GATE CSE NOTES

by

Joyoshish Saha

Introduction

* **Operating System:** Program that controls the execution of application programs & acts as an interface between the user of a computer & the computer hardware.

→ OS controls the allocation & use of the computing system's resources among the various user & tasks.

→ It provides an interface between the computer hardware & the programme that simplifies & makes feasible for coding, creation, debugging of application programs.

* **Operating System Services.**

Functions that are helpful to the user:

a) User interface: Varies between command-line (CLI), GUI, batch.

b) Program execution: Loading program & run it.

c) I/O operations:

d) File system manipulation

e) communications. (Shared memory or packets)

f) Error detection: Taking action & debugging

Functions for efficient operation:

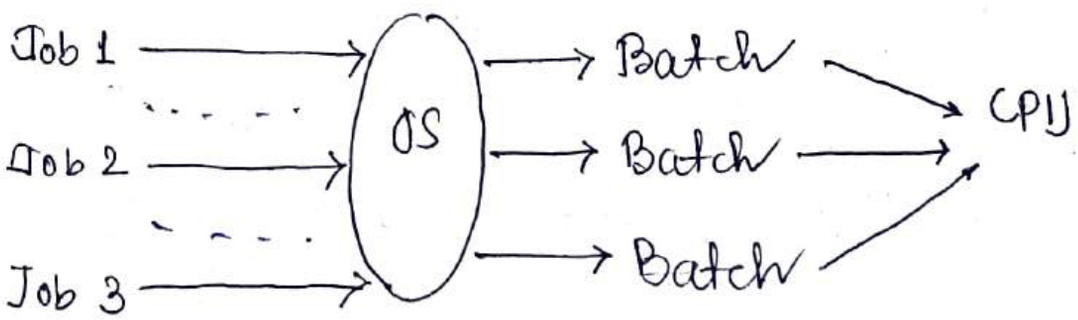
a) Resource allocation

b) Accounting (Keep track of which users use how much & what kinds of computer resources.)

c) Protection & Security

* Types of OS.

1. Batch OS. : Operator takes similar jobs having same requirement & group them into batches.



Adv: Multiple users can share the batch systems.

The idle time for batch system is very less.

It's easy to manage large work repeatedly in batch systems.

Disadv.: Priority can't be set for jobs.

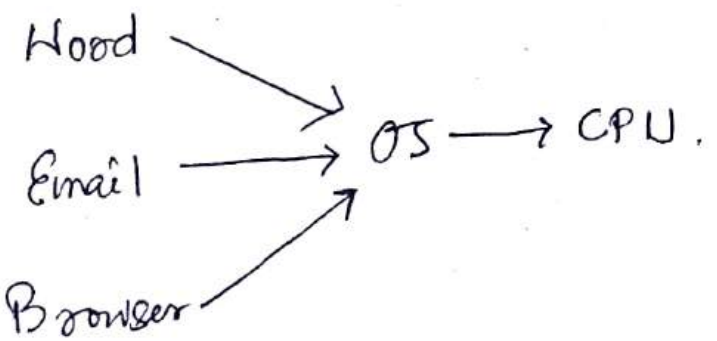
May lead to starvation because of a batch taking long time for execution.

Hard to debug.

eg. Payroll system, Bank statements etc.

2. Time Sharing OS.

Each task is given some time quantum to execute.

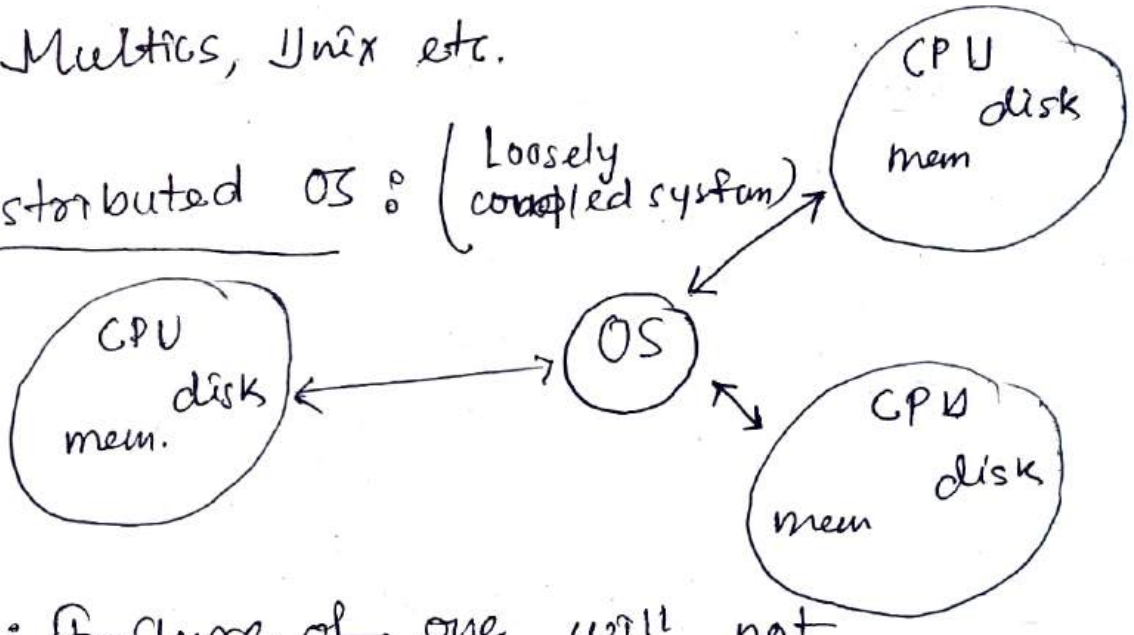


Adv.: Each task gets equal opportunity, less chances of duplication of S/W, CPU idle time can be reduced.

Disadv.: Reliability problem, data communication problem.

eg. Multics, Unix etc.

3. Distributed OS: (Loosely coupled system)



Adv.: Failure of one will not affect the other network communication.

Computation highly fast (as resources are shared).

Scalable & less load on host computer.

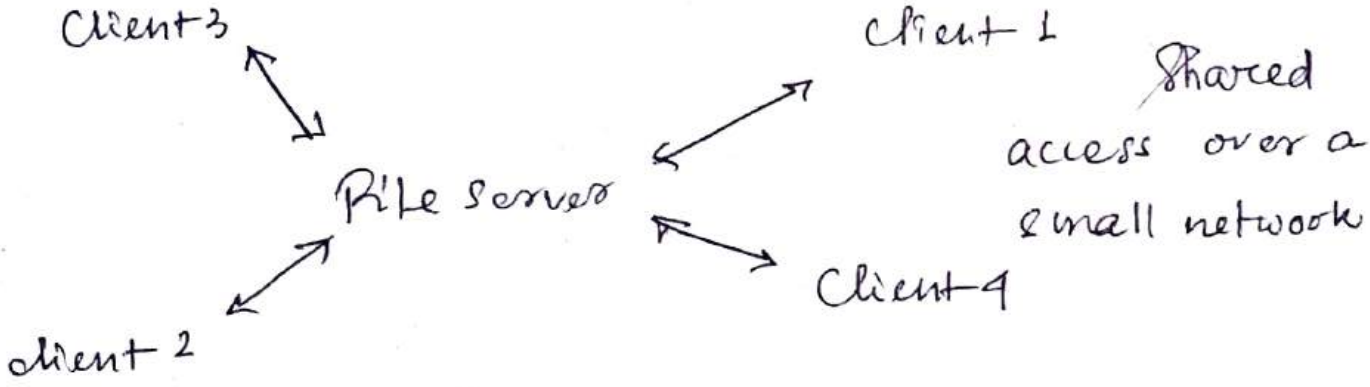
Delay on data processing reduces.

Disadv.: Failure of main network will stop the entire communication.

Very expensive.

eg. LOCUS.

4. Network OS. : (Tightly coupled system).



Adv.: Highly stable centralised servers.

Security concerned handled through servers

Server accesses are possible remotely.

Disadv.: Costly servers.

Maintenance & updates required regularly.

Eg.: Ms Windows server 2003, Windows server 200

BSD etc.

5. Real time OS (RTOS):

Time interval & response time are very small.

2 types:

i) Hard RTOS: Time constraints are very strict. Virtual memory never found.

ii) Soft RTOS: Time constraint less strict.

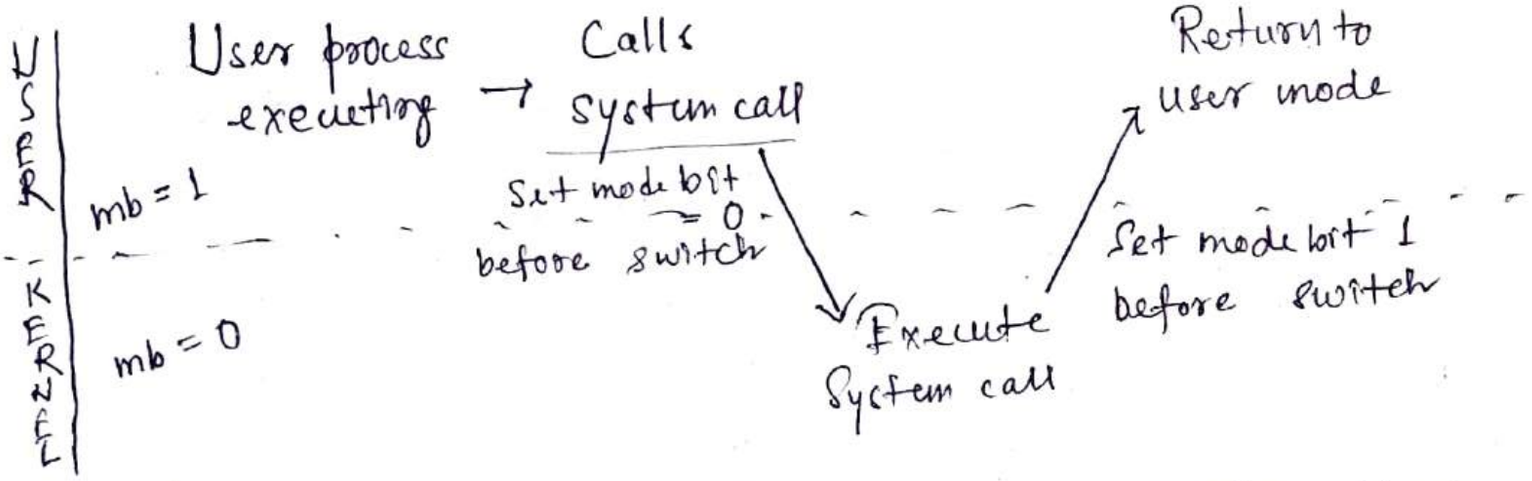
Adv.: Maximum consumption, task shifting, focus on application, RTOS can be used in embedded systems, error free memory allocation best managed.

Disadv.: Limited tasks, heavy system resources usage, complex algo., needs specific device drivers & interrupt signals.

eg.: Weapon systems, air traffic control system, QNX, VxWorks, RTLinux, FreeRTOS

* Dual Mode operations in OS :

1) **User Mode :** When the computer system runs user app. then the system is in user mode. When the user app. requests for a service from the OS or an interrupt occurs or system or system call, then there will be transition from user mode (1) to kernel mode (0).



2) **Kernel Mode :** When system boots then H/W starts in kernel mode & when OS is loaded then it starts user app. in user mode. We have privileged instructions that execute in kernel mode.

↳ [handling interrupts, I/O mgmt.]

* Privileged instructions.

Can run only in kernel mode.

→ Any attempt to execute p-ins^ns in user mode, is treated as an illegal ins^n. H/W traps it to the O/S.

→ Any ins^n that can modify the contents of the timer is a privileged ins^n.

→ P. ins^ns are used by the OS in order to achieve correct operation.

→ eg. I/O ins^ns & halt ins^ns, turn off all interrupts, setting timer, context switching, clearing memory, modifying entries in device-status table.

* Non-privileged instructions.

Can run only in user mode.

eg. Reading status of processes, reading system time, generating any trap ins^n, sending final printout of printer.

* Functions of an OS

Processor management, Device management, Buffering, Spooling (Simultaneous peripheral operation on line), memory mgmt., partitioning Virtual memory, file mgmt.

* System Calls.

Programming interface to the services provided by the OS. System calls are accessed by programs via a high level API.

Types:

a) Process control
b) File mgmt.
c) Device mgmt.
d) Information maintenance
e) Communications

a) fork(), exit(), wait()

b) open(), read(), write(), close()

c) ioctl(), read(), write()

SetConsoleMode | read console ...

d) getpid(), alarm(), sleep()

e) pipe(), shmget(), mmap()

create file mapping

map view of file.

Protection —

chmod(), set file security

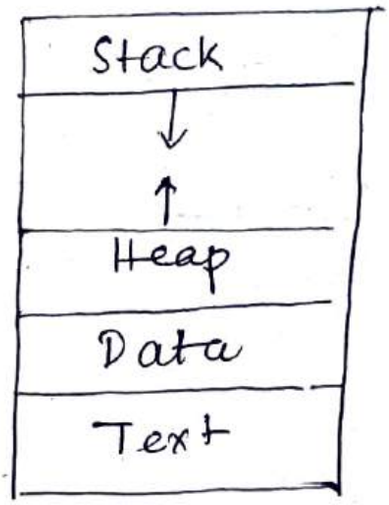
chown()

Set security descriptor group

Process Management

*Process : A program in execution.
(Active entity)

*Process in memory



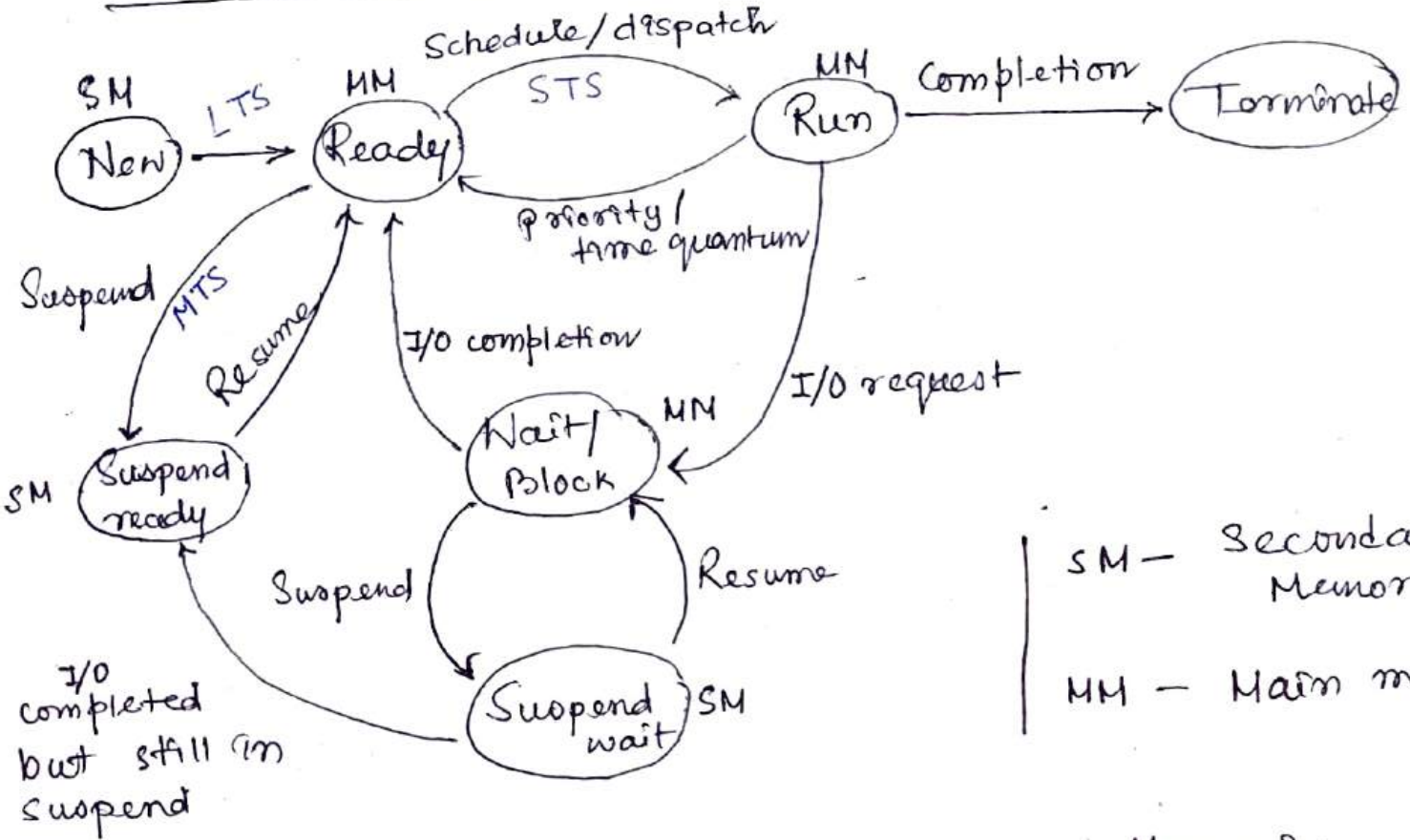
*Process attributes.

i) Identifier : Unique id for process
ii) State
iii) Priority
iv) Program counter
v) Memory pointers.
vi) Context data
vii) I/O status information
viii) Accounting information

Suspend ready: Process that was initially in the ready state but were swapped out of MM & placed onto external storage by scheduler are said to be in suspend ready. They will transition back to ready whenever the process is again brought onto MM.

Suspend wait: Process was performing I/O & lack of MM caused them to move to SM

* Process States.



→ A process necessarily goes through minimum 4 states. (new, ready, run, terminate).
Process requiring I/O, min. no. of states is 5.

→ A single processor can execute only one process at a time.

→ Moving a process from wait to suspend wait is a better alternative, as the process is already waiting for some blocked resource.

* Context Switching

Process of saving context of one process & loading the context of another process.

Context switching happens when -

a) a high priority process comes to ready state

b) an interrupt occurs

c) User & kernel mode switch

d) preemptive CPU scheduling used.

* CPU - bound & I/O - bound process

CPU - bound processes require more CPU time or spends more time in running state. (Intensive in CPU operation)

I/O - bound processes require more I/O time. More time in waiting state. (Intensive in I/O operations)

* Mode switch time < Context switch time

* Multiprogramming : Multiple processes in ready state.

1. With preemption –

Forcefully removed process from CPU.

(Time sharing, multitasking)

2. Non-preemption –

Not removed until execution.

Degree of multiprogramming is the maximum no. of processes that can be present in the ready state.

→ Optimal degree of multiprogramming means average rate of process creation is equal to the average departure rate of processes from main memory.

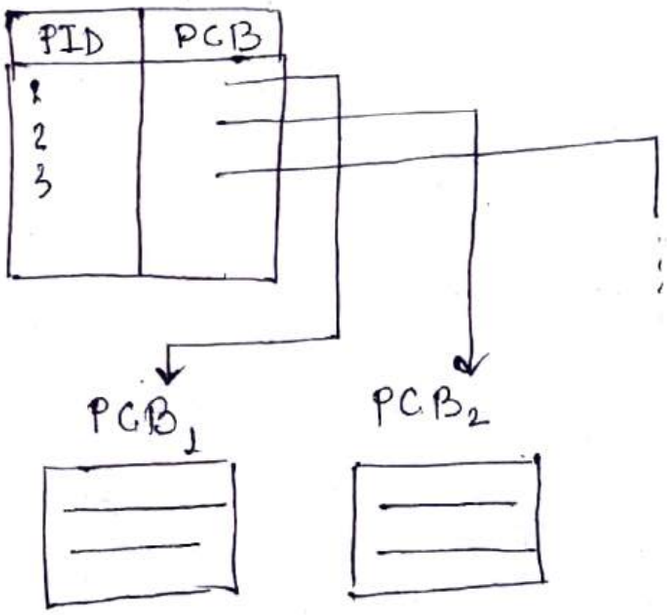
* Process table

Data structure maintained by the OS to facilitate context switching & scheduling

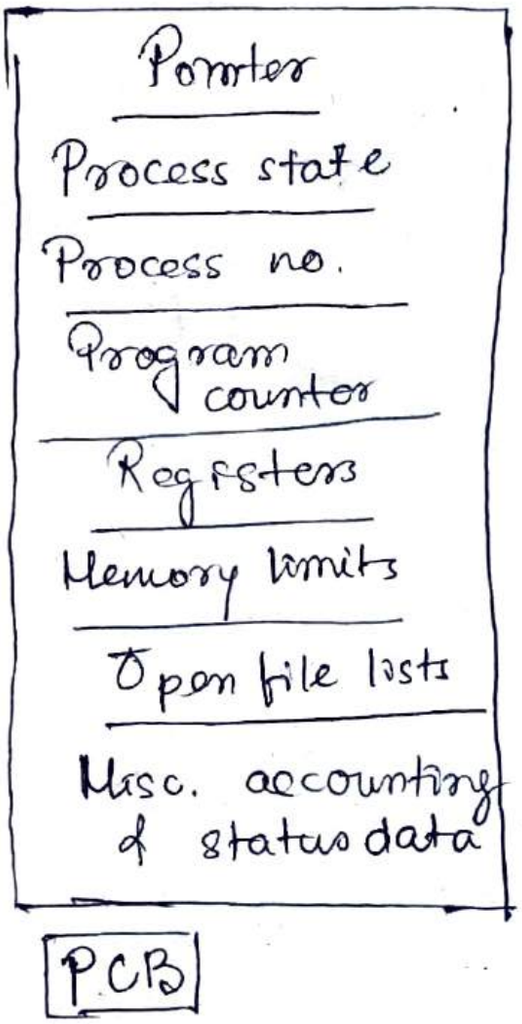
It contains all the current processes' information or PCBs in the system.

*Process Control Block.

Data structure that contains information of the process related to it.



→PCB of each process resides in MM.



* Schedulers.

Help in scheduling the processes in various ways. Responsible for selecting the jobs to be submitted into the system & deciding which process to run.

→ Types of schedulers.

1. Long term scheduler. : Also job scheduler.
performance
It selects a balanced mix of I/O bound & CPU-boun processes from the secondary memory (new) Then it loads the selected processes onto the MM (ready) for execution.

Primary objective of LTS is to maintain a good degree of multiprogramming. (New → Ready)

2. Short term scheduler : Also CPU scheduler.
context switching time
It decides which process to execute next from the ready queue. After STS decides the process, dispatcher assigns the decided process to the CPU for execution. (Ready → Run)

Primary objective of STS is to increase the system performance.

Software that moves processes from ready to run state and vice versa.

3. Medium term scheduler.

swapping time

Swaps out the processes from MM to secondary memory to free up the MM when required. It reduces degree of multiprogramming. After some time when MM becomes available, MTS swaps in the swapped out processes to the MM & its execution is resumed from where it left off.

Primary objective is swapping. Swapping may be required to improve the process mix.

→ Dispatcher.

A dispatcher is a special program which comes into play after the scheduler. Dispatcher takes the process to the desired state/queue after scheduler selects the process.

It involves - context switch, switching to user mode, jumping to the

proper location on the user program to restart that program.

Dispatcher	Scheduler.
i) Module that gives control of CPU to the process selected by STS.	i) Selects process among various processes.
ii) It's a code segment. No types.	ii) 3 type, L, M, S.
iii) Dependent on scheduler for working.	iii) Works independently.
iv) No specific algo.	iv) Works on various algo - SJF, FCFS etc.
v) Time taken is dispatch latency.	v) Time taken is negligible.
vi) Also responsible for context switches, switching to user mode.	vi) Only work is selection of processes.

* Various times related to process.

1. Arrival time: Process enters the ready queue.

2. Waiting time: Time spent by the process waiting in the ready queue for getting CPU.

Waiting time = Turn around time − Burst time

3. Response time: Time after which a process gets the CPU for the first time after entering the ready queue.

Response time = Time at which process 1st gets CPU − arrival time

4. Burst time : Time required by a process for executing on CPU.

Also called running time. It can only be known after the process has been executed.

5. Completion time :

Time at which a process completes its execution on the CPU & takes exit from the system.

6. Turn around time :

Total time spent by a process in the system.

⇐ Turn around time = Burst time + Waiting time

or

⇐ Turn around time = Completion time − Arrival time

*Objective of Process Scheduling Algorithm.

1. Maximum CPU utilisation
2. Fair allocation of CPU
3. Maximum throughput [No. of processes that complete their execution per time unit]
4. Minimum Turn around time
5. Minimum waiting time
6. Minimum response time.

* Different Scheduling Algorithms.

- i) First come first serve (FCFS)
- ii) Shortest job first (SJF)
- iii) Longest job first (LJF)
- iv) Shortest remaining time first (SRTF)
- v) Longest remaining time first (LRTF).
- vi) Round robin scheduling
- vii) Priority based scheduling (Non-preemptive)
- viii) Highest response ratio next (HRRN)
- ix) Multilevel queue scheduling
- x) Multilevel feedback queue scheduling.

* Preemptive Scheduling.

Used when a process switches from running state to ready state or from waiting state to ready state. Resources are allocated to the process for the limited amount of time & then is taken away & the process is again placed back in the ready queue if it has burst time remaining.

Algo based on this —

Round Robin, SJF, priority etc.

* Non-preemptive Scheduling

Used when a process terminates or a process switches from running to waiting state. Once the resources are allocated to a process, process holds the CPU till it gets terminated or it reaches a waiting state.

Algo based on this -
SRTF, priority etc.

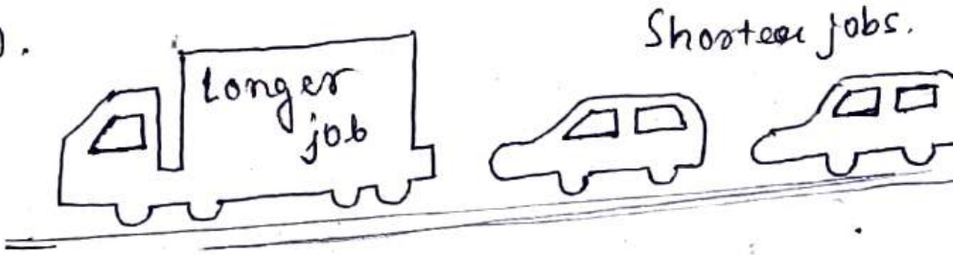
* First Come First Serve (FCFS) Scheduling

Process arriving first in the ready queue is firstly assigned the CPU. In case of a tie, process with smaller process id is executed first. It's always non-preemptive in nature..

Adv.: Simple & easy, doesn't lead to starvation.

Disadv.: • Doesn't consider priority or burst time of process.

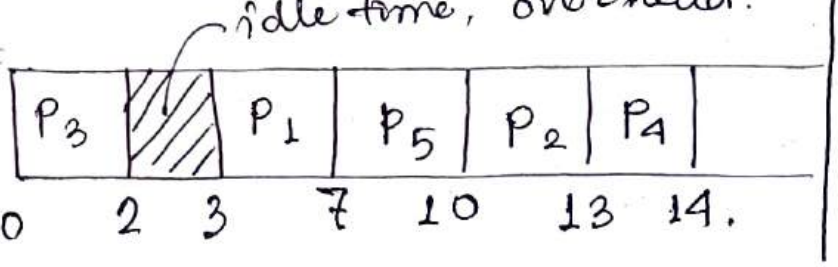
• Convoy effect - Processes with higher burst time arrived before the processes with smaller burst time. Then, smaller processes have to wait for a long time for longer processes to release the CPU.



eg.

PID	Arrival Time	Burst time	Exit time	Turn Around time	Waiting time
P1	3	4	7	4	0
P2	5	3	13	8	5
P3	0	2	2	2	0
P4	5	1	14	9	8
P5	4	3	10	6	3

Gantt chart.



avg. TAT = $\frac{4+8+2+9+6}{5}$

= 5.8

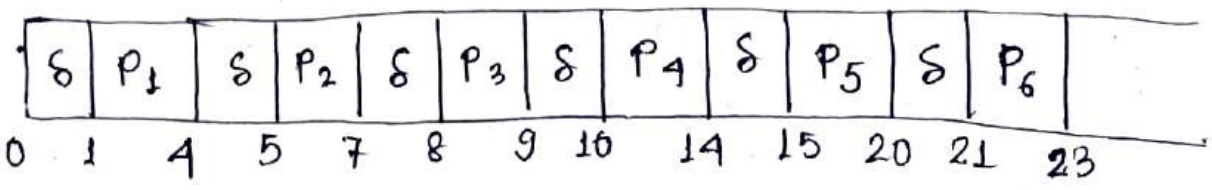
avg WT = 3.2

eg. 1 unit of overhead in scheduling the processes. Find out efficiency of algo.

PID	AT	BT
1	0	3
2	1	2
3	2	1
4	3	4
5	4	5
6	5	2

$\eta = \left(1 - \frac{6}{23}\right) \times 100\%$

$= 73.91\%$



* SJF , SRTF Scheduling.

SJF - CPU is assigned to the process having lowest burst time.
In case of a tie, it is broken by FCFS.

SJF

- Non-preemptive mode
- Preemptive Mode (SRTF)

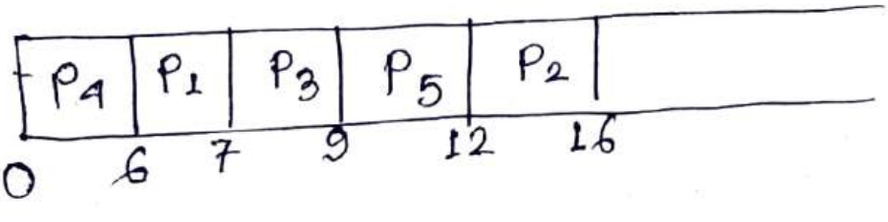
Adv. : SRTF is optimal & guarantees the minimum average waiting time.

Disadv. : Can't be implemented practically since burst time can't be known in advance. It leads to starvation of processes with larger burst time. Priorities can't be set for the processes. Processes with larger burst time have poor response time.

eg.

PID	AT	BT	Exit time	TAT	WT
1	3	1	7	4	3
2	1	4	16	15	11
3	4	2	9	5	3
4	0	6	6	6	0
5	2	3	12	10	7.

SJF

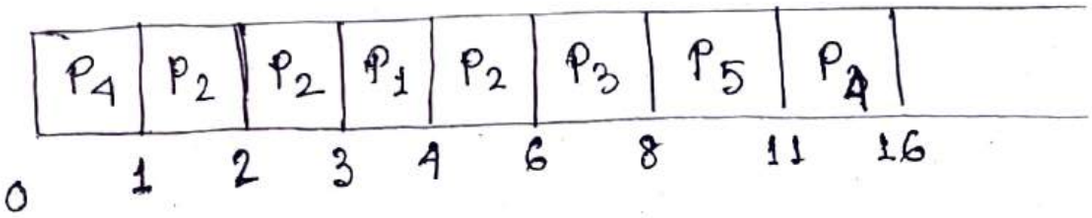


eg

PID	AT	BT	ET	TAT	WT
1	3	1	4	1	0
2	1	4	6	5	1
3	4	2	8	4	2
4	0	6	16	16	10
5	2	3	11	9	6

SRTF

preemptive SJF



Implementation.

Practically can't be implemented, but theoretically can be implemented.

