

Software: A set of programs

* collection of programs working together to achieve some specific goal.

Types of sw

Application sw

Ex: ms office
User friendly

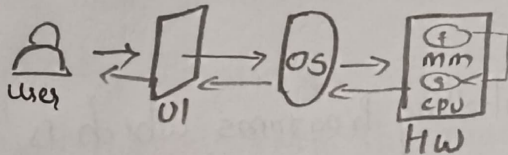
System sw

Ex: OS, compiler etc
System friendly

Operating System

- ① * A sw which acts as an interface b/w hardware & user
- ② * It's a resource manager
- ③ * It's a resource allocator
- ④ * It's a ~~system~~ control program set of

①



* OS acts as an interface b/w user & Hardware.

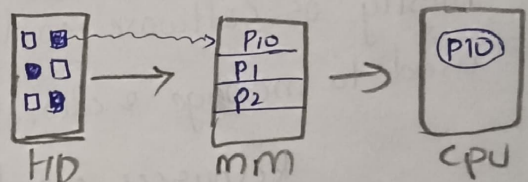
* When user interacts with UI

like CLI (command line interface)
or GUI (Graphical User Interface)

it generates events that handled by OS

* like if we open file, then OS loads it in MM & execute it in CPU by scheduling then o/p of CPU is back again to OS & it shows on screen

② Resource manager



To load mm
* When file needs to be loaded in
Ex: ppm for execution

* OS ask file system, where file is located on disk.

* OS Reads from it

* OS manage MM whether space is there or not if not how to free it & loads

the file content in MM

To display
* OS manage execution in CPU as it schedules which instruction

to be performed next on CPU to open file on screen

* on print statement execution OS sends data to display device driver.

③ Resource Allocation

OS allocates Hardware Resources like HD, MM, CPU, I/O etc

* when file is clicked to ~~open~~ saved

OS uses free disk space management approaches to find which block is available & save the data into it

* when loading data into RAM.

OS Allocates specific Area of ~~space~~ In user space to load data from HD.

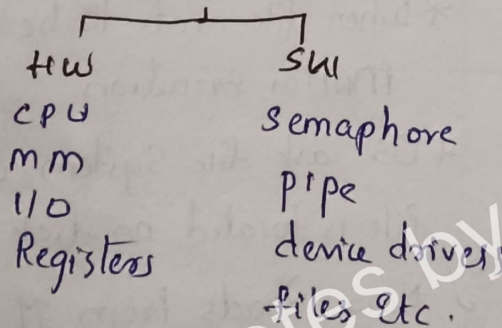
* Schedules CPU time.

OS Allocates specific time slot for each process.

④ Set of control Programs.

mostly OS software contains controlling programs which is used to manage & allocated Resources Efficiently

Resources can be



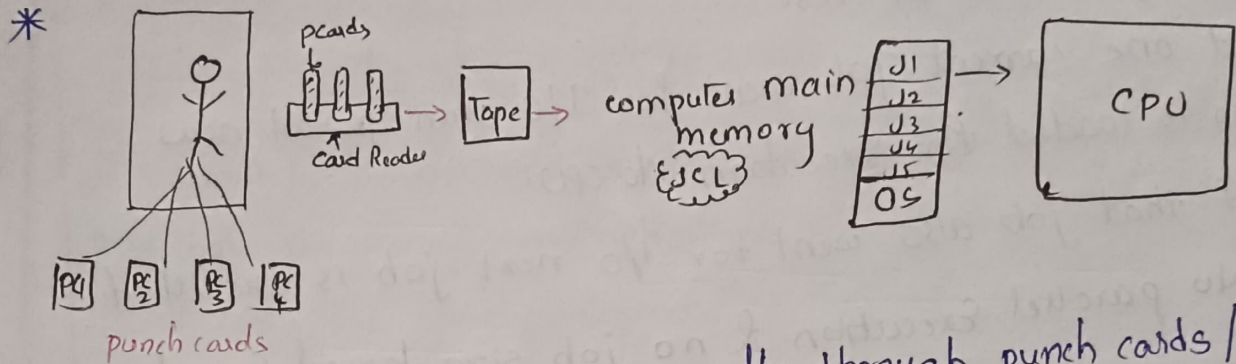
Goals of OS.

- ① Convenience: user friendly & Easy to use.
- ② Efficient: efficiently utilize Resources
- ③ Reliable: Error free → Robust Avoid deadlocks
- ④ Portable: Run on multiple Architectures

Types of OS

① Batch OS:

A type of OS in which jobs are grouped into batches and executed automatically one after other (No HDD jobs are loaded by Operator manually)



- * Jobs were submitted to mm manually through punch cards / tapes
- * Operator Physically loads cards into computer
- * Main memory reads the programs & jobs are grouped & executed one after another Automatically.
- * Jobs are not stored in disks etc,

Efficiency ↓↓↓↓
Throughput ↓↓↓↓

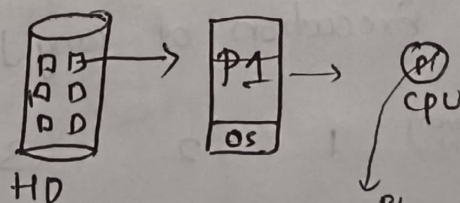
② Uniprogramming OS:

* It's an OS capable to load only one job from disk to main memory and execute at CPU

* No multi-tasking

* As Jobs are stored in disk, large no. of jobs can be stored at a time & can be serially executed on CPU.

Efficiency ↓↓↓↓
Throughput ↓↓↓↓



Notes by Abdul Muqsith

③ Multiprogramming OS

In this type of OS, the multiple processes can be loaded into main memory, if one job is executing in CPU other jobs are at rest

→ If one current job went for I/O then ~~parallel~~ new job is loaded for execution into CPU.

→ If that job also went for I/O next job is executed.

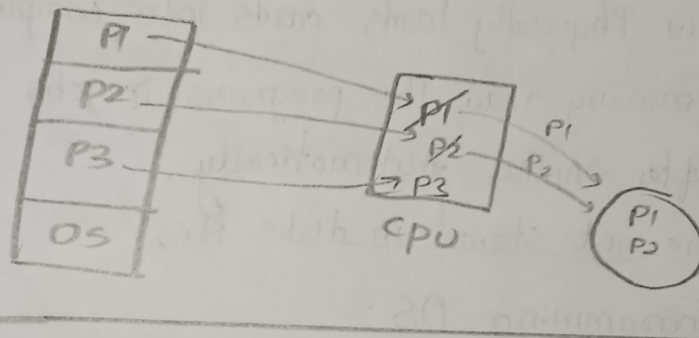
→ No parallel execution & no job size based execution

Efficiency ↓↓

Throughput ↓↓

No starvation x

But Convey Effect ✓



④ Time sharing / multitasking OS.

→ It's an extension of multiprogramming MP + preemption

→ CPU time is divided among multiple jobs

→ Each job gets fixed amount of time to execute

→ Even if larger job came first, shorter at last. The each job gets fixed time for execution

→ Before completely execution of long job → shorter is completed

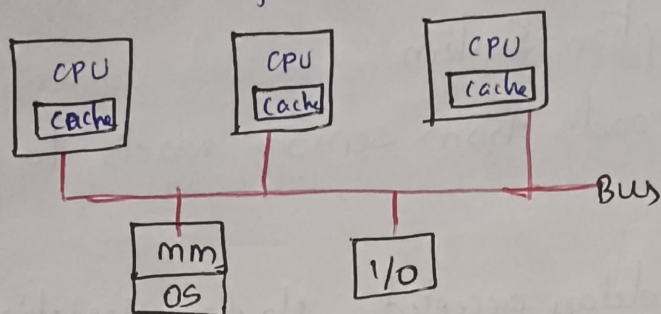
Jobs (arrival time)	1	2	3	4	let TQ = 5
cpu time for execution	20	5	40	2	
	15	0	35	0	

* CPU takes 5 time units for each job

* Job 2 & 4 are executed first

④ Multiprocessor System

→ It's also called as parallel systems



Throughput ↑
Efficiency ↑

- * several processors - common bus, memory, I/O devices & system clock
- * Single OS manage multiple processors
- * all CPU's are closely interconnected together

⑤ Distributed OS

* Multiple independent computers located at diff geographic locations connected through internet

① A DOS consists of local OS on each node along with a logical distributed layer of services

② local OS → manage node independently - (each node handle its own)

Distributed layer → handles coordination b/w nodes

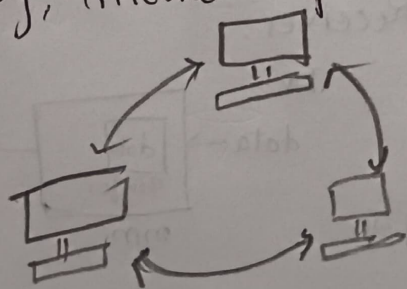
DOS = local OS + Distributed Service layer

* When one task arrives it's divided among different nodes
* Each node process it parallelly.

* Throughput ↑↑ Efficiency ↑↑

* Resource sharing, fault tolerance, scalable.

* security, internet dependency, slower than local.



Real time OS

RTOS is the OS which reacts to inputs & responds them quickly

*Ex: missile interception system

→ RTOS, typically reads from sensor & reacts to actuators

→ 2 types

Hard RTOS = No delay accepted, no time violations or flitting
Ex: missile, air bag

Soft RTOS = some delays & time variations are accepted
Ex: Video games

→ (Execution of some other job + handling I/O of other job)
Spooling: spooling is an OS technique in which data which

Spooling	Buffering
1) I/O of one job + Execution of other job	1) I/O of one job + Execution of same job
2) Secondary memory is used (Disk)	2) primary memory is used (RAM)
3) Ex: printing + gaming	3) Ex: for Buffering Video in advanced while watching
4) used for very slow device	4) Speed matching.
5) CPU wait time & No	5) Reduced.

Notes by Abdul Muqasith

Job	execution of same job	mm	Buffer data
Ex printing + gaming	Ex: Buffering video in advanced while watching youtube video.		

Components of OS.

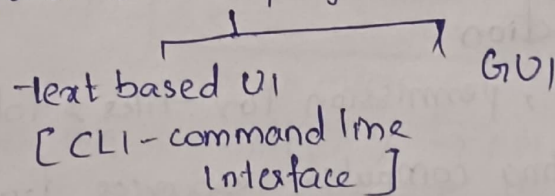
- ① Process management = create, suspend, Resume process etc
- ② main memory management = Allocation & managing mm
- ③ Secondary memory " = Read/writing technique
- ④ File management = file operations handling
- ⑤ I/O management = I/O device enabling / data receiving & sending etc.

Services of OS

OS provides various services to the users so, that user can easily understand & interact with the system.

① User Interface (UI)

User interface helps user to interact with services provide in computer or to program it



② Program Execution

- * loading data, allocating RAM & CPU time
- * Returning result.

③ File System management.

- * create, deletes, opens, closes files,
- * Organizes files into folders.

④ I/O operation:

- * controls flow of data between CPU & devices
- * speeds up performance using spooling & buffering
- * hide details of I/O to the user using GUI

⑤ Communication

* OS enables different processes to communicate with each other securely.

* Processes may be in same computer (pipes) or diff computers (sockets)

⑥ Resource Allocation

Allocates CPU time, schedules tasks, allocates memory etc.

⑦ Error detection

Error may occur in CPU, MM, I/O, user programs & based on error OS gives alerts, warnings & fixes it safely

⑧ Networking Services.

manage network connections between computers & handles protocols, routing & data transfer

⑨ Security & Protection

* Login systems, permission for files & folders (^{Security} ~~internal protection~~)

* ~~Securing~~ protecting computer resources from malware or illegal access (protection) as well as protecting processes from

⑩ Accounting

logs users activity [what, who, when accessed] used for statistics, Billings & Improving System

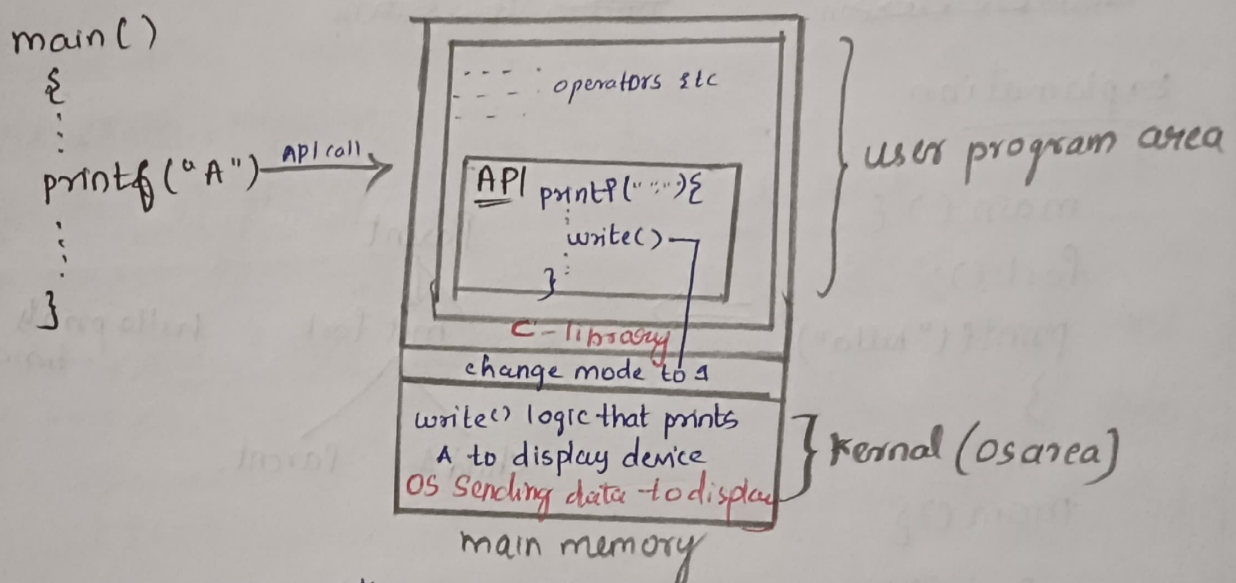
System call

System call provides the services of OS to user programs

* It's a special call/request made by a program to the OS to perform a task that program itself cannot do directly.

Example

printing data on screen using `printf()` of C library.



* When user writes `printf("A")` → API call

API call = A program calls a function written in a library.

* ~~the~~ using library `printf` prepares data

* `printf` call `write()` → which is a system call

* CPU enters in kernel mode

* OS uses display drivers

* letter 'A' gets printed on screen.

Types of Services.

- ① Process control : used to manage processes Ex: `fork()`, `exec()`, `exit()`
- ② File management : used to work with files `open()`, `read()`, `close()`
- ③ device management : used to control HW device `read()`, `write()`
- ④ Information ^{maintenance} : used to get/set system information `getpid()`
- ⑤ command : used to communicate b/w processes `pipe()`, `shmget()`

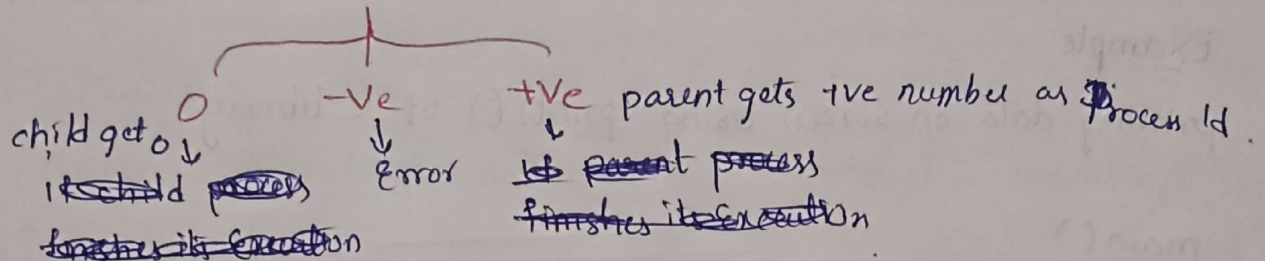
fork()

It's a system call of process control

* Used to create child processes.

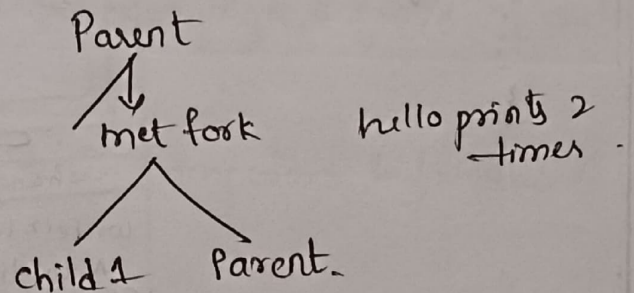
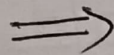
* Child is almost a duplicate of parent process [same code, etc.]

* fork returns 3 values types

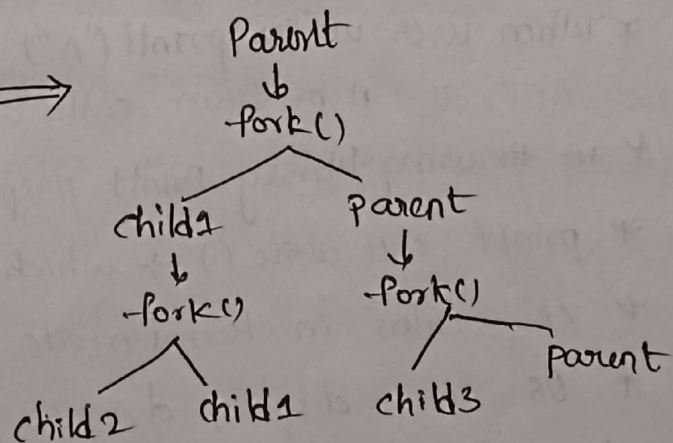
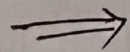


Explanation

```
main() {  
    fork();  
    printf("hello")  
}
```



```
main() {  
    fork();  
    fork();  
    printf("hello")  
}
```



Conclusion

In a program where `fork` is called, from that line of code new process [child] is created, code before `fork` is ~~not~~ copied in child but in entire program runs once.

