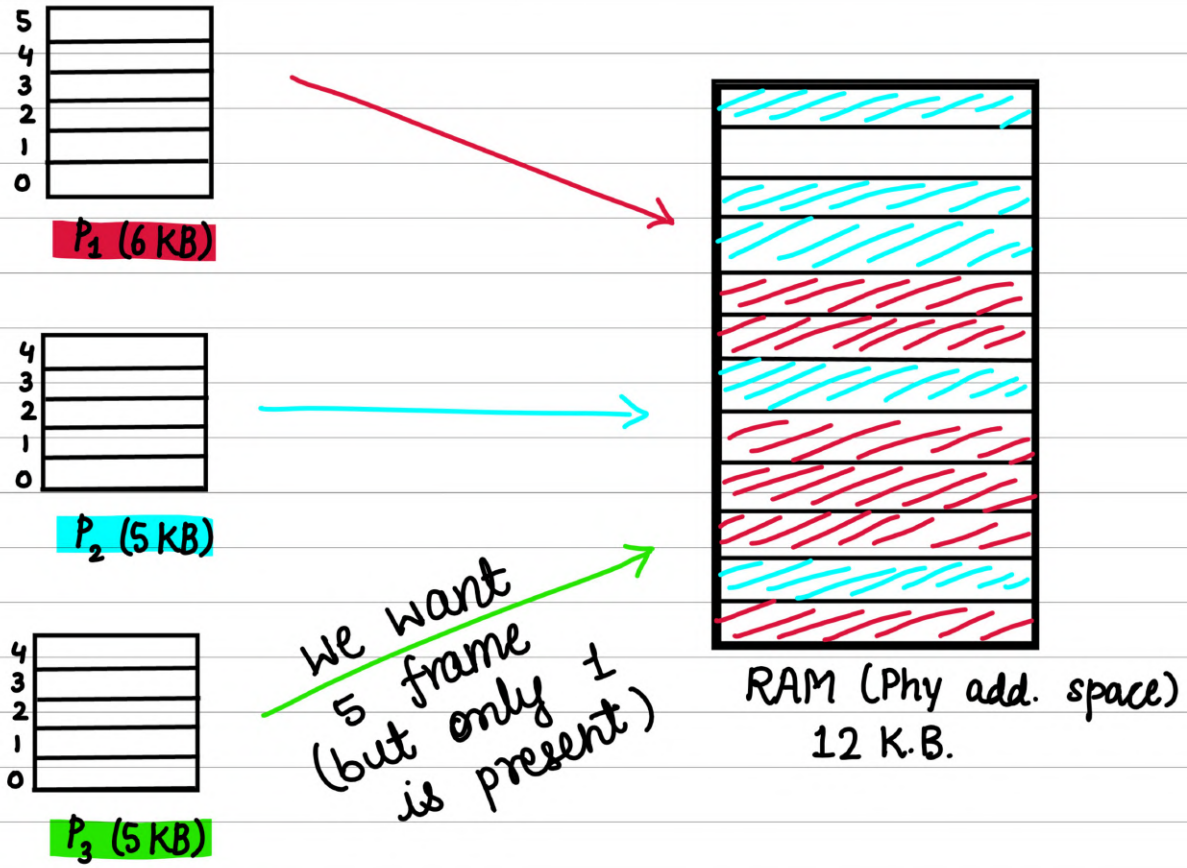
a. External fragmentation.
b. The different size of segment is not good that the time of swapping.

Modern System architecture provides both segmentation and paging implemented in some hybrid approach.

In paging there is internal fragmentation

program = 15 KB page size → 2KB

Virtual Memory Management



To handle this, we use virtual memory.

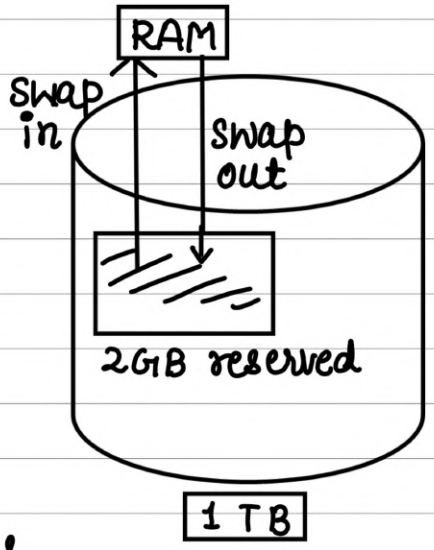
Disk → 1 TB

RAM → 8 GB

We only keep those pages in RAM which are needed for running the program.