Min heap can be used, where root element contains the least burst time proces

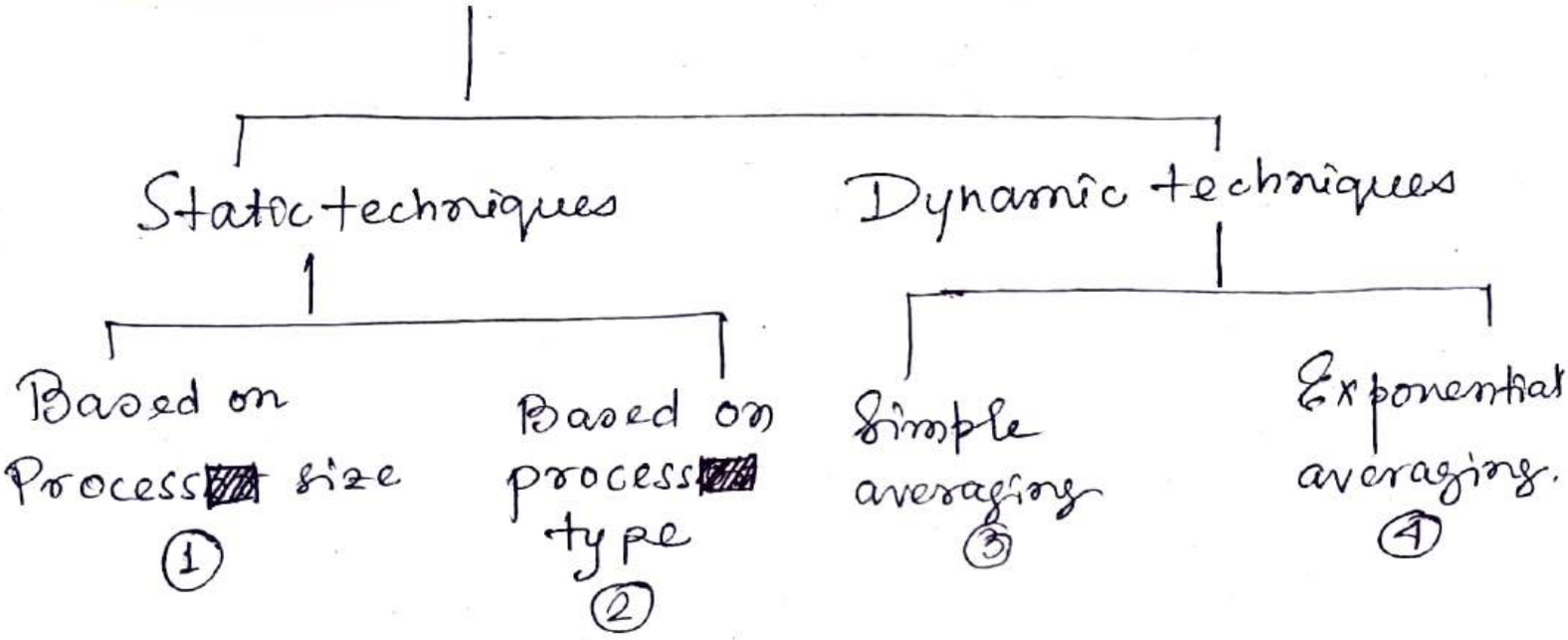
Time complexity for n processes

$$n \times (\log n + \log n) \equiv n \log n$$

adding element on min heap

deleting element from min heap

* Techniques to predict burst time.

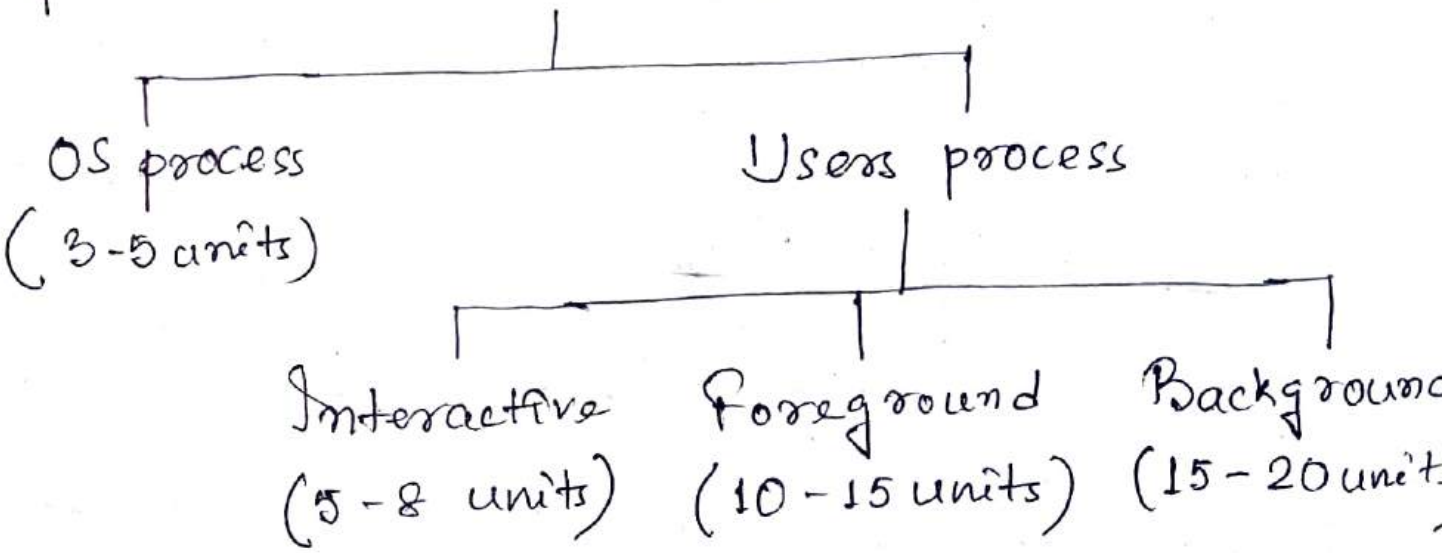


① Process size : This technique predicts the burst time for a process based on its size. Burst time of the already executed process of similar size is taken as the burst time for the process to be executed.

Predicted burst time may not always be right. This is because the burst time of a process also depends on what kind of a process it is.

(2) Process type:

Predicts the burst time for a process based on its type.



(3) Simple averaging

Given n processes of burst time of each process P_i is t_i, then predicted burst time for process P_{n+1} is

$$T_{n+1} = \frac{1}{n} \sum_{i=1}^{n} t_i$$

avg of all burst times till now.

(4) Exponential averaging

$$T_{n+1} = \alpha t_n + (1-\alpha) T_n$$

α – Smoothening factor $(0 \leq \alpha \leq 1)$

t_n – actual burst time of process P_n

T_n – predicted burst time of P_n

eg. $T_5 = ?$ | $t_1 = 4$, $t_2 = 8$, $t_3 = 6$, $t_4 = 7$ | $T_1 = 10$, $\alpha = 0.5$

$$T_2 = 0.5 \times 4 + 0.5 \times 10 = 7$$

$$T_3 = 0.5 \times 8 + 0.5 \times 7 = 7.5$$

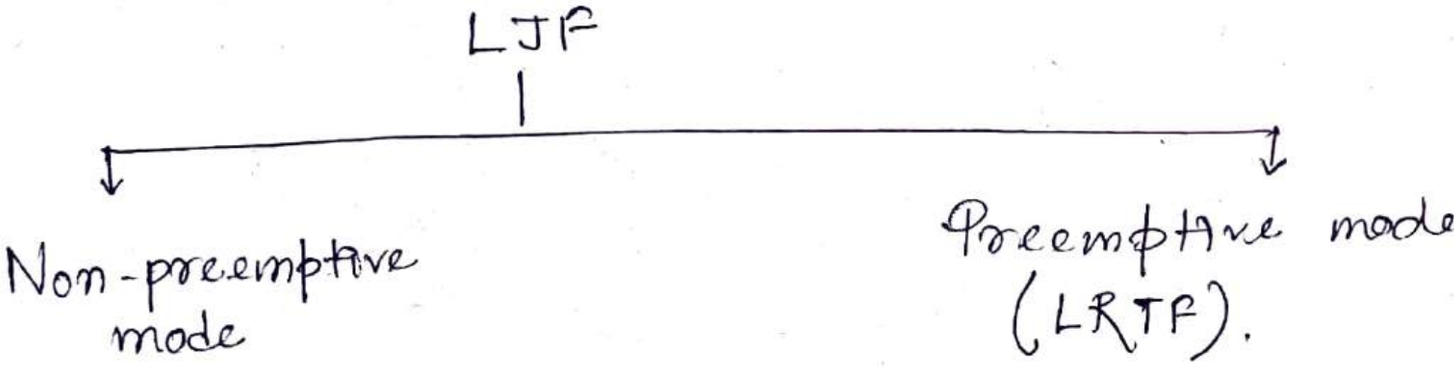
$$T_4 = 0.5 \times 6 + 0.5 \times 7.5 = 6.75$$

$$T_5 = 0.5 \times 7 + 0.5 \times 6.75 = 6.875$$ (Ans)

* LJF, LRTF Scheduling.

LJF - CPU is assigned to the processes, having largest burst time.

In case of a tie, it is broken by FCFS.



Adv.: No process can complete until the longest job also reaches its completion

All the processes approx. finishes at same time.

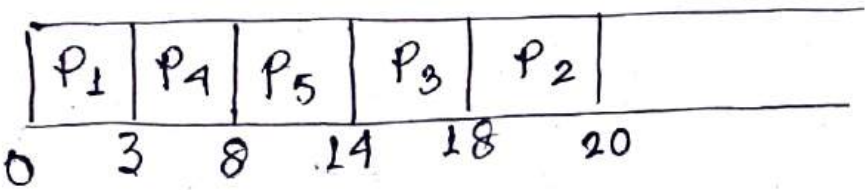
Disadv. : The WT is high.

Processes with smaller BT may starve for CPU.

eg.

LJF Non-preemptive

PID	AT	BT	ET	TAT	WT
1	0	3	3	3	0
2	1	2	20	19	17
3	2	4	18	16	12
4	3	5	8	5	0
5	4	6	14	10	4.

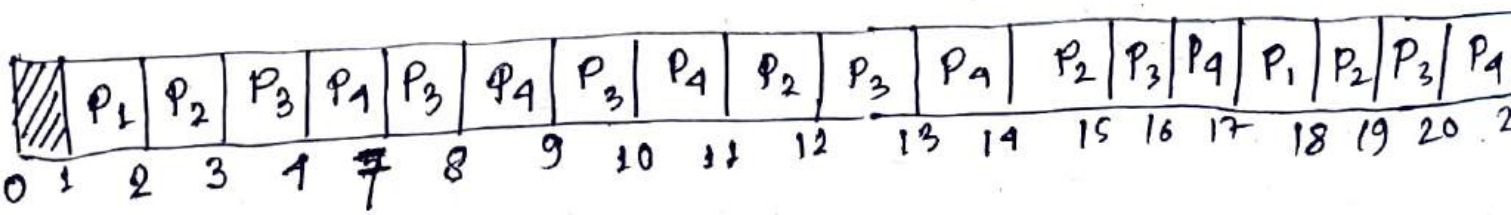


** eg.

LJF preemptive.

LRTF

PID	AT	BT	ET	TAT	WT
1	1	7 1	18	17	15
2	2	4 3 2 1	19	17	13
3	3	6 5 4 3 2 1	20	17	11
4	4	8 7 6 5 4 3 2 1	21	17	9.



* HRRN Scheduling.

$$\text{Response ratio} = \frac{WT + BT}{BT}$$

CPU is assigned to the process having highest response ratio. In case of tie, it is broken by FCFS. Operates only in non-preemptive mode.

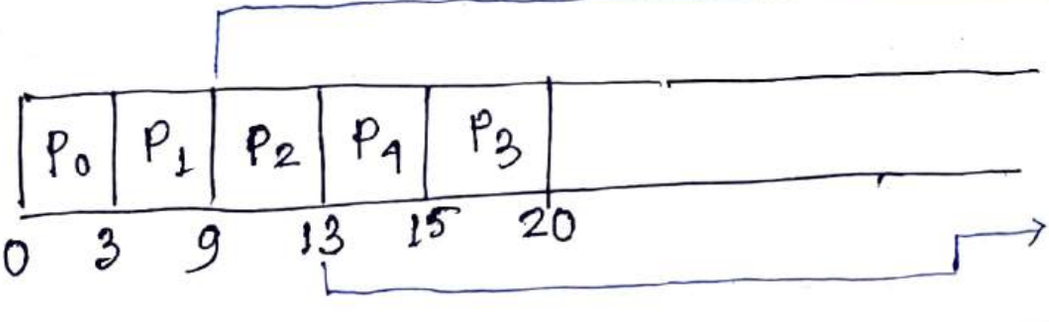
Adv.: It performs better than SJF.
It not only favours the shorter jobs but also limits the waiting time of longer jobs.

Disadv.: Can't be implemented practically, as burst time can't be known in advance.

eg.

PID	AT	BT	ET	TAT	WT
0	0	3	3	3	0
1	2	6	9	7	1
2	1	4	13	9	5
3	6	5	20	14	9
4	8	2	15	7	5

Non-preemptive



$RR_2 = \frac{5+4}{1} = 2.25$

$RR_3 = \frac{3+5}{5} = 1.6$

$RR_4 = \frac{1+2}{2} = 1.5$

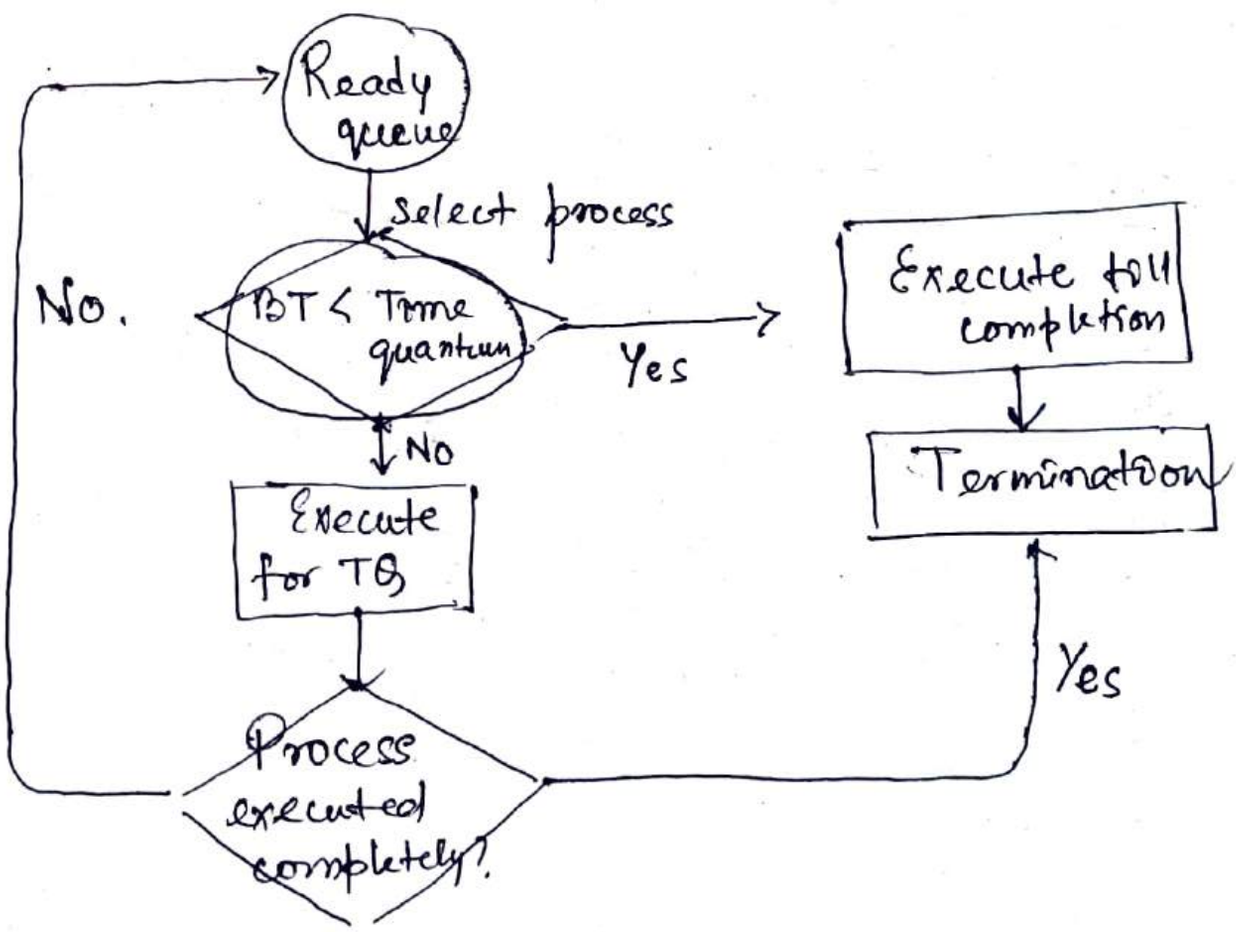
$RR_3 = \frac{7+5}{5} = 2.4$

$RR_4 = \frac{5+2}{2} = 3.5$

* Round Robin Scheduling.

CPU assigned to the process on the basis of FCFS for a fixed amount of time (time quantum). After the time quantum expires, the process is preempted & sent to the ready queue. It's always preemptive in nature.

RR scheduling is FCFS with preemptive mode.



Adv.: It gives the best performance in terms of average response time. It's best suited for time sharing system, client server architecture & interactive system.

Disadv.: It leads to starvation for processes with larger burst time as they have to repeat the cycle many times. Performance depends on TQ. Priorities can't be set for the processes.

→ With decreasing value of TQ,

i) No. of context switch increases

ii) Response time decreases.

iii) Chances of starvation decreases.

Smaller TQ is better in terms of response time.

→ Higher value of TQ is better in terms of no. of context switches.

→ With increasing value of TQ, RR tends to become FCFS.

When TQ → ∞, RR becomes FCFS.

→ Value of TQ should be neither too big nor too small.

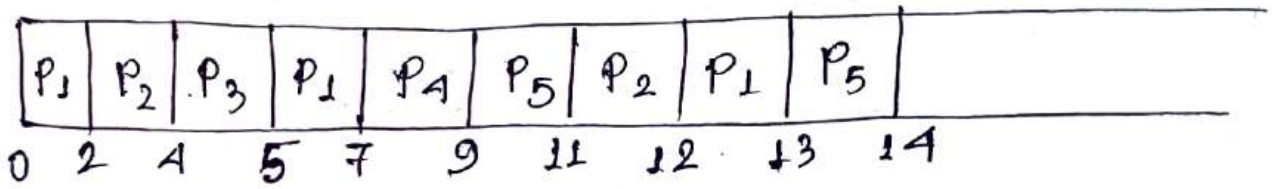
Eg.

PID	AT	BT	ET	TAT	WT
1	0	5 ~~3~~ 1	13	13	8
2	1	~~3~~ 1	12	11	8
3	2	~~1~~	5	3	2
4	3	~~2~~	9	6	4
5	4	~~3~~ 1	14	10	7

TQ = 2

Avg. TAT = $\frac{13+11+3+6+10}{5}$

$= \frac{43}{5} = 8.6$



Ready queue: ~~P1~~ ~~P2~~ ~~P3~~ ~~P1~~ ~~P4~~ ~~P5~~ ~~P2~~ P1 P5

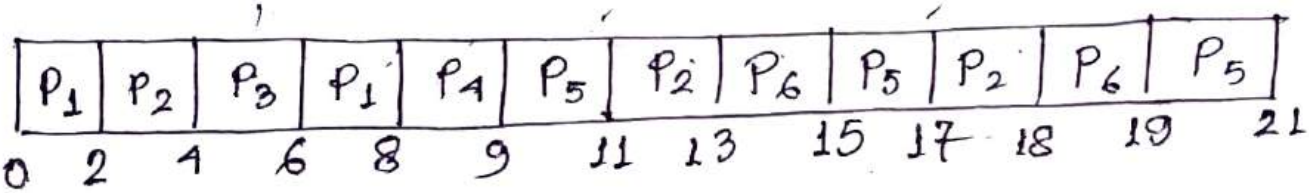
eg.

PID	AT	BT
1	0	4 2
2	1	5 3 1
3	2	2
4	3	1
5	4	6 4 2
6	6	3 1

TQ = 2

FCFS with preemption

In queue, first add processes which has come, then add the concerned process if BT is remaining.



Queue. P1 P2 P3 P1 P4 P5 P2 P6 P5 P2 P6 P5

* Selfish Round Robin Scheduling. Better service to processes that have been executing for a while than to newcomers.

Implementation: i) Processes in ready list are partitioned into NEW & ACCEPTED lists. ii) New processes wait while accepted processes are serviced by RR. iii) Priority of a new process increases at a rate 'a' while the priority of an accepted process increases at a rate 'b'. iv) When the priority of a new process reaches the priority of an accepted process, that new process becomes accepted. v) If all accepted processes finish, the highest priority new process is accepted.

Algorithm: i) No ready processes initially, when 1st process A arrives. It has priority 0 to begin with. Since, there are no other accepted processes, A is accepted immediately. ii) After a while another process B arrives. As long as $b < a$, B's priority will eventually catch up to A's, so it's accepted. Now, both A & B have same priority. iii) All accepted processes share a common priority (which rises at rate b); that makes this policy easy to implement i.e. any new process's priority is bound to get accepted at some point. So, no process has to experience starvation. iv) Even if $b > a$, A will eventually finish & then B can be accepted.

Adjusting a & b:

- if $b/a \geq 1$, a new process isn't accepted until all the accepted processes have finished
  SRR becomes FCFS.
- if $b/a = 0$ all processes accepted immediately
  SRR becomes RR
- if $0 < b/a < 1$ accepted processes are selfish, but not completely.

* Priority based scheduling: (Preemptive & non-preemptive)

CPU is assigned to the process having highest priority. In case of a tie, it's broken by FCFS.

Adv.: Considers priority of the processes & allows the important processes to run first. (Priority scheduling in preemptive mode is best suited for real time OS.)

Disadv.: Processes with lesser priority may starve. There's no idea of response time & waiting time.

→ Waiting time for the process having the highest priority will always be zero in preemptive mode, & may not be zero in non-preemptive mode.

→ Priority scheduling in preemptive & non-preemptive mode behaves exactly same under following conditions:

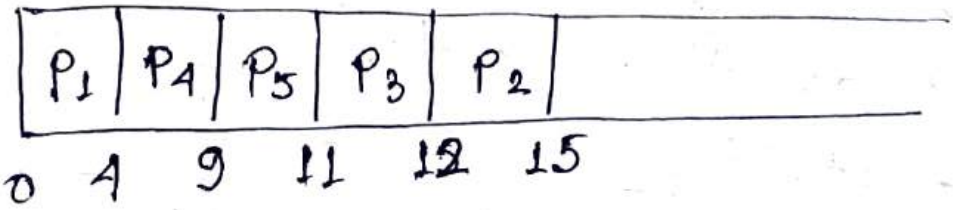
i) arrival time of all processes same,

ii) all processes become available.

eg.

non-preemptive

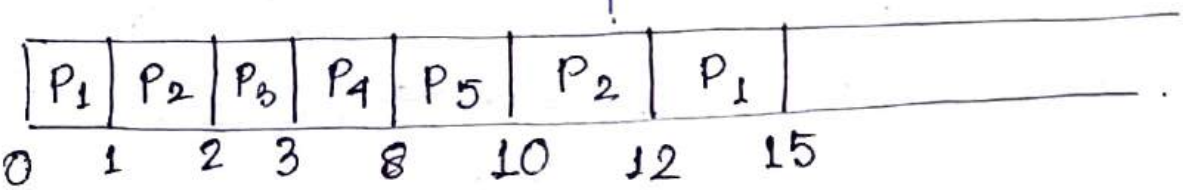
PID	AT	BT	Prio
1	0	4	2
2	1	3	3
3	2	1	4
4	3	5	5
5	4	2	5



eg.

	PID	AT	BT	Prio
preemptive	1	0	13	2
	2	1	32	3
	3	2	1	4
	4	3	5	5
	5	4	2	5

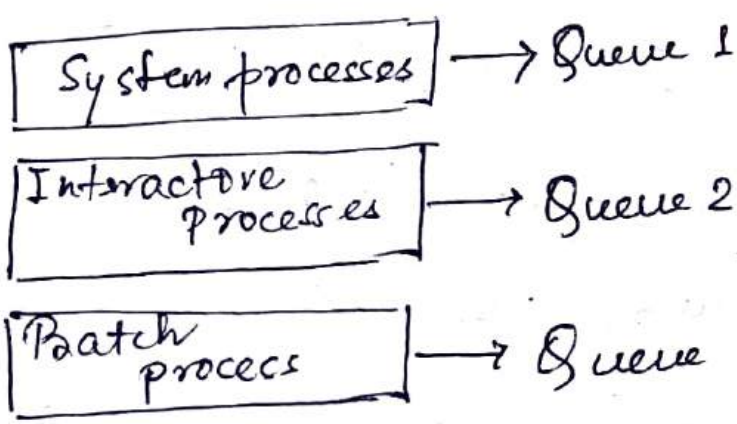
* Each queue gets a certain portion of CPU time & it can then schedule among its various processes eg. foreground q can be given 80% of CPU time for RR sched, whereas background q receives 20% of CPU to give to its processes on an FCFS basis.



* Multilevel Queue Scheduling:

Ready queue divided into separate queues for each class of processes (eg. foreground (interactive) & background processes (batch))

High priority



Low priority

Each queue has its own scheduling algorithm.

Different sched. needs. Foreground processes may have priority (externally defined) over background processes.

Each queue has own sched. algo. Processes permanently assigned to one queue. Foreground queue – RR sched; Background – FCFS. There must be scheduling among queues (fixed prio. preemptive)
- Time slice among queues: → *

→ For scheduling among the queues –

i) Fixed priority preemptive scheduling method.

ii) Time Slicing.

Each queue has absolute priority over lower priority queue.

MQS has low sched. overhead but it's inflexible.

Each queue gets certain portion of CPU time & can use it to schedule its own processes.

eg.

PID	AT	BT	Q No.
1	0	4	1
2	0	3	1
3	0	8	2
4	10	5	1

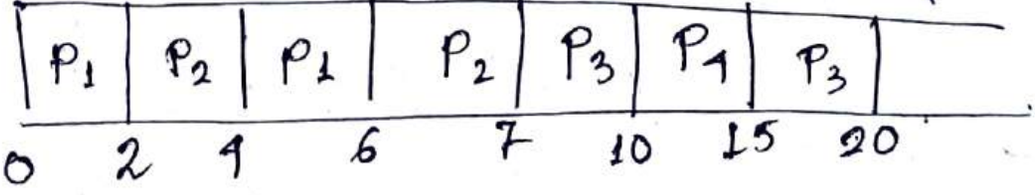
Queue 1 has higher priority.

ref. gfg

Queue 1 – RR (TQ = 2)

Queue 2 – FCFS.

because of fixed priority preemption among queues.



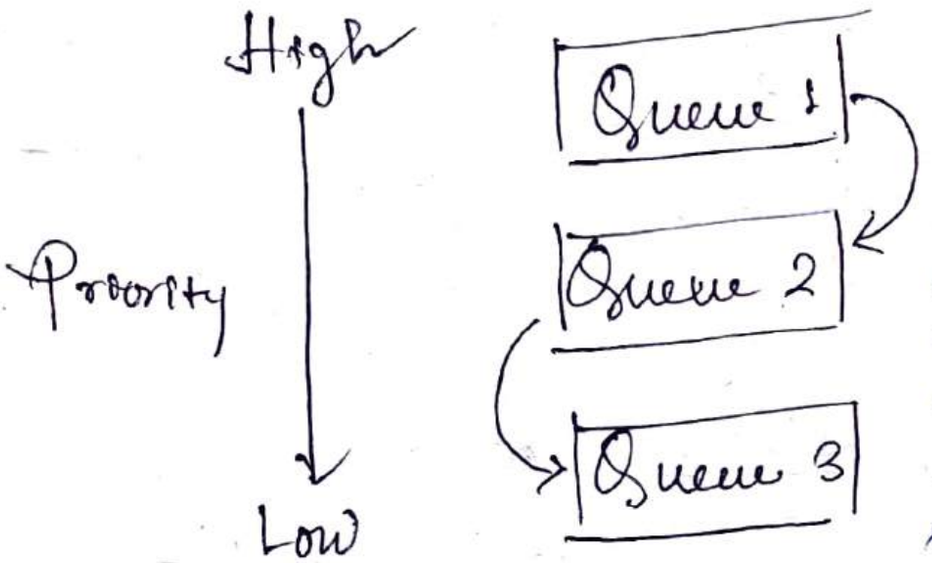
* Multilevel Feedback Queue Scheduling.

(MLFQ).

Process can move between queues.

MLFQ keeps analysing the behaviour of processes & according to which it changes its priority.

ref. gfg



If a process uses too much CPU time, it will be moved to a lower-prio queue. This scheme leaves I/O & interactive processes in the higher prio queues.

- MFQS defined by parameters -
  i) No. of queues
  ii) Sched. algo for each queue
  iii) Methods to determine when to upgrade/demote a process to higher/lower prio queue.

Process that waits for too long in a lower prio queue may be moved to a higher prio queue. This form of aging prevents starvation.

- Most general CPU scheduling algo.
- Can be configured to match a specific system under design.
- Most complex algo too.

* Starvation.

Indefinite blocking on which a process ready to run for CPU can wait indefinitely because of low priority (or some other reasons).

Solution to starvation:

Aging: Gradually increasing the priority of processes that wait on the system for a long time.

→ When deadlock occurs no process can make progress, while in starvation apart from the victim process other processes can progress or proceed.

* Thread :

A thread is a path of execution within a process. A process can contain multiple threads.

→ Multi threading

Parallelism by dividing process into threads.

→ Process vs Thread.

Threads within the same process run in a shared memory space, while processes run in separate memory spaces. Threads are not independent of one another like processes are.

Like process, a thread has its own program counter, register set, & stack space.

Advantages of thread over process. ~

gfg

i) Responsiveness

ii) Faster context switch

iii) Effective utilisation of multiprocessor systems.

iv) Resource sharing — Economy

v) Communication.

vi) Enhanced throughput of system.

Types of threads:

User level thread,

Kernel level thread.

Similarity between threads & processes

Only one thread or process is active at a time. Within process both execute sequentially. Both can create children.

(quasi-parallel)

Differences between threads & processes

Threads are not independent, processes are, Threads are designed to assist each other, processes may or may not do it.

User level thread (ULT)

Implemented on the user level library, not created using sys. calls. Kernel doesn't know about the ULT & manages them as if they were single threaded processes.

Adv.: Can be implemented on an OS that doesn't support multithreading.
Simple representation.
Simple to create.
Thread switching is fast.

Disadv.: No or less coordination among threads & kernel.
If one thread causes a page fault, the entire process blocks.

Kernel Level Thread (KLT).

Kernel knows & manages the threads. Instead of thread table in each process, the kernel itself has thread table (a master one) that keeps track of all the threads in the system. OS kernel provides sys. call to create & manage threads.

Adv.: Since kernel has full knowledge about the threads, scheduler may decide to give more time to processes having large no. of threads.

Good for application that frequently block.

Disadv.: Slow & inefficient. It requires thread control block so it's an overhead.

* Microkernel.

The user services & kernel services are implemented in different address space. The user services are kept in user address space & kernel services in kernel address space, thus also reduces the size of kernel & size of OS as well.

μkernel is responsible for-

i) Inter process comm"
ii) Memory management
iii) CPU scheduling

Adv.: i) Architecture of this kernel is small & isolated hence it can function better.

ii) Expansion of system is easier, it's simply added in the system application without disturbing the kernel.

eg. Eclipse IDE.

*Layered architecture - Groups components that perform similar functions into layers. Each layer communicated only with layers immediately above & below it. Processes' requests might pass through many layers before completion.

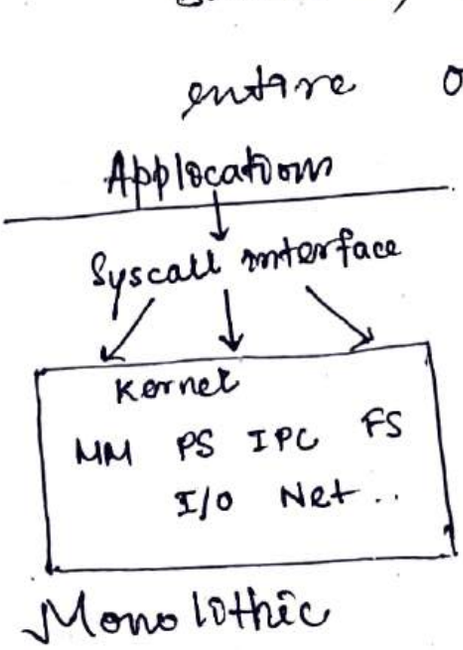
* Monolithic Kernel (Linux, BSD, Solaris)

Manages system resources between application & hardware, but user services & kernel services are implemented under same address space. It increases the size of the kernel, thus increases size of OS as well.

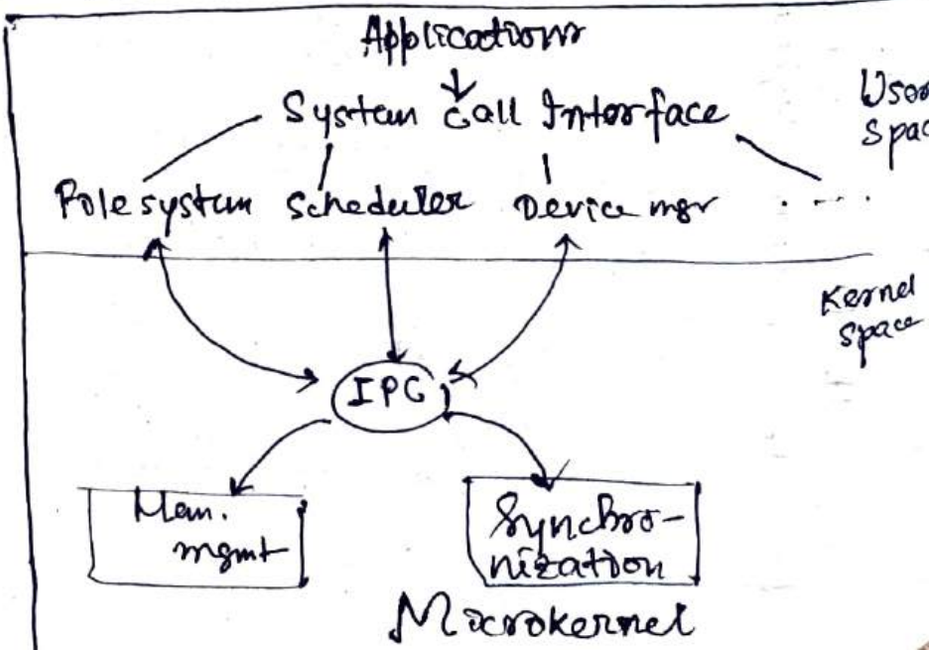
Adv.: It provides CPU scheduling, memory mgmt., file mgmt.; it's a single static binary file.