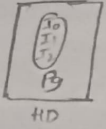
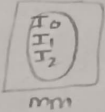
Formula :- $2^n - 1$
n = no. of times `fork` written
 2^n = no. of times program executed
childs created = $2^n - 1$

Ch-2 Process Management

- i) Program vs process
- ii) Attributes of process
- iii) context of process
- iv) context switching
- v) Process state transition diagram
- vi) CPU scheduling
 - times
 - FCFS

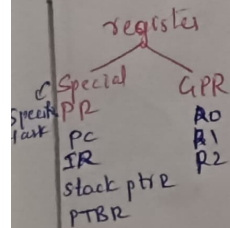
Notes by Abdul Muqsith

Process Management.

Program	Process
1) Program always remain on Hard-disk. 	1) Program when loaded inside main memory CPU or RAM  is in is in running status. (executing status)
2) Program is a single entity	2) Process can be multiple from a single program Ex: opening chrome 3 times creates 3 processes. or fork() in a program
3) Program is a passive entity	3) Process is an active entity
* It's like a deadbody	* It's like a alive person.

Attributes of the Process

- * ① pid : unique number given by OS to running process to identify/manage
- * ② state : Ready, running state etc
- * ③ priority : Importance given to each process
- * ④ program counter (pc) : which is the next instruction CPU will execute is stored in this register.
- ⑤ GPR : General Purpose Register like R0-R7 (Bank Rgs)
- ⑥ List of open files : What files are open, using, how long its using etc
- ⑦ List of open Devices : What devices are open, used, how long etc by the process
- ⑧ memory information : how much memory a process consumes etc.



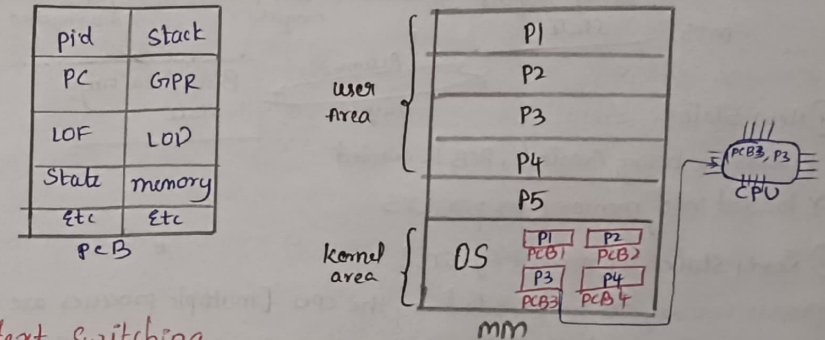
PCB

Context of the Process: It refers to the complete state of process at a particular moment of time. like PC, stack, Registers, PSW, process state (Running, ready, waiting).

PCB [Process Control Block]

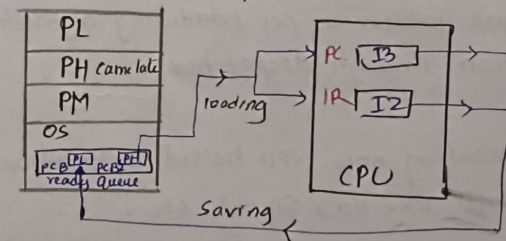
PCB is a data structure maintained by operating System. It's a structured storage of all attributes of process like PID, process state, GPR, List of files & devices, memory management, PC etc.

* OS can have multiple PCB's, each PCB stores individual process information.



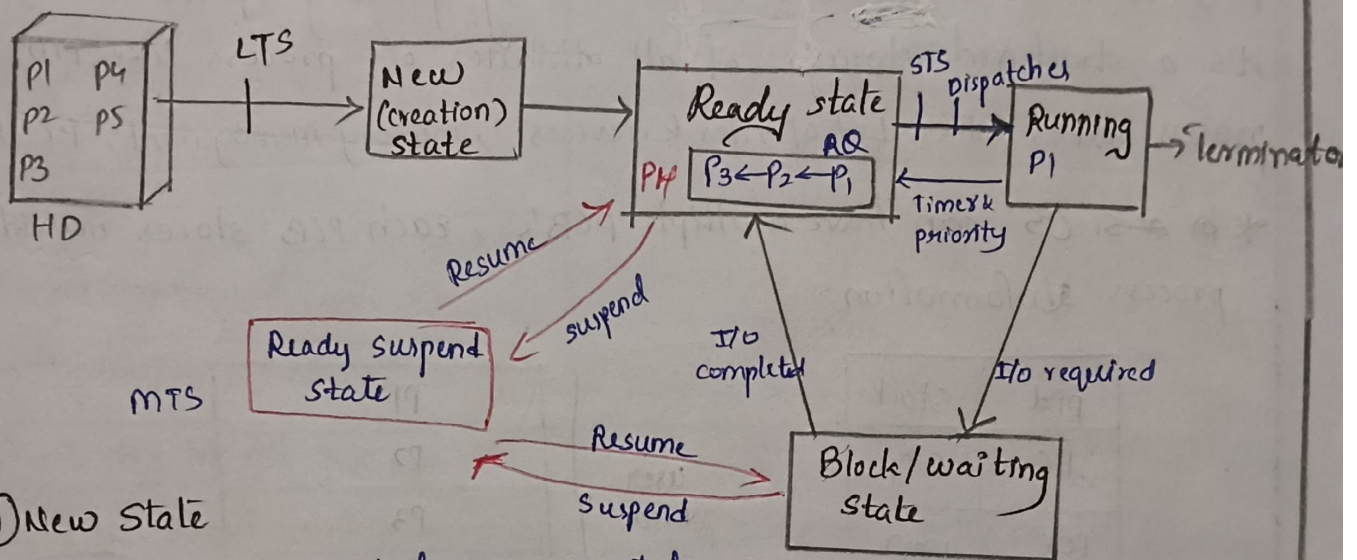
Context Switching

saving the context of running process & loading the saved context of other process is called context switching



Process State transition diagram

The process state transition diagram shows how a process moves between New, Ready, Running, waiting & terminated states during Exec



① New State

process is being created, PCB is created & loaded into memory as per LTS

② Ready state [managed by STS]

process is ready to be executed by the CPU [multiple processes are ready]
process is in ready Queue & waiting for CPU allocation by OS [PCB]

③ Running state

from a ready Queue one process as per scheduling algorithm (STS) is sent for CPU execution through dispatcher.

④ waiting / Blocked

process is not ~~used~~ executing now, CPU halted its execution & waiting for some event like I/O, Signal, etc.

- * CPU is released
- * New process from R-Q is allocated to CPU
- * PCB's are saved & loaded respectively
- * Old process is in I/O or waiting [temp stored in mem]

Suspended State (managed by MTS)

* **case 1** = process in waiting/Blocked state moved to suspended state due to new waiting process arrived & memory is insufficient.

PCB is active but process is sent to secondary memory

* **case 2** = a high priority task is arrived & needed to be loaded in R-Q but main memory is insufficient; so OS selects low-priority task and send it to suspended state in secondary memory.

PCB + memory image are sent to secondary memory. Some kernel part of PCB is still in mm.

Terminated State

Resources are released, OS releases memory, PCB & other resources.

Short-term scheduling = selects one process from the ready Queue in the main memory & allocates the CPU to it.

* Responsible for context switching.

* Based on Algorithm it can be preemptive/non-preemptive

Medium-term scheduling = It controls suspending & Resuming of processes by swapping them in/out of main memory with Secondary memory

Long-term scheduling = decides which processes are admitted into the system for ~~execution~~ making it Ready for Execution

* moves process between HD & mm (New to Ready)

* brings mixture of both I/O & CPU bonded processes so processes can run parallelly

* Controls degree of multiprogramming.

Summary

Schedulers

STS
MTS
LTS

States

Ready
Running
Block

Process is in mm

Suspended
New

process in SM

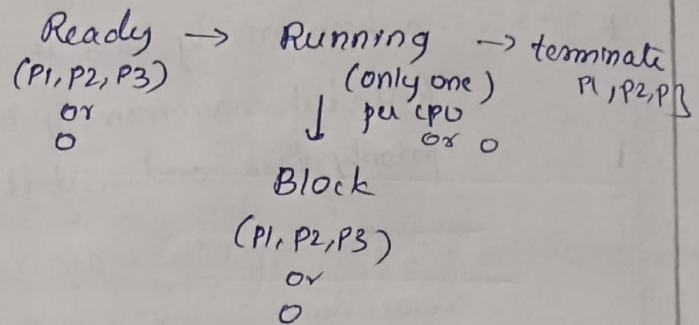
Most frequently used → STS Least → LTS

Q) Consider a system having m processes & n CPUs where $m > n$ & $n \geq 1$, find min & max no. of processes present in Ready Running & Block state

Running & Block state

	min	max
Ready	0	m
Running	0	n
Block	0	m

let processes (m) = 3 & CPU (n) = 1



Threads

smallest unit of CPU execution within a process

* It's an independent unit within process during execution but resources are shared with other threads of same process.

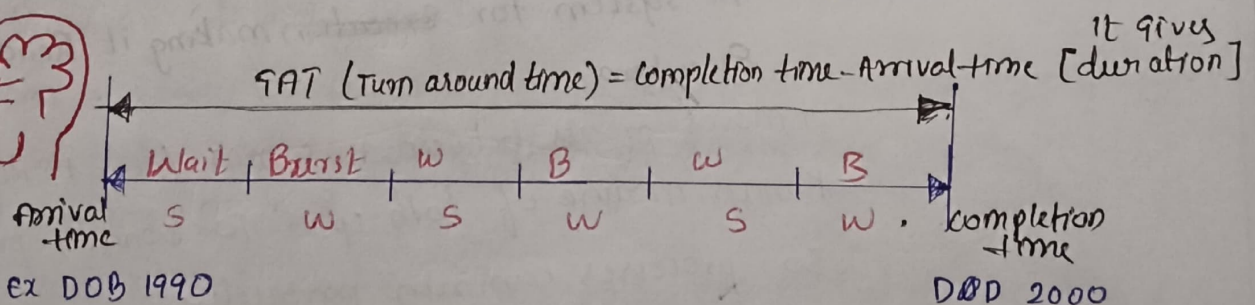
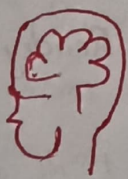
* Even in single CPU using time-slicing, CPU switches b/w threads rapidly giving illusion of parallelism.

* CPU Scheduling *

Goals

* minimize CPU utilization

* minimize avg TAT, avg W.T, avg Response time.



Arrival time = time at which process entered into ready-queue

Burst time = total time CPU spends for executing ~~one~~ process.

Total TAT (Turn around time) = total time ^{duration} taken for process execution including Burst time & waiting time.

TAT = when Process died - when it began.

Completion time = time at which a process execution is completed in the CPU & wait CPU. (measured from 0 even arrival time > 0)

waiting time = total time duration process spent ~~in~~ the ready Queue as well as in blocked

Arrival time = AT

Burst time = BT

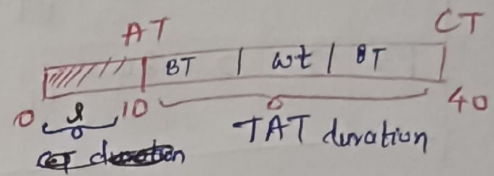
Completion time = ~~AT~~ ^{Grand chart} TAT - AT

Turn Around time = Completion time - AT (CT - AT) TAT = CT - AT

Burst time = TAT - WT

waiting time = TAT - BT

Scheduling length (L) = max completion time - minimum AT
 $\max(CT) - \min(AT)$



$$CT = TAT - AT$$

① First Come first Serve (FCFS)

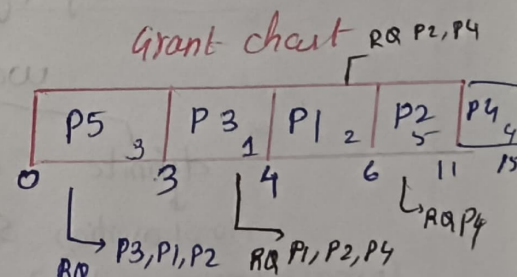
Job which arrives first is Executed first in free & fair manner.

* Each job knows when will they get Execution. [Definite waiting]

Criteria = AT

mode = non-preemptive

Pid	AT	BT	CT	TAT	WT
P ₅	0	3	3	3	0
P ₃	1	1	4	3	2
P ₁	2	2	6	4	2
P ₂	3	5	11	8	3
P ₄	4	4	15	11	7



starvation X
 Convey Effect ✓

Identify CT, TAT, WT

Pid	AT	BT	CT	TAT	WT
P1	0	20	20	20	0
P2	1	2	22	21	19
P3	2	2	24	22	20

Pid	AT	BT	CT	TAT	WT
P2	0	2	2	2	0
P3	1	2	4	3	1
P1	2	20	24	22	2

P1	P2	P3	
0	20	22	24

P2	P3	P1	
0	2	4	24

$$\text{avg wt} = \frac{39}{3} = 13$$

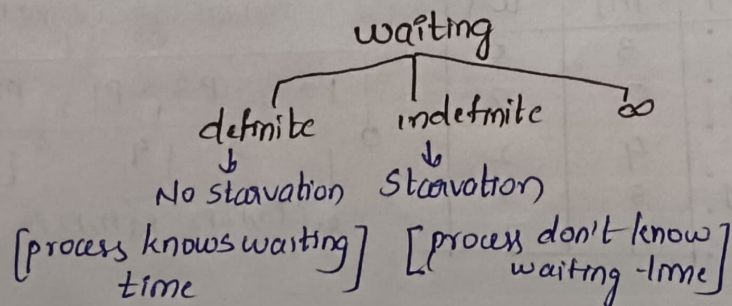
$$\text{avg wt} = 3/3 = 1$$

main disadvantage [Convey Effect]

IN FCFS if any process has the large burst-time (other than the last process) then it will effect on ^{avg} waiting-time. This effect is called convey effect.

Performance

- * It's simple to implement using singly linked list
- * No starvation, process knows when will they get executed
- * It has fair scheduling
- * But has convey effect. (Job doesn't know when will it get executed)



* Doubt? why FCFS has no starvation what if 1st process is very big, other jobs will stuck?

Answer: If this situation happens its called convey Effect. Still no starvation as the jobs know when will they get CPU

Shortest Job First (SJF)

Selection Criteria: Burst time.

Mode: Non preemptive

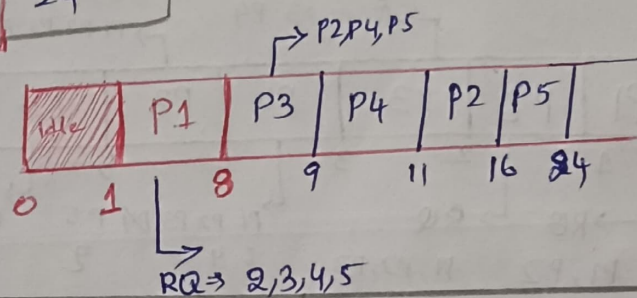
* It's an algorithm where shortest Burst-time process is executed first.

* If Burst time is same for more than 1 job then see Arrival time.

* Good but priority is not considered
* Not useful if 1st job is large BT can cause convey effect

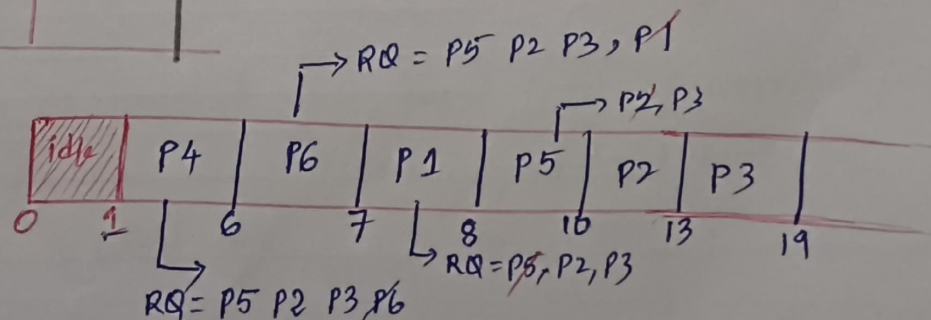
①

Pid	AT	BT	CT
P1	1	7	8
P2	2	5	9
P3	3	1	19
P4	4	2	16
P5	5	8	24



②

Pid	AT	BT	CT
P1	6	1	8
P2	3	3	13
P3	4	6	19
P4	1	5	6
P5	2	2	10
P6	5	1	7



Shortest Remaining Time First [SRTF]

Criteria: Burst time

mode < Preemptive

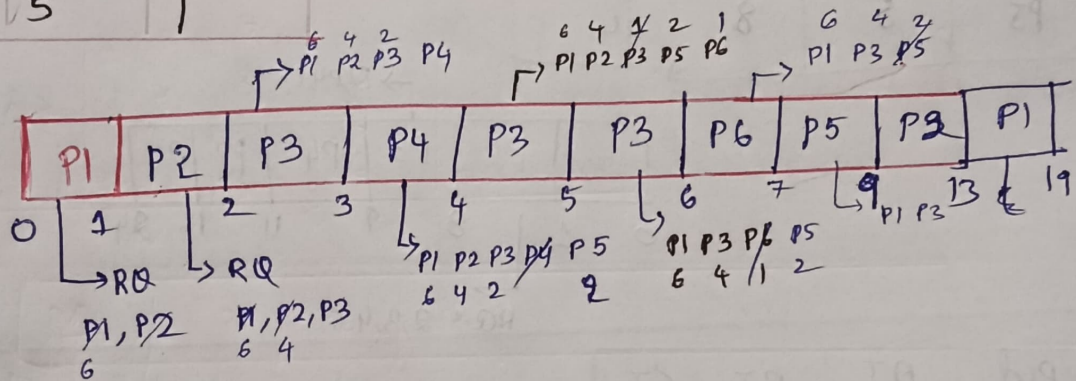
①

pid	AT	BT
P1	0	6
P2	1	4
P3	2	3
P4	3	1
P5	4	2
P6	5	1

* It checks which job is shortest

after ~~checking~~ executing Every 1 unit of time & if shortest job arrives it switches execution to it.