Rest of the pages are stored in a reserved area of the disk

Whenever a page from reserved area is needed, it is shifted to RAM. This reserved area is called **swap space.**



Advantage of this is, programs can be larger than physical memory.

Degree of multiprogramming also increases

RAM + SWAP SPACE → virtual memory

- **Virtual memory** is a technique that allows the execution of processes that are not completely in the memory. It provides user an illusion of having a very big main memory. This is done by treating a part of secondary memory as the main memory. (Swap-space)

- Programmer is provided very large virtual memory when only a smaller physical memory is available.
- **Demand Paging** is a popular method of **virtual memory management.**
- In demand paging, the pages of a process which are least used, get stored in the secondary memory.
- A page is copied to the main memory when its demand is made, or **page fault** occurs. There are various **page replacement algorithms** which are used to determine the pages which will be replaced.

Swap-in is done using page replacement algorithms. Even if all frames in RAM are filled, we will have to replace one frame.

Demand paging is done using lazy swapper. It swaps in only those pages which are needed by the program.

We are viewing a process as a sequence of pages, rather than one large contiguous address space, using the term **Swapper is technically incorrect**. A swapper manipulates entire processes, whereas a **Pager** is concerned with individual pages of a process.

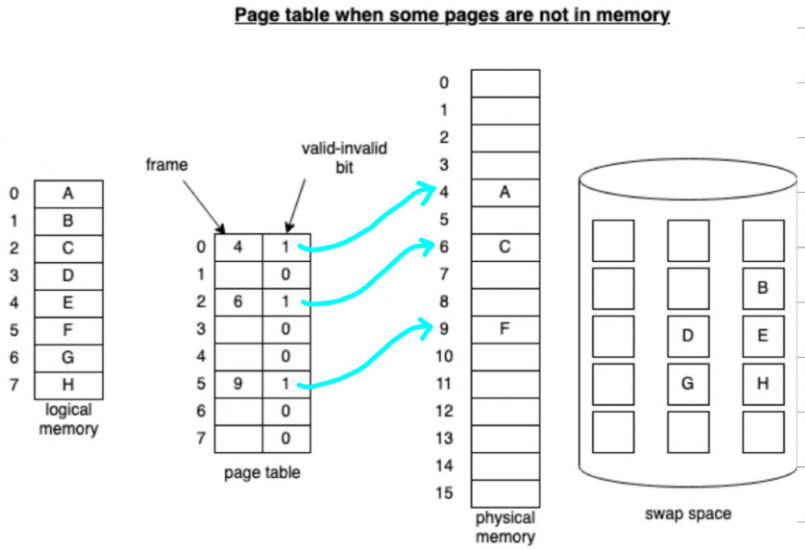
Q) How demand paging works ?

Ans: i) When a process is swapped-in, pager guesses which page will be used. It avoids reading into memory, pages that will not be used anyways.

ii) The **valid-invalid bit scheme in the page table** is used to distinguish between pages that are in memory and that are on the disk.

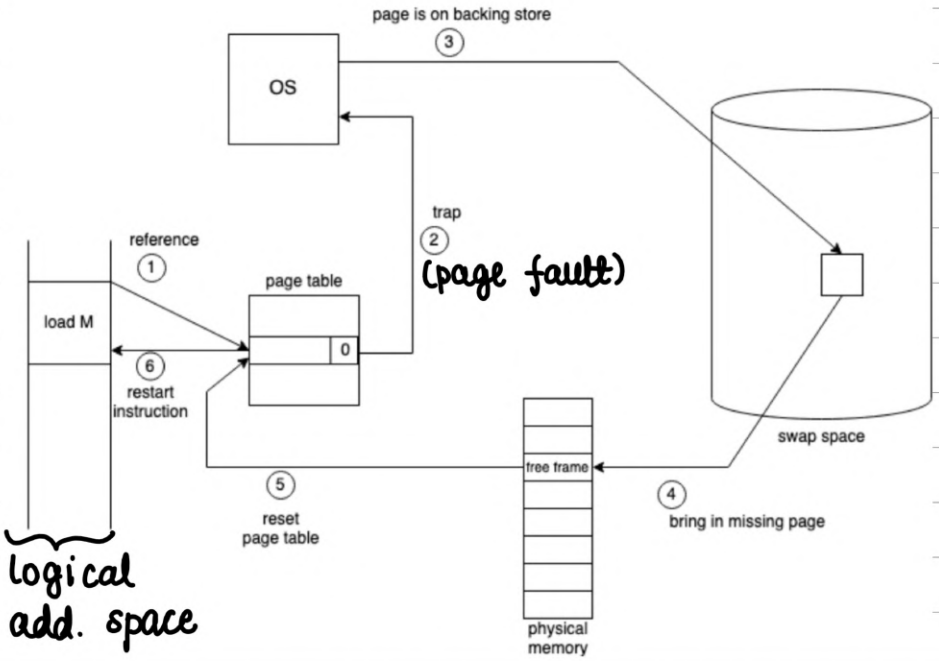
i. Valid-invalid bit **1** means, the associated page is both legal and in memory.
ii. Valid-invalid bit **0** means, the page either is not valid (not in the LAS of the process) or is valid but is currently on the disk.