Disadv.: If any one service fails it leads to entire system failure.

If user has to add any new service, user needs to modify entire OS.



Monolithic



Microkernel

* fork() in C.

System call to create a new process (child process) & it runs concurrently with process (which called system call fork), parent process. After a new process (child) created, both processes will execute the next instn following the fork() sys. call. A child process uses the same PC, same CPU registers, same open files which use in the parent process. It takes no parameters & returns an integer value.

Return values by fork().

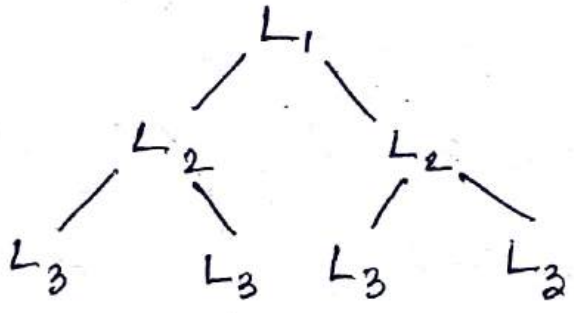
i) -ve value: Creation of a child process was unsuccessful.

ii) Zero: Returned to the newly created child process.

iii) +ve value: Returned to parent or caller. Value contains PID of newly created child process

Total no. of processes = 2^n where n is number of fork sys. calls.

```
fork();
fork();
fork();
```

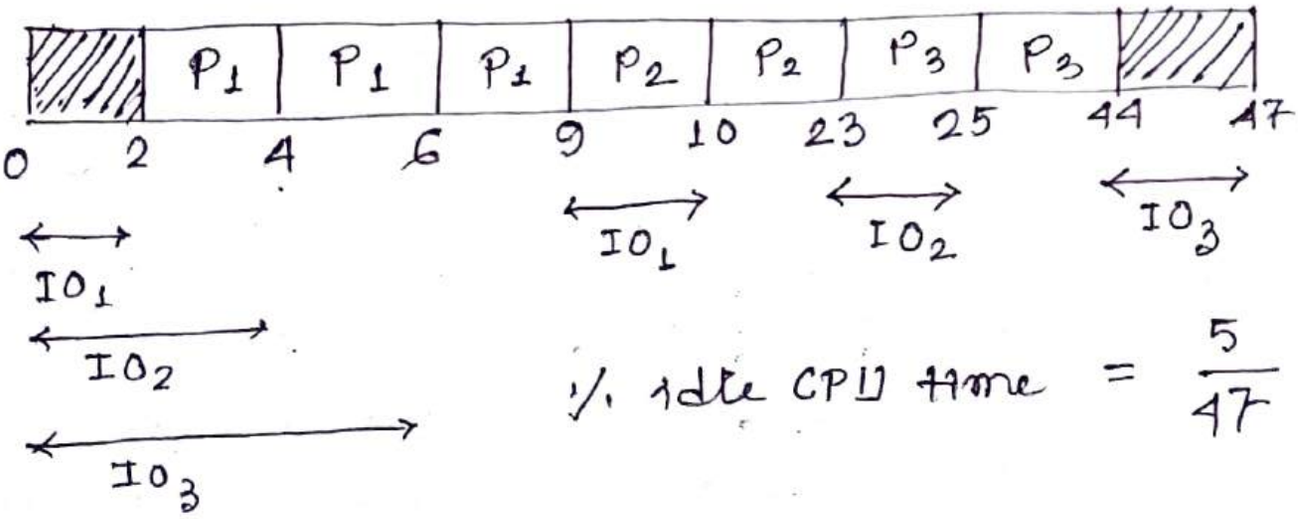


CPU scheduling problems.

eg. ***

PID	Total BT	I/O BT	CPU BT	I/O BT.
P_1	10	2	7	1
P_2	20	4	14	2
P_3	30	6	21	3.

SRTF.



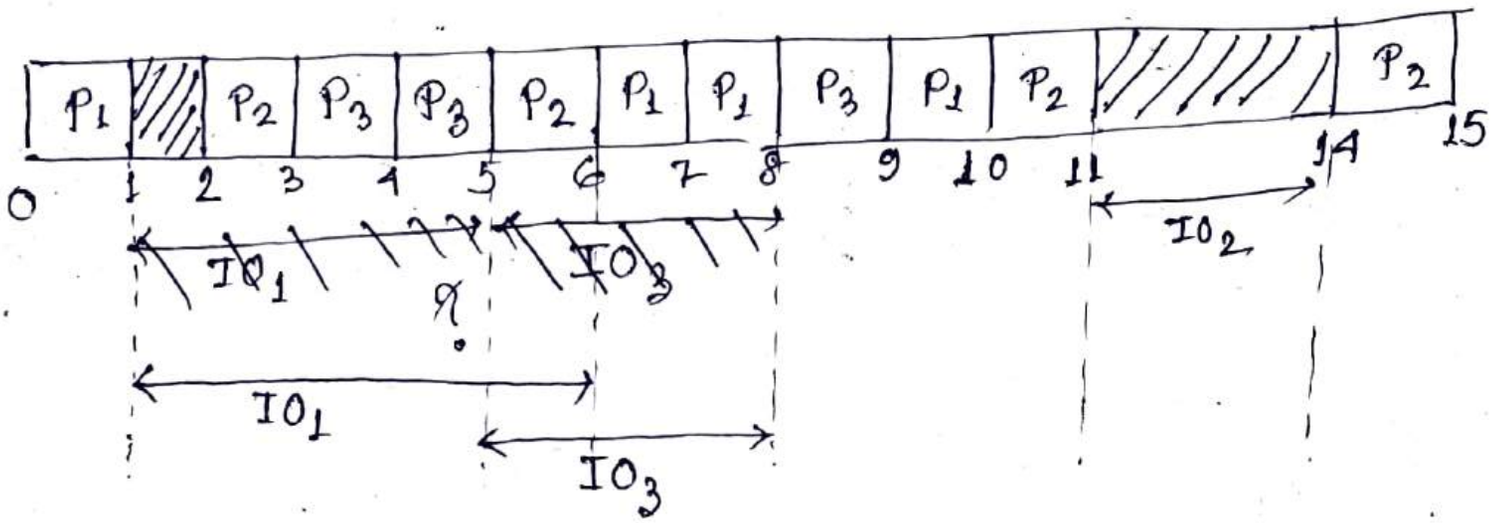
% idle CPU time = $\frac{5}{47} \times 100\%$

eg. ***

Imp.

Priority Scheduling

PID	AT	Prio	BT CPU	BT I/O	BT CPU
1	0	2	1	5	3
2	2	3	3	3	1
3	3	1	2	3	1



Process Synchronisation.

* Process synchronisation controls the execution of processes running concurrently to ensure that consistent results are produced.

Need of sync.:
When multiple processes execute concurrently sharing some system resources.

* On the basis of synchronisation, processes are of 2 types:

i) Independent process.: Execution of one doesn't affect the execⁿ of other.

ii) Cooperative process: Execution of one affects other's execⁿ.

* Critical Section: Critical section is a section of the program code where a process access the shared resources during its execⁿ. It can be accessed by only one process at a time. CS contains shared variables that need to be synchronised to maintain consistency of data variables. (CS needs to be executed atomically).

eg: Inconsistent results without sync.

```
P1 {                          Count          P2 {
    ----                        5                ----
    count++                                      count--
    ------                                       ------
}                                            }
```

In assembly language,

```
P1 {                                    P2 {
    ---                                     ----
1   MOV count, R0                       1   MOV count, R0'
2   Inc R0                              2   DEC R0'
3   MOV R0, count                       3   MOV R0', count
    ----                                    ----
}                                       }
```

case 1

$P_1(1), P_1(2), P_1(3), P_2(1), P_2(2), P_2(3)$

- exec^n order

Final value of count = 5.

case 2

$P_2(1), P_2(2), P_2(3), P_1(1), P_1(2), P_1(3)$

count = 5

case 3

$P_1(1), P_2(1), P_2(2), P_2(3), P_1(2), P_1(3)$

count = 6

case 4

$P_2(1), P_1(1), P_1(2), P_1(3), P_2(2), P_2(3)$

count = 4.

* Race Condition.

Situation where several processes access & process manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place.

final outcome depends on the exec^n order of ins^{ns} of different processes.

eg.

```
P1 () {                 P2 () {
1  C = B - 1;           1  D = 2 × B;
2  B = 2 × C;           2  B = D - 1;
}                       }
```

$B_2 = 2$

→ Case 1.

$P_1(1), P_1(2), P_2(1), P_2(2)$ $\quad B_{final} = 3$

case 2

$P_2(1), P_2(2), P_1(1), P_1(2)$ $\quad B_f = 4$

case 3

$P_1(1), P_2(1), P_2(2), P_1(2)$ $\quad B_f = 2$

case 4

$P_2(1), P_1(1), P_1(2), P_2(2)$ $\quad B_f = 3$

case 5

$P_1(1), P_2(1), P_1(2), P_2(2)$ $\quad B_f = 3$

Case 6

$P_2(1), P_1(1), P_2(2), P_1(2)$ $\quad B_f = 2$

* Critical Section Problem.

Inconsistent results produced ### when multiple processes access critical section concurrently.

* Synchronisation Mechanism.

Allows the processes to access CS in a synchronised manner to avoid inconsistent results.

For every CS in the program, a sync mechanism adds -
an entry section before the CS,
an exit section after the CS.

→ Entry Section:

Acts as a gateway for a process to enter inside the CS. It ensures that only one process is present inside the CS at any time. It doesn't allow any other process to enter inside the CS if one process is already present inside it.

```
Process {
  Non CS
  Entry
    CS
  Exit
  Non CS
}
```

→ Exit Section: It acts as an exit gate to leave the CS. When a process exits from the CS, some changes are made so that other processes can enter inside the CS.

* Criteria for Sync. Mechanisms.

1. Mutual Exclusion. - Processes must access the CS in a mutually exclusive manner. Only one process is present inside the CS at any time.

2. Progress - Mechanism must ensure an entry of a process inside the CS is not dependent on the entry of another. A process can freely enter the CS if there's no other process present inside it. A process enters the CS only if it wants to enter. A process is not forced to enter inside.

3. Bounded Waiting - The wait of a process to enter CS is bounded. It should not wait indefinitely.

4. Architectural Neutrality - It can run on any architectu without any problem. There's no dependenc

---

Progress - If some process P_i does not want to access its CS, then it will not stop some other process P_j to access its CS.
(In case of deadlock, there is no progress)

→ Mutual exclusion & progress are the mandatory criteria. Bounded waiting & architectural neutrality are optional criteria.

* Interprocess Communication (IPC)

gfg

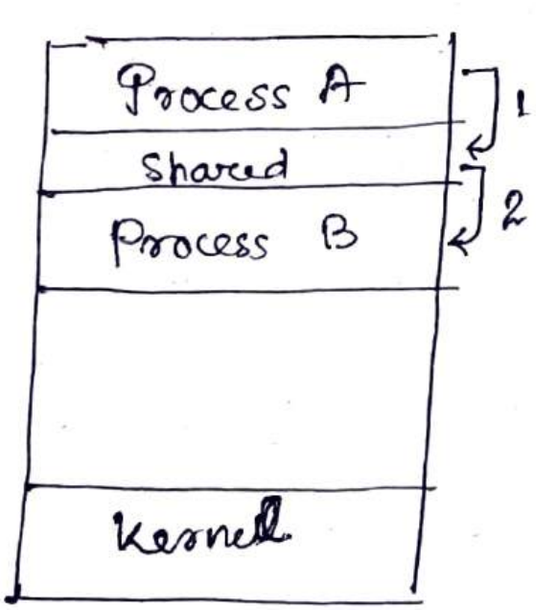
Two models -

1. Shared-memory

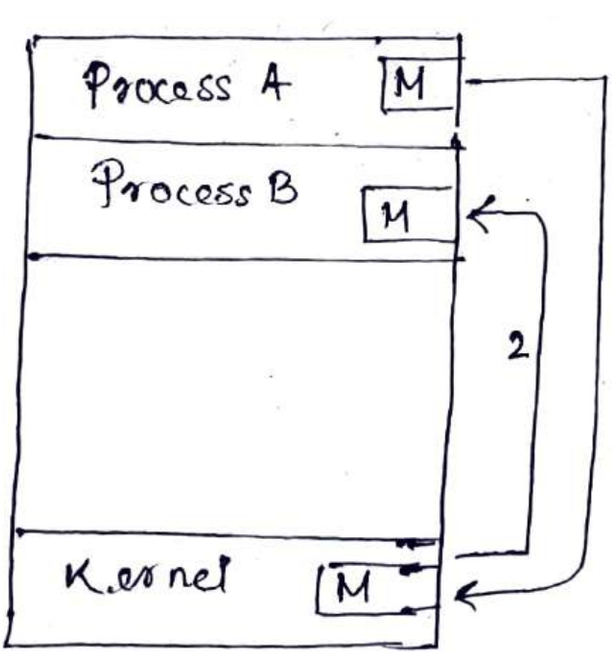
A region of memory that's shared by cooperating processes. Processes can exchange information by reading & writing data to the shared region.

2. Message passing

Communication by means of message exchanged between the cooperating processes.



Shared memory



Message passing

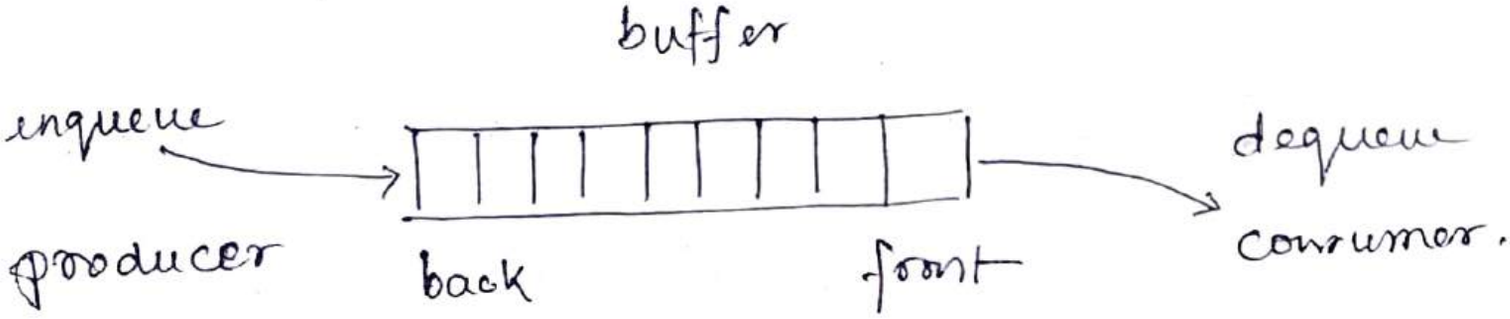
* Producer - Consumer Problem.

(Bounded buffer problem)

Producer, consumer have a fixed size buffer which they share. Producer produces items or data to put on buffer. Consumer works to remove data from buffer & consume it. Producer should not produce data when buffer is full & consumer should not remove data when buffer is empty (obvio!).

Producer should go to sleep when buffer is full. Next time when consumer removes data it notifies the producer & producer starts producing data again. Consumer should go to sleep when buffer is empty. Next time when producer add data it notifies the consumer & consumer starts consumption.

This soln can be achieved using semaphores (variables on which read, modify & update happens atomically in kernel mode (no preemption)).



* IPC Methods.

1. Pipes (same process): Flow of data in one direction only. Must have common origin

2. Names pipes (different processes): Pipe with specific name. It can be used where processes don't have a shared common process origin. eg. FIFO.

3. Message queuing: Messages pass between processes using either a single queue or several message queues

4. Semaphores: To avoid race condition.

5. Shared memory

6. Sockets: Used to communicate over a network between a client & a server.

* Lock variable sync. mechanism.

Uses a lock variable to provide synchronisation among the concurrent processes; it's a software mechanism implemented in user mode, i.e no support required from the OS. It is a busy waiting soln (keeping CPU busy even when it's technically waiting). Can be used for more than two processes.

→ Implementation:

```
While (lock != 0);      Entry
    lock = 1
-------------------
    CS
-------------------
    Lock = 0            Exit
```

lock value = 0 means the CS is currently vacant & no process is present inside it.
Lock = 1 means the CS is occupied & a process is present inside it.

→ Working:

Scene 1. P_0 arrives → executes lock != 0 (since lock = 0 initially, while breaks)
→ P_0 sets lock = 1 & enters CS. →
Now, even if P_0 gets preempted in the middle, no process can enter CS. → P_0 terminates & sets lock = 0 → Other process gets chance

Scene 2. P_1 arrives. → executes lock != 0 instr (since lock = 1, while loop starts) → Loop keeps P_1 busy until the lock value becomes 0 & loop breaks.

Scene 3. P_0 comes out of the CS, & sets lock = 0 → while loop of P_1 breaks → P_1 sets lock = 1 & enters CS → No process can enter CS → Other process can enter only after P_1 completes & sets lock = 0.

→ Failure:

Scene 1. P_0 arrives → lock = 0, so P_0 ~~gets into CS~~. breaks the while loop → P_0 gets preempted before it sets lock = 1.

Scene 2. P_1 arrives → lock is still 0, so lock = 1, P_1 gets into CS & gets preempted in the middle of CS.

Scene 3. P_0 gets scheduled again → P_0 sets lock value 1 & gets into CS.

Then P_0 & P_1 get to present inside the CS at the same time.

Similarly, if there are n processes, then all of them may be present inside the CS at the same time. This happens when each process gets preempted immediately after breaking the while loop.

Characteristics.

i) Busy waiting soln

ii) Does not fulfill mutual exclusion, progress,

iii) Can be used for any no. of processes.

iv) Never used.

* Test & Set Sync. Mechanism.

Uses test & set mech to provide the synchronisa among processes.

Test & Set mech:

Returns the old value of a memory location & sets the memory location value to 1 as a single atomic operatio If one process is currently executing a test & se no other process is allowed to begin another test & set until the 1st is finished

Implementation :

```
While (test_&_set (Lock));     Entry
        CS.
    Lock = 0.                  Exit.
```

Initially lock is set to 0.

Lock = 0 means CS is currently vacant & no process is present inside.

Lock = 1 means CS is occupied & a process is present inside it.

Working :

Scene 1 P_0 arrives → returns 0 to while & set lock to 1 → P_0 enters CS → no process can enter CS until P_0 gets terminated.

Scene 2 P_1 arrives → as lock = 1 P_1 can't break while loop & trapped in an infinite loop.

Scene 3 P_0 comes out & sets lock = 0 → P_1 sets lock = 1 → While loop breaks → P_1 enters CS & gets executed.

Characteristics.

i) Ensures mutual exclusion.

ii) Deadlock free

iii) Doesn't guarantee bounded waiting & cause starvation.

iv) Suffers from spin lock.

v) Not architecture neutral since it requires the OS to support test & set instn.

vi) It's a busy waiting soln.

→ In the test & set instn the two operations (returning old value of lock & updating lock to 1) are simultaneous (acts as an atomic operation). It ensures mutual exclusion.

→ Other processes can enter CS only after the process that has begun the test & set finishes & set lock to 0.

→ There might exist an unlucky process which when arrives to execute the cs finds it busy. So, it keeps waiting in the while loop & eventualy gets preempted. When it gets rescheduled & comes to execute the cs, it finds another process executing the cs. This may happen several times leading the process to starve.

→ Cause of spinlock.

Assuming priority scheduling algorithm.

Higher prio. process comes to execute the cs → sync. mechanism doesn't allow it to enter the cs before lower prio. process ends. → lower prio process can't be executed before the higher prio process completes → execution of both is blocked.

(Spinlocks don't cause preemption).

Busy waiting | Spinlock is a sol^{n} for threads, so that two threads can't access same resource at a time.

* Turn Variable Sync. Mechanism.

Provides sync. among 2 processes, & uses a turn variable to provide synchronisation.

Implementation :

Process P_0

```
while (turn != 0);
    CS
turn = 1 ;
```

Process P_1

```
while (turn != 1);
    CS
turn = 0.;
```

Initially, turn value is set to 0.

Turn value = 0 means it is the turn of process P_0 to enter CS.

Turn value = 1 means it is the turn of process P_1 to enter CS.

Working :

Scene 1 P_0 arrives → breaks while loop (as turn = 0) → enters CS → P_1 can't enter until P_0 gets terminated and set turn to 1.

Scene 2

P_0 comes out of CS & sets turn = 1 → While loop of P_1 breaks and P_1 enters CS & gets executed → P_1 sets turn to 0 P_0 can't enter until P_1 ends & sets turn to 0.

Characteristics.

i) It ensures mutual exclusion.

ii) Follows strict alternation approach.
(Processes have to compulsorily enter the CS alternatively whether they want it or not)

iii) It does not guarantee progress since one's execution is dependent on another. One process can't get a chance, until other gets executed.

iv) It ensures bounded waiting since processors are executed turn wise one by one & each process is guaranteed to get a chance.

v) It ensures prevention from starvation.

vi) It's deadlock free.

vii) It's architecture neutral.

viii) It's a busy waiting solution that keeps the CPU busy when the process is actually waiting.

* Interest Variable Sync. Mechanism.

Sync. mechanism that provides sync. among two processes, using an interest variable.

Implementation.

Process P_0.

```
interest[0] = True;
while (interest[1] == True);

  CS

interest[0] = False;
```

Process P_1

```
interest[1] = True;
while (interest[0] == True);
        CS

interest[1] = False
```

interest[0] = False

P_0 is not interested to enter CS

& so on.

All interests are initialised to false.

Characteristics.

i) It ensures mutual exclusion.

ii) It does not follow strict alternation.

iii) It ensures progress since if a process is not interested to enter the CS, it never stops the other process to enter the

iv) It's architecture neutral since it does not require any support from the OS.

v) It is a busy waiting soln.

vi) It suffers from deadlock.

vii) It does not guarantee bounded waiting.

Suffering deadlock

1. P_0 arrives → interest[0] = True → P_0 gets preempted & P_1 gets scheduled.
2. P_1 arrives → interest[1] = True → P_1 gets preempted.
3. P_0 gets rescheduled → P_0 can't break while loop → P_0 keeps waiting indefinitely.
4. P_1 gets rescheduled → P_1 can't break loop → P_1 keeps waiting.

Both P_0, P_1 deadlocked.

* Semaphores – Sync. Mechanism.

It's a simple integer variable, used to provide sync. among multiple processes running concurrently.

Semaphores

- Counting semaphore
- Binary semaphore or mutex.

* → Counting Semaphore implementation.

```
Struct semaphore {
        int value;
        Queue type L; }
Wait (semaphore S) {
        S.value = S.value - 1;
        if ( S.value < 0 ) {
            put process (PCB) in L;
            sleep (); }
        else return; }
Signal (Semaphore S) {
        S.value = S.value + 1;
        if (S.value <= 0) {
            select a process (PCB) from L;
            wake up (); }
}
```

no problem of busy wait

Wait	signal
P	V
Down	Up

add process to queue.

Process calls wait before entering CS and executes sign after exit.

remove process fro waiting queue.

→ Working: Counting semaphore has 2 components: an integer value, an associated waiting list (queue).

Value of counting semaphore being +ve indicates the no. of processes that can be present in the CS at the same time.
-ve value of counting semaphore indicates the no. of processes that are blocked in the waiting list.

Waiting list of counting semaphore contains the processes that got blocked when trying to enter the CS. In waiting list, the blocked processes are put to sleep.
Using a queue in waiting list ensures bounded waiting.

Wait operation is executed when one process tries to enter the CS. Wait decrements the value of counting semaphore by 1. Now, if semaphore value >= 0, process is allowed to enter CS. If semaphore value < 0, process is put to sleep in waiting list. (making the queue of waiting or blocked processes).

Signal operation takes place when a process takes exit from the CS. It increments the value of counting semaphore by 1. Now, if semaphore <= 0, a process

is chosen from the queue & it is woken up to execute. If semaphore > 0, no action is taken.

By adjusting the value of counting semaphore, no. of processes that can enter the CS can be adjusted. If the value of counting semaphore is initialised to N, then maximum N processes can be present in the CS at any time (multi-instance resource usage).

To implement mutual exclusion, the value of semaphore is initialised to 1

GV

→ Binary Semaphore Implementation

```
Struct semaphore {
    enum value (0,1);
    Queue type L; }
down  wait (semaphore S) {
P       if (S.value == 1)
            S.value = 0
        else {
          put process (PCB) in S.L;
          sleep(); } }
up    signal (semaphore S) {
V       if (S.L is empty)
            S.value = 1
        else {
            select a process (PCB) from S.L;
            wake-up(); }
}
```

S.value = 1 will be successful to enter CS

S.value = 0 block to enter CS.

S.value = 0 ⇒ Process is inside CS, others hold.

S.value = 1 ⇒ CS is available

→ Working: A binary semaphore has 2 components – an integer value (0 or 1), an associated waiting list (queue).

Wait is executed when a process tries to enter the CS. Now, if semaphore = 1, semaphore value is set to 0 and process is allowed to enter CS. If semaphore = 0, process is blocked to enter CS & put to sleep on queue.

Signal is executed when a process exit from the CS. Then, if waiting list is empty, value of binary semaphore is set to 1. If not empty, a process is chosen from waiting list & waked up to execute.

(Binary semaphores are used to implement the order on which processes must execute.)

* Mutex is not binary semaphore.

A mutex is locking mechanism used to synchronise access to a resource. Only one task can acquire the mutex.

Semaphore is a signalling mechanism.

Mutex provides mutual exclusion.
Semaphore is generalised mutex. } gfg

mutex vs semapho

* Producer - Consumer Problem using Semaphores.

Buffer is the critical section.
To solve, we need 2 counting semaphore (Full, Empty). Full keeps track of no. of items in the buffer at any given time & Empty keeps track of no. of unoccupied slots.

initialisation — Full = 0
Empty = n

Soln for producer —

wait signal
down up

```
do {
    #produce item
    wait(empty);
    wait(mutex);
    //place in buffer
    signal(mutex);
    signal(full); } while(True)
```

When producer produces item, value of Empty is decremented by 1. After placing the item in buffer, value of full is increased by 1.

Soln for consumer —

```
do {
    wait (full);
    wait (mutex);
    // remove item from buffer
    signal (mutex);
    signal (empty);
    // consumes item
} while (True)
```

* Peterson's Solution.

S/W mechanism at user mode.
Busy waiting soln. It's a 2 process (P_0, P_1) solution. It uses turn & interest variables.

All criteria of sync. mechanism fulfilled.

```
#define N 2
#define TRUE 1        #define FALSE 0.
int interest[N] = FALSE ;
int turn ;
void entry_section (int process) {
    int other ;
    other = 1 - process ;
    interest [process] = TRUE ;
    turn = process ;
    while ( interest[other] == TRUE && turn = process) ; }
void exit_section (int process) {
    interest [process] = FALSE ;
}
```

Bakery algorithm (Lamport) | mutual exclusion

Before entering CS, P receives a ticket no. Holder of the smallest ticket no. enters the CS. If P_i & P_j receive same no., if $i < j$, P_i is served first, else P_j. Tokens generated in non-decreasing order.

Notation (ticket #, pid #)

- $(a,b) < (c,d)$ if $(a < c)$ or (if $a = c$ & $b < d$.)
- max $(a_0, \ldots, a_{n-1})$ is a no. k, s.t. $k \geq a_i$ for $i = 0, \ldots, n-1$

Shared data

var choosing : array [0...n-1] of boolean — init false
number : array [0...n-1] of integer. — init 0.

Algo

gfg

```
repeat {
    choosing[i] := true;
    number[i]   := max(number[0], ....number[n-1]) +1 ;
    choosing[i] := false;
    for j:=0 to n-1
        do begin
            while choosing[j] do no-op;
            while number[j] != 0
                and (number[j], j) < (number[i], i)
                                    do no-op.
        end;
    [critical section]
    number[i] := 0;
    [remainder]
} until false;
```

Turn = 0/1

Flag [T/F | T/F]
0 1

Peterson's solution

P_0

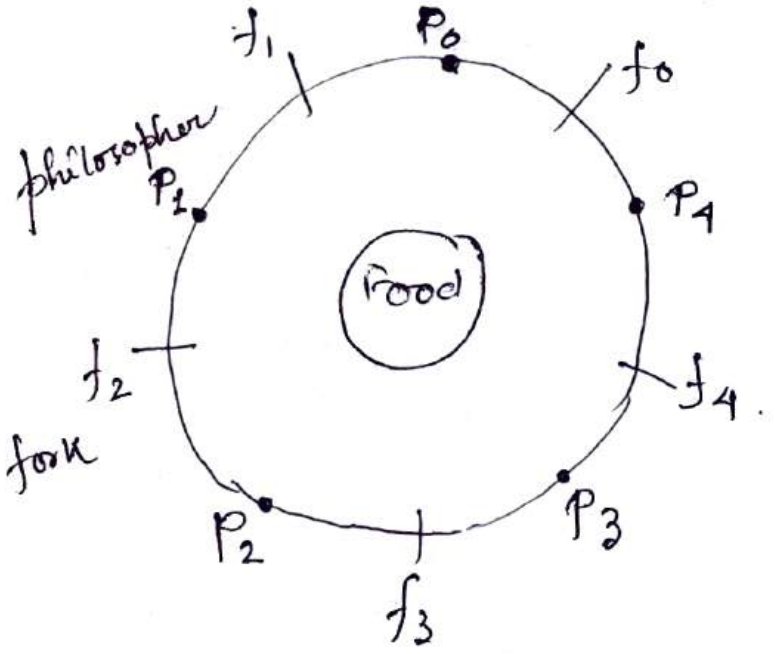
```
while (True) {
    flag[0] = T;
    turn = 1;
    while (turn == 1 &&
           flag[1] == T);
        CS
    flag[0] = F;
}
```

P_1

```
while (True) {
    flag[1] = T;
    turn = 0;
    while (turn == 0 && flag[0] == T);
        CS
    flag[1] = F;
}
```

flag ≡ interest.

* Dining Philosopher's Problem & soln using semaphores.




```
void P() {
  while(True) {
    think();
    pickf(Si);
    pickf((i+1)%N);
    EAT();
    putf(i);
    putf((i+1)%N); }
}
```

pick left fork
pick right fork

P_0 take f_0, then preempts. → P_1 arrives and takes f_1 & f_2 & put them after eating.

Soln using semaphore array S[].

```
void P() {
  while (True) {
    think;
    wait (pickf (Si));
    wait (pickf (S(i+1)%N));
    Eat;
    signal (putf (Si));
    signal (putf (S(i+1)%N)); }
}
```

P_0	S_0	S_1
P_1	S_1	S_2
P_2	S_2	S_3
P_3	S_3	S_4
P_4	S_4	S_0

(gate keepers)

More than one philosophers can eat food (eg. P_0, P_2)

Deadlock if all philosophers decide to eat at same time & all pick up their left chopstick first.

Way to avoid deadlock—

i) Don't allow all philosophers to sit & eat/think at once.

ii) Pick up both forks in a critical section.

iii) Alternate choice of first fork.
P_0 S_1 S_0

* Priority inversion : Task priorities of 3 processes in the order $L < M < H$. M doesn't share the CS. Now, L is running in CS → H also needs to run in CS → H waits for L to come out of CS → M interrupts L & starts running → M runs till completion & relonquishes control → L resumes & starts running till the end of CS → H enters CS & starts running. Neither L or H share CS with M.

Running of M has delayed the running of L & H. H is of higher pro, but H had to wait for M. Prio. of M & H got inverted. This is prio. inversion.

In short,

if the low priority process gets preempted by one or more medium-prio. processes, then the high prio. processes are essentially made to wait for the medium priority processes to finish before the low-prio. process can release the needed resource, causing a prio. inversion.

→ One solution is a priority-inheritance protocol, in which a low-prio process holding a resource for which a high priority process is waiting will temporarily inherit the high priority from the waiting process. This prevents the medium-priority processes from preempting the low prio. processes until it releases the resource, blocking the prio. inversion problem.