* Lack of Priority

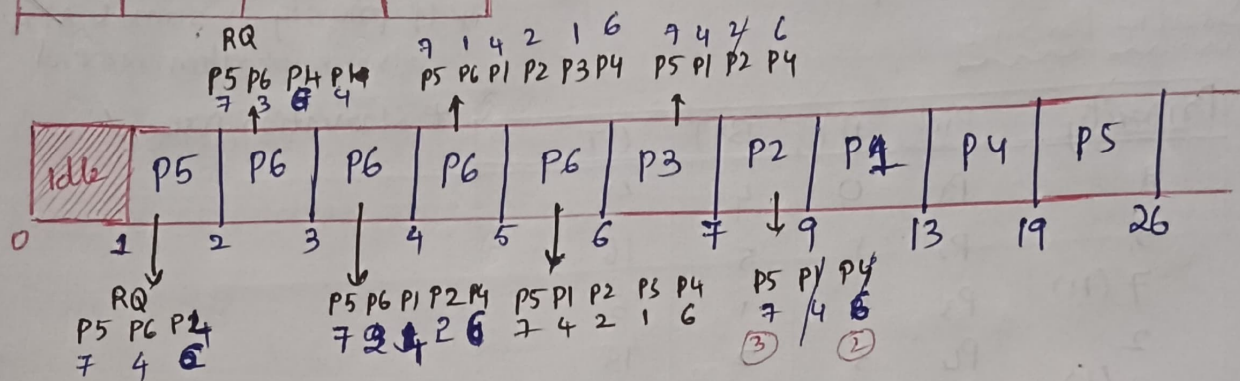


completion time-

	CT
P1	19
P2	13
P3	6
P4	4
P5	9
P6	7

Any criteria
pid ← AT

Pid	AT	BT	CT
P1	3	4	13
P2	4	2	9
P3	5	1	7
P4	2	6	19
P5	1	8	26
P6	2	4	6

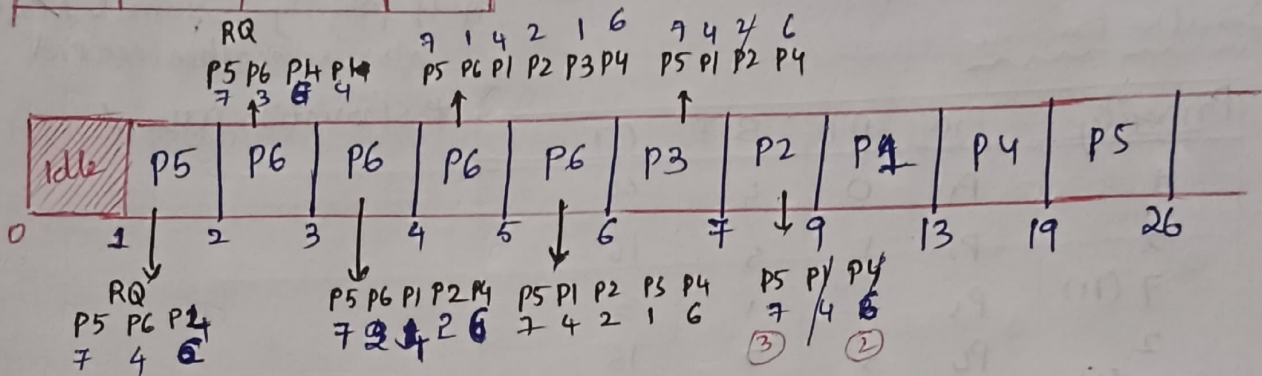


Main Issue

★ Starvation.

- ★ It favors shortest job so throughput increases
- ★ It favors shortest job so starvation for longer jobs
- ★ It is an optimal algorithm.
- ★ Not Applicable, as burst time is known after process execution
- ★ It's used as benchmark algorithm = when ^{some} other algorithm's avg CT or avg WT is near to avg CT/WT/TAT of SRTF then among them which alg is nearest to this, is considered as good.
- ★ Applicable when os predicts the burst-time of arrived job based on past behaviour. So not fully practical but Approximate ^{SRTF} ~~CT~~ is used. [Same with SJF]

Pid	AT	BT	CT
P1	3	4	13
P2	4	2	9
P3	5	1	7
P4	2	6	19
P5	1	8	26
P6	2	4	6



Main Issue

★ Starvation.

- ★ It favors shortest job so throughput increases
- ★ It favors shortest job so starvation for longer jobs
- ★ It is an optimal algorithm.
- ★ Not Applicable, as burst time is known after process execution
- ★ It's used as benchmark algorithm = when ^{some} other algorithm's avg CT or avg WT is near to avg CT/WT/TAT of SRTF then among them which alg is nearest to this, is considered as good.
- ★ Applicable when os predicts the burst-time of arrived job based on past behaviour. So not fully practical but Approximate ^{SRTF} ~~CT~~ is used. [Same with SJF]

I) Priority Based Scheduling

Priority { Static: A fixed priority value assigned to each process will not change lifetime
Dynamic: Priority of a process changes at runtime based on system conditions or process behaviour

Criteria: Priority

mode: Non preemptive

Static Priority

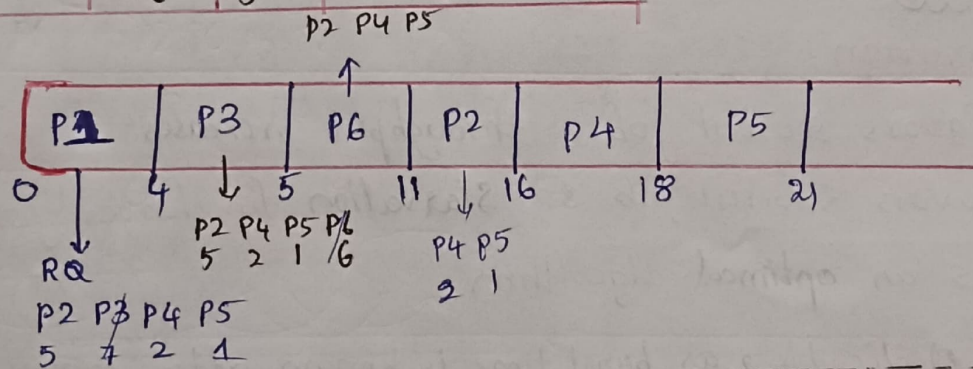
* Practically possible

* If priority is same & AT is also same - then see PID.

* Starvation issue ✓

①

Priority	Pid	AT	B.T	CT
4	P1	0	4	4
5	P2	1	5	16
7(H)	P3	2	1	5
2	P4	3	2	18
1(L)	P5	4	3	21
6	P6	5	6	11



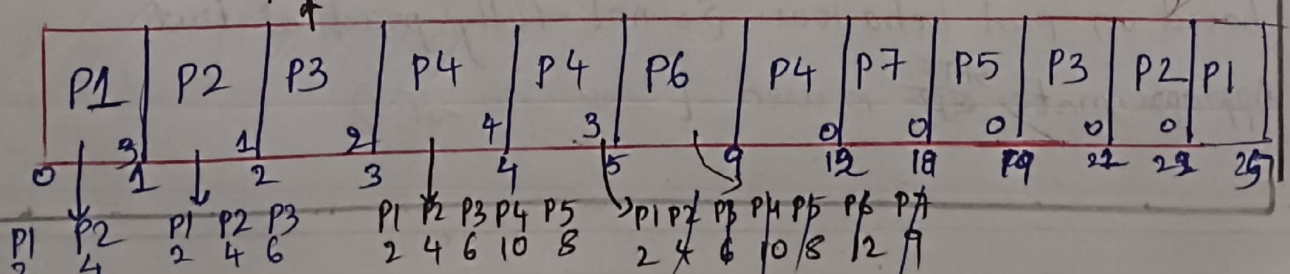
Criteria: Priority

mode: preemptive.

* After Every 1 unit of time priority is checked

* starvation ✓

Priority	Pid	AT	BT	CT
2(L)	P1	0	4	25
4	P2	1	2	22
6	P3	2	3	21
10	P4	3	5	12
8	P5	4	1	19
12(H)	P6	5	4	9
9	P7	6	6	18



Round Robin Scheduling

- * Used in time sharing systems
- * Each process gets a fixed time slice [called Time Quantum] & in that CPU is given to that process for execution until Quantum Expires
- * Processes are kept in Ready Queue.
- * If a process does not finish its execution within time Quantum it is preempted and placed at the End of the Queue.
- * Prevents starvation as Fair - Scheduling & practically possible.

Criteria: AT

mode: preemptive [TQ]

set TQ

Prepare Ready Queue

Select 1st process / Next process

Is that process
 $BT < TQ$

yes

Run the process till
completed

Terminate
that process

Remove it from RQ & check RQ
if empty then stop

Run the process till TQ Expires

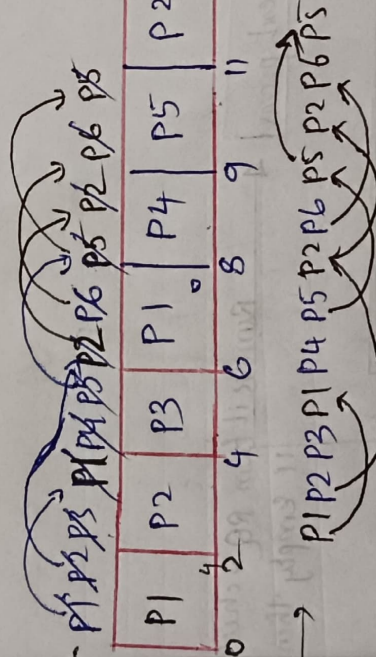
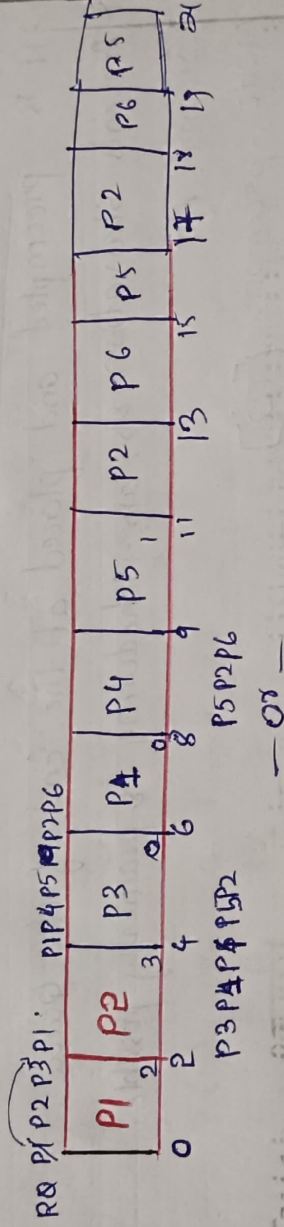
Is process
finished

NO

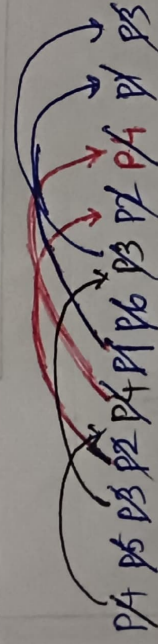
load the remaining process
[PCB] at the End of RQ

① If $TQ=2$ find avg CT

Pid	AT	BT	CT
P1	0	2	8
P2	1	8	18
P3	2	20	6
P4	3	10	9
P5	4	22	21
P6	5	31	19



Pid	AT	BT	CT
P1	5	20	22
P2	4	30	27
P3	3	40	33
P4	1	20	30
P5	2	20	6
P6	6	30	21



Pid	AT	BT	CT
P1	5	20	22
P2	4	30	27
P3	3	40	33
P4	1	20	30
P5	2	20	6
P6	6	30	21

~~WTF~~ [WTF - longest job first]

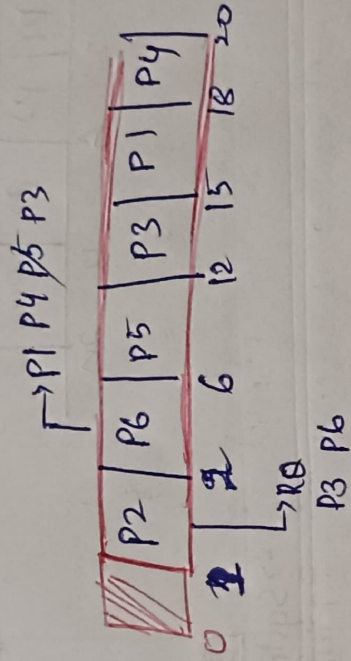
Criteria: B.T

Mode: Non-preemptive

* longest job has higher priority

* convey effect

Pid	AT	BT	CT
P1	3	3	18
P2	1	10	2
P3	2	3	15
P4	4	2	20
P5	6	50	12
P6	2	40	6

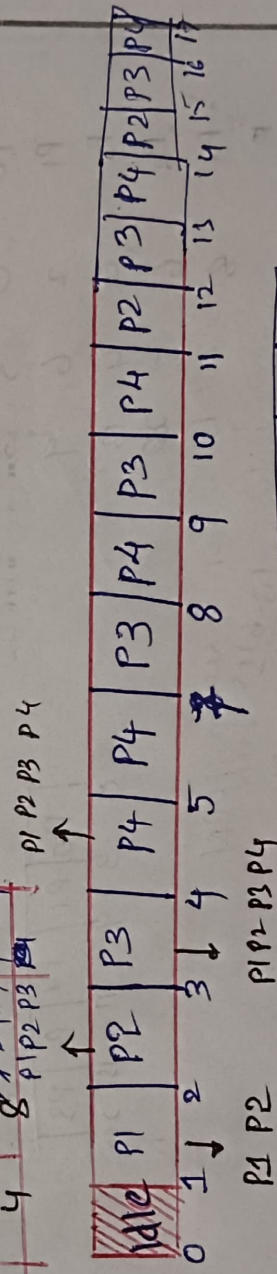


LRTF [longest remaining time first]

Criteria: BT

Mode: Preemptive

Pid	AT	BT	CT
P1	1	1	18
P2	2	4	19
P3	3	8	20
P4	4	8	21

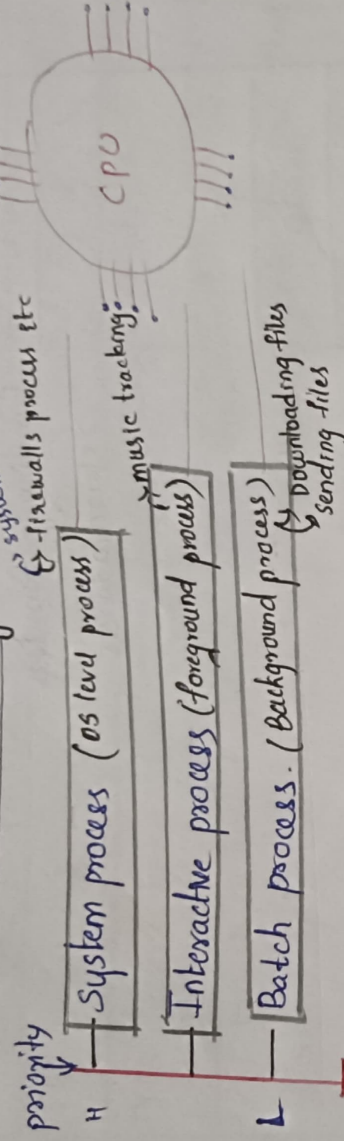


Pid	AT	BT	CT
P1	0	4	31
P2	0	8	32
P3	0	10	33
P4	0	12	34

$$\text{Total BT} = 34 + 0 \rightarrow \text{start A7}$$

$$CT = 34 + 33 + 34$$

Multilevel Queue Scheduling



* Instead of having one Ready Queue, we are making OS to contain 3 types of Queues

- 1) System Process RQ
- 2) Interactive " RQ
- 3) Batch " RQ

* priority is higher for system process RQ & lowest for Batch

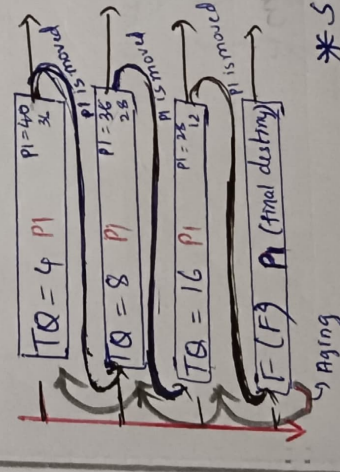
* When System process RQ is scheduling processes no, other has ability to interfere or preempt them from a CPU

* Each Queue can have different scheduling Algorithms.

* Generally high level of priority process will be placed into top level ready queue & low in low priority ready queue

* starvation issue for lower priority Queue processes.

Multilevel feedback Queue Scheduling



* Using concept of aging we can avoid starvation.