(LAS - logical address space)



Q) What happens when OS tries to access page which are in swap-space (not in physical memory)?

Ans: This scenario is called page fault.
The procedure to handle page fault is:



i. Check an internal table (in PCB of the process) to determine whether the reference was valid or an invalid memory access.
ii. If ref. was invalid process throws exception.
If ref. is valid, pager will swap-in the page.
iii. We find a free frame (from free-frame list)
iv. Schedule a disk operation to read the desired page into the newly allocated frame.
v. When disk read is complete, we modify the page table that, the page is now in memory.
vi. Restart the instruction that was interrupted by the trap. The process can now access the page as through it had always been in memory.

Pure demand paging

Pure Demand Paging

i. In extreme case, we can start executing a process with no pages in memory. When OS sets the instruction pointer to the first instruction of the process, which is not in the memory. The process immediately faults for the page and page is brought in the memory.

ii. Never bring a page into memory until it is required.

Locality of Reference

We keep pages that pager thinks will be needed in the process. This reduces page faults increasing performance.

Advantages of Virtual memory

a. The degree of multi-programming will be increased.

b. User can run large apps with less real physical memory.

Disadvantages of Virtual Memory

a. The system can become slower as swapping takes time.

b. **Thrashing** may occur.

Page Replacement Algorithms :-

During Page fault,

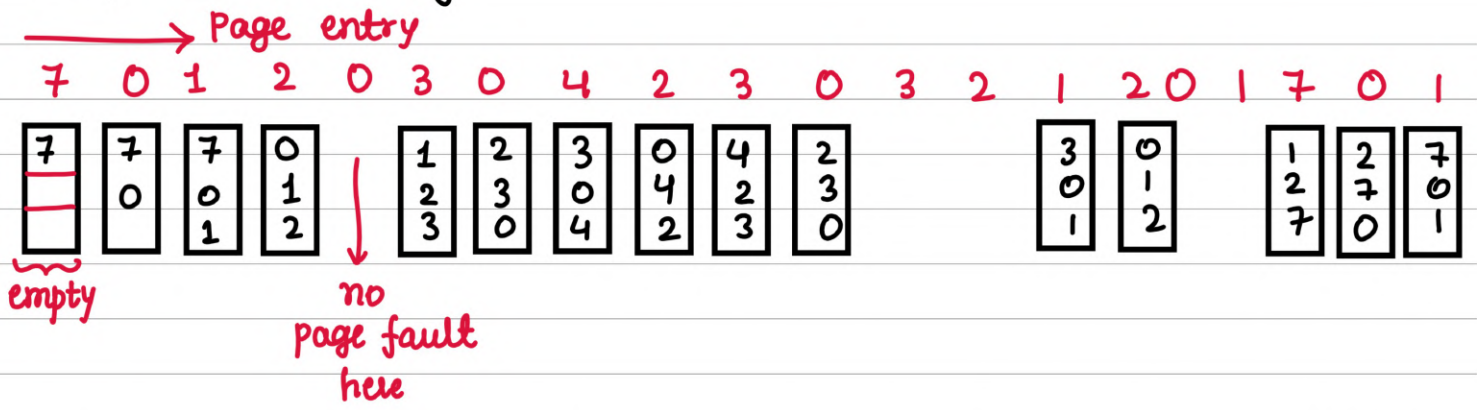
The **page replacement algorithm** decides which memory page is to be replaced. Some allocated page is swapped out from the frame and new page is swapped into the freed frame.

Algo - 1

First - in - First - out

i. Allocate frame to the page as it comes into the memory by **replacing the oldest page.**
ii. Easy to implement.