Readers - Writers Problem.

gfg

- One set of data is shared among a no. of processes.
- Once a writer is ready, it performs its write. Only one writer at a time.
- If a process is writing, no other can read it.
- If at least one reader is reading, no other can write.
- Readers may not write & only read

Soln when reader has the priority

semaphores : 1. mutex - used to ensure mutual exclusion when readcnt is updated (i.e. when any reader enters or exits from cs). 2. wrt - used by both readers & writers.

int : readcnt - no. of processes performing read in the cs, init. 0.

gfg

Writer process

```
do {
// Writer requests
  wait (wrt);
// performs write
// leaves
  signal (wrt);
} while (true);
```

Reader Process

```
do {
// Reader wants to enter
  wait (mutex);
  readcnt++;
  if (readcnt == 1)
    wait (wrt)
  signal (mutex)
// performs reading
  wait (mutex);
  readcnt--;
  if (readcnt == 0)
    signal (wrt);
  signal (mutex);
} while (true)
```

→ even 1 reader is there, no writer allowed. (reader's priority)

→ other readers can enter

→ reader wants to leave

Writer

1. Writer requests the entry to CS.
2. If allowed i.e. wait() gives a true value, it enters & performs the write. If not allowed, it keeps on waiting.
3. It exits the CS.

reader

1. Reader requests the entry to CS.
2. If allowed,
   - it increments the readcnt. ~~++~~ If this reader is the first reader entering, it locks the wrt to restrict the entry of writers if any reader is inside.
   - It then, signals mutex as any other reader is allowed to enter while others are already reading.
   - After performing reading, it exits the CS. When exiting, it checks if no more reader is inside, it signals wrt as now, writer can enter CS.
3. If not allowed, it keeps on waiting.

* Key terms.

→ Deadlock : A situation in which 2 or more processes are unable to proceed because each is waiting for one of the others to do something.

→ Livelock : A situation in which 2 or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.

→ Starvation : A situation in which a runnable process is overlooked indefinitely by the scheduler; although it's able to proceed, it is never chosen.

→ Spinlock : Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability

*Problems. (GV)

1. Void enter_CS (x)

Test & Set Lock

```
{ while (test_and_set (x)) ;
}
Void leave_cs (x)
{  x = 0; }
```

Realised using test & se

X initialised to 0 & associated with

⟶ It is deadlock free.

2.

Method by P_1	Method by P_2
While (S_1 == S_2);	while (S_1 != S_2);
CS	CS
$S_1 = S_2$	$S_2 = !\, S_1$.

Initial values of S_1 & S_2 random.

→ Check cases for

S_1	S_2
0	0
0	1
1	0
1	1

Mutual exclusion but not progress.

Strict alternation.

3. Counting semaphore S initialised to 10.
Then 6P → 4V performed on S. S_{final} = ?

→ $10 - 6 + 4 = 8$

4. Process

W	X	Y	Z
P(S)	P(S)	P(S)	P(S)
Read (x)	Read (x)	Read (x)	Read (x)
x = x + 1	x = x + 1	x = x - 2	x = x - 2
Write (x)	Write (x)	Write (x)	Write (x)
V(S)	V(S)	V(S)	V(S)

Counting semaphore. S initialised to 2.
x initialised to 0
Max. value possible of x after all complete execution.

→ To obtain maximum value of x, processes must execute in a manner that only the impact of processes X & W remain on value of x & of Y & Z gets lost on value of x.

Strategy

1) Make W read value of x = 0
2) Preempt W
3) Schedule Y & Z one by one.
4) Reschedule W.
5) W increments x to 1 & impact of Y & Z gets lost.
6) Execute X to increment x to 2.

Lost update problem (DB)

So, max. possible value of x is 2.

Again, min. " " of x is -4
(only impact of Y & Z).

5. P_i, coded as follows. $i = 1, 2, \ldots, 9$.

```
repeat {
    P(mutex)
    {CS}
    V(mutex)
} forever.
```

Code for P_{10} is identical except it uses V in place of P. Largest no. of processes that can enter CS at any moment?

→ P_{10} repeat

V(mutex) p −1

{cs} v +1

v (mutex) P_1

forever.

Assume, $mutex_{init} = 1$.

All processes may enter cs at the same time.

6. Let m[0] ... m[4] be binary semaphores & P[0] ... P[4] be processes. Each process P[i] executes the following -

wait (m[i]);

→ P_0 preempts after P(m[0]) wait (m[(i+1) % 4]);

P_1 " " P(m[1]) release (m[i]);

P_2 " " P(m[2]) release (m[(i+1) % 4]);

P_3 " " P(m[3])

P_4 " " P(m[4])

Now, m[0] = ... = m[4] = 0.

Thus, system is in deadlock, since no process can proceed now.

Max. no. of processes inside cs = 2.

7. Binary semaphores S_0, S_1, S_2 init(1,0,0)

P_0	P_1	P_2
While(true)	Wait(S_1);	Wait(S_2);
{	release(S_0);	release(S_0);
wait(S_0);		
print(0);		
release(S_1);		
release(S_2);		
}		

How many times will P_0 print '0'?

Concurrent processes.

→ P_0 arrives → P_0 executes P(S_0) → S_0 is 0 → P_0 prints '0' (1st) → P_0 executes release (S_1, S_2) → $S_1 = S_2 = 1$ → P_0 executes again due to while → As $S_0 = 0$ P_0 gets blocked.

P_1 gets scheduled → P_1 executes P(S_1) → $S_1 = 0$ → P_1 executes V(S_0) → P_0 wakes up as $S_0 = 1$ → P_0 gets scheduled → P_0 executes P(S_0) → $S_0 = 0$ → P_0 prints '0' (2nd)

→ P_0 sets $S_1 = 1$, $S_2 = 1$. → P_0 loops but gets blocked as $S_0 = 0$.

P_2 gets scheduled → P_2 executes wait(S_2) → $S_2 = 0$ → Wakes up the process P_0 after executing V(S_0).

→ P_0 gets rescheduled → P_0 prints '0' (3rd) → P_0 sets $S_1 = S_2 = 1$. → P_0 gets blocked as $S_0 = 0$

P_0 can't execute any more.

✓ Max. no. of times P_0 can print 0 is 3.

✓ Min. no. of times = 2.

↳ [P_0 fully executes (prints first '0')
↓
P_1 executes fully
↓
P_2 executes fully
↓
P_0 gets rescheduled (prints second '0')
↓
P_0 gets blocked]

8. We want to synchronise P & Q, using binary semaphores S & T.

P

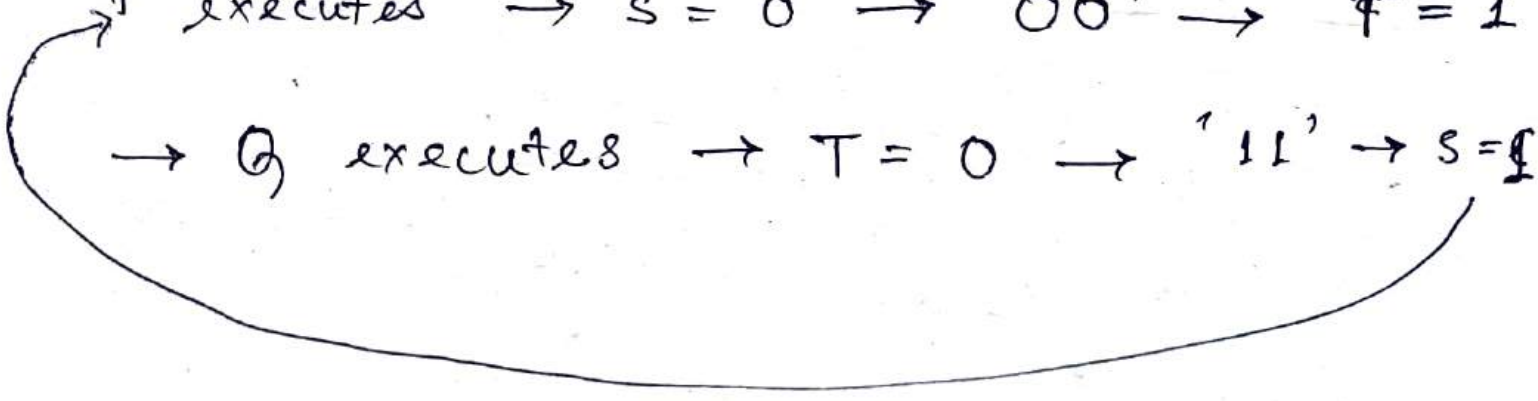
```
while (1) {
    W:
    print '0';
    print '0';
    X:
}
```

Q

```
while (1) {
    Y:
    print '1';
    print '1';
    Z:
}
```

Sync. statements can be inserted only at points W, X, Y, Z.

→ If W: P(S) Y: (P(T)) | S, T ↓ ↓ 1, 0 init
X: V(T) Z: V(S)



Prints 0011 0011 0011 ....

9. In problem 8, which will ensure that the o/p string never contains a substring of form 01^n0 or 10^n1 where n is odd?

→ Check by hit & trial

Other process must execute after the first process gets executed completely.

W: P(S)
X: V(S)

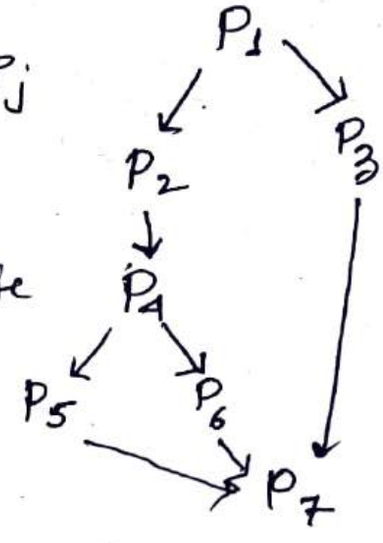
Y: P(S)
Z: V(S).

S init = 1

- **Precedence graph –**

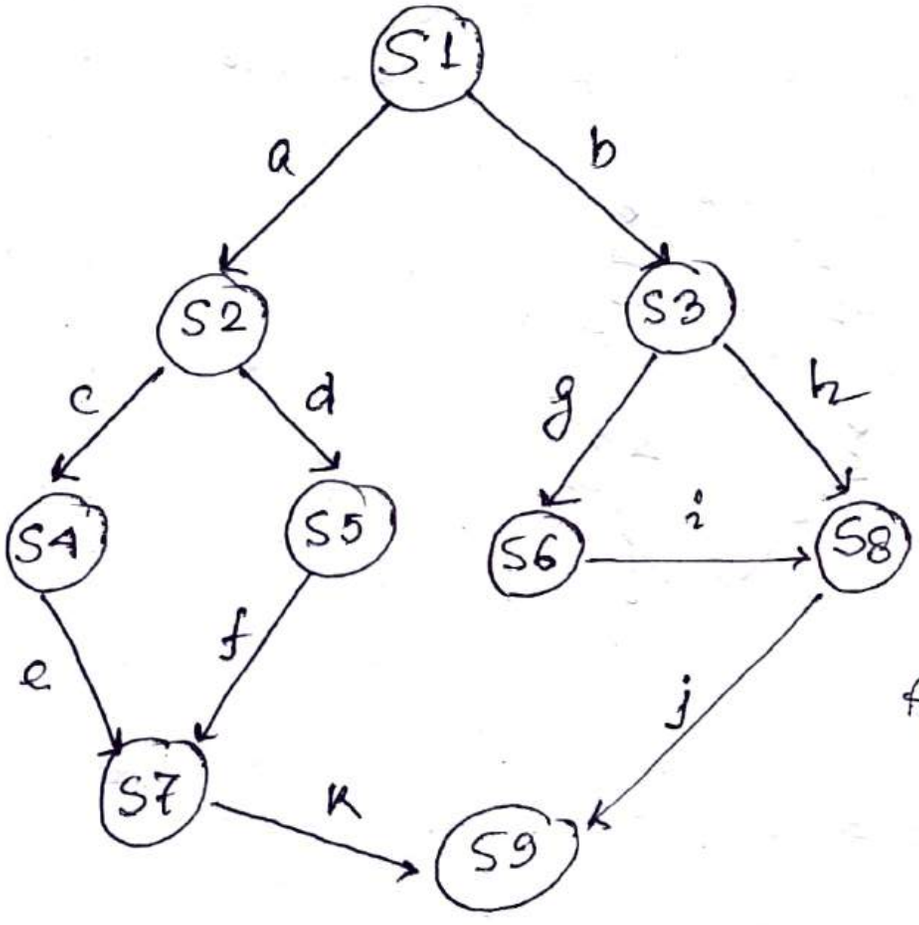
It shows the order in which activities can execute.

In the graph, edge from P_i to P_j means P_i must execute before P_j. [eg – P_2 & P_3 can execute after P_1 completes, P_7 can only execute after P_5, P_6, P_3 complete.]



eg. Concurrent processing using semaphore operators P & V for sync. Draw precedence graph for statements S_1 to S_9

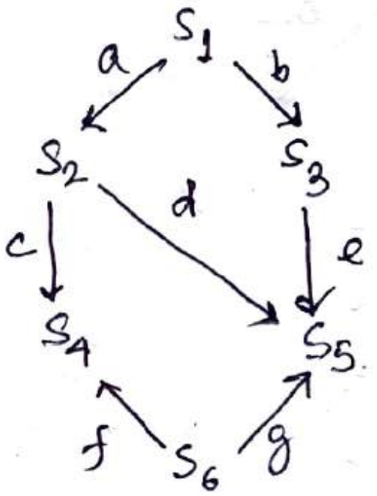
```
Var
a,b,c,d,e,f,g,h,i,j,k : semaphore ;
begin
co begin
  begin S1 ; V(a) ; V(b) end ;
  begin P(a) ; S2 ; V(c) ; V(d); end ;
  begin P(c) ; S4 ; V(e) ; end ;
  begin P(d) ; S5 ; V(f) ; end ;
  begin P(e); P(f); S(7); V(k); end ;
  begin P(b); S(3); V(g) ; V(h) ; end ;
  begin P(g); S(6) ; V(i) ; end ;
  begin P(h) ; P(i) ; S(8) ; V(j) end ;
  begin P(j) ; P(k) ; S9 end ;
coend
end ;
```



precedence graph.

- A higher level language construct for specifying concurrency as the parbegin & parend statement. Also known as cobegin & coend. All statements enclosed between parbegin & parend can be executed concurrently.

eg. Write concurrent program using cobegin & coend & semaphores.



→ Var

a, b, c, d, e, f, g : semophore ;

begin

cobegin

begin S1 ; V(a) ; V(b) end ;

begin P(a) ; S2 ; V(c) ; V(d) end ;

begin P(b) ; SB ; V(e) end ;

begin P(c) ; P(f) ; S4 end ;

begin P(d) ; P(e) ; S5 end ;

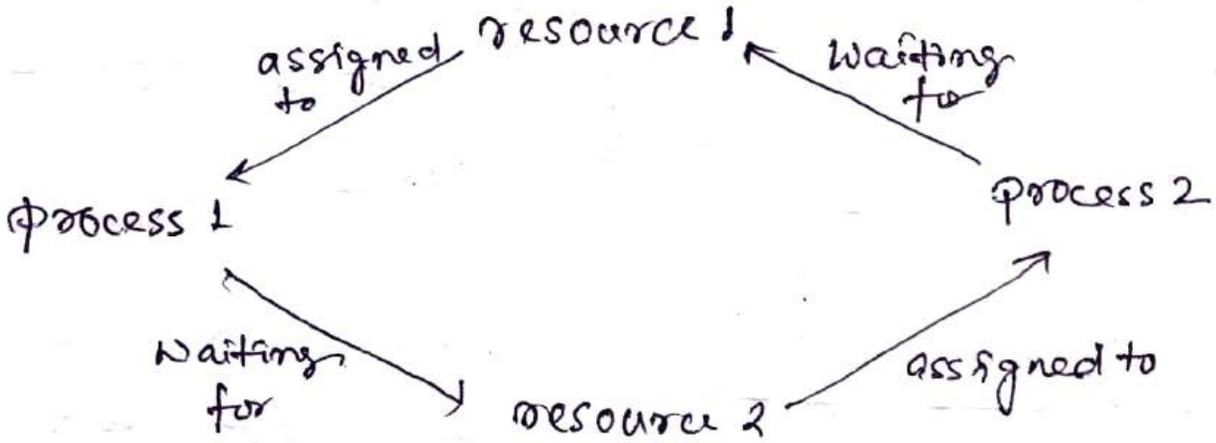
begin S6 ; V(f) ; V(g) end ;

coend

end ;

Deadlock.

* Deadlock is a situation where a set of processes are blocked because each process is holding a resource of waiting for another resource acquired by some other process.



Both processes keep waiting infinitely.

Analogically, two trains coming toward each other on same track.

* Conditions for deadlock:

1. Mutual Exclusion.: There must exist at least one resource in the system that can be used by only one process at a time. If there exists no such resource, then deadlock will never occur. Printer is an example of a resource

that can be used by only one process at a time.

2. Hold & Wait : There must exist a process that holds some resource & waits for another resource held by some other process.

3. No preemption : Once the resource is allocated to the process, it can't be preempted.

4. Circular wait : All the processes must wait for the resource in a cyclic manner where the last process waits for the resource held by the first process.

All these 4 conditions must hold simultaneously for the occurrence of deadlock.

*Deadlock Handling.

Strategies —
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection & recovery
- Deadlock ignorance.

→ Deadlock prevention.

Involves designing a system that violates one of the four necessary cond^{n}s reqd. for occurrence of deadlock. Various cond^{n}s violated as -

1) Mutual exclusion: All system resources must be such that they can be used in a shareable mode. [In a system there are always some resources that are mutualy exclusive. So, this cond^n can't be violated.].

2) Hold & Wait: Can be violated in following ways -

a) A process has to first request for all resources it requires for execution. Once it acquires all resources,

only then it can start its execution. This ensures that the process does not hold some resources & wait for other resources. [Drawbacks – Less efficient, it's not possible to predict in advance which process requires which resources]

b) A process is allowed to acquire the resources it desires at the current moment. After acquiring the resource it starts execution. Now, before making any new request, it has to compulsorily release all the resources that it holds currently. [Efficient & implementable]

c) A timer is set after the process acquires any resource. After the timer expires, process has to compulsorily release the resource.

3) No preemption.: Can be violated by forceful preemption.

A process is allowed to forcefully preempt the resources possessed by some other process only if it is a high priority process or a system process / the victim process is in the waiting state.

4. Circular wait: Can be violated by not allowing the processes to wait for resources in a cyclic manner.

Approaches -

a) A natural number is assigned to every resource. Each process is allowed to request for the resources either in only increasing or only decreasing order of the resource number. In case increasing order is followed, if a process requires a lesser no. resource, then it must release all the resources having larger number. This approach may cause starvation & never leads to deadlock.

→ Deadlock Avoidance :

Involves maintaining a set of data using which a decision is made whether to entertain the new request or not. If entertaining the request causes the system to move in an unsafe state, then it's discarded. This strategy requires that every process declares its maximum requirement of each resource type in the beginning. Main challenge is predicting the requirement of the processes before execution. [eg. Banker's algo.].

→ Deadlock detection & Recovery

Involves waiting until a deadlock occures. After deadlock occers, system state is recovered. Main challenge is detecting the deadlock. If resources have single instance, we can check for cycle in the resource allocation graph. (for multiple instances of resources, cycle in graph is necessary but not sufficient cond")

→ Deadlock Ignorance.

Involves ignoring concept of deadlock & assuming as if it doesn't exist. This strategy helps to avoid the extra overhead of handling deadlock. Windows & Linux use this strategy & it is the most widely used method. (Ostrich approach).

* Consider there are n processes where P_i require x_i units of resource R.

In worst case, the no. of units that each process holds = one less than max demand

i.e. P_i holds $x_i - 1$ units of R.

Now, had there been one more unit of R in the system, system could be ensured deadlock free. This is because unit would be allocated to one of the processes & it would get executed & then release its resource units.

Max. no. of- units that ensures deadlock

$$(x_1 - 1) + (x_2 - 1) + \cdots + (x_n - 1)$$

$$= \sum x_i - n = \text{Sum of- max needs of } n \text{ processes} - n$$

Min. no. of- units that ensures no deadlock

One more than max no of units that ensures deadlock

$$= \left(\sum x_i - n\right) + 1$$

* Banker's Algorithm. [for deadlock avoidance]

→ Data structures used.

DS.	Defⁿ	eg.
1. Available	Single dimensional array that specifies the no. of instances of each resource type currently available.	Available [R1] = K. .
2. Max	It's 2D array that specifies the maximum no. of instances of each resource type that a process can request.	Max [P1] [R1] = k
3. Allocation.	2D array that specifies the no. of instances of each resource type that has been allocated to the process.	Allocation [P1] [R1] = k.
4. Need	2D array that specifies the no. of instances of each resource type that a process more requires for execution.	Need [P1] [R1] = K

→ Safe state. : When all the processes in the system can be executed in some arbitary sequence with available no. of resources.

To implement safety - work & finish data structures are used. Work is a 1d array that specifies the no. of instances of each resource type currently available. Finish is a 1d array that specifies whether the process has finished its execution or not. Finish [P1] = 0 means P1 is yet to execute.

→ Algorithm.

1. Work & finish be two vectors of size m (#resources) & n (#process). Initialize work with available & Finish [i] = 0 for i = 1 to n.

2. Find an i such that both

i) Finish [i] = 0

ii) $Need_i \leq Work$

If no such i exists then goto step 4.

3. Work = Work + Allocation;

Finish[i] = True/1

goto step 2

4. If finish[i] = 1 for all i then the system is in ~~system~~ safe state.

[Better see some example! :-:]

eg.

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

safe sequence. —

$\langle P_0, P_2, P_3, P_4, P_1 \rangle$

Need = Max - Allocation

	A	B	C	D	
✓ P_0	0	0	0	0	1st
P_1	0	7	5	0	5th
✓ P_2	1	0	0	2	2nd.
✓ P_3	0	0	2	0	3rd
P_4	0	6	4	2	4th

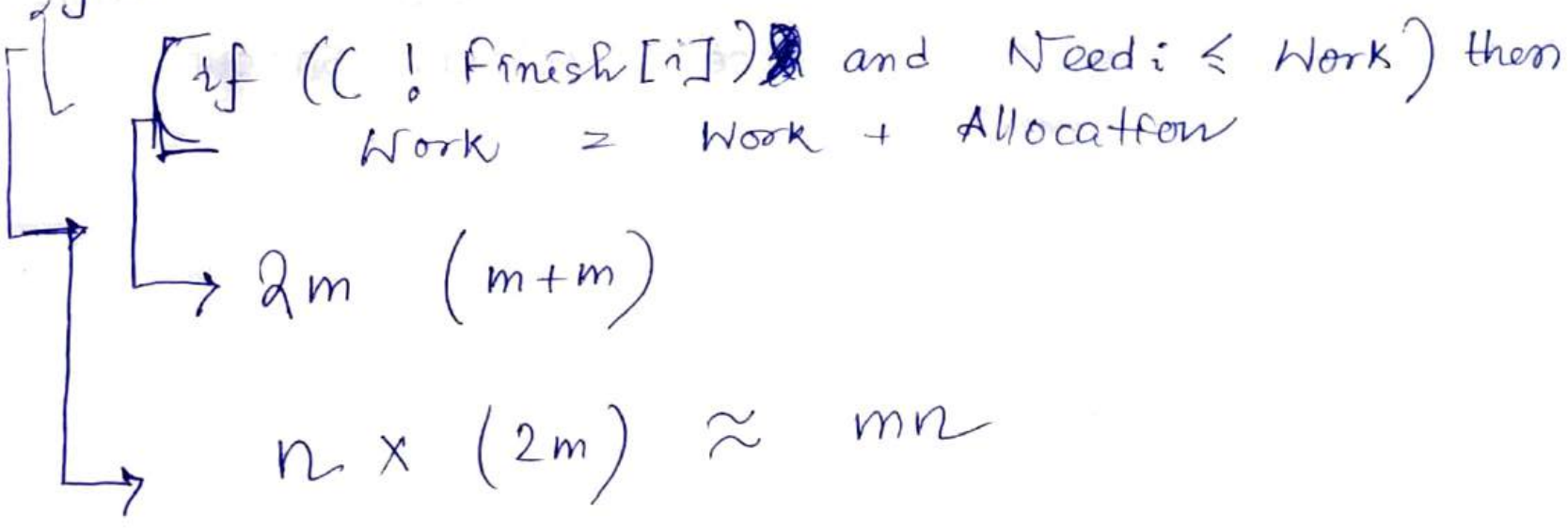
Work

A	B	C	D
1	5	2	0
0	0	1	2
1	5	3	2
1	3	5	4
2	8	8	6
0	6	3	2
2	14	11	8
0	0	1	4
2	14	12	12
1	0	0	0
3	14	12	12

Finish

0	F
1	F
2	F
3	F
4	F

Complexity:



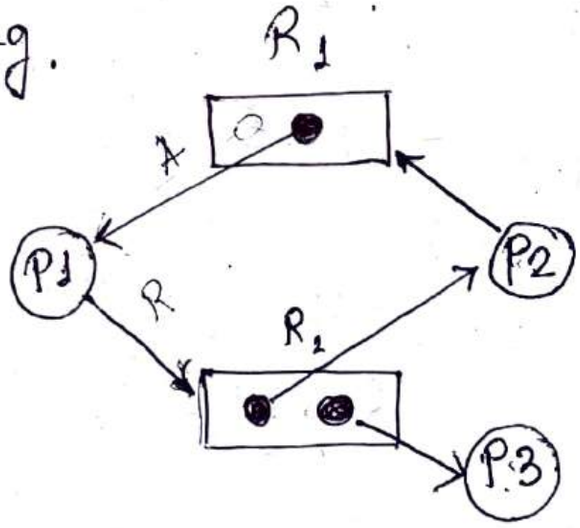
It's in repeat loop so $(n^2 m)$.

*Resource Allocation Graph.

Represents the state of a system pictorially.

- Vertices
  - Process
  - Resource
    - Single instance
    - Multi instance
- Edges
  - Assign (head of arrow points to process)
  - Request (head of arrow points to instance)

eg.



P1 holds R1 & requests one instance of R2.
P2 holds one instance of R2 & requests for an instance of R1.
P3 holds one instance of R2.

→ Deadlock detection using RAG.

In a RAG, where all the resources are single instances, if a cycle is being formed, then system is in a deadlock state. If no cycle, process system is not in deadlock state.

In a RAG, where all the resources are not single instance (i.e. multi-instance)

if a cycle is being formed, then system may be in a deadlock state. Banker's algo is applied to confirm whether system is or not in a deadlock state. If no cycle is formed, system is not in a deadlock state.

Presence of a cycle is necessary but not sufficient for occurence of deadlock.

* Methods of resource allocation to processes.

i) Resource partitioning approach -

Allocates one resource partition to each user program before the program's initiation. A resource table records the resource partition & its current allocation status.

ii) Pool based approach - OS checks the allocation status of resources whenever a program makes a request for a resource. If resource is free, it allocates the resource to program.

Solved problems.

1. 3 user processes each requiring 2 units of resources. Min no. of units that no deadlock occur.

→ $(1 + 1 + 1) + 1 = 4$

2. 6 units of resource R. & each process on the system requires 2 units of R. How many processes can be present at max that no deadlock occurs?

→ 6 units to 6 processes each.

Max no. of processes = $6 - 1 = 5$

3. 100 units, each process requires 4 units. Then how many processes can be present at max to avoid deadlock?

→ $\left\lceil \frac{100}{4-1} \right\rceil - 1 = 34 - 1 = 33$

4. m resources of same type being shared by n processes. Resources can be requested & released by processes only one at a time. System is deadlock free iff -

Max no. of units that ensures deadlock =

$$\sum x_i - n$$

For no deadlock,

no. of units must be $> \sum x_i - n$

$$m > \sum x_i - n$$

$$\sum x_i < m + n$$

---

5. System running n processes. P_i holding x_i instances ($1 \le i \le n$). Currently all instances of R are allocated. Further, for all i, process i has placed a request for an additionally y_i instances while holding the x_i instances it has. There are two processes p & q s.t. $y_p = y_q = 0$

Which of the following is necessary cond^n for no deadlock ?

After executing Processes $p \neq q$, they release $x_p + x_q$ resource units.

Thus for no deadlock,

$$x_p + x_q \geq \min_{k \neq p,q} y_k .$$

[This is a necessary condition, not sufficient]

Sufficient cond^n would be

$$x_p + x_q \geq \Sigma y_i$$

or $x_p + x_q \geq \max y_k \quad [k \neq p, q]$

6. Single processor system has resource types X, Y, Z, shared by three processes. There are 5 units of each resource. Which process will finish last?

	Alloc			Request		
	X	Y	Z	X	Y	Z
P0	1	2	1	1	0	3
P1	2	0	1	0	1	2
P2	2	2	1	1	2	0

Available. [5 5 5] − [5 4 3]

= [0 1 2].

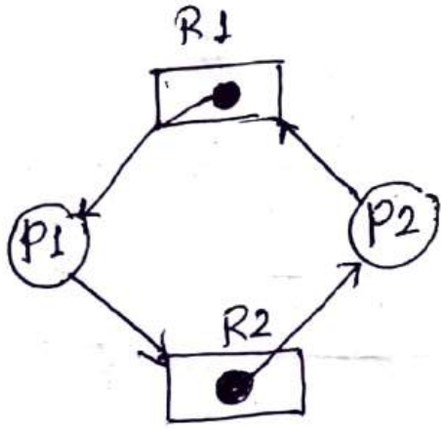
0	1	2
2	0	1
2	1	3
1	2	1
3	3	4
2	2	1
5	5	5

$\langle P_1, P_0, P_2 \rangle$

=

Ans.

7. Consider this RAG.

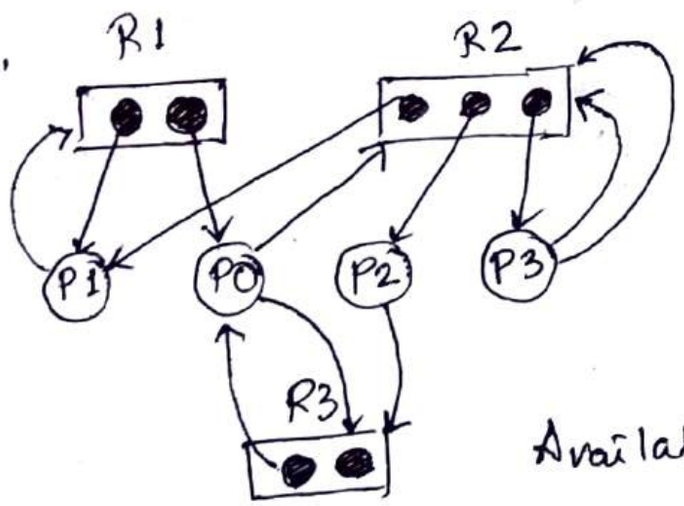


	Alloc		Need	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0

Available = [0 0]

System in deadlock.

8.



	Alloc			Need		
	R1	R2	R3	R1	R2	R3
P0	1	0	1	0	1	1
P1	1	1	0	1	0	0
P2	0	1	0	0	0	1
P3	0	1	0	0	2	0

Available = [0 0 1].

001
010
011
101
112
110
222

$\langle P_2, P_0, P_1, P_3 \rangle$ Safe sequence.

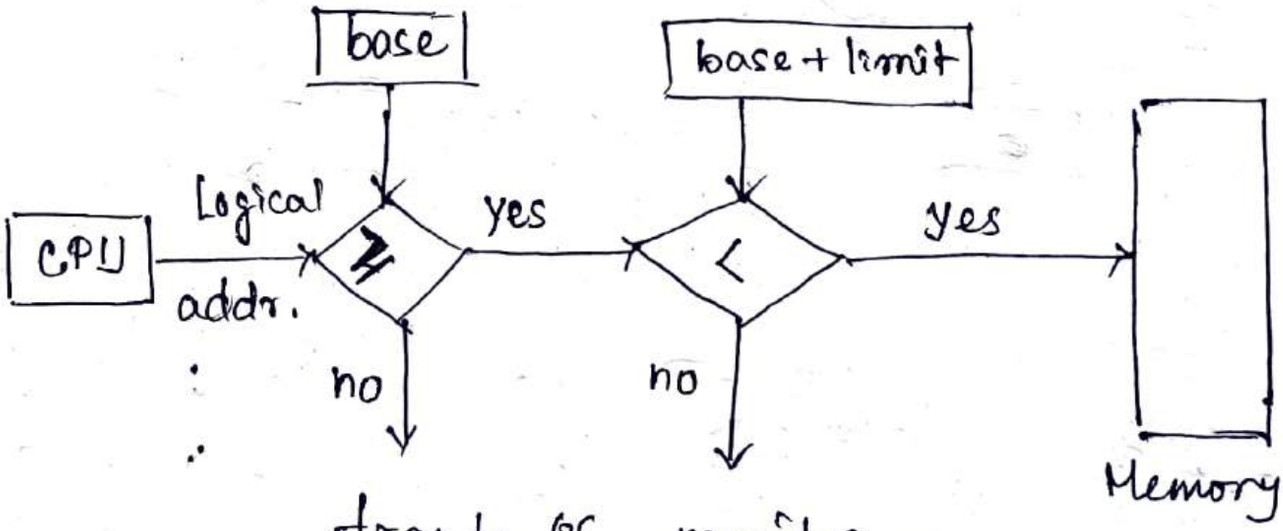
Memory Management.