* so starvation is avoided & priority is also done

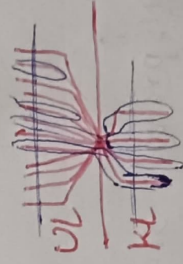
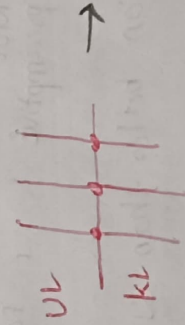
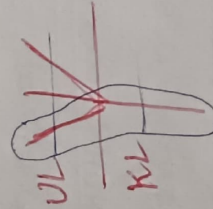
* In order to avoid starvation increase process priority at regular intervals of time by sending process from bottom level

Queue to the top level Queue. [Mechanism is called **AGING**]

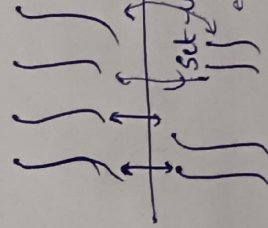
User level-thread	Kernel level Thread
1) Implemented by user [library designed by user]	1) Implemented by OS
2) OS doesn't know about ULTs if PCB is passed OS knows about it	2) OS recognizes KLTs
3) If one user-level-thread is blocking system call then entire process will be blocked [If a thread uses non-blocking system calls like fork() the entire block with process not get affected]	3) If one kernel level-thread uses blocking system call then another thread will not be affected as OS knows that thread is sending request not the process
4) No HW support is Required	4) HW support needed ex: gang scheduling OS Design or HW design changes here affects here

Multithreading models

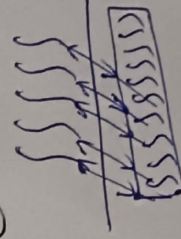
- ① many to one ② one to one ③ many to many



④ set concurrency function - dynamic thread creation



⑤ thread pooling



* thread pooling depends on main memory
greater mem ↑ greater the no. of threads ↑

Chapter - 3 Synchronization and deadlock

Synchronization

★ Effective communication among the processes

Why synchronization

→ When multiple processes make changes in critical section then it causes data inconsistency & data corruption issues

~~Data~~ ★ **Critical Section**: It's a nothing but a part of program where a process accesses or modifies shared data/resources like variables, files, memory, devices etc.

★ **Data inconsistency**: Data inconsistency refers to a situation where logically incorrect data is produced when multiple processes or threads access / modifies the same critical section without sync

Real world Example:

In a program, if multiple threads are created & each thread simultaneously accesses a shared global variable and increments it, variable may not be incremented correctly. Some updates may be missed & the final output becomes logically incorrect.

* as this operation is **not atomic** [individual steps] so all may take **old value** as input **simultaneously update it** & data gets incorrect

* **Synchronization mechanism** prevents this issue by allowing only one thread/process at a time to enter critical section

Ex:- $x = 10$

def func:

$x = x + 1$

thr1 = thread(target=func)

thr2 = thread(target=func)

thr1.start() & thr2.start()

thr1.join() & thr2.join()

→ example of thread issue

→ use **Assembly language** as topic is OS related

Communication Approaches

- ★ ① Shared Memory ✓
- ② Sockets (process from diff systems want to communicate)
- ③ Pipe (2 process from same system but diff prog want to communicate)
- ④ Signal

Some terminologies

① Race condition: A condition where 2 or more processes/threads racing to access critical section at the same time but the result is depended on order of the execution.

② co-operating / co-ordinating process.

If one process is affecting the execution of other process then it's called co-ordinating processes if not then they are called independent processes.

③ Atomicity :

It means an operation or process execution is done without any interruption by other process or thread while it is executing.

* process is executed with any preemption done by OS. All its instructions are executed in one go. X No middle interventions

④ Synchronization:

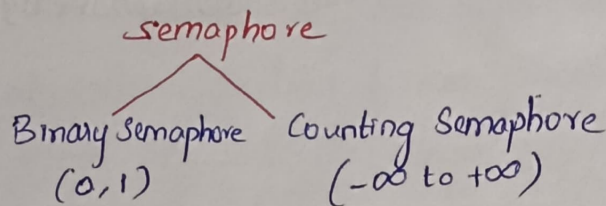
process synchronization: ensures that only one process/thread is allowed to enter & execute the critical section at a time where shared resources are accessed & modified.

⑤ Mutual Exclusion

At any point of time only one process should be present inside the critical section. implemented using semaphore.

* Semaphore

- * It is an integer variable which can be access by using some atomic operations (like Down()/up())
- * It is an OS resource
- * It is invented by *dijkstra* also known as *dijkstra Conditional Variable*
- * Operations are Down()/up() or wait()/signal() or P()/V()



Counting semaphore

structure Semaphore

```

{
  int value
  Queue_type L // keep blocked process
}

```

Semaphore S // reference variable of struct

S.value = 3

Down(S)

<Critical Section>

up(S)

Down(S)

{

S.value = S.value - 1

if (S.value < 0)

{

block the process & keep it in queue

}

}

up(S)

{

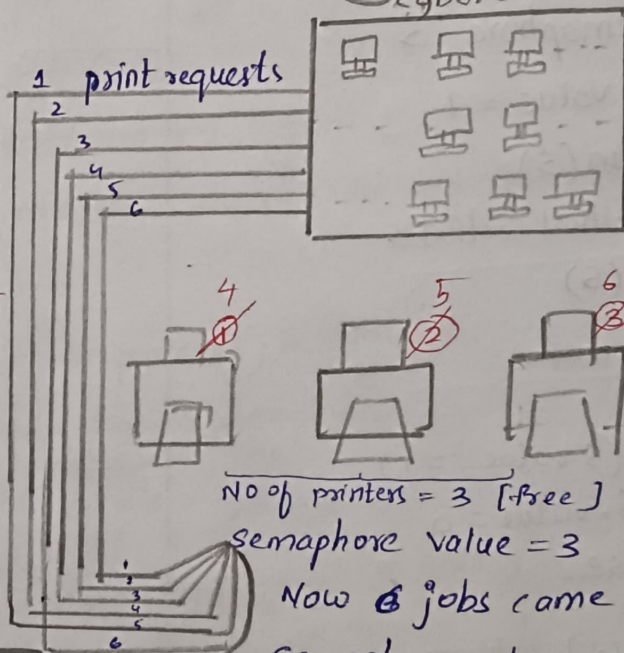
S.value = S.value + 1

if (S.value >= 0) {

select a blocked process from Queue & wake up

}

Counting Semaphore
CyberCafe



It assigned 3 first jobs to printers
so Now, remaining 3 are in Queue

- * counting semaphore maintains the count of available resources
- * It decrements its count when new process access the free resource of CS
- * When No resources are free semaphore value = 0 & New processes are added in Queue & semaphore value logs negative values for new processes
- * Helps avoid race condition & resource conflicts
- * **Down()** / **wait()** / **P()**: a atomic operation that Decrement Semaphore
- * **up()** / **signal()** / **V()**: atomic operation that Increases semaphore by 1

Binary Semaphore

Structure Bsemaphore {

enum value(0,1)

Queue type L

}

Bsemaphore S

S.value = 1

Down(S)

<critical section>

Up(S)

Down(S)

{

if (S.value == 1)

S.value = 0

else

{

Block the process &

keep in Queue

}

}

up()

{

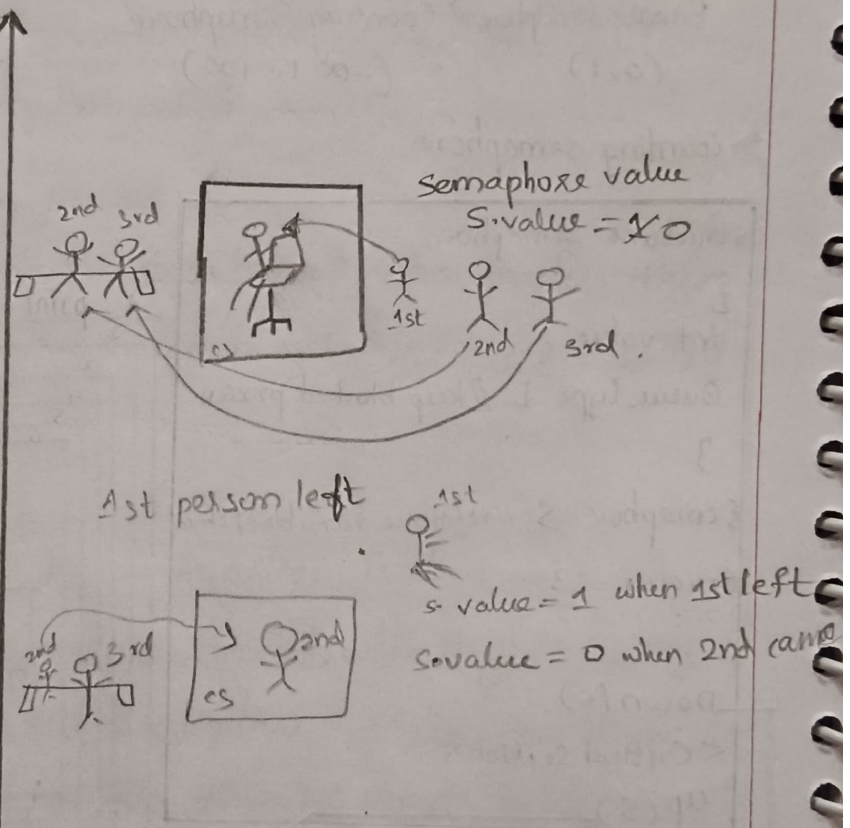
if (Queue is not empty)

{

//select a blocked process from Queue & wake up

}

else S.value = 1 }

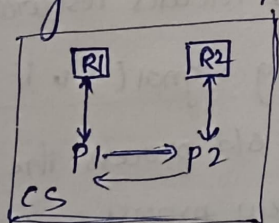


NOTES:

- * Binary semaphore has only 0 and 1 values for variable as it has only one resource in critical section
 - * When critical section resource is freely available then variable value gets 1 or else if its busy then its value will be 0
 - * When semaphore value is 1 = New process can use resource & value gets down to 0 until its terminated
 - * When semaphore value is 0 = New process cannot use resource of critical section & new process need to wait in Queue.
 - * When process inside CS is terminated = Semaphore value down to 0 based on scheduling algorithm new process from the Queue is selected & allocated for Resource.
- [If FCFS is used {No starvation} If ^{priority}RR is used {starvation}]

Why counting Semaphore Causes deadlock?

- * Counting semaphore allows multiple process to use resources if they are free at a time
- * So, if one process wait for other process, & that process is also waiting for this process then it causes deadlock.

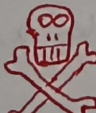


Ex $R1 = 2$ $R2 = 1$
 $S1 = 4$ $S2 = 1$

P1: $\text{down}(S1) \Rightarrow S1 = 0$
 $\text{down}(S2) \Rightarrow$ waits $\rightarrow S2$

Resource was booked

P2: $\text{down}(S2) \Rightarrow S2 = 0$
 $\text{down}(S1) \Rightarrow$ waits
 $S1$ is already busy
Resource was booked

* It caused deadlock 

Monitors

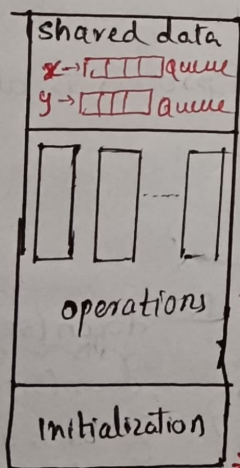
Monitors are high-level synchronization mechanism which avoids deadlocks by implementing **mutual exclusion (ME)** among process

- * Monitors allows only one process to use resources inside monitor
- * Even if only 1 resource is used by currently running process it doesnot allows other processes to use other free resources of monitor
- * Single Monitor so can't cause deadlock.
- * But Multiple monitor, suppose one monitor need process of other monitor which also need that called monitor process then it causes circular deadlock.

* Monitor contains **shared variables, condition Variables & procedures**

- * Process running outside monitor can't access internal variable but can access internal procedures.

Structure of Monitor



* Monitor only allow one process to enter in-to it.

* Even if internal process needs other process then it holds present process in x or y Queue & releases resources **[wait(Pi)]**

* Then new process is allowed using **signal()** & it executes

* Now even new process needs old process the old process is **signalled()** then new process is queued

* **Condition Variables** ? They are 2 different operations **[wait(Pi)]**

are performed on the condition variables x, y i.e 1) **wait()** 2) **signal()**

* x & y are variables which holds processes in a Queue.

* wait()

when a process in any procedure or operation uses **wait()** with x [$x \cdot \text{wait}(Pi)$] etc then it is suspended & Resources are released

* That suspended process gets appended in x/y Queue.

Signal() : It wakes suspended Process from the Queue

signal()

Block Queue of respective
Condition variable is Empty

→ No Suspended Process in Queue

→ signal() has no effect & is lost

→ some Blocked processes are in Queue

→ process will be resumed from the
Block Queue

ex program (Java)

Monitor monitor-name {

{

variables --- ; // shared data

conditional Variables ---, x & y

Procedure P1 { --- }

{

≡

}

procedure Pn(---)

{

≡

}

procedure initialize(---)

{

≡

}

// initialize data

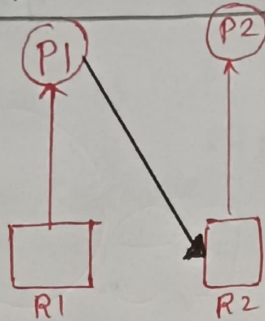
}

Deadlock: when a set of processes are said to be in deadlock if each process in the set is waiting for an event or waiting for other process in the set that ^{will} never be generated.

~ or ~

when a set of processes are waiting for an event that never happens

(i)
No deadlock.

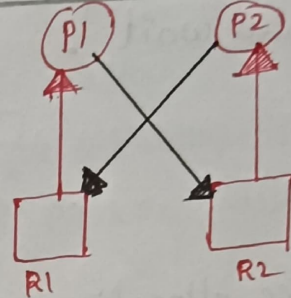


Here R1 resource is hold by P1 & R2 by P2

* when P1 needs resource R2 it gets blocked as R2 is under P2

* So when P2 completes, it releases R2 - then P1 takes it & then complete its Execution.

(ii)
Deadlock



* P1 needs resource R2 but it was under P2 and P1 gets blocked

* P2 needs resource R1 but it was under P1 & P2 gets blocked

* No one is Executed.

(iii)

$a=1 \rightarrow$ binary semaphore
 $b=1 \rightarrow$ binary semaphore

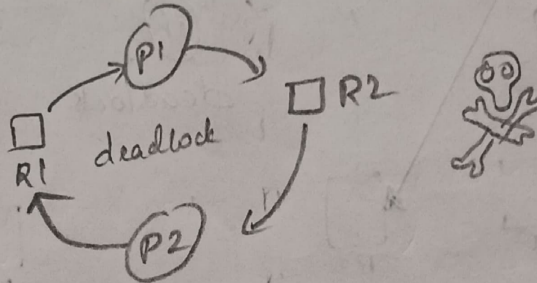
resource a booked
P1 $a=0$
down(a)
down(b)
P1 is blocked
as resource b is busy

resource b booked
P2 $b=0$
down(b)
down(a)
P2 is blocked
as resource a is busy

both are blocked resources gets locked.

Necessary Conditions for deadlocks

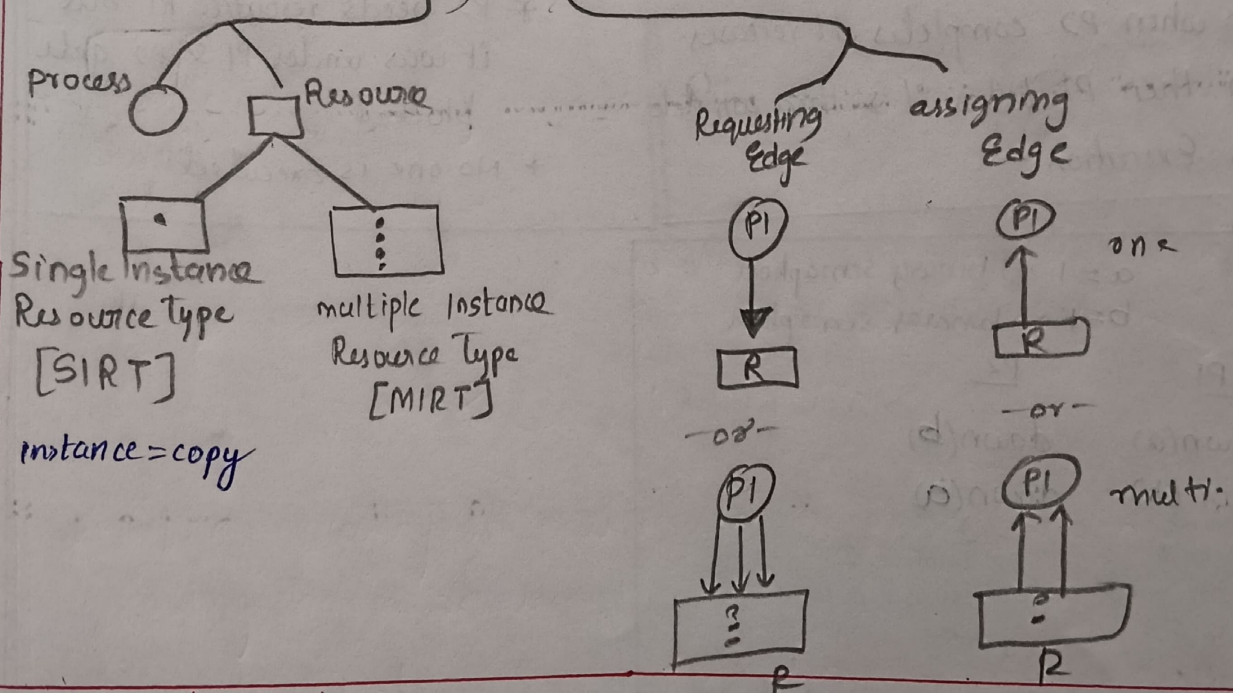
- ① Mutual Exclusion: when process are waiting for cs only one at a time is allowed
- ② Hold and wait: when process hold some resource & wait for other res
- ③ No preemption: no stealing of ~~process~~ resource from running state to ready process
- ④ Circular wait:



① Resource allocation Graph (RAG)

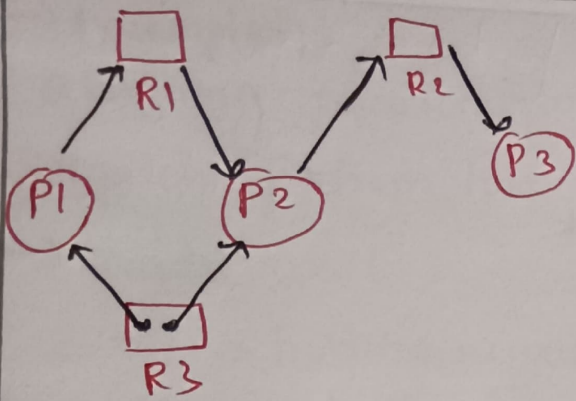
→ Used to model the deadlock.