Reference string



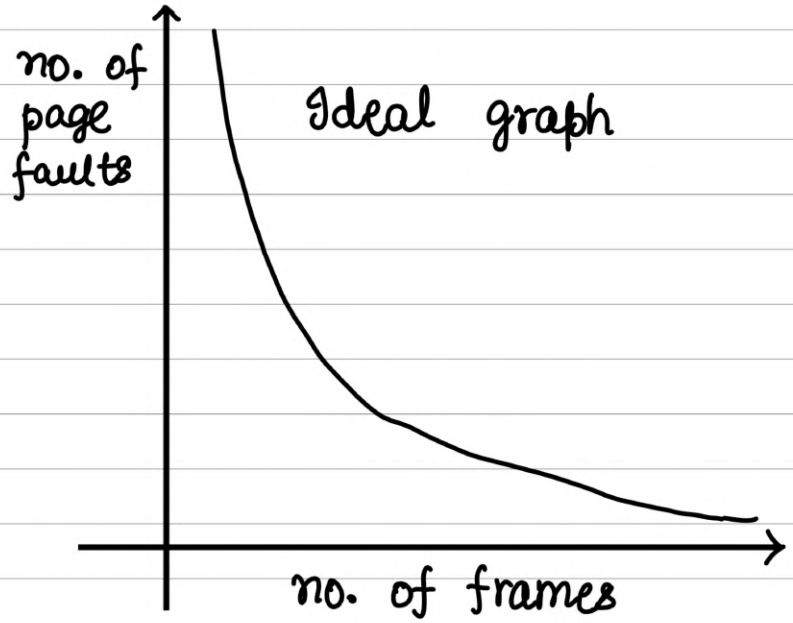
In FIFO → 15 page faults are there. For each page-fault we will have a page fault service time (as we need to add page from swap-space which needs time.)

Performance is **not** always good

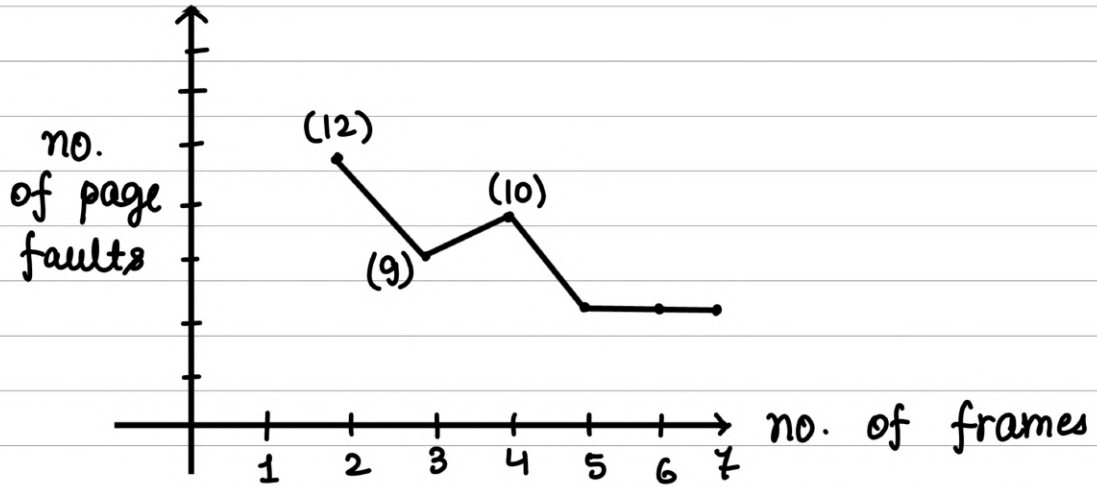
1. The page replaced may be an initialization module that was used long time ago **(Good replacement candidate)**
2. The page may contain a heavily used variable that was initialized early and is in content use. (**Will again cause page fault**)

Belady's Anamoly :

Even when no. of frames increases, sometimes, page faults in this algorithm increases, which is not desired.



Ref : 1 2 3 4 5 1 2 3 4 5



Algo - 2 : Optimal page Replacement

→ Best → Page faults minimum
→ impossible to implement

i. Find if a page that is never referenced in future. If such a page exists, replace this page with new page.
If no such page exists, find a page that is **referenced farthest in future**. Replace this page with new page.
ii. **Lowest** page fault rate among any algorithm.
iii. Difficult to implement as **OS requires future knowledge of reference string** which is kind of impossible. (Similar to SJF scheduling)

Algo 3: LRU (least recently used)

→ approximation based on past.