*Requirements of Memory Mgmt.

1. Relocation : Relocating the process in the memory (may be after swapping).

2. Protection : To protect unwanted interference among processes.

Between relocation & protection, a trade-off occurs as the satisfaction of relocation requirement increases the difficulty of satisfying the protection requirement.



Hardware address protection with base & limit registers.

3. Sharing: Several processes accessing same portion of main memory.

4. Logical organisation.:

Internal logical organisation of the main memory (linear or 1D address space that consists of sequence of bytes or words.).

5. Physical Organisation.: Organisation of the flow of information between main & secondary memory.

* Address Binding: Process of mapping the program's logical or virtual addresses to corresponding physical or main memory address.

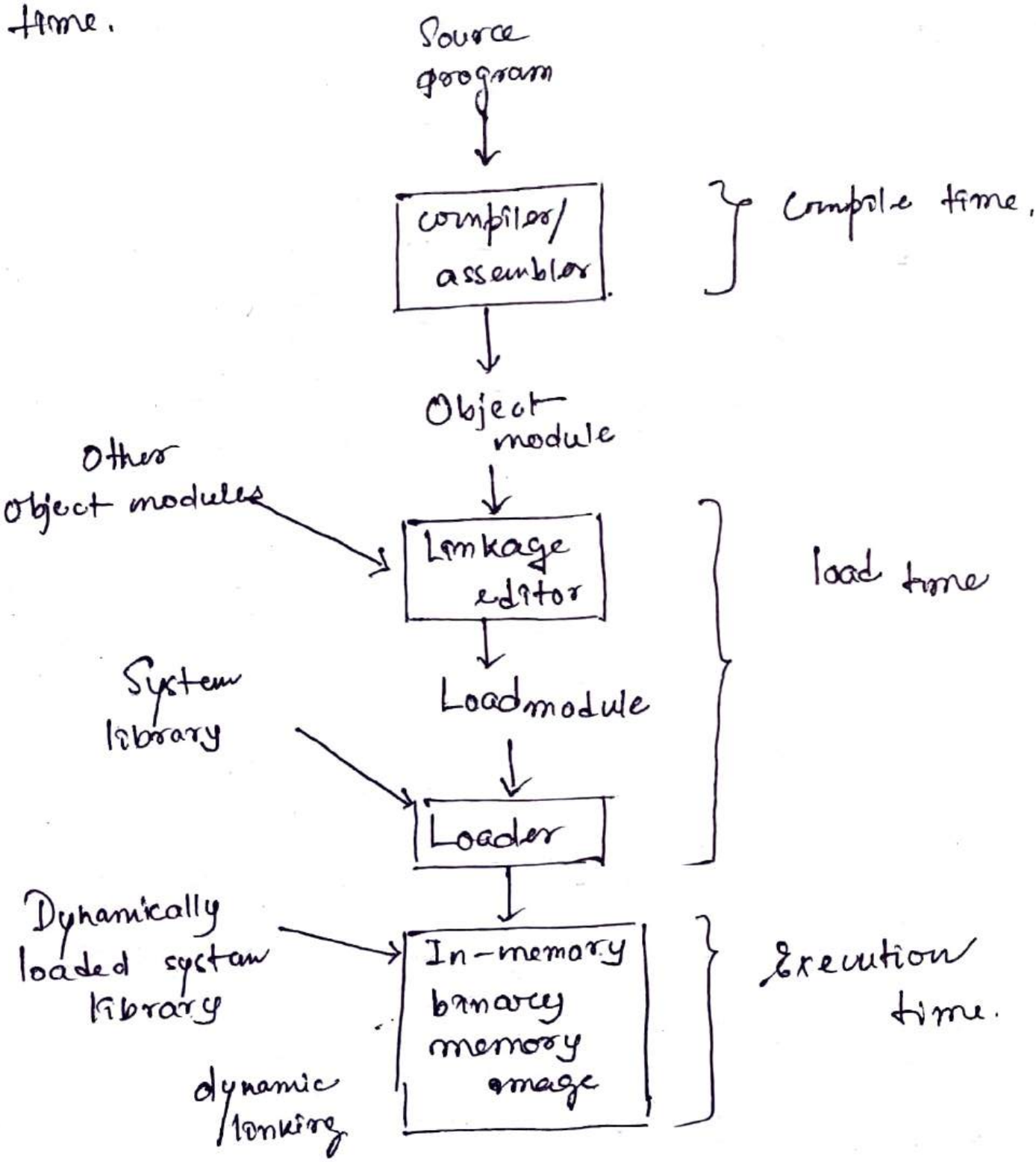
Schemes.

i) Compile time. : If it's known at compile time where a program will reside on physical memory, then absolute code can be generated by the compiler, containing actual physical addr. However, if the load address changes at some later time, then the program will have to be recompiled.

ii) Load time : If the location at which a program will be loaded is not known at compile time, the compiler must generate relocatable code, which references addresses relative to the start of the program. If that starting address changes, program must be reloaded but not recompiled.

iii) Execution time : If a program can be moved around on memory during the course of its execution, then binding must be delayed until exec^{n}

time.



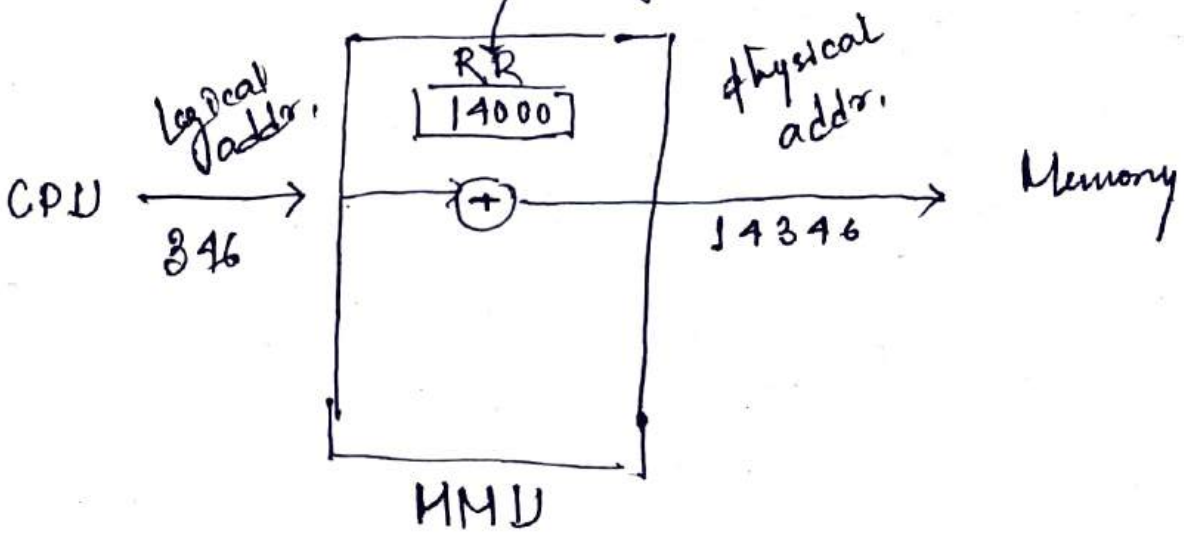
* Logical vs Physical Address space.

Addr. generated by CPU is a logical address, whereas address actually sent to the memory hardware is seen by it as a physical address.

Addresses bound at compile time or load time have identical logical & physical addresses.

Addresses created at execution time have different LA & PAs. In this case, LA is also known as virtual address. Set of all logical addresses used by a program comprises the logical address space & the set of all corresponding physical addresses comprises the physical address space.

Run time mapping of logical to physical addresses is handled by memory mgmt. unit (MMU). Here, one form of MMU can be the base & limit register. Base register, here, is called relocation register.



* Dynamic Loading:

Rather than loading an entire program into memory at once, dynamic loading loads up each routine as it's called. The advantage is that unused routines need never be loaded, reducing total memory usage & generating faster program startup.

* Static Linking

Library modules get fully included in executable modules, wasting both disk space & MM usage.

* Dynamic Linking

Only a stub (small piece of code containing references to the actual library module) is linked into the executable module at run time.

It saves disk space.

* Swapping

If there is not enough memory available to keep all running processes in memory at the same time, then some processes who are not currently using the CPU may have their memory swapped out to a fast local disk called the backing store.

Standard Swapping

If compile time or load time binding is used, then processes must be swapped back onto the same memory location from which they were swapped out. If execution time binding is used, then the processes can be swapped back into any available location.

* Relocation

Finding a way to map virtual addresses onto physical addresses.

Static relocation.

Performed before or during the loading of the program into memory by a relocating linker / loader.

Dynamic relocation.

Runtime mapping from virtual address to physical address performed by MMU.

* Contiguous Memory Allocation.

Allows to store the process only in a contiguous fashion. Thus, entire process has to be stored as a single entity at one place inside the memory.

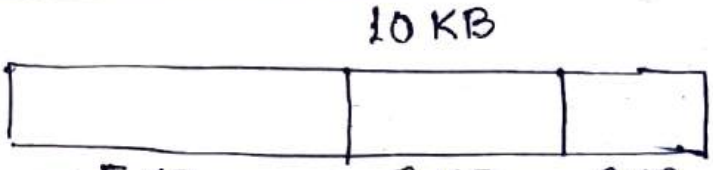
Contiguous memory allocation techniques —

i) Static partitioning / Fixed size partitioning

ii) Dynamic partitioning / Variable size partitioning

→ Static Partitioning.

Main memory is predivided into fixed size partitions. Each partition is allowed to store only one process.



Algorithms for partition allocation

1. First fit algorithm. Starts scanning the partitions serially from the starting. When an empty partition that is big enough to store the process is found, it's allocated to the process.

partition size ≥ process size.

2. Best fit algorithm. Scans the empty partitions. Allocates the smallest size partition to the process.

3. Worst fit algorithm. Scans all empty partitions. Allocates largest size partition to the process.

• For static partitioning, best fit algorithm works best, because space left after the allocation inside the partition is of very small size. Thus, internal fragmentation is least.

• Worst fit algorithm works worst. Internal fragmentation is maximum.

• Internal fragmentation.

Occurs when the space is left inside the partition after allocating the partition to a process. This space can't be allocated to other processe

Internal fragmentation occurs only in static partitioning.

• External fragmentation.

Occurs when the total amount of empty space required to store the process is available in the MM; but as the space is not contiguous, so process can't be stored.

- Translating logical address into Physical address.

CPU generates a logical address. A physical address is needed to access the main memory.

1. Translation scheme uses 2 registers that are under the control of OS. During context switching, the values corresponding to the process being loaded are set in the registers.

Relocation register stores the base address of the process in the MM.

Limit register stores the size or length of the process.

2. CPU generates a logical address containing the address of the row that it wants to read.

3. Logical address is compared with the limit of the process.

3.a. Generated address ≥ limit

A trap is generated. This helps to prevent unauthorised access.

3.b. Generated address < limit

Address must always lie in the range [0, limit - 1]. Treated as valid. Generated address is added with the base addr. of the process.

- Advantages of Static Partitioning

Simple & easy to implement. Supports multiprogramming since multiple process can be stored inside the MM. Only one memory access is reqd. which reduces the access time.

- Disadvantages. Suffers both internal fragmentation & external fragmentation. It utilises memory inefficiently. Degree of multiprogramming is limited equal to no. of partitions. There is limitation on the size of process since processes with size greater than the size of largest partition can't be stored & executed

→ Dynamic Partitioning

Performs the allocation dynamically. When a process arrives, a partition of size equal to the size of process is created. Then, the partition is allocated to the process.

• Partition allocation algorithms

Processes arrive & leave the MM. As a result, holes of different size are created in the MM. These holes are allocated to the processes that arrive in future.

First fit, best fit, worst fit.
(Same as before).

• For dynamic partitioning, worst fit algo. works best. Because, space left after allocation inside the partition is of larger size. There is high probability that this space might suit the requirement of arriving processes.

• Best fit algo works worst. There is low probability that the space left after allocation might suit the requirement of arriving processes.

• Translating logical address into Physical address.

Same as static partitioning.

• Advantages. Doesn't suffer from internal fragmentation.

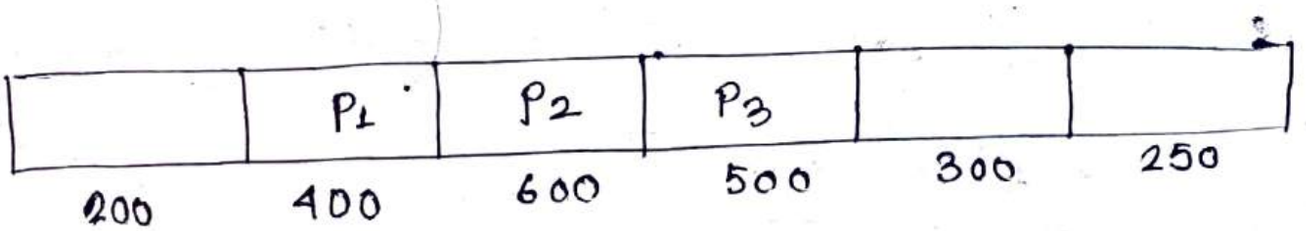
Degree of multiprogramming is dynamic. There is no limitation of the size of processes.

• Disadvantages.

Suffers from external fragmentation. Allocation & deallocation of memory is complex.

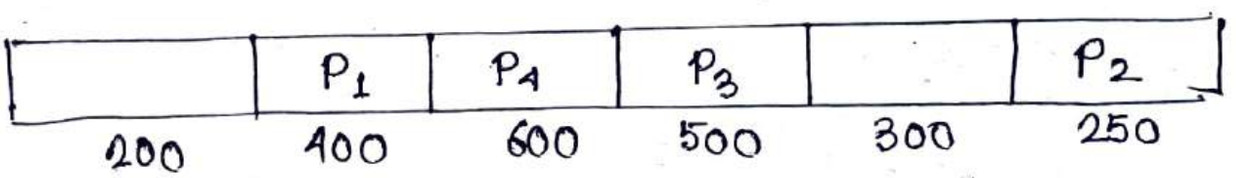
eg. Six memory partitions of size 200 KB, 400 KB, 600 KB, 500 KB, 300 KB & 250 KB. These partitions need to be allocated to four processes of sizes 357 KB (P_1), 210 KB (P_2), 468 KB (P_3), 491 KB (P_4) in that order.

– Allocation using First Fit algo =

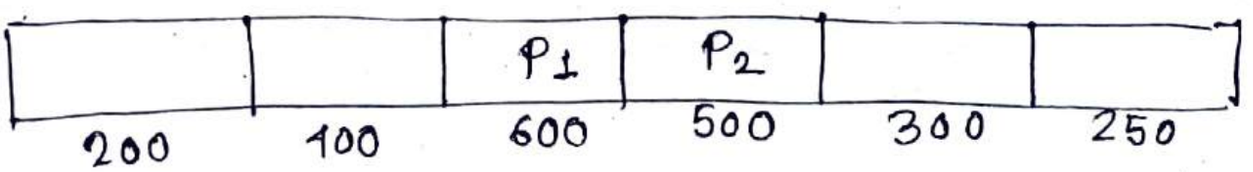


P_4 can't be allocated.

– Using Best fit algo =

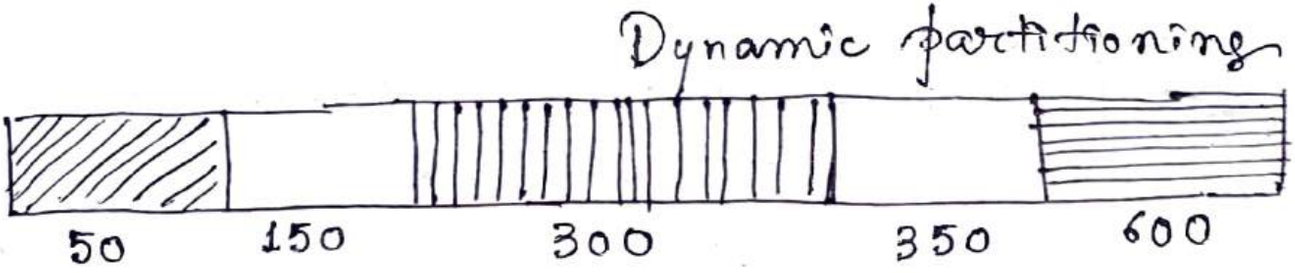


– Using Worst fit algo =



P_3 & P_4 can't be allocated.

eg.



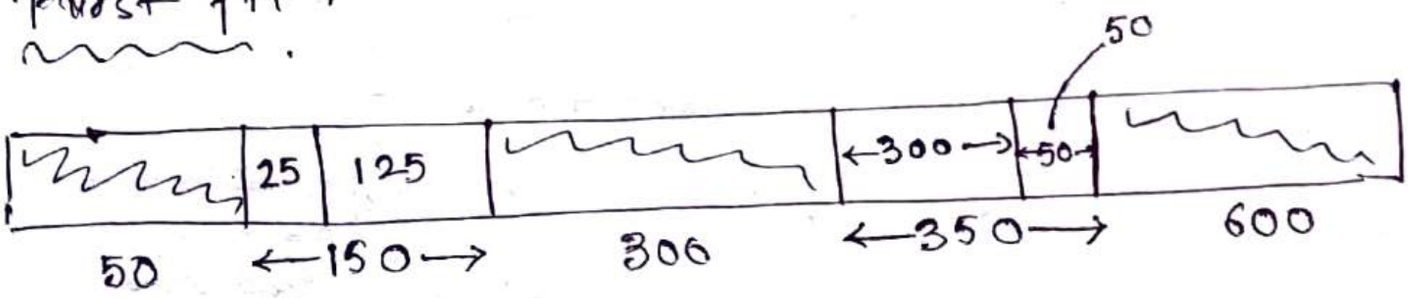
Sequence of requests for blocks of size 300, 25, 125, 50 can be satisfied if we use

i) Either first fit or best fit

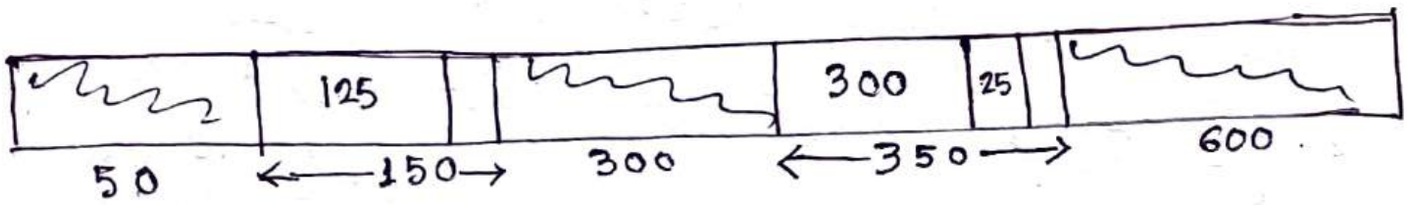
ii) First fit but not best fit

iii) Best fit but not first fit

First fit.



Best fit



50 can't be allocated.

* Non-contiguous Memory Allocation.

Allows to store parts of a single process in a non-contiguous fashion.

Techniques for NCMA -

i) Paging ii) Segmentation.

→ Paging: It's a fixed size partitioning scheme. Secondary & main memory are divided into equal fixed size partitions. Partitions of secondary memory are called ~~frames~~ pages, that of main memory are called ~~pages~~ frames.

Each process is divided into parts where size of each part is same as page size. Size of last part may be less than the page size. Pages of process are stored in the frames of MM depending upon their availability.

Physical address space divided into a no. of frames; Logical address space divided into pages.

- Addr. generated by CPU is divided into

  page number (p): No. of bits reqd. to represent the pages in logical addr. space or page no.

  page offset (d): No. of bits reqd. to represent particular word in a page or page size of logical address space or word no. of a page.

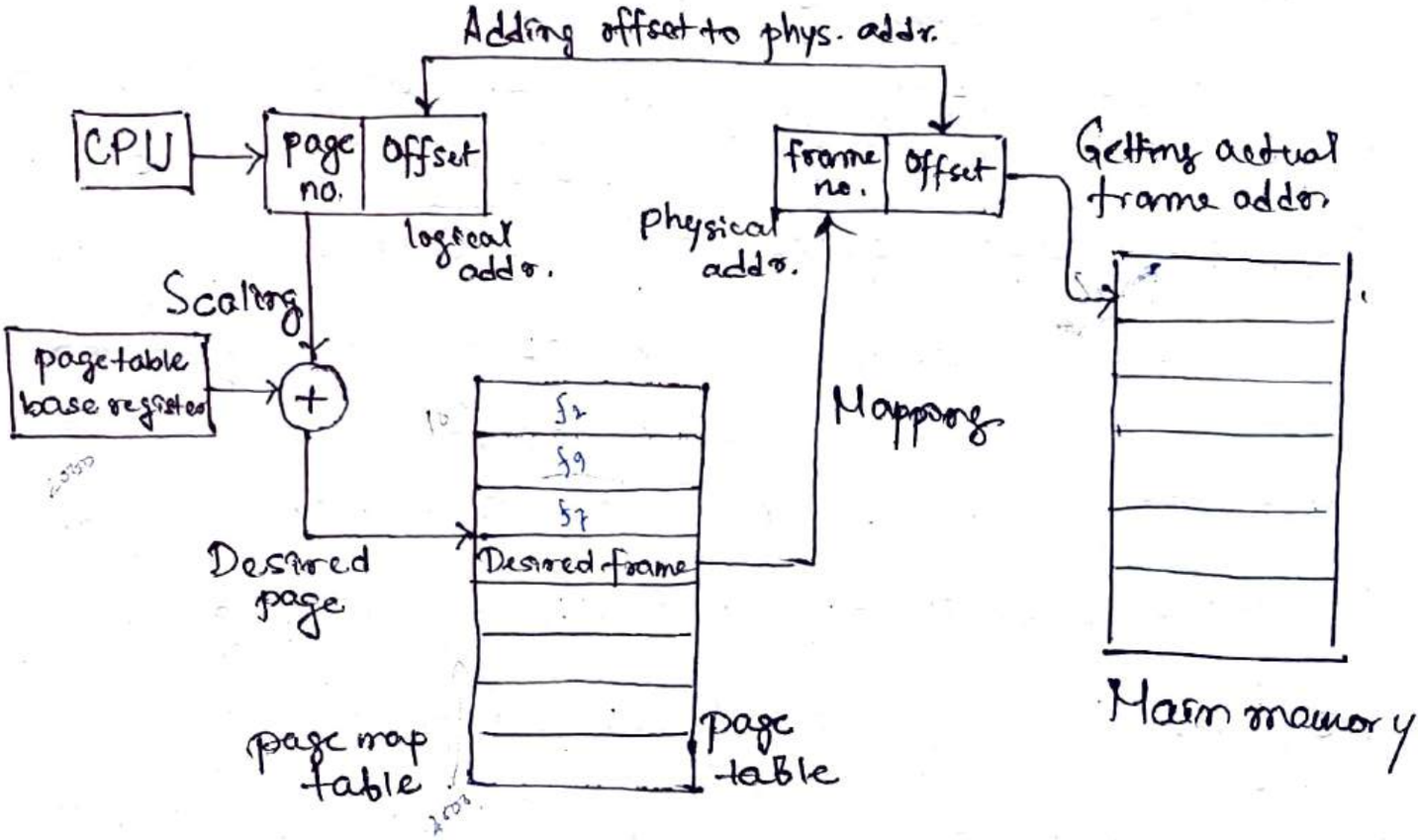
- Physical addr. is divided into

  frame no. (f): No. of bits reqd. to represent the frame of physical addr. space or frame no.

  frame offset (d): No. of bits reqd. to represent particular word in a frame of physical addr. space.

→ Working: CPU generates a logical addr consisting of page no. & page offset. Page table provides the corresponding frame no. where that page is stored in the main memory. The frame no. combined

with the page offset forms the reqd. physical addr. Frame no. specifies the specific frame where the reqd. page is stored in main memory.



→ To determine the actual page no. of process, CPU stores the page table base in a special register. Each time the addr. is generated, the value of the page table base register is added to the page no. to get the actual loc^{n} of the page entry on the table. This is called scaling.

→ Offset of physical addr. & logical addr. are similar (as page size & frame size are equal.).

• Advantages of Paging

i) It allows to store parts of a single process in a non-contiguous fashion.

ii) Solves the problem of external fragmentation.

• Disadvantages:

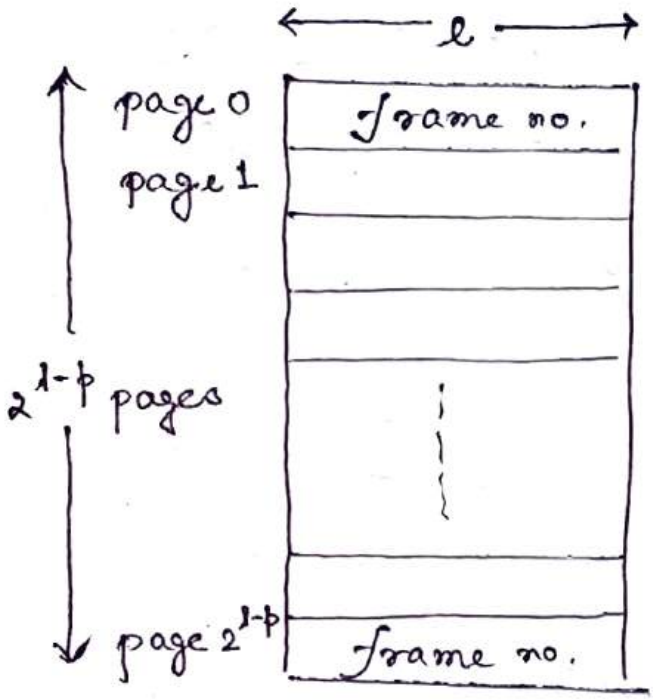
i) Suffers from internal fragmentation.

ii) Overhead of maintaining a page table for each process.

iii) Time taken to fetch instr increases as 2 memory accesses are reqd.

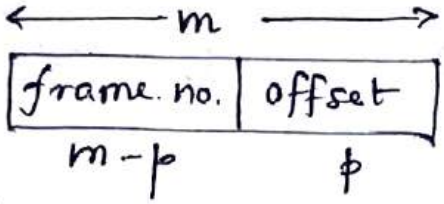
→ Page Table: Data structure that maps the page no. referenced by the CPU to the frame no. where the page is stored.

Page table is stored in the main memory. No. of entries in the page table is equal to the no. of pages in which the process is divided.

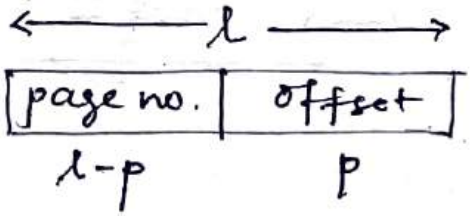


page table.

Phys. addr.



Log. addr.



page table size = $2^{l-p} \times e$

$e = m - p$ bits

• Page table entry has following information -

1. Frame no. Compulsory field. No. of bits on the frame no. depends on the no. of frames in the MM.

2. Present/Absent bit.

Also called valid/invalid bit. Specifies whether that page is present on the MM or not.

If the required page is not present on the MM, then it is called page fault. A page fault requires

page initialisation. The reqd. page has to be initialised (fetched) from the secondary memory.)

3. Protection bit. This bit is called read/write bit. Specifies the permission to perform read & write on the page. [If only read then bit = 0, if both read, write bit = 1.]

4. Reference bit. Specifies whether that page has been referenced in the last clock cycle or not. (If referenced recently then bit = 1).

Reference bit is useful in page replacement policy.

5. Caching enabled/disabled. Whenever fresh data is reqd. caching is disabled using this bit.

6. Dirty bit. Also called modified bit. Specifies whether the page has been modified or not.

Normal PT.

Frame no	Page no
f_0	0
f_3	1
⋮	2
f_{10}	3

Inverted PT.

Frame no.	Page no.	Process id.
0	p_0	P_1
1	p_1	P_9
2	p_7	P_1
⋮	⋮	⋮
	p_1	P_7

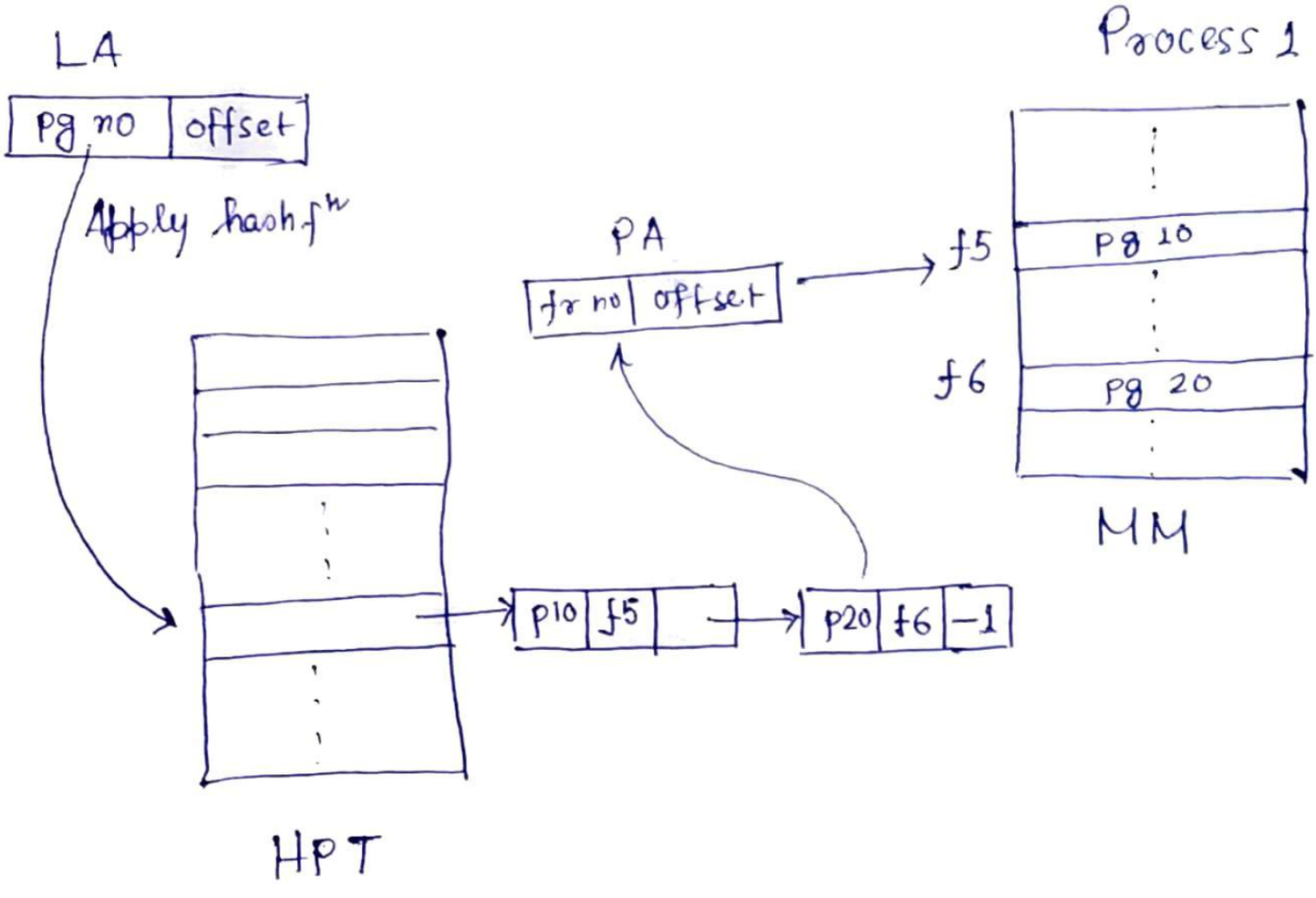
Each P has its own PT.

PT will be in MM.

IPT - no. of entries - no. of frames.
(global PT).

Overhead of searching.

Hashed PT.


