Date / /

Types of threads

1.) User Threads – Supported above kernel & are managed without kernel support.

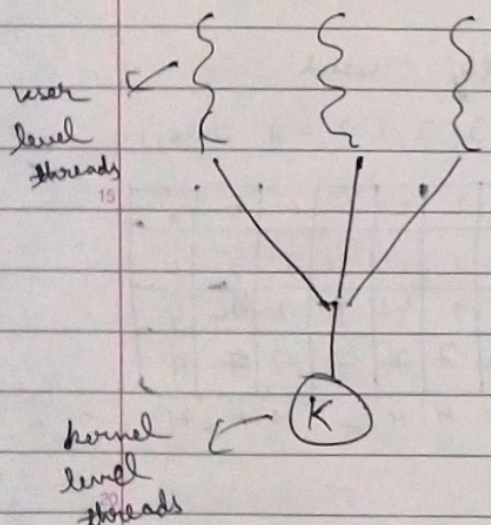
2.) Kernel Threads – Supported & Managed by OS.

Multithreading Models

For system to function properly, there must be relationship between user threads & kernel threads. Models used for defining Relationship :-

1.) Many-to-One model 2.) One-to-One Model 3.) Many-to-Many model

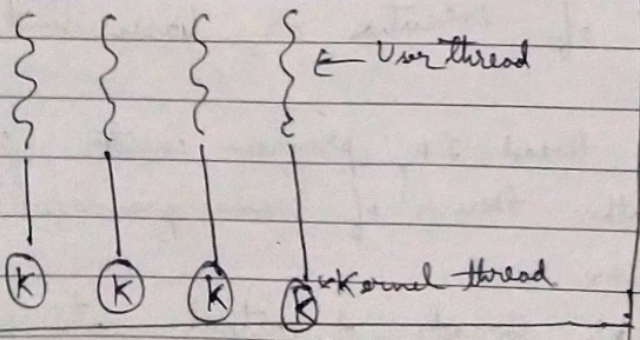
→ **Many-To-One Model**



Disadvantage
Only one thread can access kernel at a time so parallel running is not possible

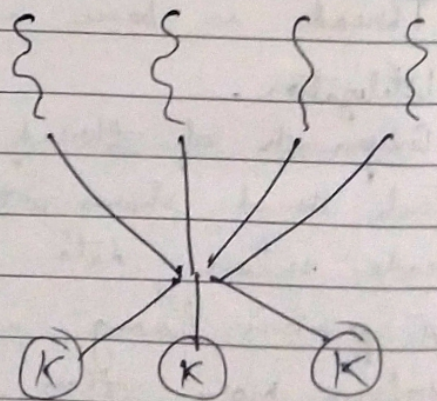
→ Thread management done by thread library in user space, so it is effecient.

→ **One to One Model**



Hyperthreading or Simultaneous Multi Threading (SMT)

Many-To-Many model



(Many user level threads map to smaller or equal no of kernel threads)

→ **LRU algo (Least Recently Used)**

Replace the least recently used page.

eg:- Reference string :- 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

f4				2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
f3			1	1	1	1	1	4	4	4	4	4	4	1	1	1	1	1	1	1
f2		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f1	7	7	7	7	7	3	3	3	3	3	3	3	3	3	3	3	7	7	7	
	*	*	*	*	Hit	*	Hit	*	Hit	Hit	Hit	Hit	Hit	*	H	H	H	*	H	H

7 was ~~was~~ came before 0, 1, 2

Page Faults = 8 & Page Hits = 12

→ **MRU algo (Most Recently Used)**

Replace the page which was recently used

eg:- Reference string :- 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

f4				2	2	2	2	2	2	3	0	3	2	2	2	2	0	0	0	0
f3			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f2		0	0	0	0	3	0	4	4	4	4	4	4	4	4	4	4	4	4	4
f1	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
	*	*	*	*	H	*	*	*	H	*	*	*	*	H	H	H	*	H	H	H

0 was most recently used

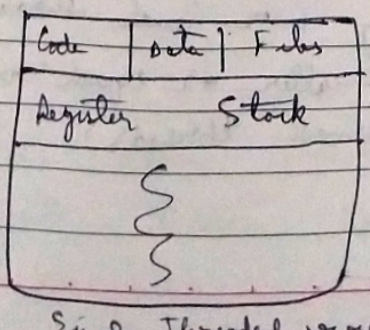
Threads

→ Thread is basic unit of execution or basic unit of CPU Utilization.