How to implement?

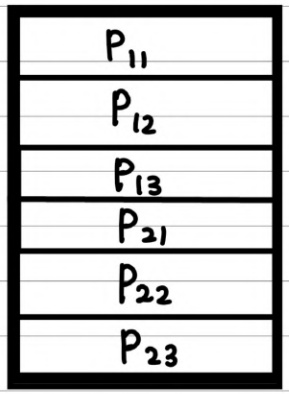
1. **Counters**
    a. Associate time field with each page table entry.
    b. Replace the page with smallest time value.
2. **Stack**
    a. Keep a stack of page number.
    b. Whenever page is referenced, it is removed from the stack & put on the top.
    c. By this, most recently used is always on the top, & least recently used is always on the bottom.
    d. As entries might be removed from the middle of the stack, so Doubly linked list can be used.

Algo 4: Counting based page replacement:

i. Least frequently used (**LFU**)
    1. Actively used pages should have a large reference count.
    2. Replace page with the smallest count.
ii. Most frequently used (**MFU**)
    1. Based on the argument that the page with the smallest count was probably just brought in and has yet to be used.
iii. Neither MFU nor LFU replacement is common.

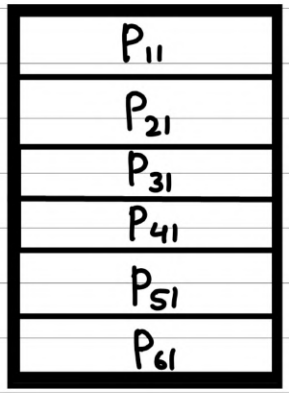
Thrashing

Case I



RAM

Case II



RAM

P_{ij} → j^{th} page of process i

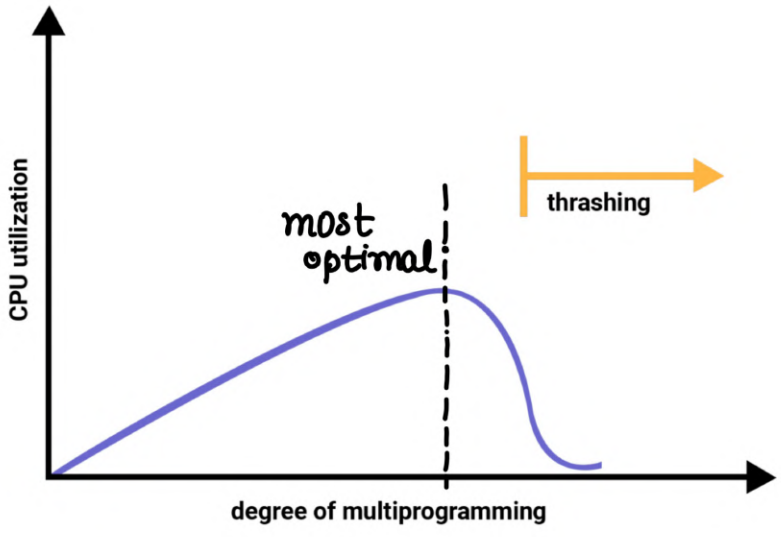
Q. Which has more no. of page faults ?

Ans: Case II, as there are less pages corresponing to each process.

Degree of multiprogramming is better in case 2 but case 1 is more efficient due to lesser page faults.

In case II, CPU will be more busy in servicing page faults.

- If the process doesn't have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.
  This **high paging activity is called Thrashing.**
- A system is Thrashing when it **spends more time servicing the page faults than executing processes.**