LA
pg no
offset
Apply hash fn
HPT
p10
f5
p20
f6
-1
PA
fr no
offset
Process 1
f5
pg 10
f6
pg 20
MM

In case the page is modified, before replacing the modified page with some other page, it has to be written back on the secondary memory to avoid losing the data. Dirty bit helps to avoid unnecessary writes. This is because if the page is not modified then it can be directly replaced by another page without any need of writing it back to the disk.

→ Formulae on paging

MAIN MEMORY

- Physical addr. space = size of MM
- Size of MM = Total no. of frames × Page size
- Frame size = Page size.
- If no. of frames in MM = 2^x, then no. of bits in frame no. = x bits
- If page size = 2^x bytes, then no. of bits in page offset = x bits (byte addressable).

PROCESS

- If size of MM = 2^x bytes, no. of bits in the physical address = x bits.

- Virtual addr. space = size of process
- No. of pages the process is divided = $\frac{\text{Process size}}{\text{Page size}}$.
- If process size is 2^x bytes, then no. of bits in physical addr. = x bits.

PAGE TABLE

- Size of page table = No. of entries * Page table entry size
- No. of entries in page table = No. of pages the process is divided
- Page table entry size = No. of bits in frame no. + No of bits used for optional fields

- In general, of the given address consist of n bits, then using n bits, 2^n locations are possible. Then, size of memory = 2^n * size of one location. = 2^n * m bytes [when 1 word = m bytes & memory is word addressable]

eg. System with byte addressable memory, 32 bit logical addresses, 4 KB page size & page table entries of 4 bytes each. Size of page table?

→ Process size = 2^{32} B

No. of entries in page table = $\frac{2^{32}}{2^{12}}$

Page table size = $2^{20} \times 4 = 2^{22}$ B $= 4$ MB

$= 2^{20}$ pag

eg. Consider a machine with 64 MB physi memory & 32 bit virtual addr. If the page size is 4 KB, what is PMT (Page map table) size?

→ No. of pages = $\frac{2^{32}}{4\text{KB}} = 2^{20}$

No. of bits in physical addr. = $\log_2 2^{26}$

$= 26$ bits

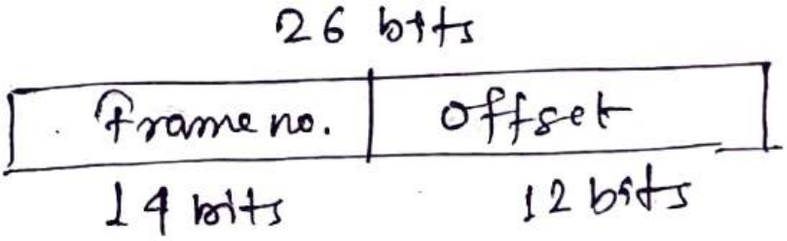
No. of frames in MM =

$$\frac{64\text{MB}}{4\text{KB}} = 2^{14}$$

No. of bits in frame no. = 14.

No. of bits in page offset = $\log_2 2^{12} = 12$

[from 4KB page]



Process size = 2^{32} B = 4GB.

Page table size = $2^{20} \times$ (14 bits)

$\approx 2^{20} \times$ (16 bits) [approx.

$= 2^{20} \times 2$ bytes

$= 2^{22}$ bytes = 4 MB.

eg. In a virtual memory system, size of virtual address is 32 bit, size of physical addr. is 30-bit, page size is 4 KB. & size of each page table entry is 32 bit. The MM is byte addressable. Which one of the following is the maximum number of bits that can be used for storing protection & other information in page table entry?

→ Size of virtual address = 32 bit

Size of physical address = 30 bit

Page size = 4 KB.

Size of each page table entry = 32 bit

Size of MM = 2^{30} B = 1 GB.

No. of frames on MM = $\frac{1GB}{4KB} = 2^{18}$

No. of bits in frame no = 18

Max no. of bits that can be used for storing protection & other information

= 32 bits − 18 bits.

= 14 bits. (Ans).

- Overhead in Paging.

i) Overhead of page tables.

Paging requires each process to maintain a page table.

ii) Overhead of wasting pages

Wasting last page of each process if it is not completely filled.

Total overhead for one process =

$$\text{Size of its page table} + \left(\frac{\text{Page size}}{2}\right)$$ only for wasting pages.

Optimal page size

Page size that minimises the Total overhead.

$$\text{Optimal page size} = \sqrt{2 \times \text{Process size} \times \text{PMT entry size}}$$

Derivation

Total overhead due to one process =

$$\text{Size of page table} + \frac{\text{page size}}{2}.$$

$$= \text{No. of entries} \times \text{Entry size} + \frac{\text{page size}}{2}$$

$$= \frac{\text{Process size}}{\text{Page size}} \times \text{Entry size} + \frac{\text{Page size}}{2}$$

For minimum overhead,

$$\frac{d}{d(\text{page size})} \text{ overhead} = 0$$

Process size = P

page size = p_a

entry size = e

P, e constant

$$\therefore \frac{d}{dp_a}\left[\frac{P}{p_a} \times e + \frac{p_a}{2}\right] = 0$$

$$\Rightarrow -\frac{1}{p_a^2} \cdot P \cdot e + \frac{1}{2} = 0$$

$$\Rightarrow p_a = \sqrt{2Pe}$$

eg. Virtual addn space is 4 KB & PMT entry size 8 B. What should be the optimal page size ?

$$\rightarrow \sqrt{2 \times 4KB \times 8B}$$

$$= \sqrt{2 \times 2^{12} \times 2^3} = 2^8 \text{ B}.$$

eg. In a paging scheme, virtual addr. space is 16 MB & entry size is 2 B. What should be the optimal page size?

→ $(2 \times 16\,MB \times 2B)^{1/2} = 8\,KB.$

✦ Translation Lookaside Buffer (TLB)

Major disadvantage of paging is it increases the effective access time due to increased no. of memory accesses. One memory access is reqd. to get the frame no from the page table. Another memory access is reqd. to get the word from the page.

TLB is a solution that tries to reduce the effective access time. Being a hardware, the access time of TLB is very less as compared to the MM.

TLB consists of 2 columns - page number & frame number.

Page No.	Frame No.

Translating logical address into physical address.

In a paging scheme using TLB, the logical addr. generated by the CPU is translated into the physical address using following steps –

1. CPU generates logical addr. consisting of 2 parts page no. & page offset.

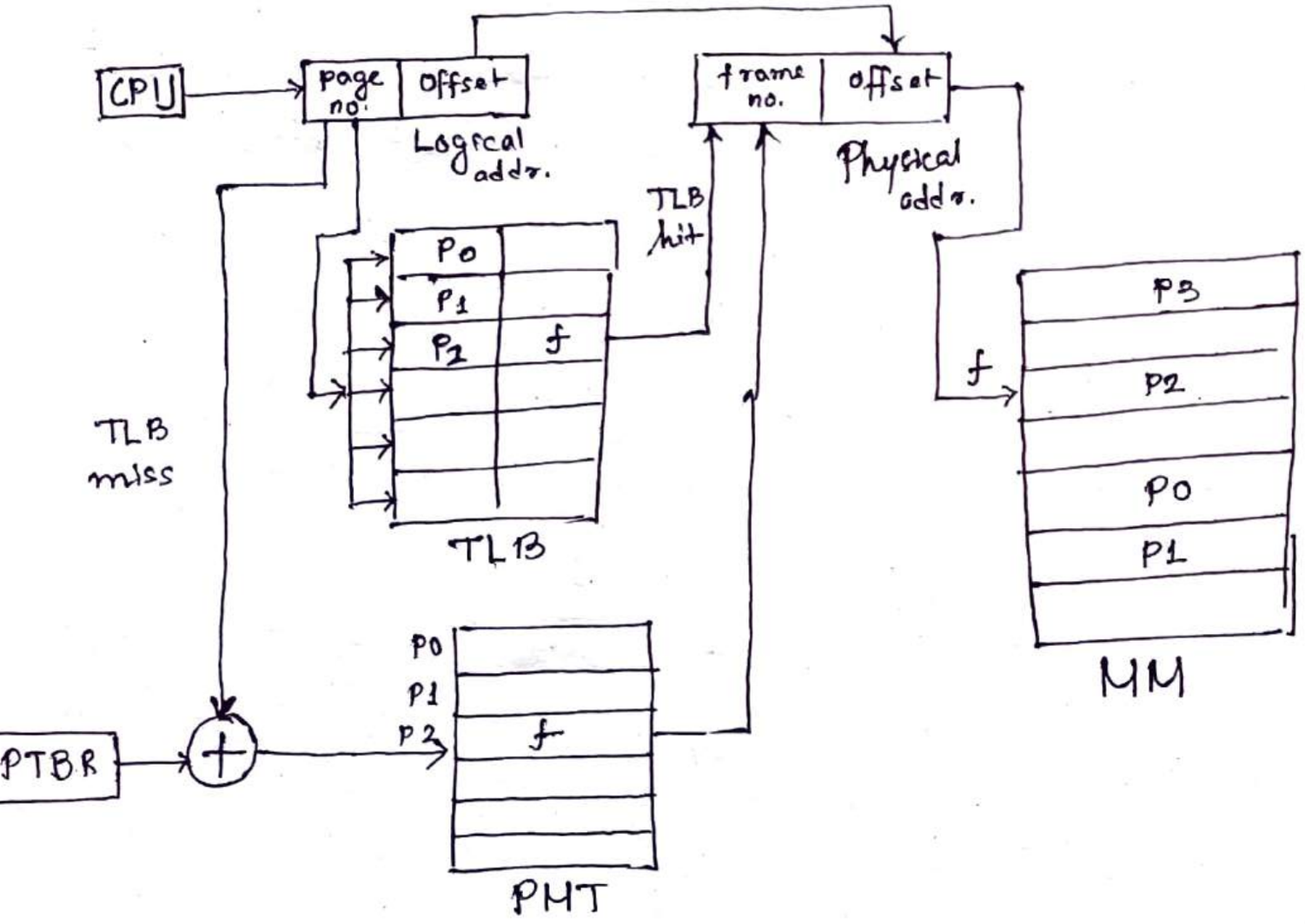
2. TLB is checked to see if it contains an entry for the referenced page no. The referenced page no. is compared with the TLB entries all at once. Now, two cases arise –

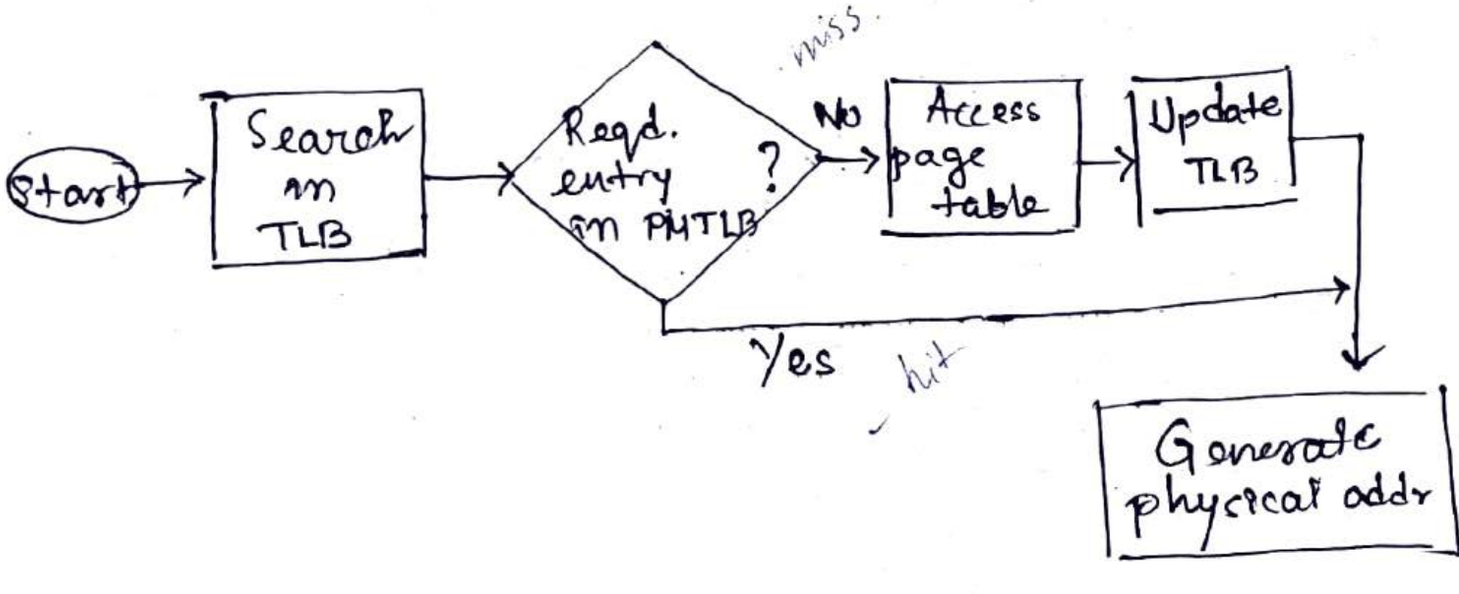
2a. If TLB hit – If TLB contains an entry for the referenced page no., TLB entry is used to get the corresponding frame no. for the reference page no.

2b. If TLB miss – PMT is used to get the corresponding frame no. for the referenced page no. Then TLB is updated with the page no. & frame no. for future references.

3. After the frame no. is obtained, it is combined with the page offset (≡ frame offset) to generate the physical address. Then physical addr. is used to read the reqd. word from the MM.

[Assumed, no page fault].





• Unlike page table, there exists only one translation Look aside buffer in the system. Whenever context switching occurs the entire content of TLB is flushed & deleted. TLB is then updated with currently running process.

• When a new process gets scheduled, initially TLB is empty. So, TLB misses are frequent. With every access from the page table, TLB is updated & eventually TLB hits increase.

• Time taken to update TLB after getting the frame no. from the page table is negligible. TLB is updated in parallel while fetching the word from the MM.

• Advantages of TLB.

Reduces effective access time. Only one memory access reqd. when TLB hit occurs.

• Disadvantages.

After some time of running the process when TLB hits increases & process starts to run smoothly, a context switching occurs. The entire content of TLB is flushed. TLB is again updated with currently running process.

* TLB can hold the data of only one process at a time. As it is a special hardware, it requires additional cost.

- Effective access time.

Single level paging using TLB,

$$EAT = \text{Hit ratio of TLB} \times \{\text{Access time of TLB} + \text{Access time of main memory}\} +$$

$$\text{Miss ratio of TLB} \times \{\text{Access time of TLB} + 2\times \text{access time of MM}\}$$

[No page faults]

For multilevel paging,

$$EAT = p(t+m) + (1-p)(t+km+m).$$

$k \rightarrow$ levels.

eg. A paging scheme uses a TLB. A TLB access takes 10 ns. & a MM access takes 50 ns. What is the EAT if TLB hit ratio is 0.9 & there's no page fault?

$\rightarrow$ $0.9\{10+50\} + 0.1\{10+2\times 50\}$

$= 65$ ns.

• Multilevel Paging

Paging scheme where there exists a hierarchy of page tables.

The need of multilevel paging arises when - the size of page table is greater than the frame size. As a result, the page table can't be stored in a single frame in MM.

Working: The page table having size greater than the frame size is divided into several parts. The size of each part is same as frame size except possibly the last part. The pages of page table are then stored on different frames of the MM. To keep track of the frames storing the pages of the divided page table, another page table is maintained. As a result, the hierarchy of page tables get generated. Multilevel paging is done till the level is reached where the entire ~~the~~ page table can be stored on a single frame.

Illustration.

Logical addr. space = 4 GB

Physical addr. space = 16 TB

Page size = 4 KB.

No. of bits in Physical addr. $= \log_2(16\,TB)$

$= 44.$

No. of frames in MM $= \frac{16\,TB}{4\,KB}$

$= 2^{32}$

No. of bits in frame no. = 32.

No. of bits in offset = 44 - 32 = 12.

No. of pages the process is divided =

$$\frac{4GB}{4KB} = 2^{20}$$

Now, inner page table keeps track of the frames storing the pages of process.

Inner page table size $= 2^{20} \times 32$ bits

$= 4\,MB$

Observe, the size of inner page table is greater than the frame size (4KB). Thus, inner page table has to be divided into pages.

No. of pages of inner page table = $\frac{4\,MB}{4KB}$

$= 2^{10}$

These 2^{10} pages of inner page table are stored in different frames of the MM.

No. of page table entries in one page of inner page table =

$$\frac{\text{page size}}{\text{page table entry size}} = \frac{4KB}{32\,bits} = 2^{10}$$

One page of inner page table contains 2^{10} entries.

No. of bits reqd. to search a particular entry in one page of inner page table

$= 10$ bits

Now, outer page table is reqd. to keep track of the frames storing the pages of inner page table.

Outer page table size =

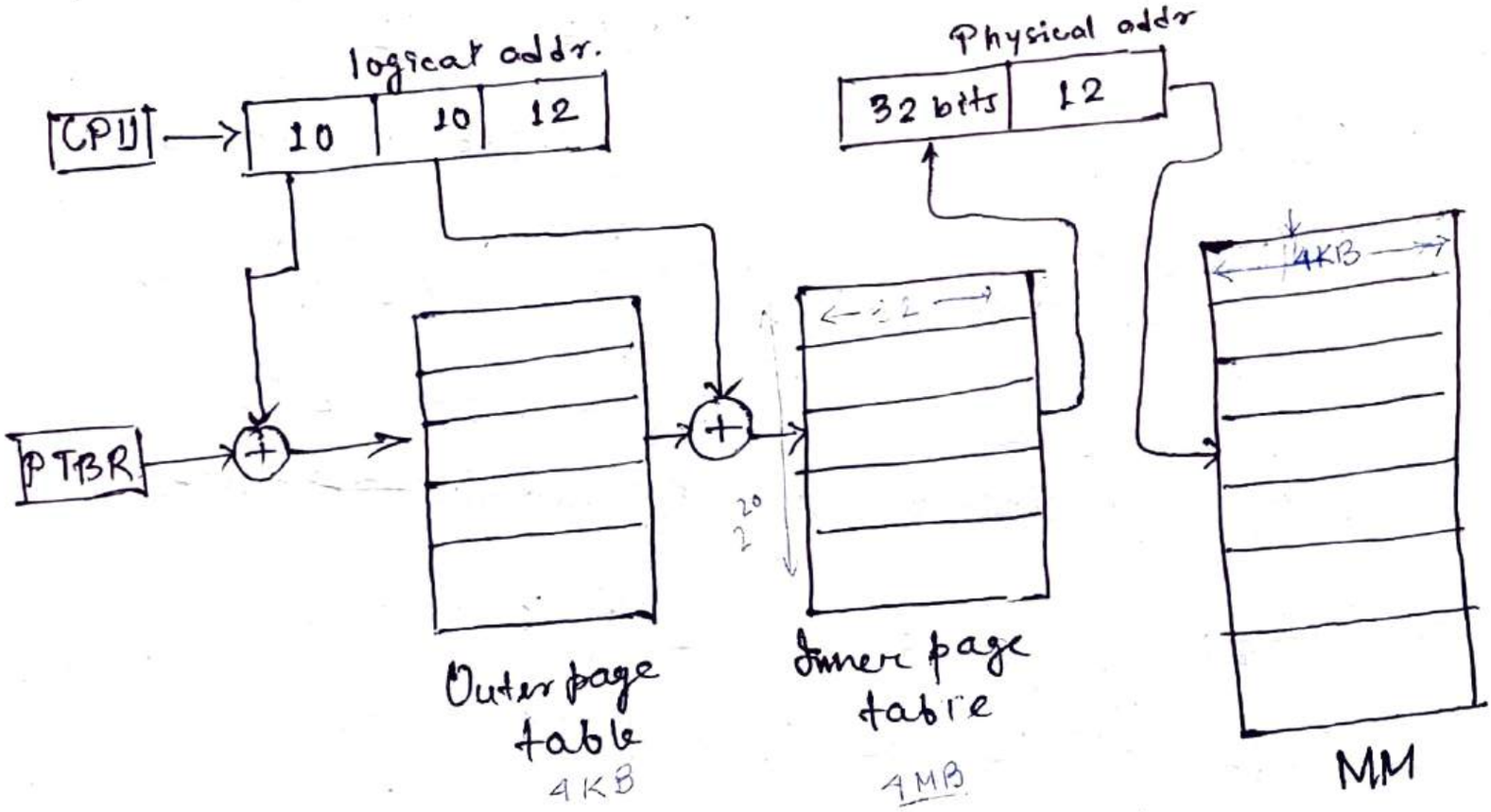
No. of pages the inner pagetable is divided × No. of bits in frame no.

$= 2^{10} \times 32$ bits $= 4$ KB.

Observe, the size of outer page table is same as frame size (4KB). Thus, outer page table is stored in a single frame. So, for a given system, we will have 2 levels of page table. PTBR will store the base address of the outer page table.

Outer page table contains 2^{10} entries.

No. of bits reqd. to search a particular entry an outer page table = 10 bit



eg. Consider system using multilevel paging. The page size is 16 KB. The memory is byte addressable & virtual address is 48 bits long. Page table entry size is 4 B. How many levels of page table reqd?

→ No. of bits in frame no = $\log_2(4B)$

$= 4B$

$\neq \log_2 2$

$= 32$ bits.

No. of frames in MM $= 2^{32}$ frames.

Size of MM $= 2^{32} \times 16$ KB

$= 64$ TB $= 2^{46}$ B

No. of bits in physical addr. = 46 bits.

No. of bits in offset = $\log_2(16KB)$

$= 14$ bits.

No. of bits in virtual addr. = 48 bits.

Process size $= 2^{48}$ B $= 256$ TB

No. of pages of process $= \frac{256\ TB}{16\ KB} = 2^{34}$ pages

[no. of pages the process is divided into

Inner page table size $= 2^{34} \times 4$ B

$= 2^{36}$ B $= 64$ GB.

No. of pages of inner page table $=$

$$\frac{64 \text{ GB}}{16 \text{ KB}} = 2^{22} \text{ pages.}$$

No. of page table entries on one page of inner page table $= \frac{16 \text{ KB}}{4 \text{ B}} = 2^{12}$ entries.

No. of bits reqd. to search an entry in one page of inner page table = 12 bits.

Q) Outer page table-1 size $= 2^{22} \times 4$ B

$= 16$ MB

No. of pages the outer page table-1 is devided $= \frac{16 \text{ MB}}{16 \text{ KB}} = 2^{10}$ pages

No. of page table entries on one page of outer page table-1 $= \frac{16 \text{ KB}}{4 \text{ B}}$

$= 2^{12}$ entries.

No. of bits reqd. to search a particular entry on one page of outer page table-1

$= 12$ bits

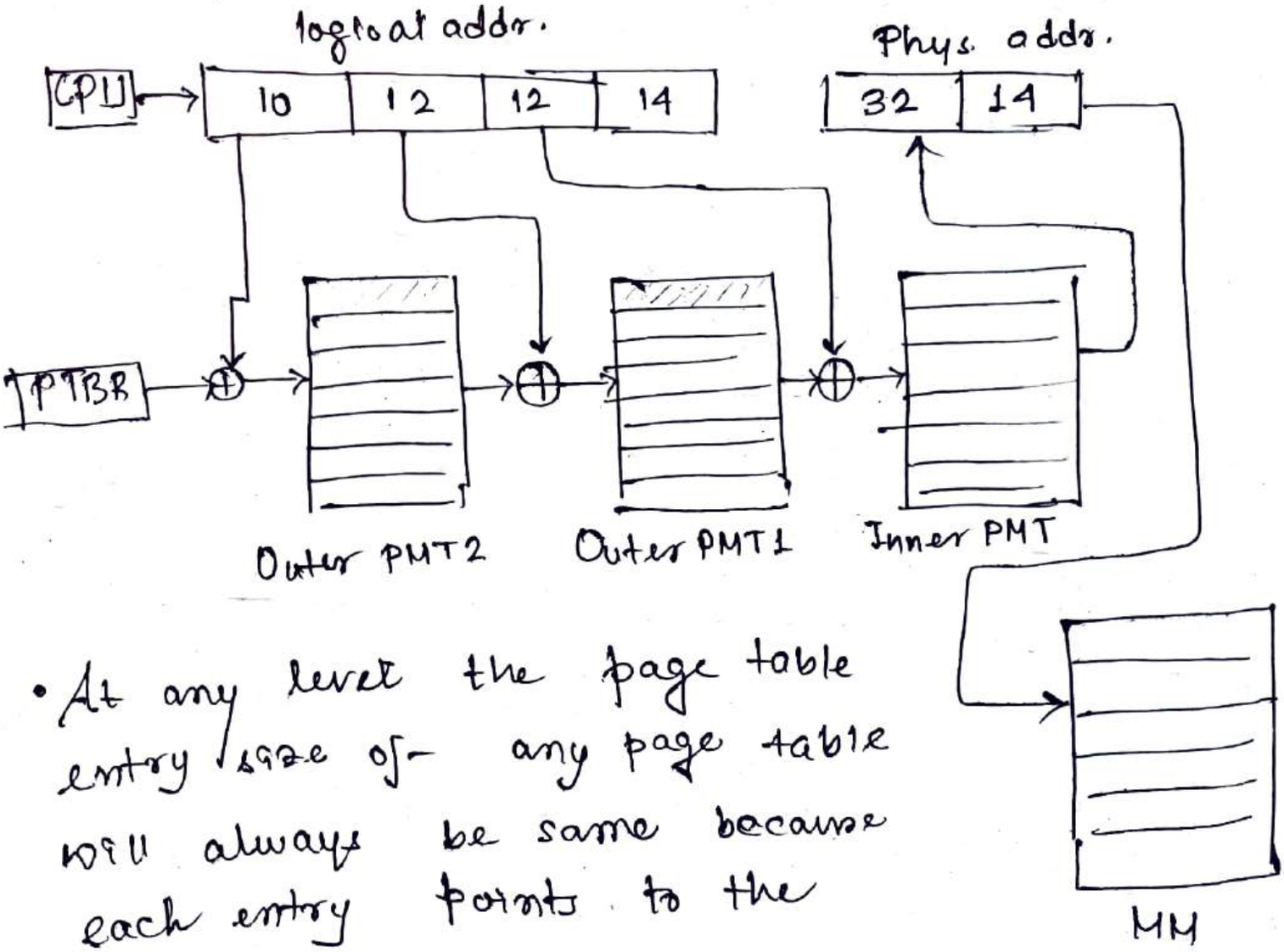
Outer page table-2 size = $2^{10} \times 4B$

$= 4KB$

Outer page table-2 size is less than the frame size (16KB).

No. of bits reqd. to search an entry in outer page table-2 = 10 bits

[as there are 2^{10} entries]



- At any level the page table entry size of any page table will always be same because each entry points to the frame no.

• When there is only one level of paging, there is only one page table whose size is less than or equal to frame size.

* Page fault : When a page referenced by the CPU is not found on the MM it is called page fault. When a page fault occurs, the reqd. page has to be fetched from the secondary memory onto the MM.

Translating logical address into Physical address.

In a paging scheme using TLB with possibility of page fault, the logical addr. generated by the CPU is translated onto the physical addr. using the following steps—

i) CPU generates logical address.

ii) TLB checked if contains an entry for the referenced page no. If TLB hit, TLB entry used. If TLB miss, page table used to get the frame no.

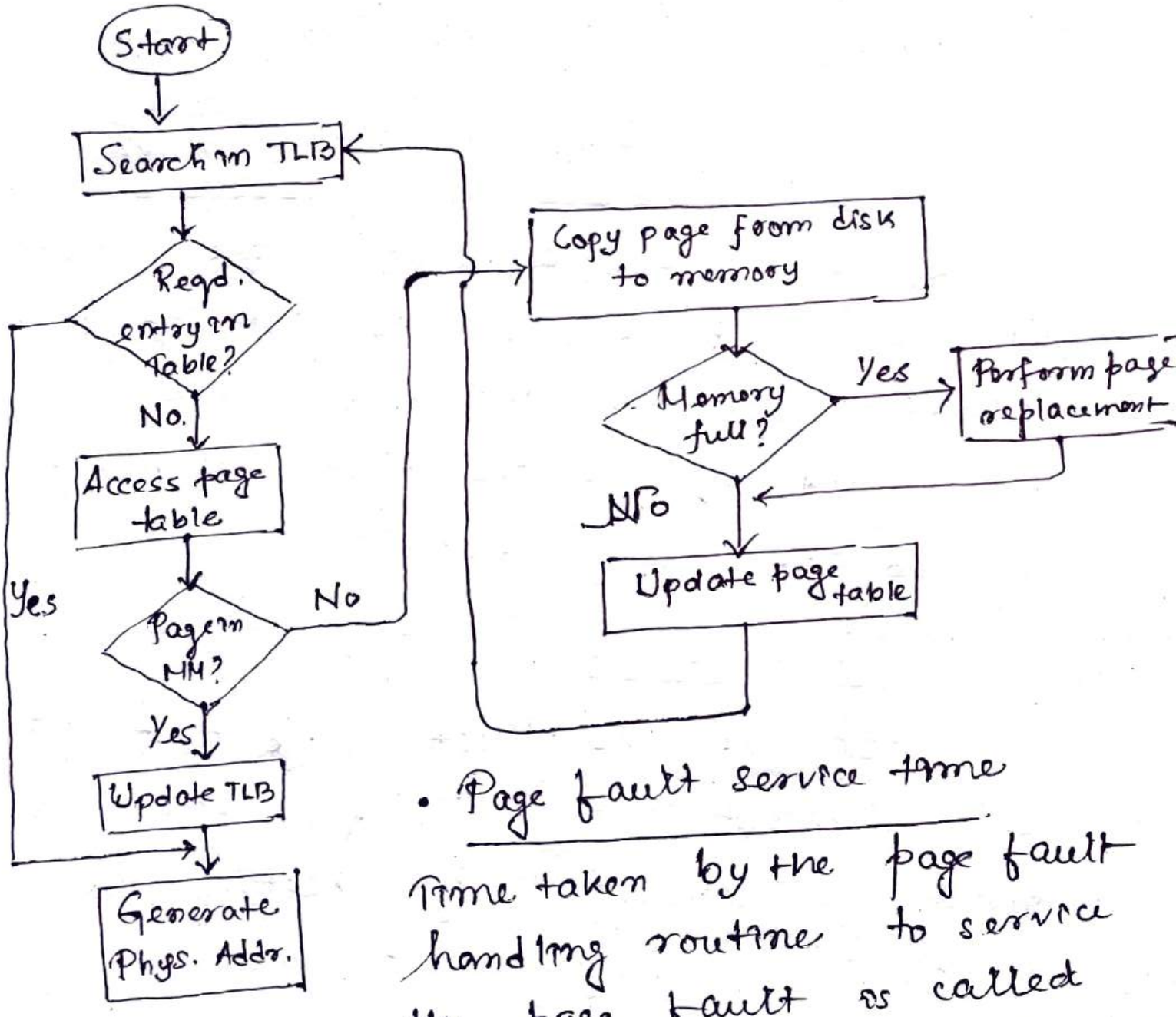
The valid/invalid bit of the page table entry indicates whether the referenced page is present on MM or not.

If valid/invalid bit is set to 1, it indicates that the page is present in the MM. Then, PMT is used to get the frame no. for the referenced page no. TLB is updated with the page no. & its frame no. for future reference

If valid/invalid bit is set to 0, it indicates that the page is not present in the MM. A page fault occurs. The occurrence of page fault calls the page fault interrupt which executes the page fault handling routine.

The currently running routine is stopped & context switching occurs. The referenced page is copied from the sec: memory to the MM. If the MM is full, a page is replaced to create room for the referenced page. After copying the referenced page successfully in the MM, the page table is updated. When the exec^{n} of process is resumed, step-2 repeats.

iii) After the frame no. is obtained, it's combined with page offset to generate the physical memory address.



- Page fault service time

Time taken by the page fault handling routine to service the page fault is called as page fault service time. Page fault service time is much greater than MM access time. It increases EAT.

* Page Replacement. (GV.)

Process of swapping out an existing page from the frame of a MM & replacing it with the reqd. page.

→ Page replacement algorithms

Helps to decide which page must be swapped out from the MM to create a room for the incoming page.

A good algo. is one that minimizes the no. of page faults.

1. FIFO algo.

Replaces the oldest page that has been present in the MM for the longest time. Implemented by keeping track of all the pages in a queue.

2. LIFO algo

Replaces the newest page that arrived at last in the MM. It's implemented by keeping track of all pages in a stack.

3. LRU Algo (Least recently used)

Replaces the page that has not been referred by the CPU for the longest time.