$G(\text{Vertex}, \text{Edge})$



identify deadlock

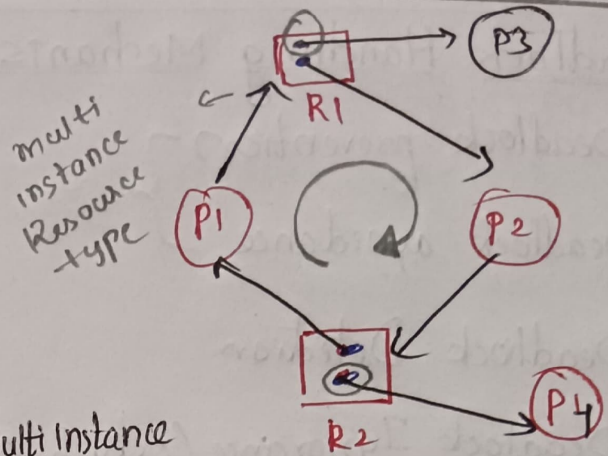
- ① check cycle is there
- ② check deadlock
- ③ No cycle No deadlock.



No deadlock

order of Execution = P3, P2, P1

Single Instance Resource type



Multi Instance Resource type

No deadlock

* They are 2 Resources for R1 & R2

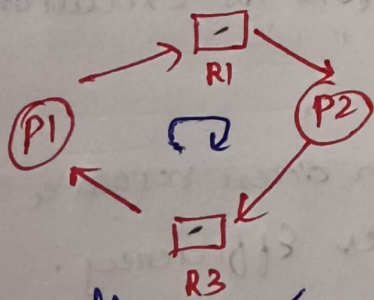
* when P3 & P4 finishes R1 & R2 2nd resource is free

* P1 gets R1 1st port when P3 finishes

* R1 2nd port used by P2

* R2 1st is used by P1

* P2 uses R2 port 2 when P4 finishes



Deadlock ✓
[SIRT]

SIRT + cycle = Deadlock

SIRT + MIRT + cycle = maybe deadlock

But MIRT < process requesting it.

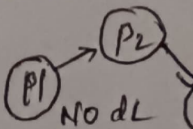
Wait for Graph ECET - T4

A. WFG is a simplified form of RAG which only shows nodes [Processes] & does not show resources

Nodes → Processes; Edges - P1 → P2 means = P1 is waiting for a resource held by P2

* Simple way to find cycle

cycle present - deadlock



Deadlock Handling Mechanism

- ① Deadlock prevention
 - ② Deadlock avoidance
 - ③ Deadlock Detection
 - ④ Deadlock Ignorance (ostrich Alg)
- } Before deadlock
- } After deadlock

① Deadlock prevention.

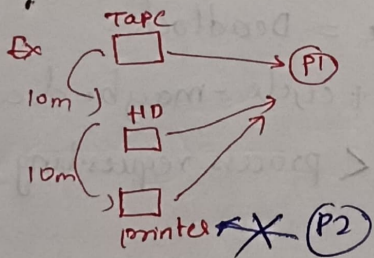
By dissatisfying any one / more necessary conditions for DL like ME, preemption, hold & wait resource, circular wait.

→ Mutual exclusion

Not possible to apply to all resources bcoz some resources are sharable [File] but some are not [CPU] [Printer]

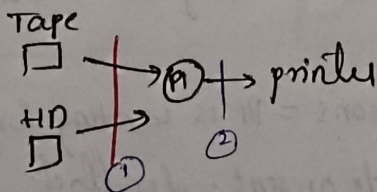
→ Hold & wait

* Allocate all necessary resources to the process before its execution starts.



No other process can access resource even its free violates **Efficiency**.

* Before requesting for new resource the process should release existing held resources.



* If process need to hold resource without releasing it then it should make sure that no other process need that resource

* Starvation ✓

→ Preemption

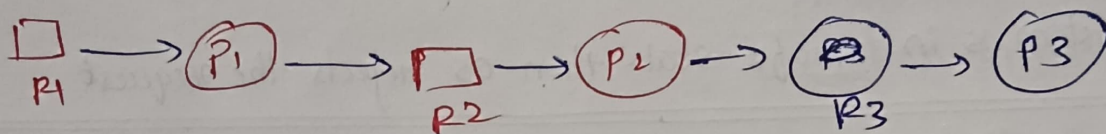
assigning resource to high priority requested resource - By snatching from low priority process avoids deadlock.

→ circular wait

$\gamma_p \rightarrow$ holding resources

$\gamma_j \rightarrow$ request resources.

$\boxed{\gamma_p < \gamma_j}$ * If holding resource id of process $<$ the requested resource id. then process can request for that ~~process~~ resource.



P1 can request resource R3 as its holding resource R1
 $R1 < R3$

P3 can't request resource R1 as its holding R3
 $R3 \not< R1$

② Deadlock Avoidance

Implemented by using Banker's Algorithm

→ useful in Batech

- ① Safety Alg
- ② Resource Request Alg

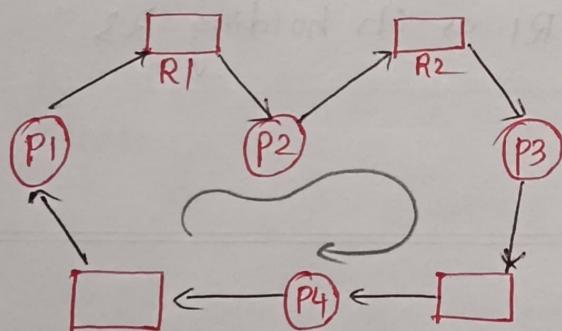
Inshort,

* Whenever the process is requesting the resource what o.s does is, it will run safety Algorithm and whenever the system is in safe state then os accept that process request that is → assign the requested resource

* If system is in unsafe state then os rejects the request.

③ Deadlock Detection

i) Single Instance Resource type



→ Run cycle detection algorithm
→ If cycles are not detected then time gets wasted.

→ wastes cycles of cpu → throughput ↓

ii) Multiple Instance Resource type

$m=5$

$m=3(A, B, C) = (7, 2, 6)$

need Banker's Algorithm

skip this.

Data Structure

→ available[i] = k

→ max(i, j) = k

→ Allocation(i, j) = k

→ Need(i, j) = k. need = Max-Allocation.

Recovery

Termination

Kill all the processes

P_1 90%
 P_2 15%
↑
near to end

Kill one process & run D Detection Alg
if DL present delete one more process, continue until no deadlock detected

preemption

preempting one resource from one process & allocating to others.

disadvantage
preempted resource from a process is not guaranteed will be given back (starvation)

* **Fun Fact**: Practically not even one among Detection, prevention, Avoidance & Recovery is implemented in our systems

Banker's Alg demerit: it need which process needs how many resources initially which is not possible.

But implementing one Alg .i.e ostrich Alg (Ignorance)

* Deadlock hua-toh system Restart kardiye -to kaam khatam kyu detect, avoid aur recovery ke uper-focus karna.

How many min processes can i safely allow

* **$R \geq n(m-1) + 1$** → for Question asking how deadlock is prevented if R no. of resource instances & n no. of processes & m no. of resources to any process.

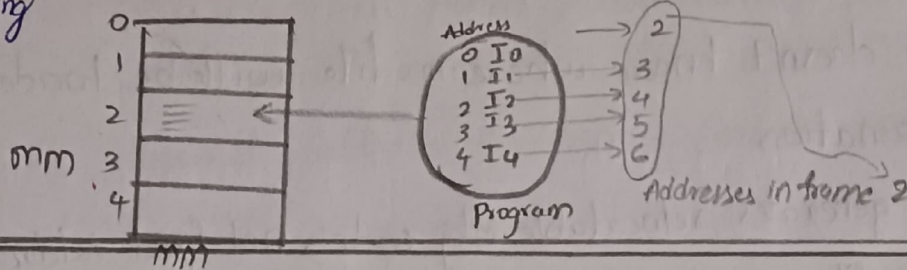
↑
ECET

Memory Management

Address Binding

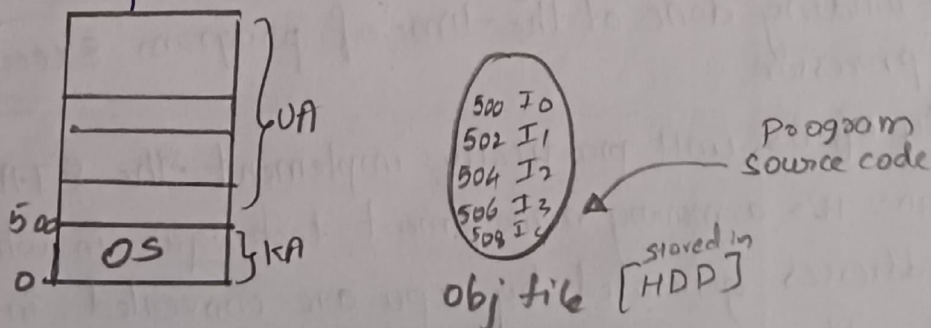
Association of program instructions and data to actual physical memory locations is called Address Binding

linking, connecting
Binding



Compile time Address Binding [Absolute Binding]

- * If the compiler takes the responsibility of association or linking of the program instructions and the data of the actual physical memory location is called compile time Address Binding.
- * Before loading instructions & data into mm the addresses to each instruction & data is hard coded by the compiler into the program.
- * In this approach, compiler assigns addresses to each inst/data while converting ~~them~~ ~~into~~ source code into obj code.
- * While execution of the program, it must be placed at exactly same memory locations decided during compilation, if mismatch then Recompile.



compiler interacts with OS
- to get details of free space in mm

Load-time Address Binding

* Address binding done after loading the program into main memory.

→ This type of address binding will be done by OS mm manager i.e. loader.

→ Compiler doesn't know where this file will be loaded in MM
clear Explanation

* compiler generates relocatable obj code, not fixed addresses

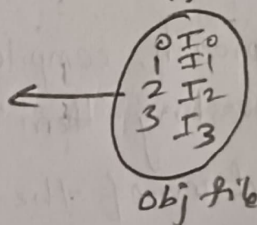
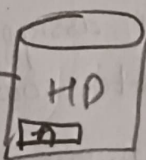
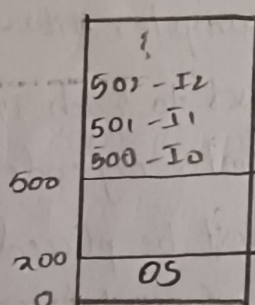
* that obj file is stored on disk with logical addresses.

* When the program is loaded into RAM, the loader

choose a free
memory location

convert relative
add into absolute
physical addresses

which was
* Then the file is loaded into RAM is executed → if suspended then should be loaded at same location



Execution time Address Binding

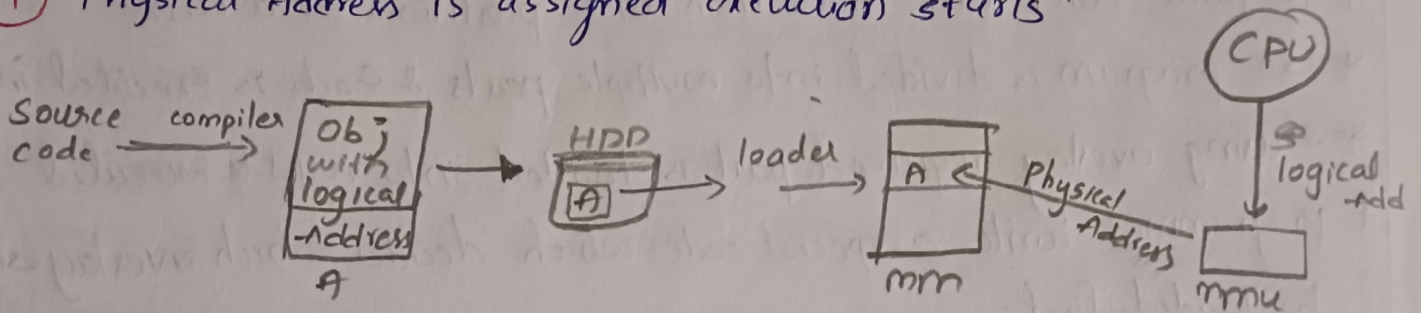
→ Address binding done at the time of program execution by the processor

→ majority of OS will practically implement the ETAB

* Definition: It's a memory management technique in which logical addresses generated by CPU are converted into physical addresses during program execution using mmu

Explanation

- ① **Compiler**: generates logical addresses of each instruction in obj-file & save in HDD
- ② **Loader**: Loads that file into main-memory so can be executed but at available space.
- ③ **CPU access**: CPU when need that instruction (as per pc) then it sends logical (virtual) address which is converted into physical address by mmu (memory management unit).
- ④ Physical Address is assigned Execution starts.



Compile-time AB	Load time AB	Execution-time AB
① compiler knows where the inst/data will be loaded into RAM so it assigns the hardcopy of address while compiling	② Compiler doesn't know where its inst/data will be loaded so it assigns relocatable addresses to its file ② Loader assigned the physical addresses to the inst/data after loading file into RAM	① Neither compiler nor loader assigns the physical address to the obj-file ② During Execution only mmu translates cpu generated logical add into physical address -

Overlays

Overlays are a memory management technique in which only the required part of a program is loaded into main memory at a time, instead of loading entire program.

→ Used when a program is larger than available partition of RAM

How?

- * Program size is larger than size of partitions in RAM
- * Program can't take more than one partition at a time
- * So program is divided into multiple parts & each is executed independently
- * & using overlay required part is loaded into MM & executed
- * programmer write a special code which decides which overlay should be loaded first.

* ADVANTAGES

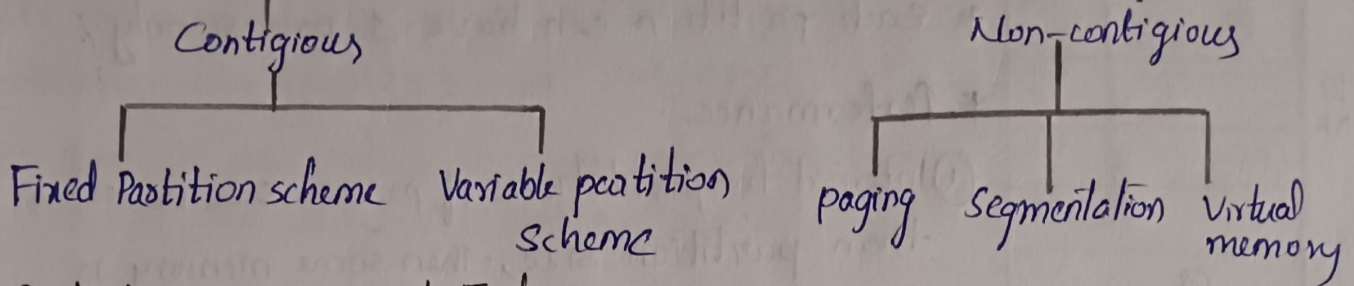
saves memory & executes tasks

* DISADVANTAGES

can't run if each program overlay is interdependent
program design becomes complex.

Notes by Abdul Muqsooth

Main Memory Management Techniques



Contiguous management Techniques

A process occupies one contiguous region of a physical memory.

- * All instructions related to that process is allocated a single, contiguous / continuous (adjacent) block of main memory.

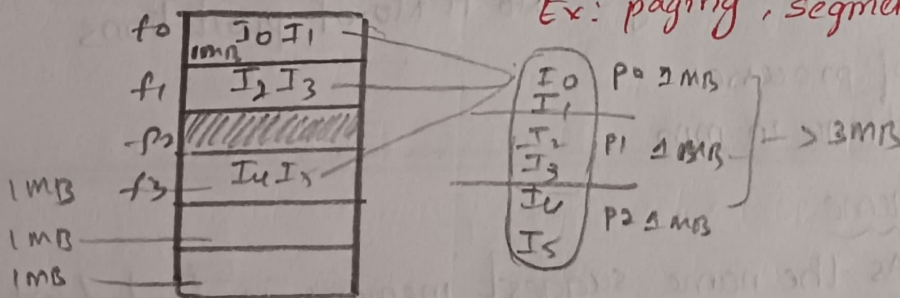
Ex: Fixed / variable partitions

Non-contiguous

A process is divided into multiple parts and each is stored in different (non-adjacent) locations in the main memory.

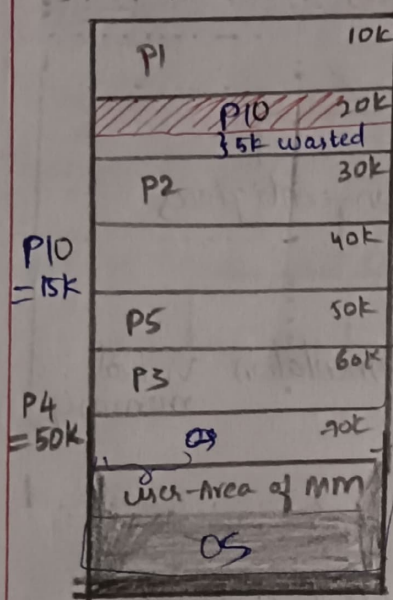
- * All instructions related to that process is also divided into groups. Each group is at different location in the main memory called frames.

Ex: paging, segmentation & VM.