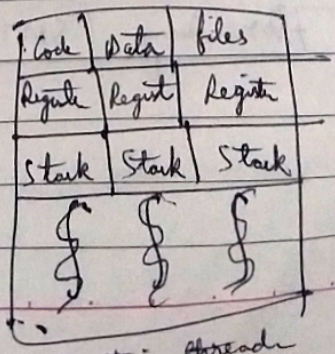
Components of thread :- thread ID, program counter, register set and stack

→ Each thread shares with other threads of same process :- code section, data section.

→ A process having multiple threads of control, it can perform more than one task at a time.



Single Threaded process



Multi-threads

Hard disk (S.M.) is very slow in comparison to M.M

Page Replacement Algorithm

→ **FIFO (First In First Out)**

eg:- Given reference string :- 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 1, 2, 0

Find no of page faults if initially there were 3 empty frames

f3			1	1	1	x	0	0	0	3	3	3	3	2	2
f2		0	0	0	0	3	3	3	2	2	2	2	1	1	1
f1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0
	*	*	*	*	Hit	*	*	*	*	*	*	Hit	*	*	Hit

x → page fault

So no of page faults = 12

no of page hits = 3.

Hit ratio = $\frac{3}{15} = \frac{1}{5}$ &

miss ratio = $\frac{12}{15} = \frac{4}{5}$.

/ page-fault ratio

Problem with FIFO (Belady's anomaly)

With more no of ~~fr~~ empty frames, no of page faults should decrease but in FIFO opposite happens. By increasing frames, no of page faults also increases. This principle is known as Belady's anomaly.

→ **Optimal Page Replacement Algo**

In this algo, we replace the page which is not used in longest dimension of time in future.

eg:- Reference string :- 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Sol-

f4				2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
f3			1	1	1	1	x	4	4	4	4	4	4	1	1	1	1	1	1	1
f2		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f1	7	7	7	7	7	3	3	3	3	3	3	3	3	3	3	3	3	7	7	7
	*	*	*	*	Hit	*	Hit	*	Hit	Hit	Hit	Hit	Hit	*	Hit	Hit	Hit	*	Hit	Hit

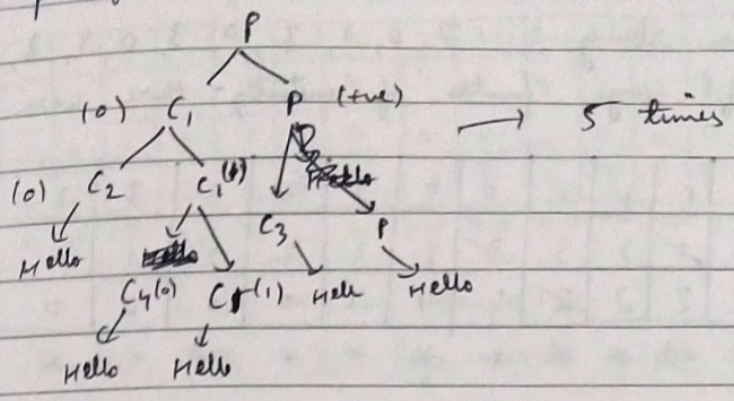
we replace 7 because it is used at last after 0, 1 & 2

we can also replace 3 as both 3 & 4 are not required in future

Camlin Page
Date / /

Que 2.) if (fork() || fork())
fork()
printf("Hello")

Sol-



Process Vs Threads in OS

→ Process means any program is in execution. Process Control block controls the operation of any process. PCB contain info about processes like process id, process priority, registers, etc. Process takes more time to terminate & it is isolated with other processes.