4. Optimal Algo

Replaces the page that will not be referred by the CPU in future for the longest time. Practically impossible to implement. Such pages can't be predicted

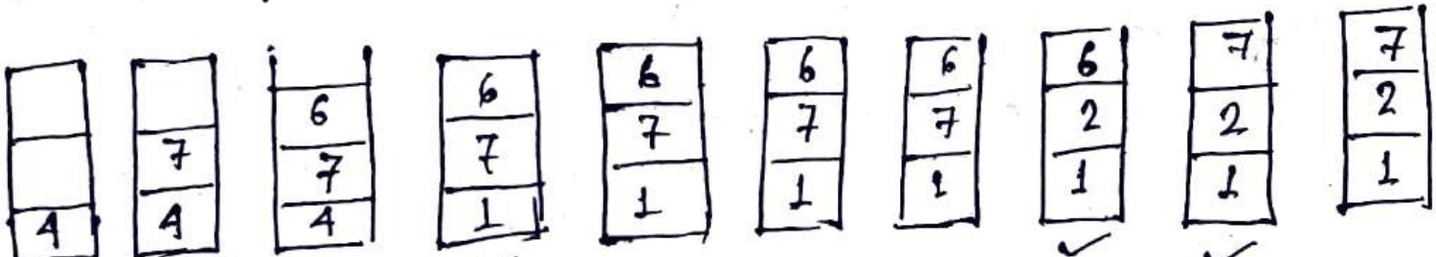
5. Random Algo

Randomly replaces any page.

eg. 3 page frames for storing process pages in MM. Uses FIFO. Assume all frames are initially empty. Total no. of page faults that will occur while processing the page reference string given

4, 7, 6, 1, 7, 6, 1, 2, 7, 2

→ No. of references = 10

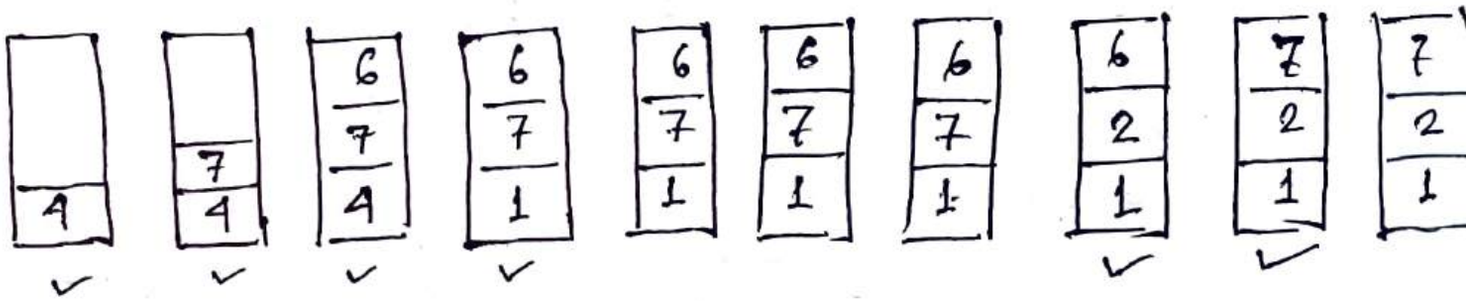


4 7 6 1 7 6 1 2 7 2

No. of page faults = 6.

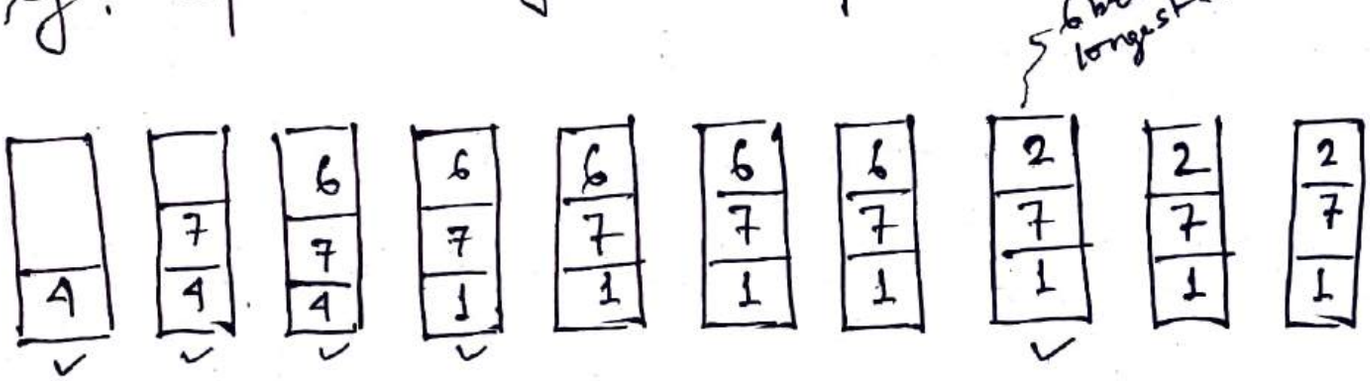
eg. LRU on previous problem.

4, 7, 6, 1, 7, 6, 1, 2, 7, 2.



Total page faults = 6.

eg. Optimal algo on previous.



Total page faults = 5.

*Belady's Anomaly

eg. 0 1 2 3 0 1 4 0 1 2 3 4
Check with 3 & 4 frames using FIFO

(GV) Phenomenon of increasing number of page faults on increasing the no. of frames in MM.

FIFO, random & second chance algorithms (page replacement) suffer from Belady's anomaly.

Unusual behaviour of increasing page faults is observed only sometimes.

Reason. An algo suffers from this iff it does not follow stack property. Stack based algos don't suffer from Belady's anomaly.

LRU, optimal algos don't suffer from this anomaly. This is because these algos assign priority to a page for replacement that is independent of the no. of frames in the MM.

The set of pages that were present in the MM, when no. of frames is m will be compulsorily present in the corresponding stages in MM when the no. of frames is increased to m+1.

*Important results about page replacement algorithms (GV)

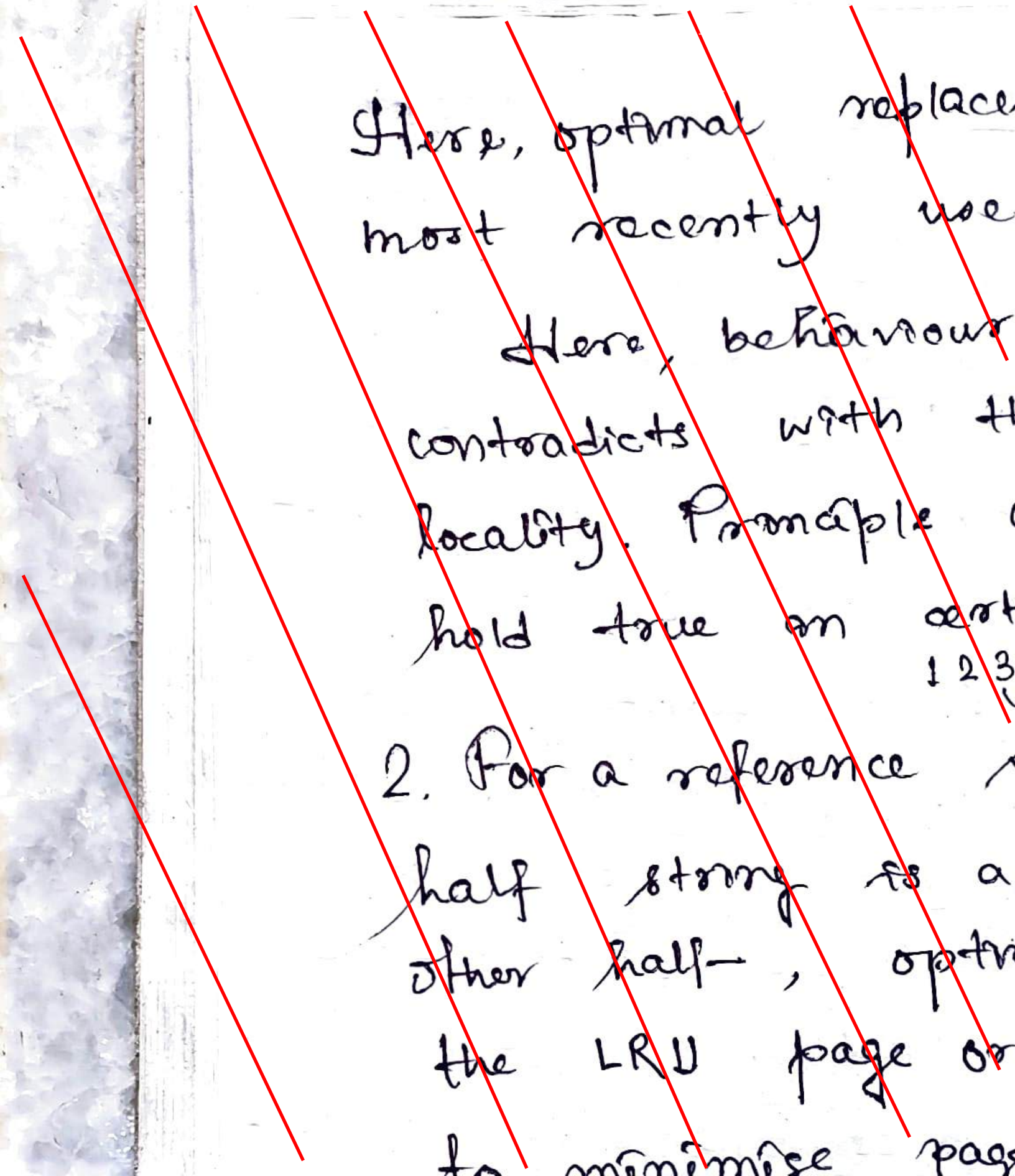
1. For a reference string consisting of repeated sequence of page numbers, optimal algo replaces the most recently used page to minimise page faults.

Here optimal replacement algo acts as Most Recently Used (MRU) algo.

Behaviour of optimal algo contradicts with the principle of locality. Principle of temporal locality does not hold true in certain scenarios.

1 2 3 1 2 3 -> 3 will be replaced (recently used)

2. For a reference string where first half of the string is a mirror image of the other half (1 2 3 3 2 1), optimal algo replaces the LRU page or firstly arrived page to minimise page faults. Thus, optimal algo acts as LRU and FIFO algo.



Thrashing - ↑ Degree of multiprogramming ⇒ CPU utilisation increased to a certain limit then it decreases for highere page faults.

Solution:

i) Working set model: Working set is a collection of pages a process is using actively & which must thus be memory-resident to prevent this process from thrashing. If the sum of all working sets of all runnable threads exceeds the size of memory, then stop running some of the threads for a while. It's based on concept of locality. Working set defines a working set window of length delta. Whatever pages are included in the most recent delta page references are said to be in the process's working set window, & comprise its current working set.

Selection of delta is critical. If it is too small then it does not encompass all of the pages of the current locality & if it's too large, then it encompasses pages that are no longer being frequently used (also catches multiple localities).

The total demand D is the sum of the sizes of working sets for all processes. If D exceeds the total number of available frames, then at least one process is thrashing, because there are not enough frames available to satisfy minimum working set. If D is significantly less than the current available frames, then additional processes can be launched.

Δ too small → Not encompass entire locality

Δ too large → Overlap several localities

Total demand $D = \Sigma WSS_i$

Working set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, optimizes CPU utilisation. Difficulty with working set model is keeping track of the working set. Working-set window is a moving window. At each memory reference, a new reference appears at one end, & the oldest reference drops off the other end. A page is in working set if it is referenced anywhere in the working-set window.

eg. Page reference stream - Window size 6

3 2 4 3 4 2 2 3 4 5 6 7 7 6 5 4 5 6 7 2 1.

WS [3] [3 2] [3 2 4] [2 4 3] [2 3 4] [3 4 2] [3 4 2] [4 2 3]

Page fault: 1, 2 (1+1), 3 (2+1), 3, 3, 3, 3, 3

[2 3 4] [2 3 4 5] [2 3 4 5 6] [2 3 4 5 6 7] [3 4 5 6 7] imp

3, 4 (3+1), 5, 6, 6

[4 5 7 6] [7 6 5] [7 6 5 4] [7 6 4 5] [7 4 5 6]

6, 6, 7, 7, 7

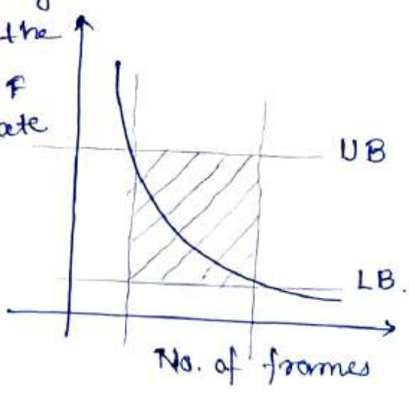
[4 5 6 7] [4 5 6 7 2] [4 5 6 7 2 1]

7, 8

(9) Ans. Total page faults.

ii) Page fault frequency: If PF rate is too high it indicates that the process has too few frames allocated to it. A low PF rate indicates that the has too many frames.

Upper & lower limits can be established on the desired page fault rate. If the PF rate falls below the lower limit, frames can be removed from the process. Similarly, if PF rate exceeds upper limit, more no. of frames can be allocated to the process.



Graphical state of the system should be kept limited to the rectangular region formed.

Buddy System :- Memory allocation and management algorithm that manages memory in power of two increments. Larger memory block is divided into small parts to best fit the request. Two smaller parts of block are of equal size and called as buddies. In the same manner one of 2 buddies will further divide into smaller parts until the request is fulfilled.

+ Memory size being 2^U. and size of S is reqd.
- If $2^{U-1} < S \leq 2^U$ allocate the whole block.
- else recursively divide the block equally and test the condition at each time. When satisfied, allocate block and get out of loop.

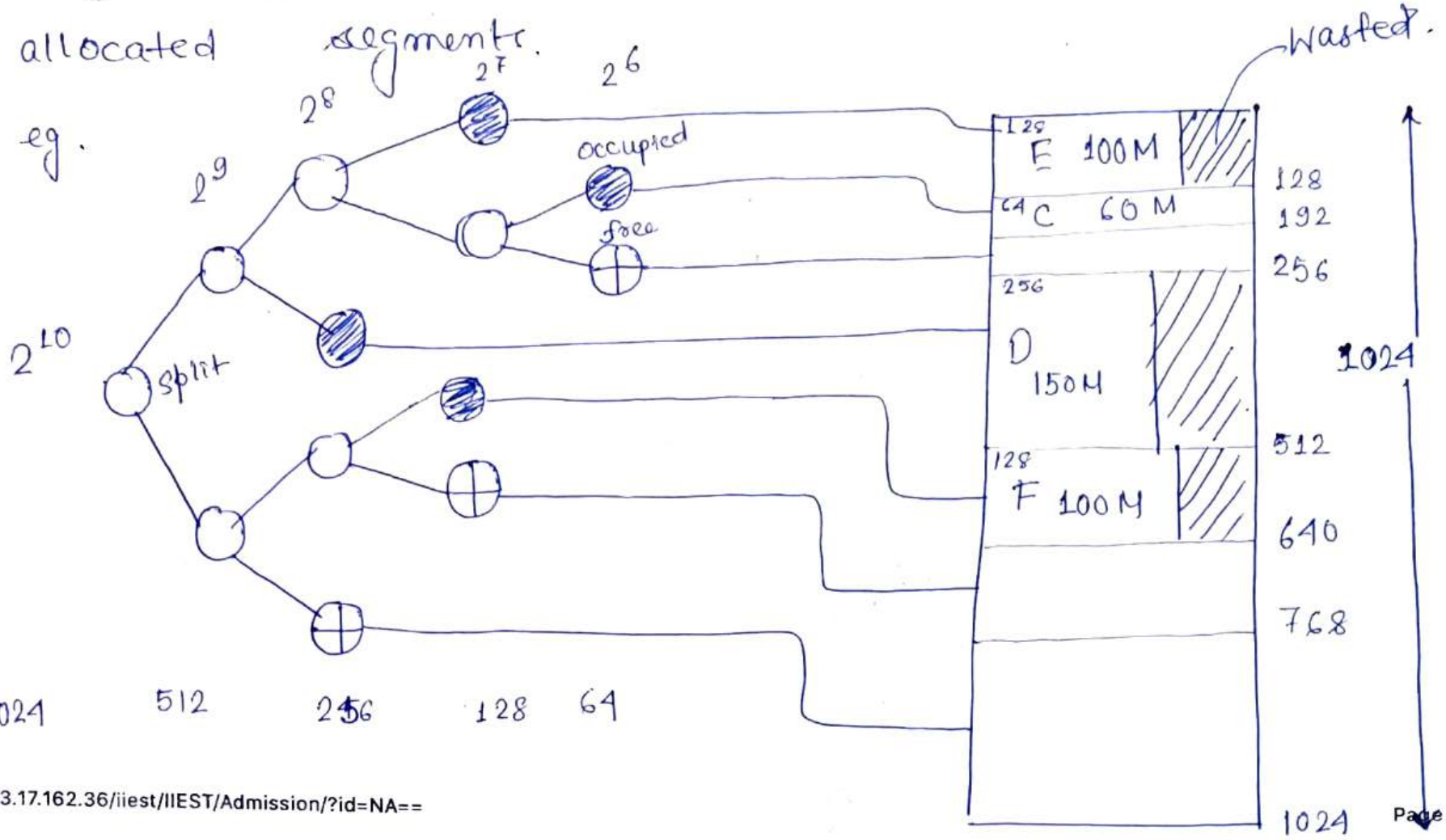
System keeps the record of all unallocated blocks each & can merge these to make one big chunk.

+ 4 types of Buddy system.
i) Binary ii) Weighted iii) Tertiary
iv) Fibonacci : Blocks divided into sizes that are fibonacci numbers. ($Z_i = Z_{i-1} + Z_{i-2}$).

+ Advantages. : i) In comparison to other simpler techniques such as dynamic alloc^n, buddy system has little external fragmentation.
ii) Buddy memory system is implemented with the use of Binary tree to represent used, unused split memory blocks. iii) Buddy system is very fast to allocate & deallocate memory. iv) Addr. calculation easy. v) Adjacent buddies can be quickly combined to form larger segments using coalescing.

+ Drawback: Internal fragmentation - rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments.

eg.

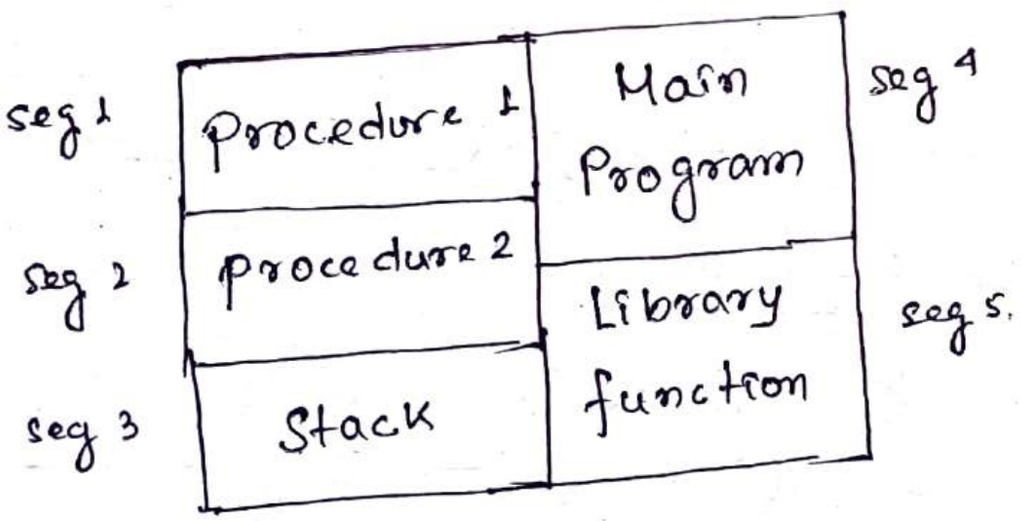


* Segmentation. Non contiguous memory allocation technique, where process isn't divided blindly into fixed size pages. Process is divided into modules for better visualization.

• Characteristics.

Segmentation is a variable size partitioning scheme. Sec. memory & MM are divided into partitions of unequal size. Size of partitions depend on the length of modules. Partitions of sec. memory are called as segments.

eg. Program divided into 5 segments



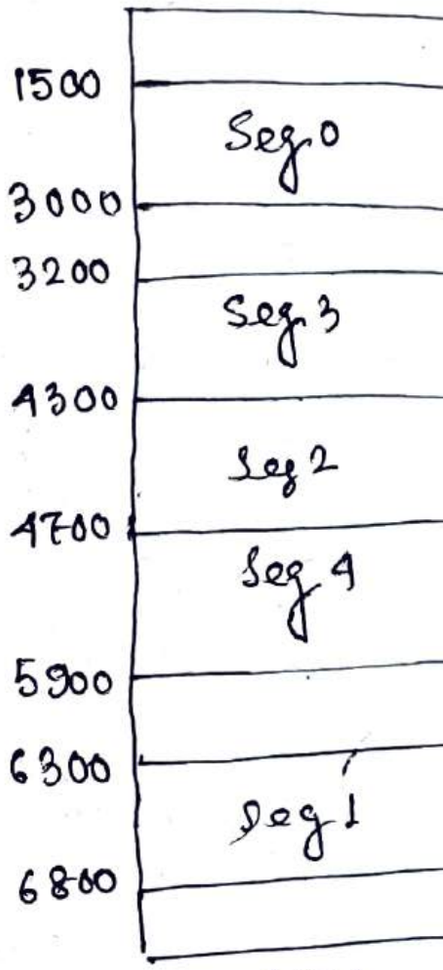
• Segment Table.

Stores the information about each segment of a process. Two columns — first stores the size or length of segment, second stores the base addr. of segment in MM. Segment table is stored as a separate segment in the MM. Segment table base register (STBR) stores base addr. of segment table.

eg.

	Limit	Base
Seg 0	1500	1500
Seg 1	500	6300
Seg 2	400	4300
Seg 3	1100	3200
Seg 4	1200	4700

Segment table.



MM.

- Translating Logical addr. into Physical addr.

1. CPU generates a logical addr. consisting of 2 parts — segment no. (specifies the specific segment of the process from which CPU wants to read the data), segment offset (specifies the specific word in the segment that CPU wants to read).

2. For the generated segment no., corresponding entry is located in the segment table. Then, segment offset is compared with the limit of the segment. Now, 2 cases arise —

2a. Segment offset ≥ limit

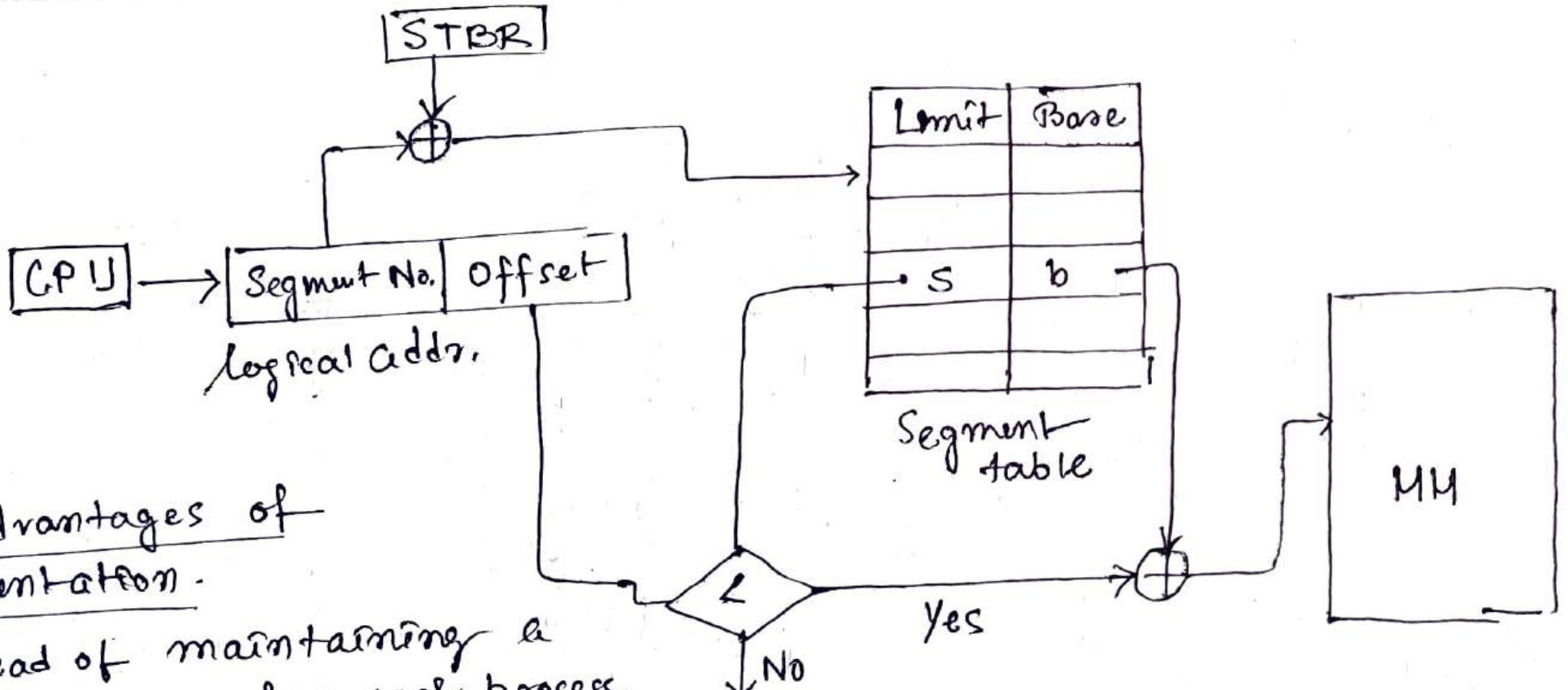
A trap is generated.

2b. Segment offset < limit

Valid request. Segment offset is added with the base addr. of the segment. The result obtained after addition is the address of the memory loch storing the reqd. word.

- Advantages.

Allows to divide the program into modules which provides better visualization. Segment table consumes less space as compared to PMT. Solves external fragmentation.



- Disadvantages of Segmentation.

Overhead of maintaining a segment table for each process.
Time taken to fetch instn increases since 2 mem. accesses are reqd.
Segments of unequal size are not suitable for swapping. It suffers from external fragmentation as the free space gets broken down into smaller pieces with the processes being loaded & removed from the MM.

* Segmented Paging:

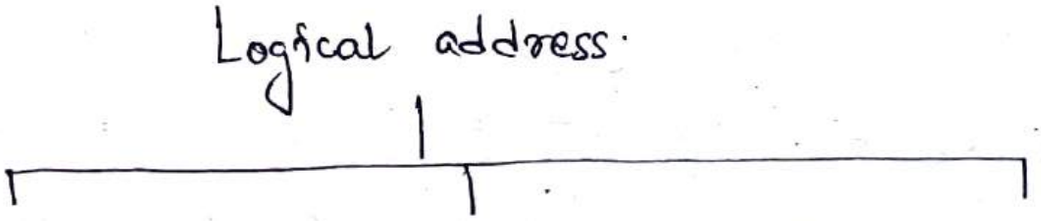
Implements combination of paging & segmentation.

- Working. Process is first divided into segments & then each segment is divided into pages. These pages are then stored on the frames of MM. A PMT exists for each segment that keeps track of the frames storing the pages of that segment. Each PMT occupies one frame on the MM. No. of entries in the page table of a segment = no. of pages that segment is divided. A segment table exists that keeps track of the frames storing the page tables of segments. No. of entries in the segment table of a process = no. of segments that process is divided. The base address of the segment table is stored in the segment table base register (STBR).

- Translating logical address into physical address.

CPU always generates a logical address. A physical address is needed to access the main memory.

Step 1. CPU generates a logical address consisting of 3 parts — Segment no., page no., page offset.



Segment no. Page no. Page offset.

- Segment no. specifies the segment from which CPU wants to read the data.
- Page no. specifies the specific page of that segment from which CPU wants to read the data.
- Page offset specifies the specific word on that page that CPU wants to read.

Step 2. For generated segment no., corresponding entry is located in the segment table. Segment table provides the frame no. of the frame storing the page table of the referred segment. The frame containing the page table is located.

Step 3. For the generated page number, corresponding entry is located in the page table. Page table provides the frame no. of the frame storing the reqd. page of the referred segment. The frame containing the reqd. page is located.

Step 4 The frame no. combined with the page offset forms the reqd. physical address. For the generated page offset, corresponding word is located in the page.

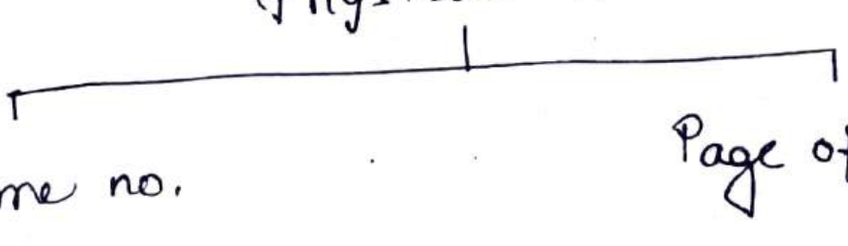
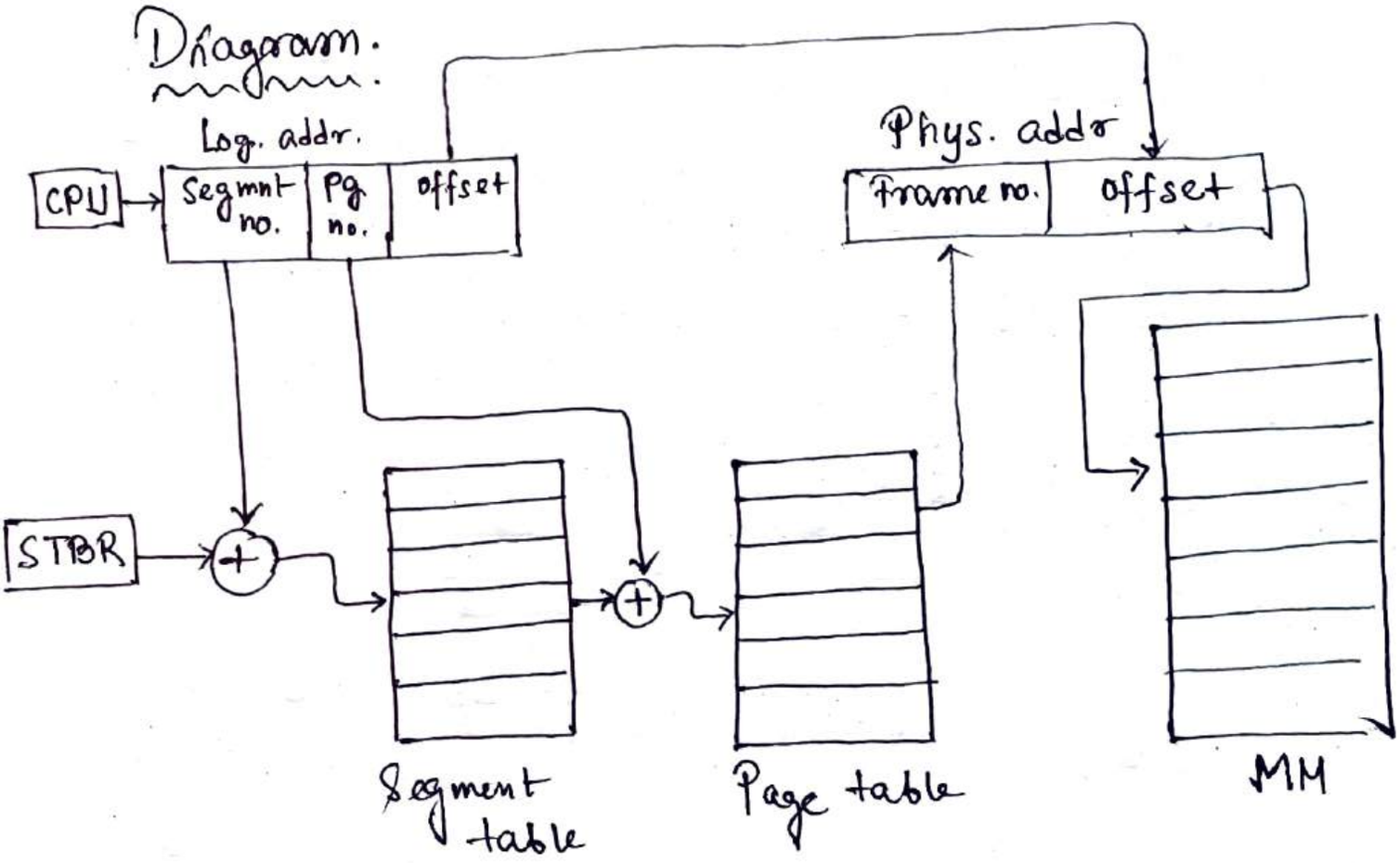


Diagram:



- Advantages.

i) Segment table contains only one entry corresponding to each segment.

ii) It reduces memory usage.

iii) The size of page table is limited by the segment size.

iv) It solves problem of external fragmentation.