Fixed Partition Scheme of memory management

Fixed partition



* Main memory is divided into fixed no. of partitions

* Each partition can have 1 process only

* Performance

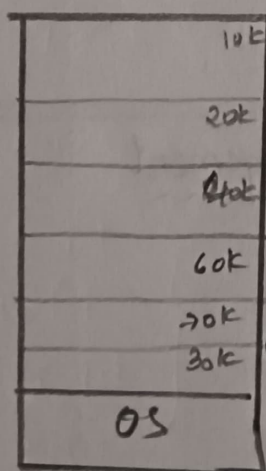
① **Internal fragmentation**: If process size is less than partition size, then some memory is wasted called as Int fragmentation

② **External fragmentation**: Though the total memory space is enough to hold new process but due to 1 partition = 1 process the 1 partition single cannot hold new process alone -- this called as Ex fragm
Ex 3 partitions of 10K size each are free but a process came of 30K size. In presence of total 30K size also we can't allocate memory to it

③ **Maximum process size** = maximum partition size

④ **Degree of multi-programming** = No. of partitions
↓
refers to total no. of processes can be present at once in mm

Variable Partition Scheme



* As the name suggest memory is divided into partitions based on incoming processes

* At run time process gets

① **No-internal fragmentation**

② **Maximum Process size** = memory size

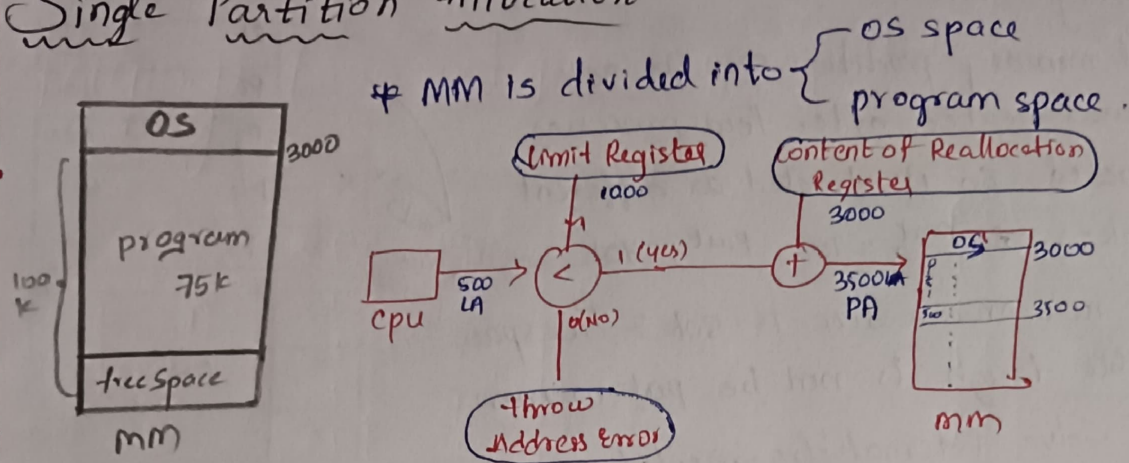
③ **Degree of multi-programming** =

* It depends on program sizes

* Smaller the program sizes maximum & larger the

no. of processes

Single Partition Allocation



* How it works ?

- Only one program can be kept at once in main memory
- Suitable for uni-programming system
- If size of program < size of memory → memory wastage

workflow

- * Program is loaded in MM → CPU sends Logical Address
- * Limit register checks whether CPU sent LA is within limit of LA
- * If yes then its converted into Physical Address using content of Reallocation Register addition
- * What it does means, suppose OS occupies 3000 KB size now CPU want to access instruction at 3500 KB so it sends LA 500 which is added with 3000 & finally it can be referred in MM as physical Address.

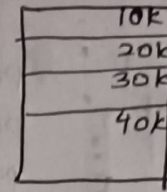
Partition Allocation methods

- ① First fit
 - ② Best fit
 - ③ worst fit
 - ④ Next fit
- } Algorithms

First fit: In a partition memory, search for free space from the top of the memory and allocate 1st free sufficient partition

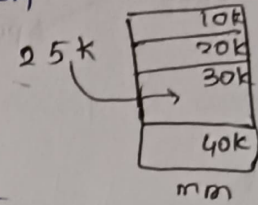
ex:

25k
PI

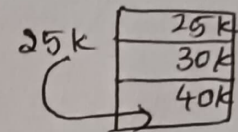


Best fit: Allocate smallest free sufficient partition

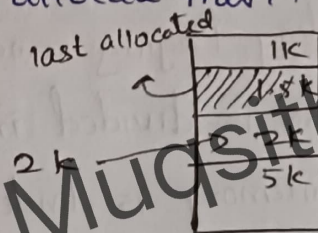
↑ time ↑ search all partitions
then best/worst among them



Worst fit: Allocate ~~smallest~~ largest free sufficient partition

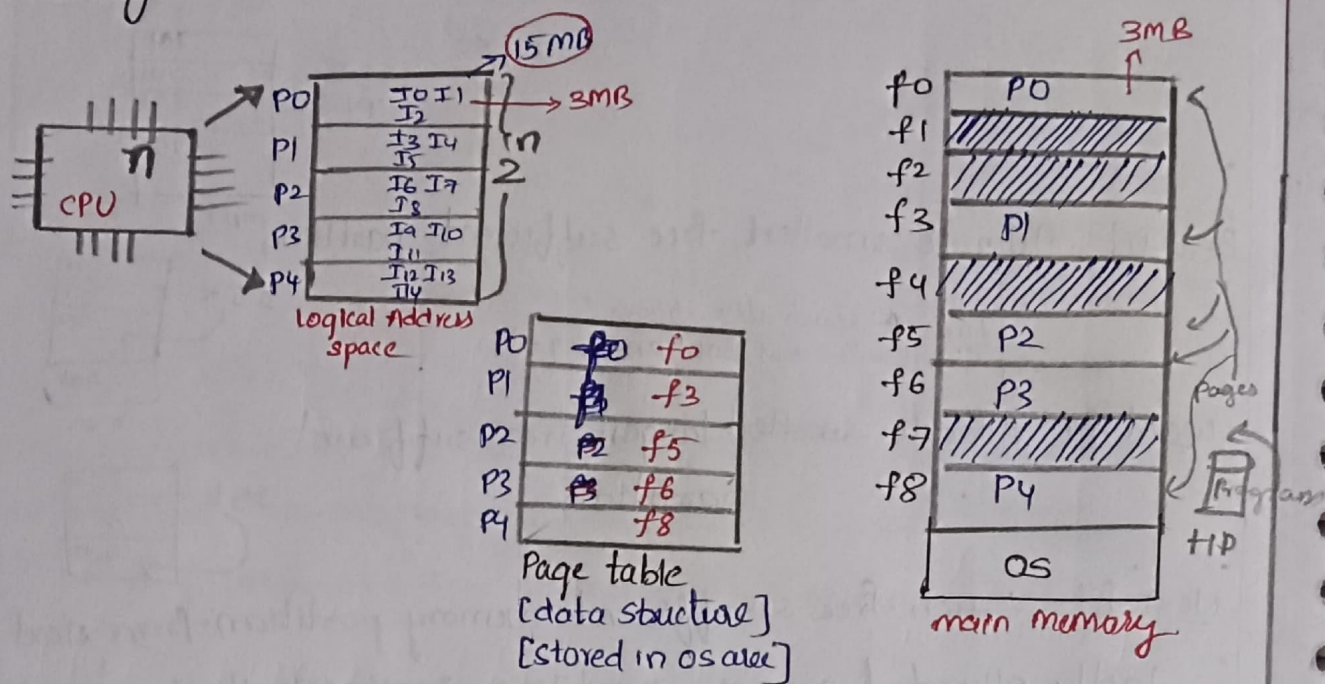


Next fit: search free & sufficient memory partition from next to lastly allocated memory partition & allocate there.



Notes by Abdul Mugith

Paging (non-contiguous management technique)



Understanding Paging concept

- * A process is divided into pages
- * Main memory is divided into frames
- * Pages are placed in free frames (**non-contiguous**) & CPU can access them.
- * CPU does not know where pages are physically stored. to access them
- * So OS provides table [a data structure] which contains page numbers & their respective frame addresses
- * So, now CPU can have page knowledge but which instruction in that page? to know that CPU ^{creates} ^{assumes} a logical address space which has 2 parts one is page & other is their word number (instruction numbers)
- * here CPU does not physically create anything but ^{designed to} ^{assumes} page number and word number which then sent to page table which extracts frame number of respected page. Now frame + word number becomes Physical Address used to ^{for} ~~give~~ accessing.

~~from~~ instructions from main memory.

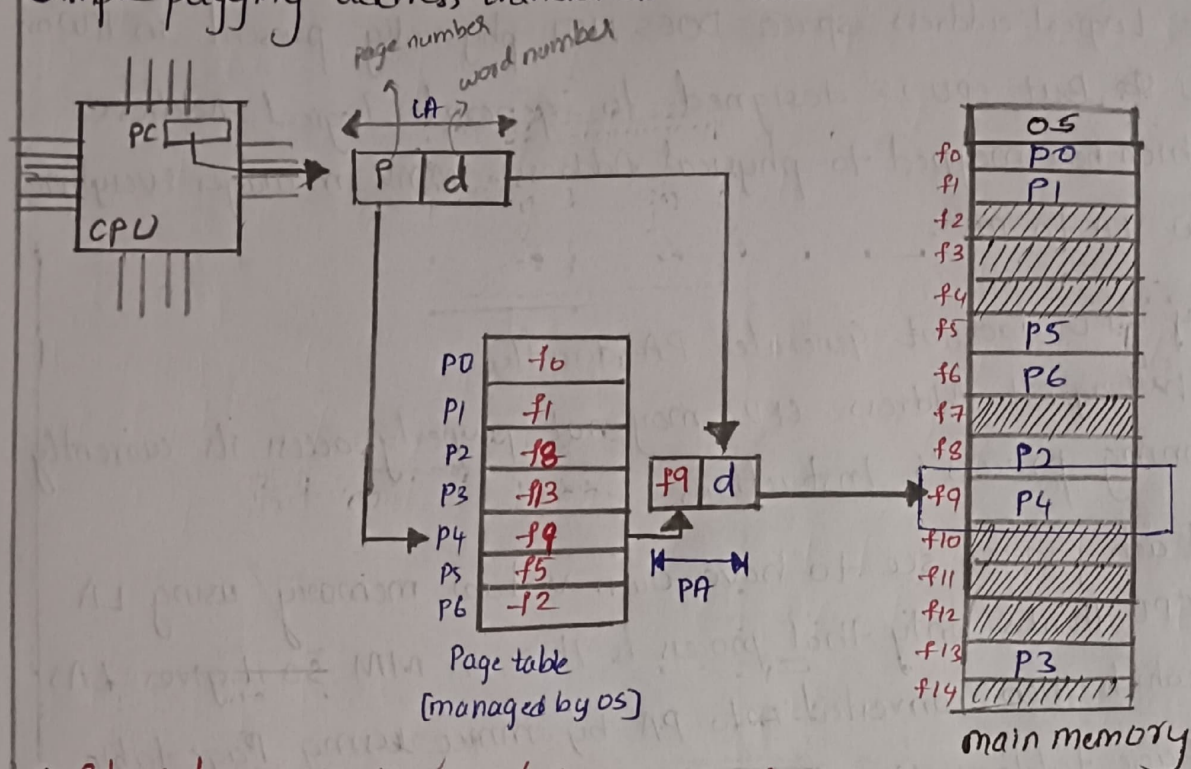
Note: Logical address space DOES NOT physically present in HW or SW ~~But~~ But CPU is designed to generate logical address which then mapped to physical address. MMU manages everything b/w LA to PA.

Why CPU doesnot generate PA directly

with^{out} Logical Address CPU may not properly access its currently running process's instructions.

- * Each process see to have own virtual memory/using LA
- * CPU thinks only that process is there in MM so it gives LAs which then converted into PA by mmu using Page table
- * Page table is created & managed by OS which has all process & frame address of currently running process.
- * Each process has its own page table. Each process is divided into pages and each page contains group of instructions & data
- * Degree of multi-programming = High as total no. of pages are in main memory.
- * **NOTE** = page table is a data structure which keeps all ~~page number~~ lists of page information of a process irrespective of their presence in RAM. with an extra bit 'valid/invalid bit' which show whether referred page is inside MM or Not.

Simple paging address translation Scheme



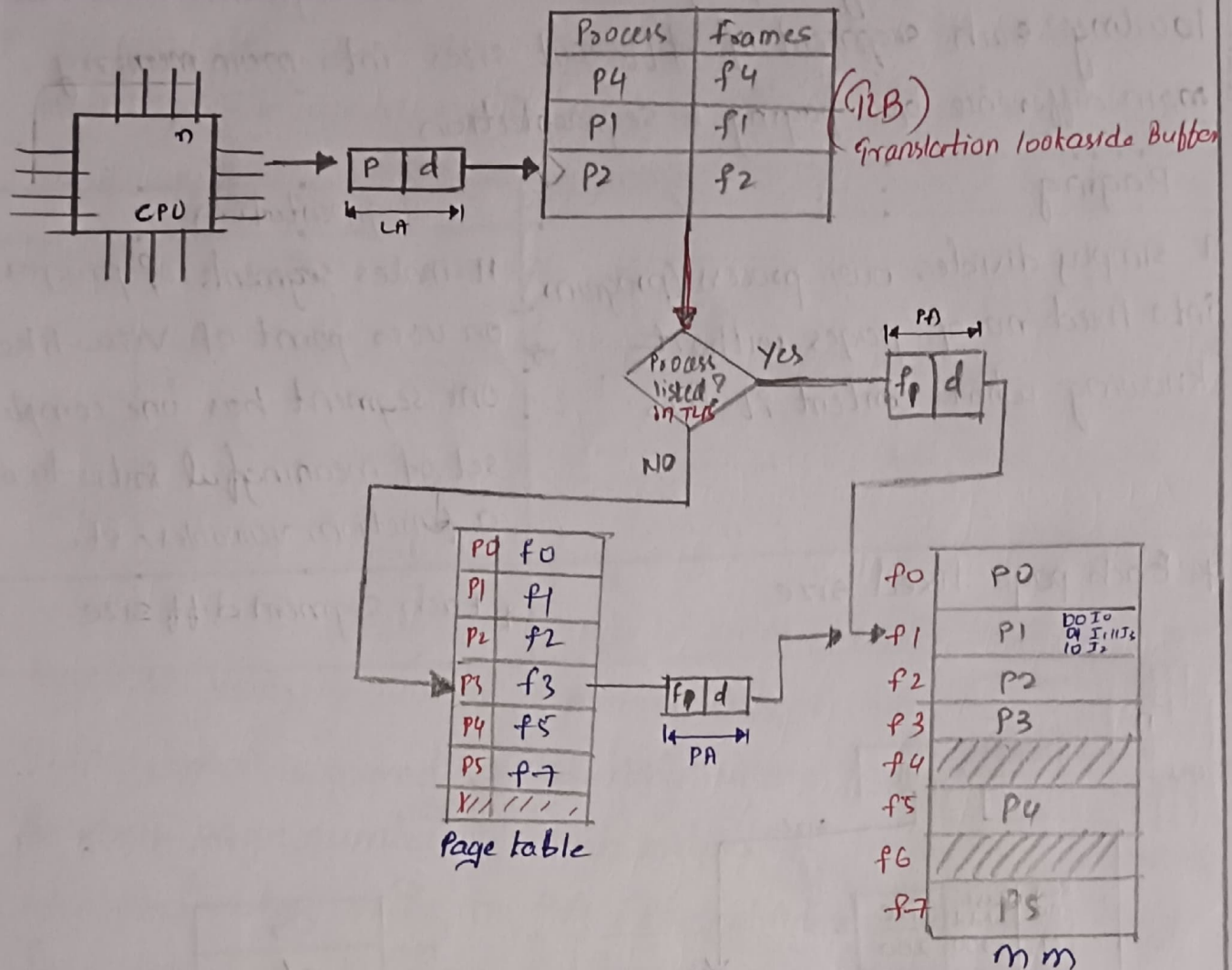
- * Steps how paging used by system for memory management
- * Program is divided into pages of fixed size & are allocated in mm
- * When PC in CPU loads new instruction, then CPU generates Logical Address
- * LA contains Page number & word number (instruction number)
- * It is given to page table managed by OS
- * Then OS maps Page number $\rightarrow$ to frame number of main memory
- * Main memory + word number is combined & instruction is fetched
- * $LA = \text{Page number} + \text{word number}$
- * $PA = \text{frame number} + \text{word number}$

Performance :

$$EMAT (\text{Effective memory access time}) = (m + m) \rightarrow \text{main memory access time for OS + actual frame table}$$

- * Size of each frame = Size of each page
- * n bit processor $\rightarrow 2^n$ pages assumption $\rightarrow$ limited pages in mm (till actual program size) if not then loaded later by OS from disk

Simple paging with TLB (Translation lookaside Buffer)



* As Logical Address in without TLB Paging mechanism is slow process $\rightarrow$ a time access request so TLB as small H/w chip is used as intermediate b/w LA & PA so