*** Cause of Thrashing:**

1 Initial low CPU utilization - - -↑ ing degree of multi-programmings.

2 A Global Page replacement algo, replaces pages w/o regard to the process

3 A process may need more frames _ . . cause faults again

4 Other processes 's frames are replaced & they may need those soon

5 As a result CPU utilization decreases

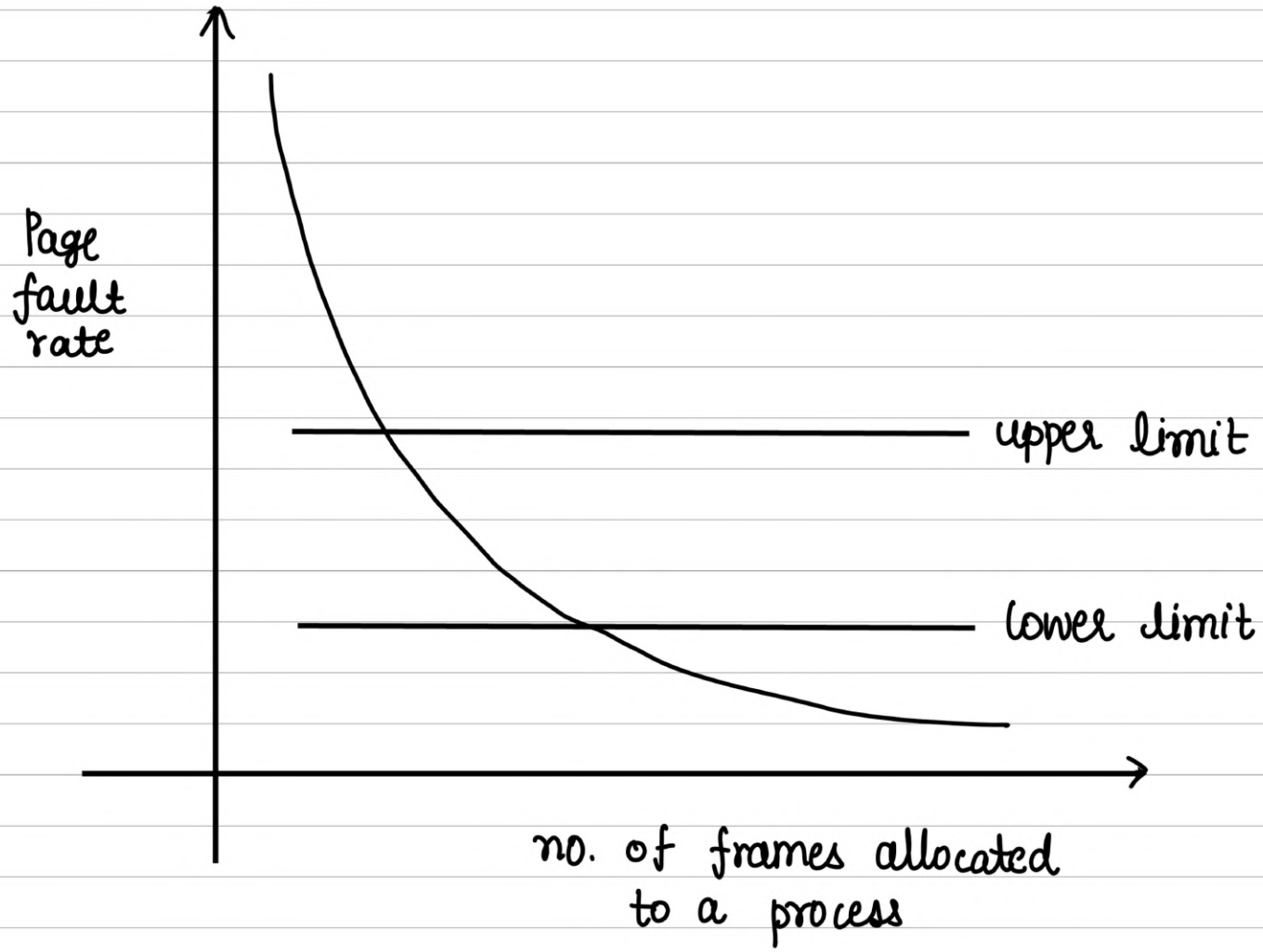
6 CPU scheduler now, may increase CPU utilization by increasing degree of multi-programming

7 Ultimately , CPU utilization drops drastically

Techniques to handle Thrashing:

i. **Working set model**
   1. This model is based on the concept of the **Locality** Model.
   2. The basic principle states that if we allocate enough frames to a process to accommodate its current locality, it will only fault whenever **it moves to some new locality.** But if the allocated frames are lesser than the size of the current locality, **the process is bound to thrash.**

ii. **Page Fault frequency**
   1. **Thrashing** has a high page-fault rate.
   2. We want to **control** the page-fault rate.
   3. When it is too high, the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames.
   4. We establish upper and lower bounds on the desired page fault rate.
   5. If pf-rate exceeds the upper limit, allocate the process another frame, if pf-rate fails falls below the lower limit, remove a frame from the process.
   6. By controlling pf-rate, thrashing can be prevented.


Page fault rate
upper limit
lower limit
no. of frames allocated to a process