• Disadvantages:

i) Suffers from internal fragmentation.

ii) Complexity level is much higher as compared to paging.

File System, IO and Protection.

* Access methods. – Sequential, Direct, Indexed access.
* File Operations: Creating, writing, reading, repositioning, deleting, truncating.
* File attributes: Name, identifier, type, location, size, protection, time.
* Information required for accessing a file.

i) File pointer ii) File open count iii) Disk location of a file iv) Access rights.

– Per process table (File ptr, access right)
– Global file table / Open file table.

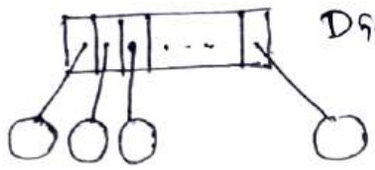
• Directory (Contains file location details/metadata

Operations – Search, create, delete, list, traverse, rename

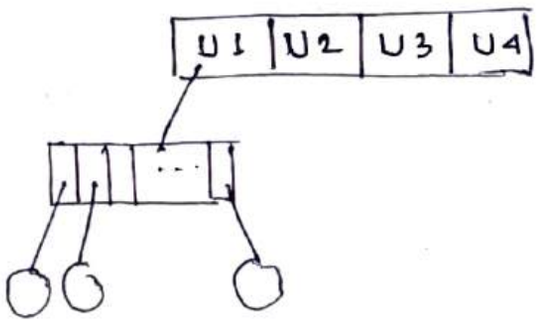
Single level & Multilevel directory (2level, Tree)

Acyclic graph, graph.

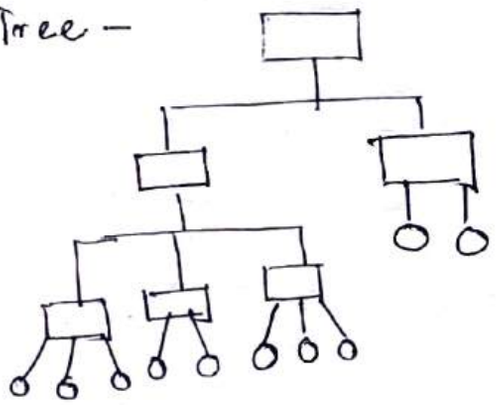
→ Single Level -



Two Level -

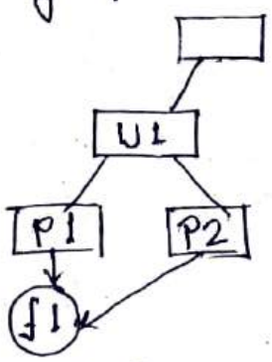


Tree -



Pathnames - absolute & relative. (in ca user in pwd & go to next leve

Acyclic graph -



- dangling pointer - proble

* File System: Provides the mechanism for online storage & access to file contents.

Issues -

i) File structure
ii) Allocating disk space
iii) Recovering freed space
iv) Tracking locations of data
v) Interface other parts of OS to secondary storage

Application programs

⇓

Logical file system (metadata, directory structure)

⇓

File organization module (logical block → physical block map)
(free space mgmt)

⇓

Basic file system (commands to the I/O control)
Buffering

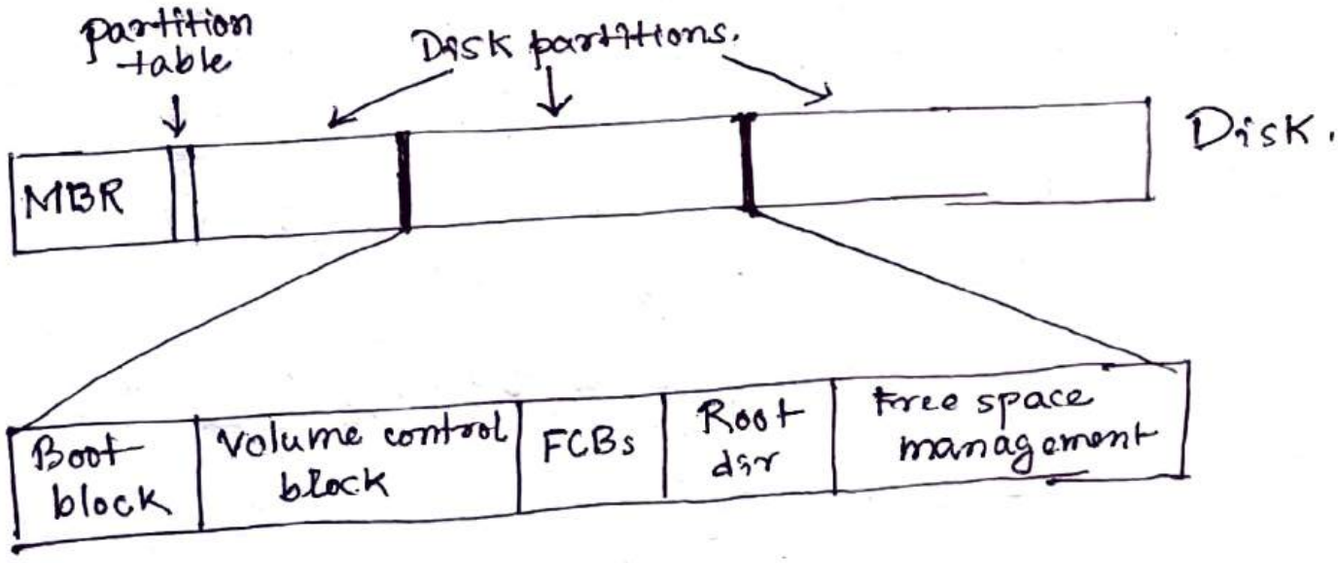
⇓

I/O control (consist device drivers, interrupt handlers)

⇓

Devices.

* MBR.



* Data structures.

Several in memory & on disk structures are used to implement a file system. These structures vary depending on the OS & the file system but general points are —

1) Boot control block (per volume):

It contains information needed by the system to boot an OS from that volume.

In UNIX file system it is called boot block.

In NTFS, it's partition boot sector.

ii) Volume control block : Contains volume details such as the number of blocks in the partitions, size of the block.

In UNIX, it's super block. In NTFS, master file table.

iii) Directory structure per file system: Used to organise the files. In UNIX, this includes file names & associated

iv) File control block (FCB) : Details about the files.

[File permissions, file dates (creation, access, write), file owner, group, ACL, file size, file data blocks/ptrs to file data blocks]

— In-memory data structures:

Loaded at mount time & discarded at dismount.

i) In memory mount table : Information about each mounted volume.

ii) In memory directory structure cache: Contains directory information of recently accessed directory.

iii) System wide open file table: Contains FCB of each open file.

iv) Per process open file table:

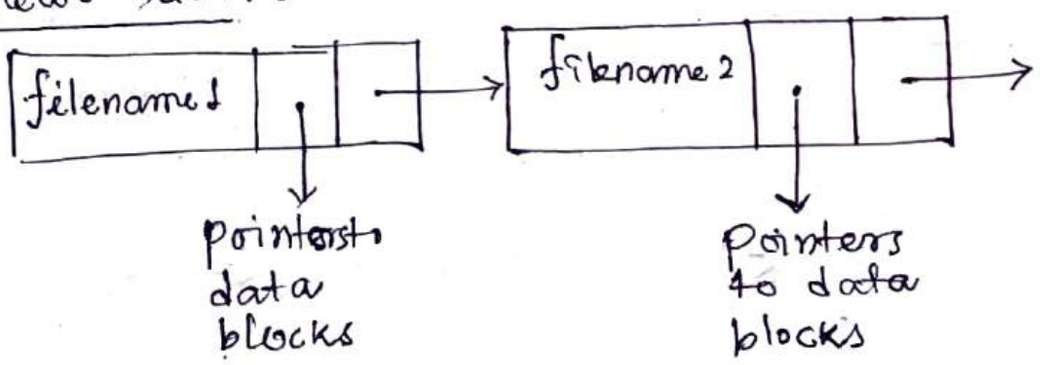
Contains the pointer to the appropriate entry on the system wide open file table.

* Directory Implementation.

Selection of directory allocation & directory mgmt. algorithms significantly affects the performance & reliability of file system.

Algorithms.

i) Linear list.:

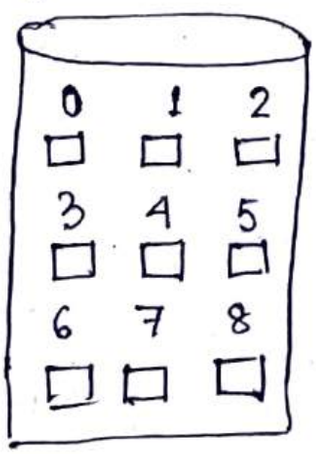


ii) Hash table.

* Allocation methods.

Allocating space to the files so that disk space is utilised effectively & files can be accessed quickly.

- Contiguous allocation -



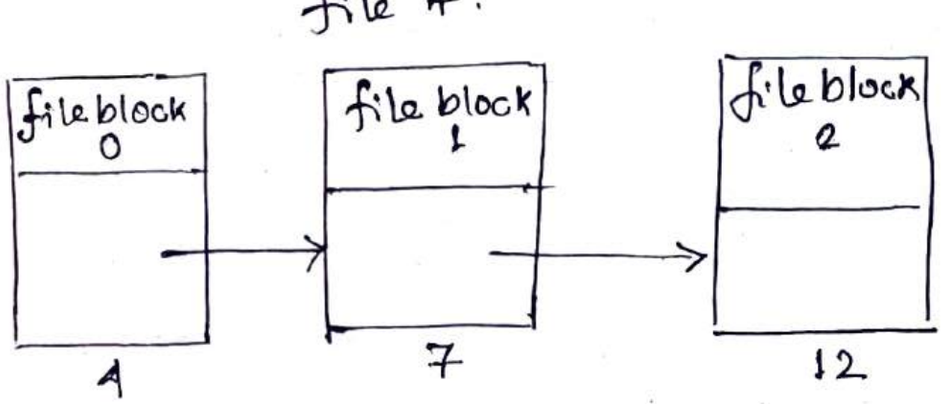
directory

file	start	length
temp	0	2
tr	3	3
list	6	1

Adv. Sample to implement, good read performan

Disadv. Disk becomes fragmented.

- Linked list allocation -



physical blo

Adv. Each disk block can be used.
No space lost due to disk fragmentatio

Disadv. Random access is slow.
Pointer takes up few bytes.

- File Allocation Table:

Every entry shows where the next block of the file is. (Implemented on LL allocation). In-memory data structure.

adv. random access easier.

Q. G'14. A FAT based file system is being used & the total overhead of each entry in the FAT is 4B in size. Given a 100×10^6 B disk on which the file system is stored & data block size 10^3 B, the max size of a file that can be stored on disk in units of 10^6 B is 99.6

FAT size = $\frac{100 \times 10^6}{10^3} \times 4 = 0.4 \times 10^6$ B.

Available space = $(100 - 0.4) \times 10^6$ B

= 99.6×10^6 B.

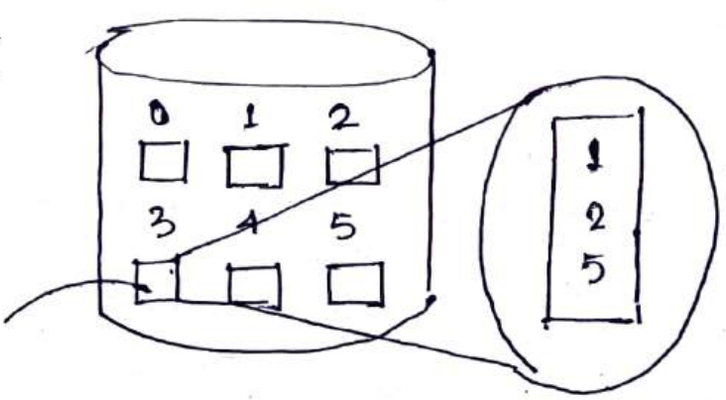
* Indexed allocation:

Linked allocation solves the external fragmentation & size declaration problem of contiguous allocation.

In the absence of FAT, linked allocation can't support efficient direct access.

Indexed allocation solves the problem by bringing all the pointers together into one loc^n: the index block.

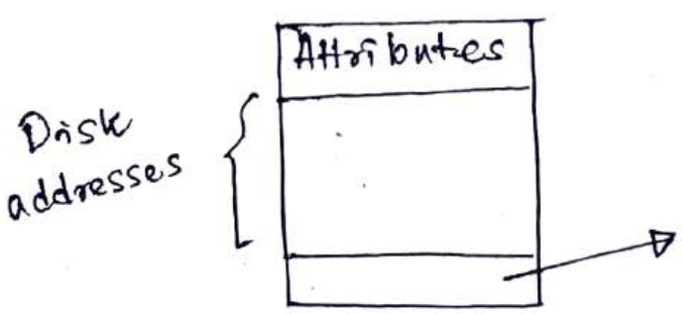
File	Index Name
File A	3



Index block

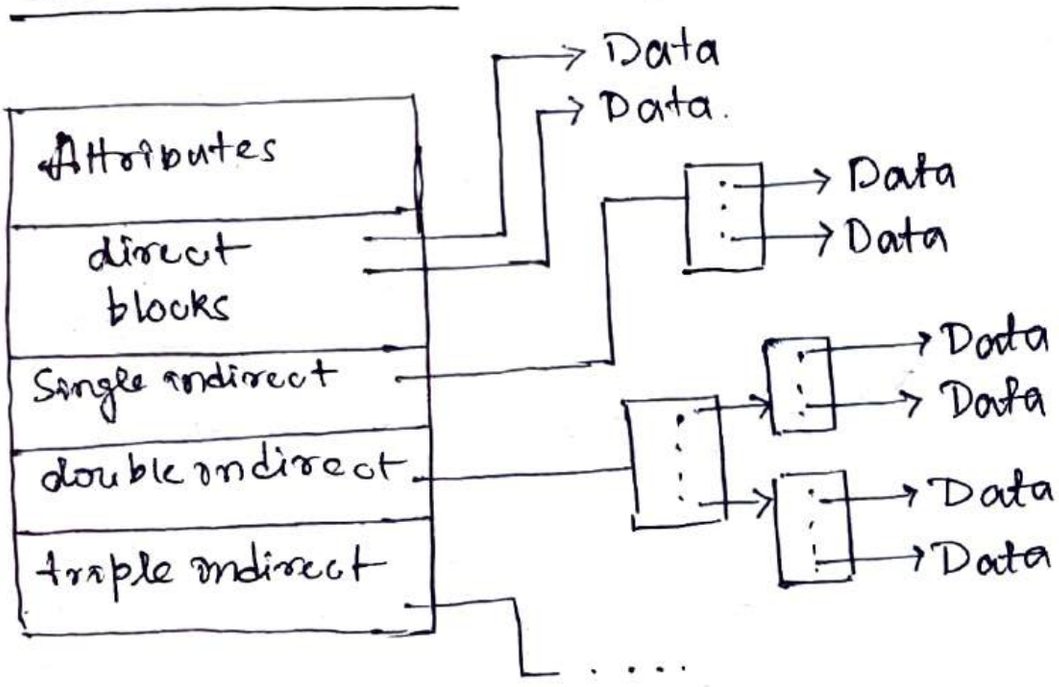
- If the index block is too small, however it will not be able to hold enough ptrs to hold for a large file.

Linked schema - We can link several index blocks.



Also multilevel index is possible.

- Combined scheme.



G'04. A unix style inode has 10 direct ptrs & one single, one double & one triple indirect pointer Disk block size is 1KB, disk block address is 32b & 48b integers are used. What's the max possible file size?

→

$(10 + 170 + 170^2 + 170^3) \times 1024\,B$

Size of disk block = 1024 B

Disk block address = 4B

No of addresses / block $= \frac{1024}{4} = 2^8$ addresses.

1 SI = 2^8 Indirect $\times 2^{10} = 2^{18}$ B

1 DI = $(2^8)^2 \times 2^{10} = 2^{26}$ B

1 TI = $(2^8)^3 \times 2^{10} = 2^{34}$ B.

$(10 + 2^{18} + 2^{26} + 2^{34}) \approx 2^{34}$ B (Ans).

G'12 A file system with 300 GB uses a file descriptor with 8 direct block address. One indirect block address & one doubly indirect block address. The size of each disk block is 128 B & the size of each disk block address is 8B. The max. possible file size in the file system is —

→ Direct block addressing will point to 8 disk blocks =

$$8 \times 128B = \underline{1KB}.$$

Singly indirect block addressing will point to 1 disk block that has $\frac{128}{8}$ disc block addresses

$$= \frac{128}{8} \times 128\ B = \underline{2KB}$$

Doubly indirect block addressing will point to 1 disk block that has $\frac{128}{8}$ addresses to disk blocks that in turn has $\frac{128}{8}$ n to disk blocks =

$$16 \times 16 \times 128\ B = \underline{32KB}$$

Total = 35KB.

* Free Space Management.

1. Bit vector. : The free space list is implemented as a bit map or bit vector. Each block is represented by 1b. If the block is free, bit is 1 else 0.

2. Linked list. : Free list. Link together the free disk blocks keeping a pointer to the first free block in a special location on the disk & caching it in memory.

* Disk Scheduling:

→ magnetic disk in COA (GV)

Technique used to schedule multiple requests for accessing the disk.

Purpose of disk scheduling algo is to reduce the total seek time.

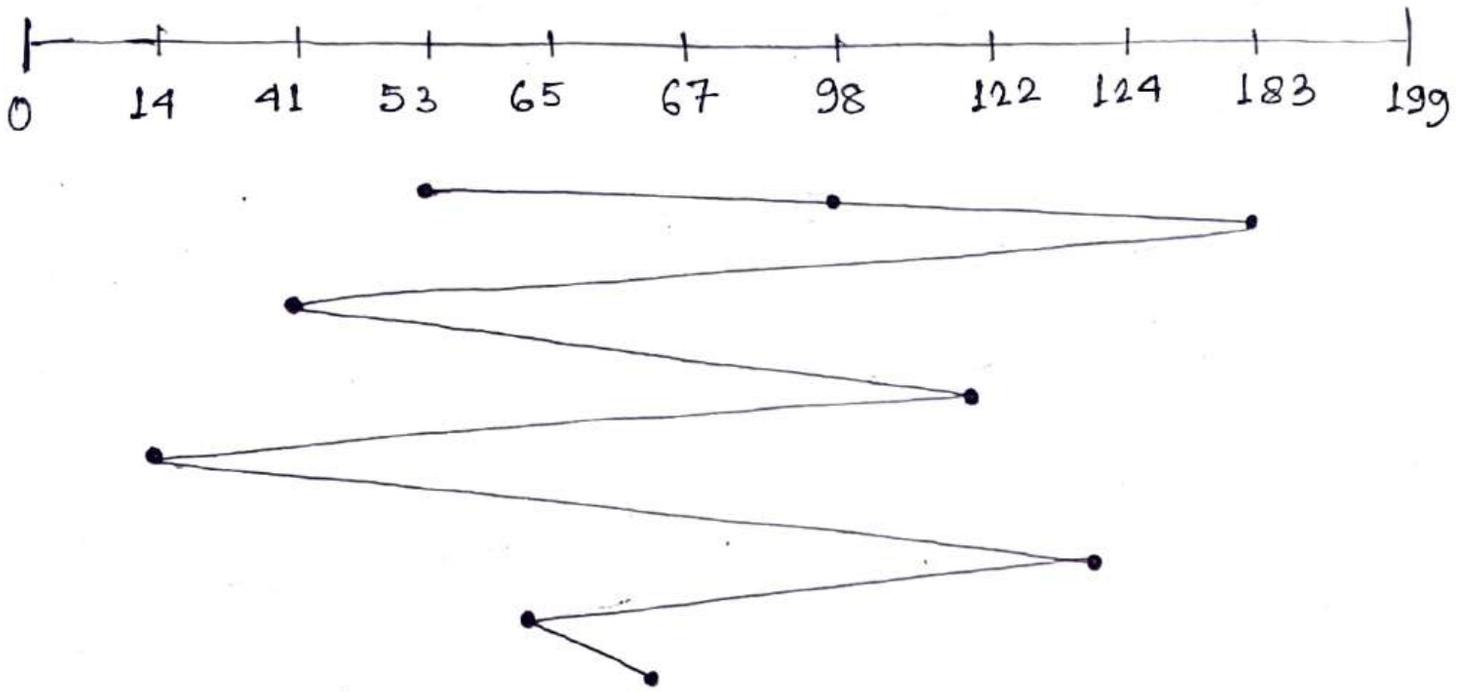
- Various algorithms :-

i) **FCFS** : Requests in the order they arrive in the disk queue.

adv. Doesn't cause starvation to any request. (may suffer from convoy effect)

disadv. Results in increased total seek time. It's inefficient.

Q. Disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The FCFS algo is used. The head is initially at cylinder no 53. The cylinders are numbered from 0 to 199. The total head movement (in no. of cylinders) incurred while servicing these requests is —



Total head movements =

$$(98-53)+(183-98)+(183-41)+(122-41)+(122-14)+$$
$$(124-14)+(124-65)+(67-65) = 632.$$

ii) SSTF (Shortest seek time first):

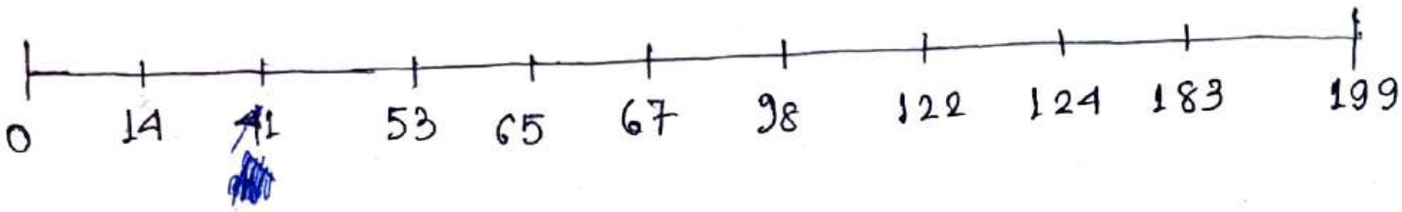
Services that request next which requires least number of head movements from its current position regardless of the direction. It breaks the tie in the direction of head movement.

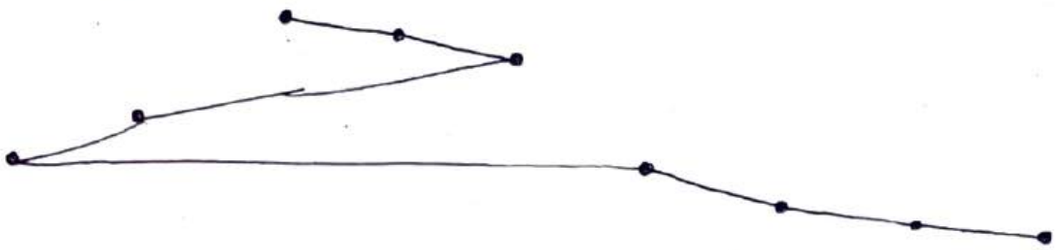
adv.: Reduces total seek time as compared to FCFS. Provides increased throughput. Provides less avg response time & waiting time.

disadv.: Overhead of finding out the closest request. The requests which are far from the head might starve for the CPU. Provides high variance in response time & waiting time. Switching direction of head frequently slows down the algo

Q. 98, 183, 41, 122, 14, 124, 65, 67

Head initially at 53 cylinder





Total head movements incurred =

$$(65-53) + (67-65) + (67-41) + (41-14) + (98-14) + (122-98) + (124-122) + (183-124)$$

$$= 236.$$

iii) SCAN algorithm.

Scans all the cylinders of the disk back & forth. Head starts from one end of the disk & move towards the other end servicing all the requests in between. After reaching the other end, head reverses its direction & move towards the starting & servicing all requests in between. Same repeats.

Also called elevator algorithm.

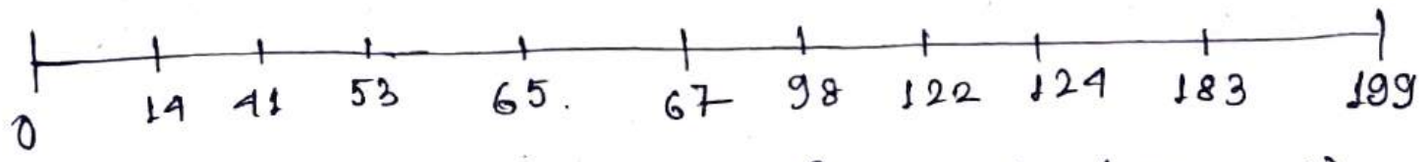
adv.: It does not lead to starvation. Low variance in response time & waiting time

disadv.: Causes long waiting time for the cylinders just visited by the head.

Causes the head to move till the end of the disk even if there are no requests to be serviced.

Q. 98, 183, 41, 122, 14, 124, 65, 67.

Head at 53.



Total head movements = (199 - 53) + (199 - 14)

= 331.

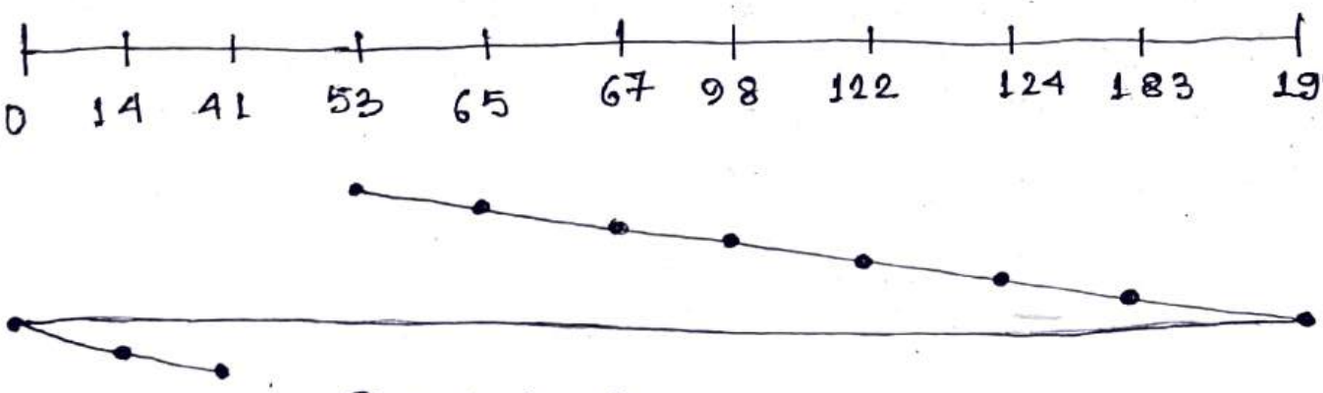
iv) C-SCAN

Head starts from one end of the disk & move towards the other end servicing all the requests in between. After reaching the other end, head reverses its direction. It then returns to the starting end without servicing any request in between.

adv.: Waiting time for the cylinders just visited by the head is reduced as compared to SCAN. Provides uniform waiting time, better response time.

disadv.: More seek movements as compared to SCAN. Causes the head to move till the end of the disk even if there are no requests to be serviced.

Q.5)



Total head movements =

(199 - 53) + (199 - 0) + (41 - 0) = 386.

v) LOOK algorithm.

Improved version of SCAN. Head starts from the first request at one end of the disk & moves towards the last request at the other end servicing all the requests in between. After reaching the last request at the other end, head reverses its direction. It then returns to the first request at the starting end servicing all requests in between. Same process repeats.

- Main difference between SCAN & LOOK is

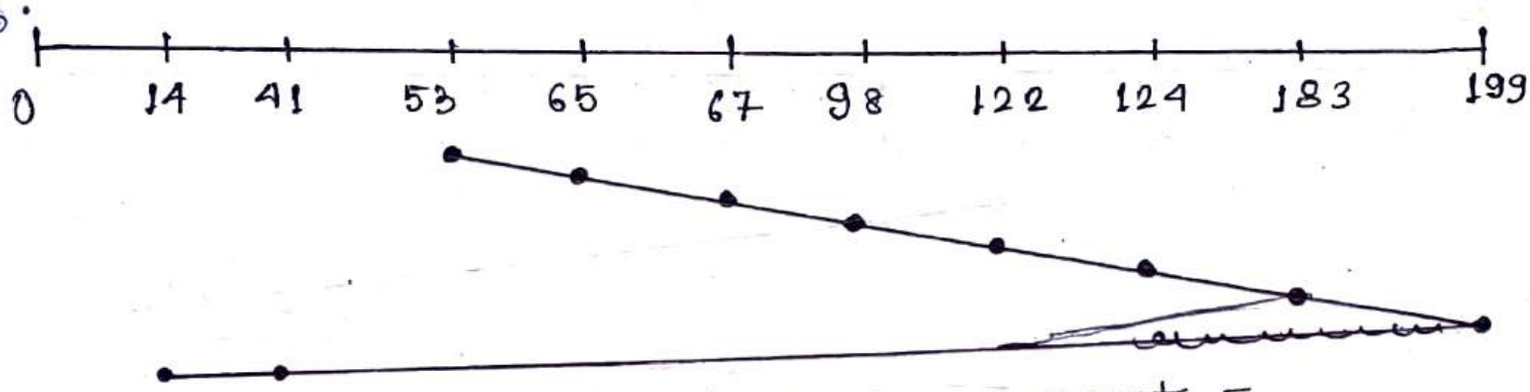
~ SCAN scans all cylinders start from one end ~~even~~ to the other end even if there are no requests at the ends.

~ LOOK scans all cylinders starting from the 1st request at one end to the last request at the other end.

adv. : It does not causes the head to move till the ends of the disk when there are no requests to be serviced. It provides better performance as compared to SCAN. It does not lead to starvation. It provides low variance in response time & waiting time.

disadv. : There's an overhead of finding the end requests. It causes long waiting time for the cylinders just visited by the head.

Q. 9



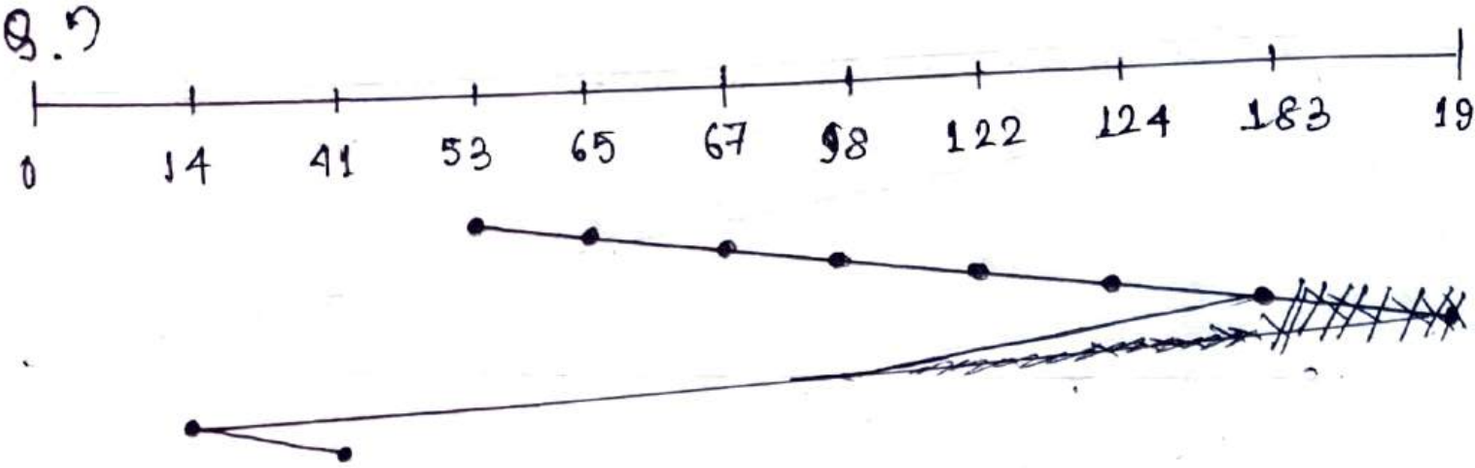
Total head movements =
$(183 - 53) + (183 - 14) = 299$.

vi) C-LOOK. Improved version of LOOK. Head starts from the first request at one end & move towards the last request at the other end servicing all requests in between. After reaching the last request at the other end, head reverses its direction. It then returns to the 1st request at the starting end without servicing any request in between. Same repeats

adv.: Does not cause the head to move till the ends of the disk when there are no requests to be serviced. It reduces the waiting time for the cylinders just visited by the head. Provides better performance as compared to LOOK. It does not lead to starvation. Provides low variance in response time & waiting time.

disadv.: Overhead of finding the end requests.

Q.7



Total head movements =

$(183 - 53) + (183 - 14) + (41 - 14)$

$= 326$