* first PA is mapped in TLB - Buffer if frame & process is listed in it then firstly mm is accessed. if not, then again page table is used to map mm

$$EMAT [\text{effective memory access time}] = x(c+m) + (1-x)(c+m+m)$$

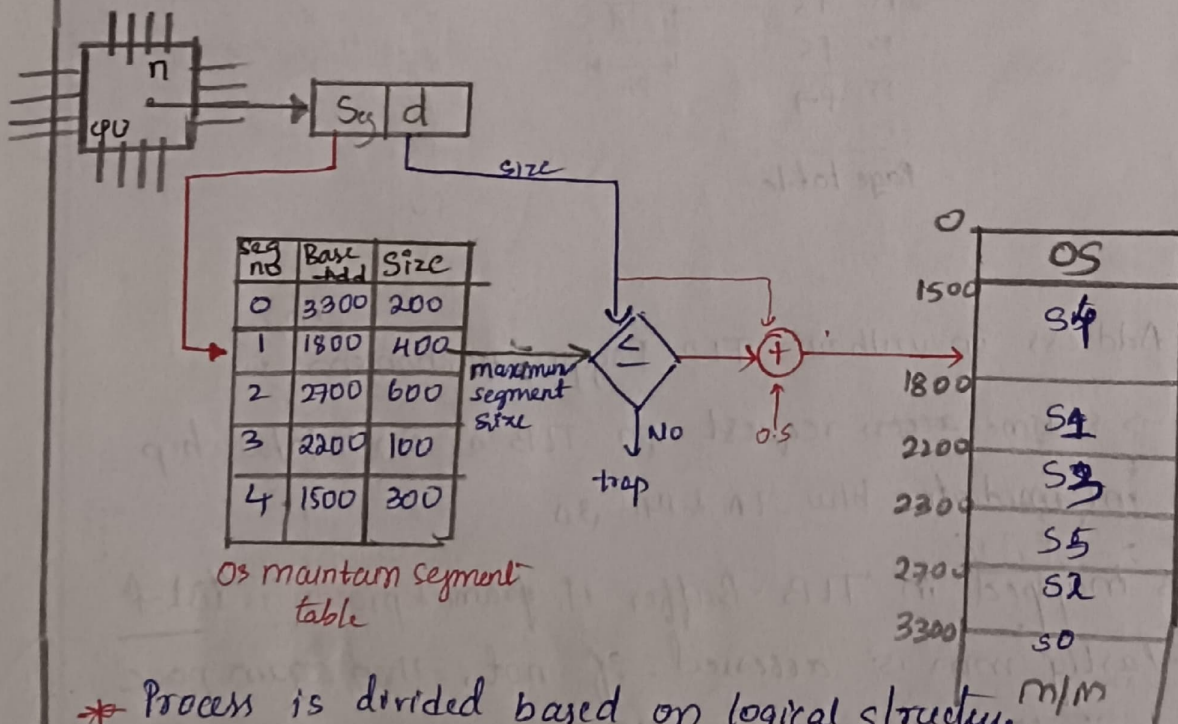
x : TLB hit ratio
 c : TLB access time
 m : mm access time
 $(1-x)$: TLB miss ratio
 c : page table access time
 m : frame access time

$$x(c+m) + (1-x)(c+m+m)$$

Segmentation

It refers to dividing a program into related segments and then loading each segment of different sizes into main memory.
main difference b/w paging & segmentation

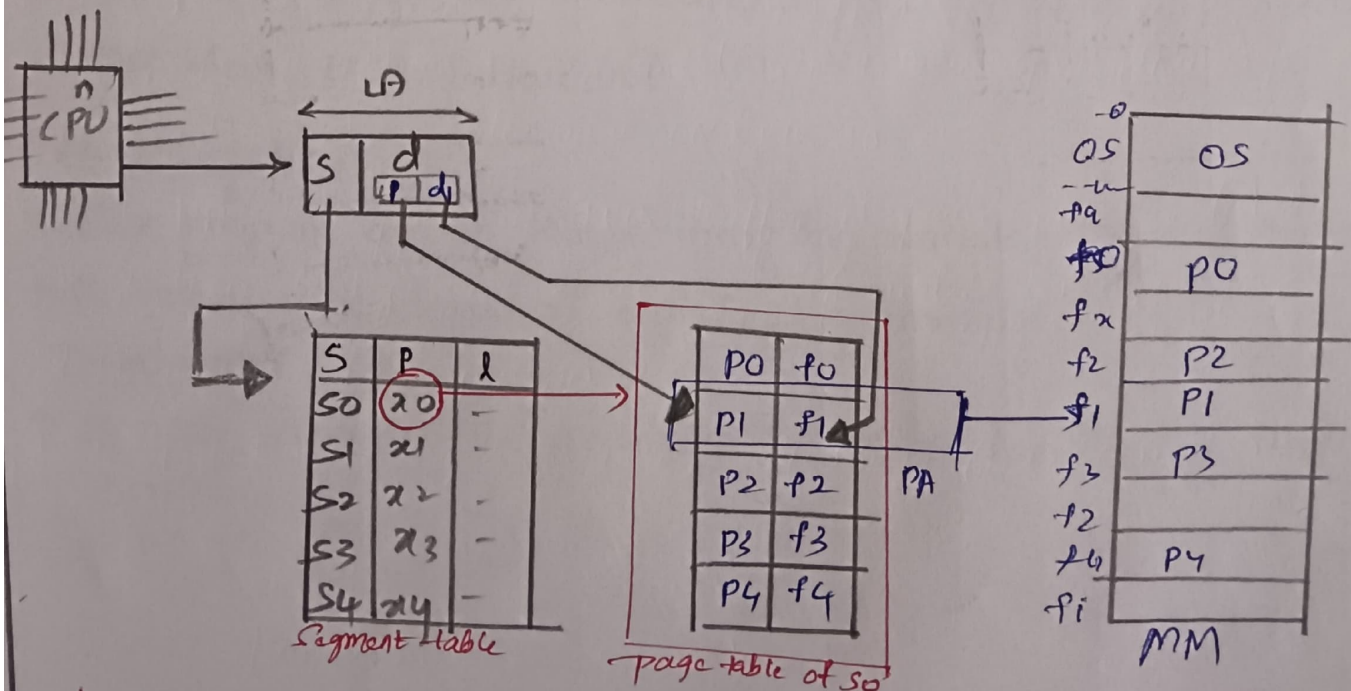
Paging	Segmentation
It simply divides our process (program) into fixed no. of pages without knowing what content it has.	It makes segments of program on user point of view like one segment has one complete set of meaningful instruction like a function, variables, etc.
* Each page fixed size	* Each segment diff size



- * Process is divided based on logical structure
- * segments are stored in non-adjacent locations
- * Supports protection & sharing
- * No internal fragmentation but External fragmentation

Segmentation with paging.

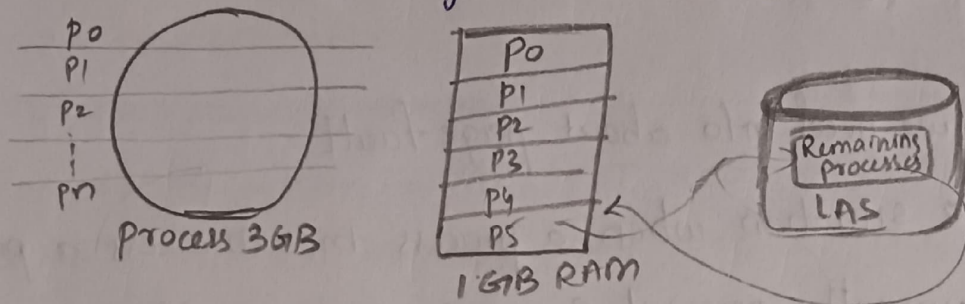
- * Paging is applied on each Segment
- * CPU generates LA which contains 2 parts segment number and ~~table~~ page information Section → page information section is divided into 2 parts page number and instruction/word number
- * Segment number is used to map the page table addresses of specific segment from segment table.
- * Segment table has 3 cols segment no., page table of each segment, length of segment.
- * When ever CPU sends LA firstly Segment table - from OS is used to identify which page table it belongs to (Address of page table)
- * Then control is moved to identify where page table is located & which page number & which instruction inside that page is identified, which refers to PA (Physical Address)
- * Then instruction is accessed from MM



* No External Fragmentation

Virtual Memory

It is a technique that allows execution of processes that may not reside completely in the memory.



- * Virtual memory allows execution of process whose size is ~~far~~ larger more than RAM size
- * Small portion of Harddisk called Logical address space is used as main memory.
- * A process is divided into pages, each page may not be loaded in RAM so how many free space is available that much pages are loaded remaining pages are stored in LAS area of HDD
- * When CPU needs page instructions which are not in MM then OS swaps not much useful pages in RAM with needed one from LAS of Harddisk.

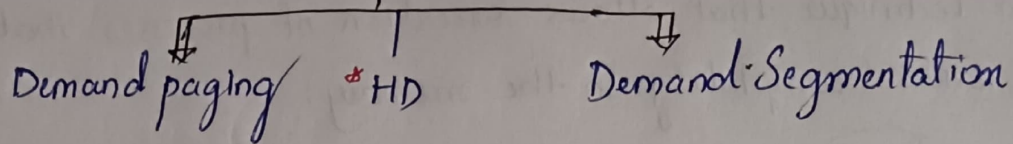
Advantages

- * User program can be larger than the available physical memory
- * It increases the degree of multi-programming.

Disadvantages

- * It may decrease the performance of the system if it is used carelessly.

Implementation of VM

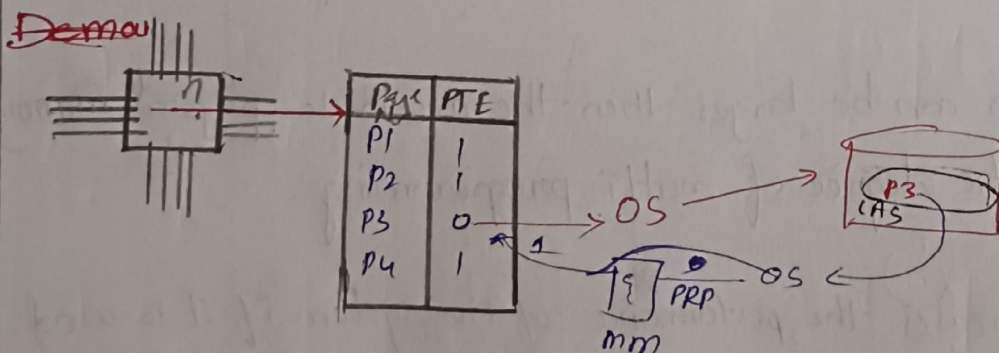


Demand Paging

* Before this we need info about page-fault.

Page-fault = a situation when a process tries to access a page that is not currently present in main memory (RAM)

- * CPU's logical Address is sent to OS
- * OS checks the page table
- * If present page table has 1 if not then 0 as extra bits
- * If not present then is called page fault.
- * That extra bit is present in frame column only called as PTE or
- * Now OS try to bring that page from HDD's LAS area and load in main memory if frames are freely available, if not then OS selects page reference policy (PRP) to replace any page in MM with new page & update bit in page table.
- * Time taken for this entire process is called Page-fault Service time (PFST) → S



Demand Paging: Whenever page-fault is occurring bring the page from LAS to PAS on demand.

$$EMAT = P * S + (1 - P) * m$$

P = Page fault rate

S = page fault Service time

$1 - P$ = Page hit ratio

m = main memory access time

Q) Consider a System with PFST 200 ns and page hit ratio is 95% & main memory access time 20 nsec what is EMAT?

$$S = 200$$

$$1 - P = 0.95 \text{ so } P = 0.05$$

$$m = 20$$

$$EMAT = P * S + (1 - P) * m$$

$$= 0.05 * 200 + 0.95 * 20$$

$$= 10 + 19$$

$$= 29 \text{ ns.}$$

Page replacement policies

FIFO

Reference String (page no.s referred by the process while execution)

Ex¹ → Page no. referred by process executor (CPU)

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

→ 4 frame MM

4 frames → 10 pf (more frames lesser Page fault) } ^{but} false
 3f → 15 pf (less frames more page fault)

Ex²

1	2	3	4	1	2	5	1	2	3	4	5
			4			4	4	4	3	3	3
		3	3			3	3	2	2	2	2
	2	2	2			2	2	1	1	1	5
1	1	1	1			5	5	5	5	4	4

4f → 10 pf

here with more frames more pf & with less frames less pf
 we got this unusual behaviour is called Belady Anomaly

↓

1	2	3	4	1	2	5	1	2	3	4	5
	3	3	3	2	2				2	4	
	2	2	2	1	1	1			3	3	
1	1	1	4	4	4	5			5	5	

3f → 9 pf

* FIFO Algorithm can cause Belady Anomaly

Optimal Page replacement policy

It states whenever a page fault is occurring replace a page which you are not going to use for longest duration of time in future.

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

4f → 8 pf

* This algorithm will not suffer with Belady Anomaly

* We cannot use this alg practically as its not possible to identify which process will be used in the future so it's a benchmark algorithm.

Least Recently Used (LRU)

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

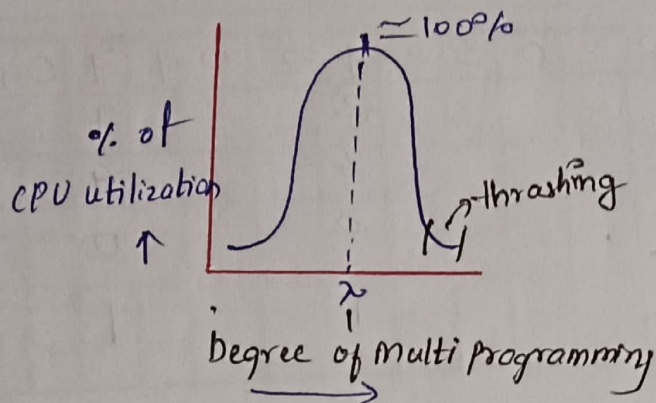
4f → 8 pf

* This algorithm is practically implementable & is close to optimal page replacement policy's performance

Thrashing

* A condition where CPU spends large amount of time in bringing pages from hard-disk.

ex:- 90% on I/O (paging activity)
10% on Execution (instruction execution)



* So, as no. of processes in the mm increases i.e. if Degree of MP increases above the threshold value then CPU utilization decreases at a point of time when it goes down it's called thrashing.

Reasons for thrashing

- ① Less memory
- ② High degree of MP
- ③ Page Replacement Policy
- ④ Page Size.

Ex: Program = 10MB

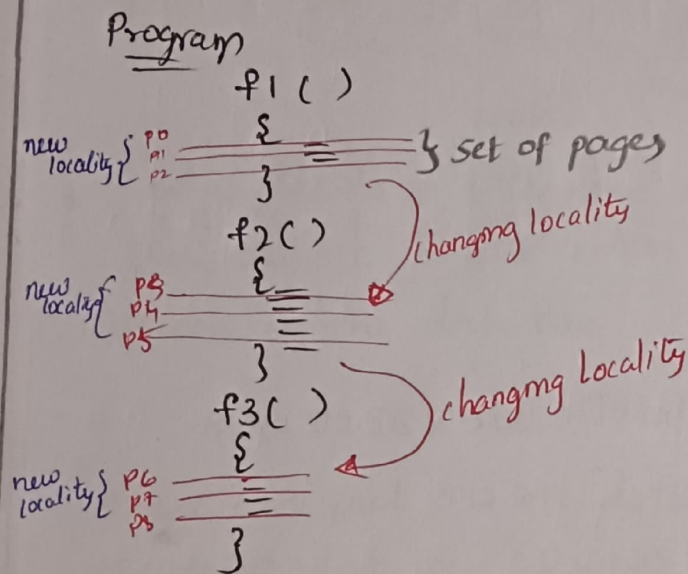
If page size = 2MB No. of pages = 5

If page size = 5MB No. of pages = 2

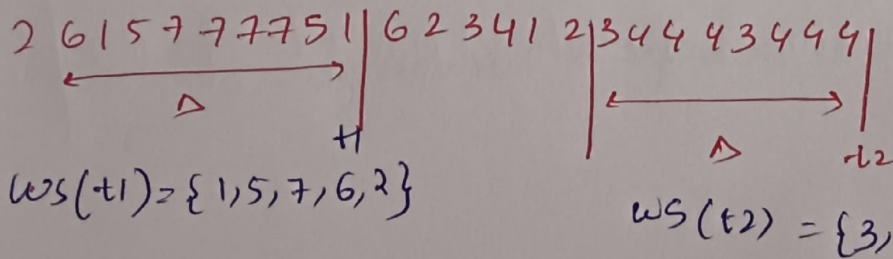
Thrashing Handling Technique.

① Working Set Model.

- It is a method used to prevent thrashing
- works based on locality model
- A locality is a set of pages that are actively used together
- locality models states when a process executes it moves from one locality to another because program is a collection of localities (functions)

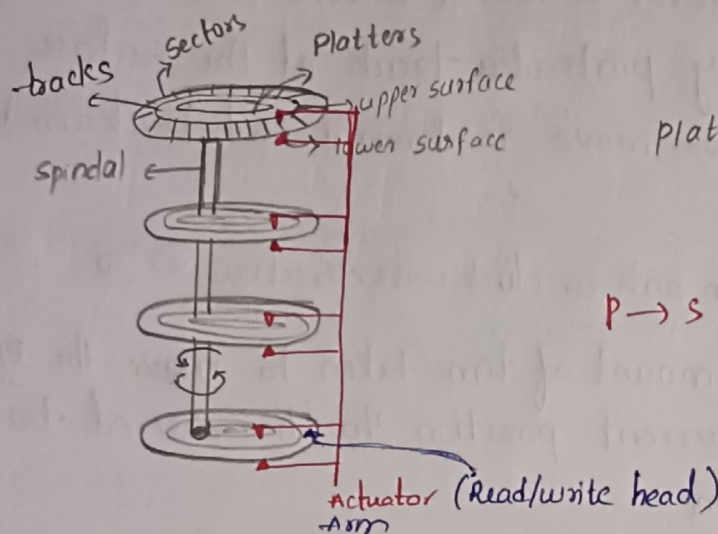


- working set model uses working set window (Δ)
- working set (ws) = The set of pages in the most recent page reference (set of unique pages in a duration of time in process)
- working set window (Δ) = no. of pages in the most recent page reference



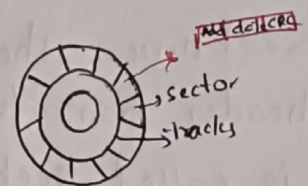
- if $\text{sum}(ws) + PO - PI > AF$ (Available frames) → Process gets suspended ^{some}
- if $\text{sum}(ws) + PO - PI < AF$ → Process Resumed.

Ch-6 Disk Structure



Platter $\rightarrow$ Surface $\rightarrow$ Track $\rightarrow$ Sectors
 $\downarrow$
 data

$P \rightarrow S \rightarrow T \rightarrow S \rightarrow D$



- * If no. of platters 8 & They are 2 surfaces & 256 tracks in Every Every surface & 512 sectors in every tracks. Then how many sectors on the disk are.

$$8 \times 2 \times 256 \times 512 = \text{Total no. of sectors}$$

If we can put 512 kB data in each sector then what is the total disk size. & what is the no. of bits required to represent disk size

$$\text{Disk size} = P \times S \times T \times S \times D$$

$$= 8 \times 2 \times 256 \times 512 \times 512 \text{ kB}$$

$$= 2^3 \times 2^1 \times 2^8 \times 2^9 \times 2^9 \times 2^{10} \text{ B}$$

$$= 2^{40} \text{ B}$$

$$\boxed{\text{Disk size} = 1 \text{ TB}}$$

$$1 \text{ k} = 2^{10}$$

$$1 \text{ M} = 2^{20}$$

$$1 \text{ G} = 2^{30}$$

$$1 \text{ T} = 2^{40}$$

∴ No. of bits required to represent Disk size = 40

Disk I/O Operation (Disk performance parameters)

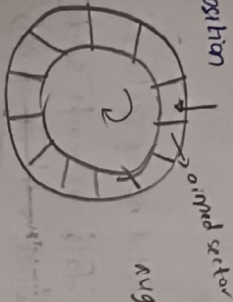
- R/W header can never be moved outside the track it will be pointing to any particular track of the surface
- R/W header will be move in forward and backward direction

(either ← or →)

- disk can rotate in anti or clockwise direction ↻

Seektime = the amount of time taken to move the R/W header from its current position to the desired track is called seektime.

Rotational latency: The amount of time taken to rotate the track when the R/W header comes to exact position (sector) from current position



avg time = $\frac{1}{2}$

Transfer time:

The amount of time taken to transfer the required data
 * Transfer time depends on the rotational rate of disk and the total size of the track.

Disk I/O operation =

Seektime + $\frac{1}{2}$ rotational time + transfer time

Disk Scheduling Algorithms

Goal: To minimize the seektime [time taken to reach up to desired track]

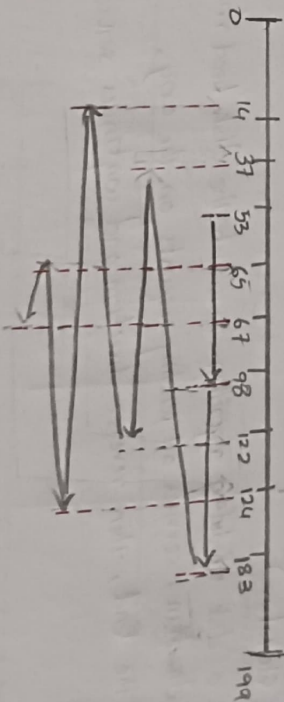
Algorithms

- 1) FCFS
- 2) SSTF (shortest seektime first)
- 3) SCAN
- 4) LOOK
- 5) ~~CSMA~~ CSAN
- 6) CLOOK

First come - first serve

Track requests Queue is 98, 183, 37, 122, 14, 124, 65, 67

Assuming 200 tracks from 0 to 199 & assume our R/W head is present at 53 initially

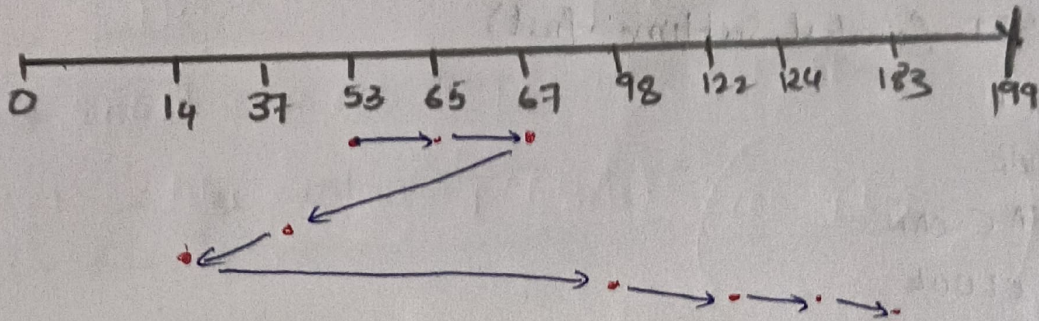


$$\begin{aligned} \text{Total Seektime} &= (98-53) + [183-98] + [183-37] + [122-37] + \\ &[122-14] + [124-14] + [124-65] + [67-65] \\ &= 640 \end{aligned}$$

* No Starvation.

Shortest Seek-time First (SSTF)

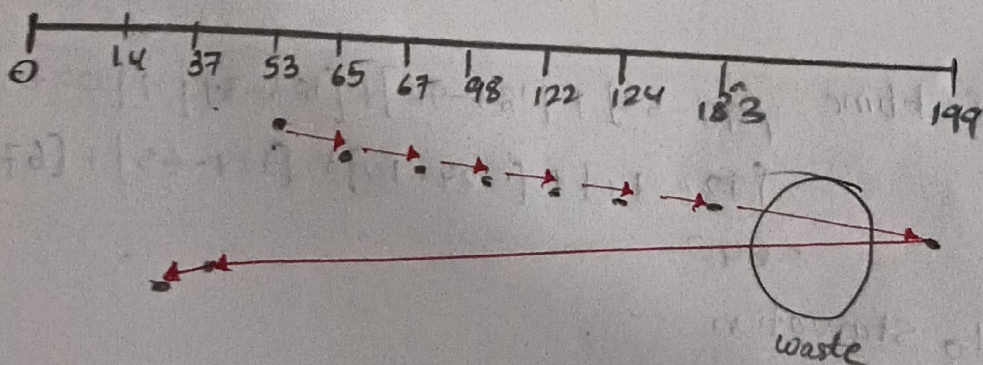
Select a request from Queue that request least movement of disk arm from its current position regardless of the direction Queue = 98, 103, 37, 122, 14, 124, 65, 67



* Starvation; If a request came first but is far away from R/W head & before reaching that request in b/w many other requests came having less seek time then starvation causes.

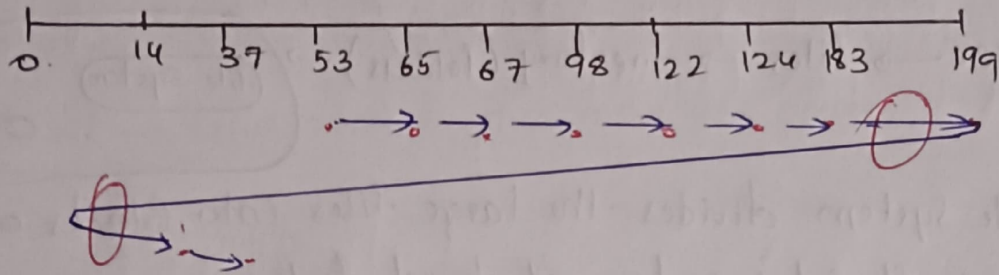
SCAN (elevator)

scan is a disk scheduling algorithm. in which the disk head moves in one direction, services all pending requests on the way, reaches the end, then reverse direction and continues servicing requests.



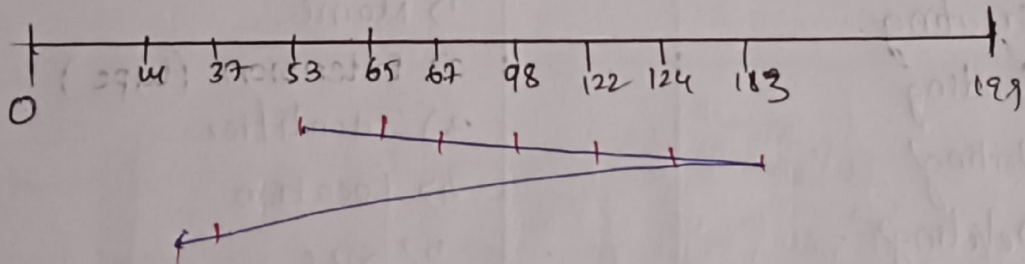
C-SCAN

CSCAN is a disk scheduling alg, in which the disk head moves in one direction $\rightarrow$ up direction, $\rightarrow$ all pending request gets served but while reverse no pending services are met it goes to first then again moves up direction & serves all.

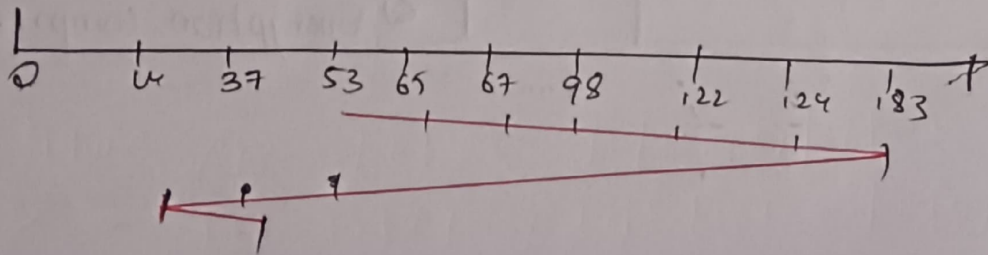


LOOK

It looks last request in direction & reach it by ^{serving all} ~~the~~ ^{met} request



CLOCK



File Access methods

- ① Sequential Access
- ② Direct Access
- ③ Indexed Access

① Sequential Access. Ex [music Taperecorder]

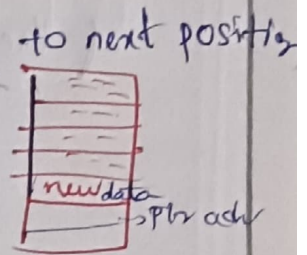
★ All records of file is accessed one after another.

★ Supports following operation

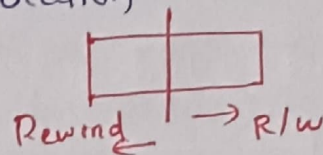
i) Readnext → Read record and move the pointer to next position

ii) writenext → write & advance the position
similar to append

iii) rewind → moving back to earlier location



Advantage
Simple to implement
disadv → slow



Direct Access

It is based on the disk model. It allows random access user can jump to any record and access that record operations

1) read n → Reading nth record.

2) write n → writing nth record

3) jump to record n

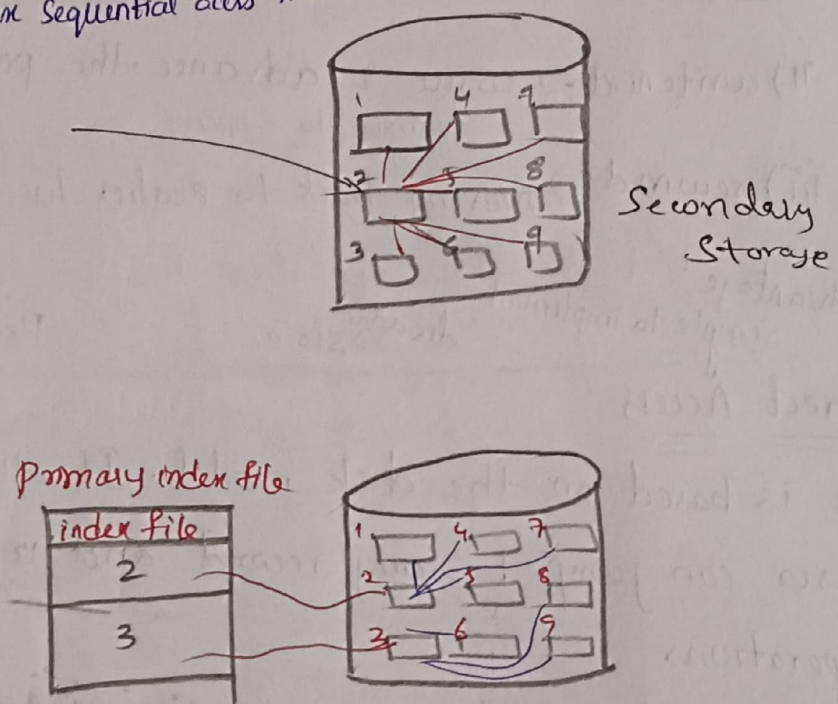
Adv
File can be immediately access decreasing avg access time
No need to transverse all blocks preceding to jump on specific block of record.

Indexed Access

An Index is maintained which contains the address (or ptrs) of the file blocks

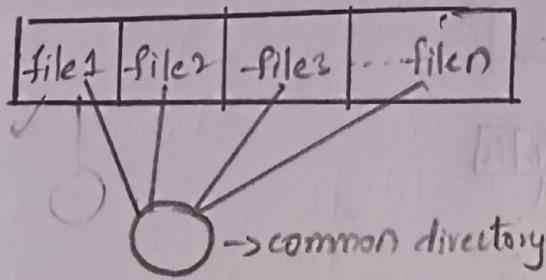
* The primary index file contains the pointers (or address) to the secondary storage index file & Secondary index file contains the actual address of the blocks of the file (ISAM)

Index Sequential access method



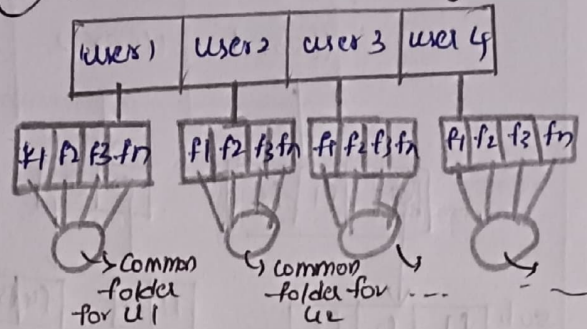
Directory Structure Organization

① Single level directory



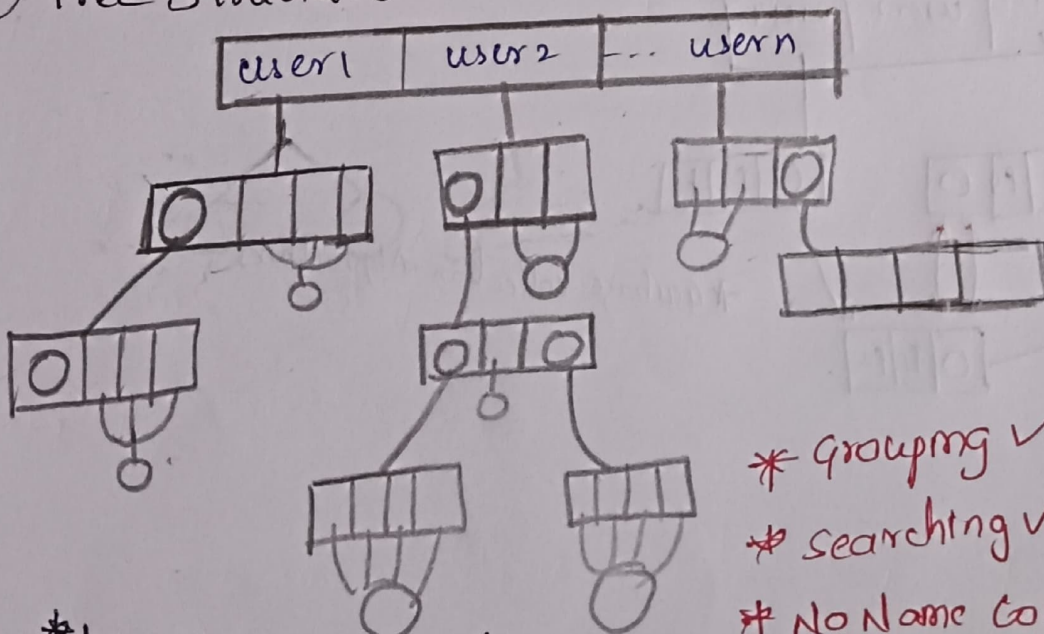
- * All files are stored in a single folder
- * **Search**: difficult to search files if large no. of files are there
- * **Grouping**: Not possible to group some files all are in one group
- * **Name conflicts**: two files can't have the same name.
- * Easy to implement. ~~no~~ multi-user ~~conflict~~

② Two level directory



- * different users have different folders
- * Each user have one single folder
- * different user can have same file name (user1 file name = user2 file name x No user level conflicts)
- * Specific user can't create sub-folders (groups) from their respective root directories

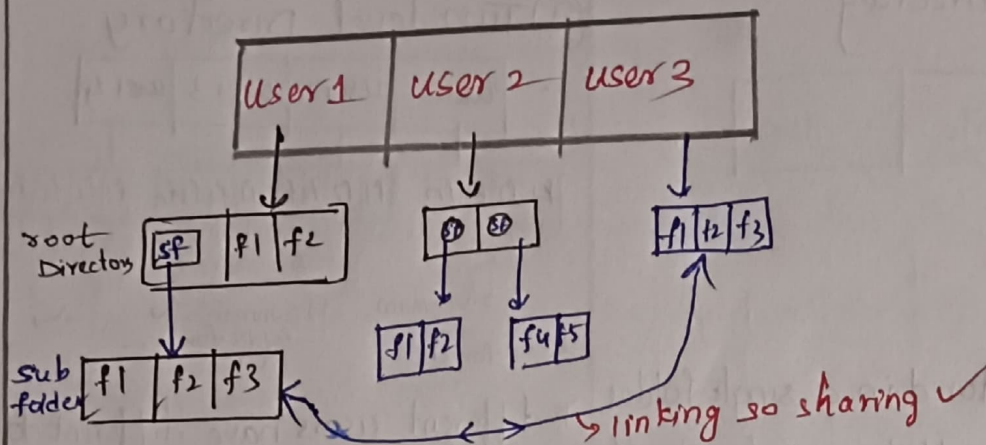
③ Tree Structure



* difficult to implement

- * grouping ✓
- * searching ✓
- * No Name conflicts
↓
Same name files can be stored in separate directories
- * sharing X Not possible

④ Acyclic Directory Structure (ADS)



Notes by Abdul Muqsith

⑤ General graph structure.