→ Thread is the segment of a process means a process can have multiple threads & these multiple threads are contained within a process. Thread have 3 states :- ready, running & blocked.

Differences

	Process	Threads
1.)	Program in execution	1.) Segment or part of a process
2.)	Process is called heavy weight process.	2.) Thread is called light weight process.
3.)	Takes more time to terminate.	3.) Takes less time to terminate
4.)	Consumes more resources	4.) Less resources
5.)	Process is isolated.	5.) Threads share memory
6.)	Process switching uses interface in OS.	6.) Thread switching does not require to call a OS & cause an interrupt to the kernel

Fork() system call

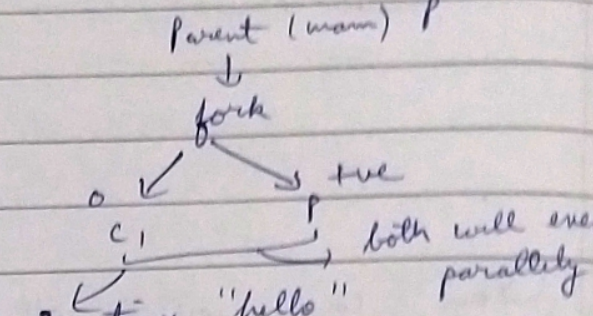
It is use to create a child process.

Fork() → 0 child (child process id is 0)

→ +ve parent (return parent process id +ve)

eg:-

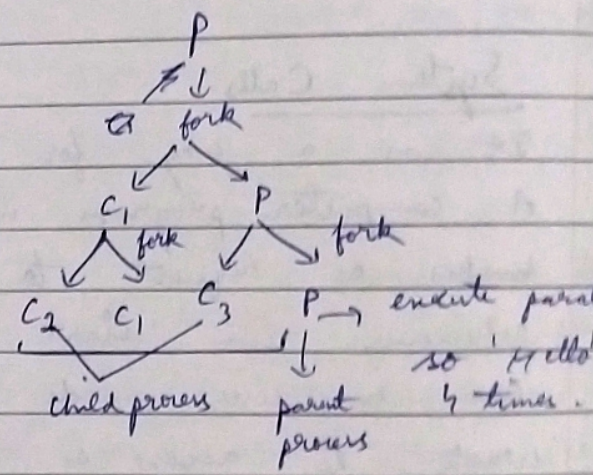
```
main() {
    fork();
    printf("hello");
}
```



So 2 times "hello"

Now if we write fork() twice :- will be printed.

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



so 'Hello' printed 4 times.

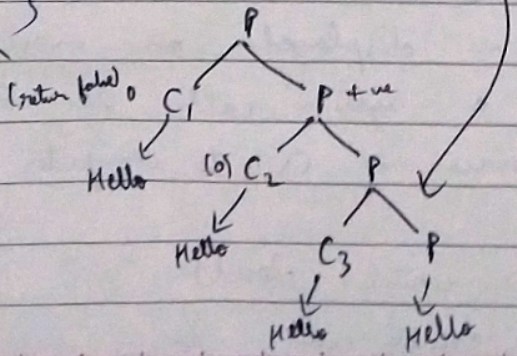
If n times fork() is used then printing will take place 2^n times & no of child processes = $2^n - 1$

Inp :-

```
#include<-->
int main()
{
    if (fork() && fork())
        fork();
    printf("Hello");
}
```

Approach

Ques:- How many times hello get printed.

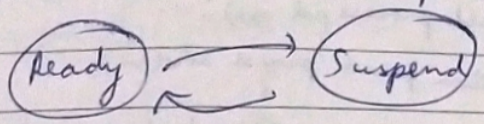


Ans :- 4 times

→ Kernel or OS controls hardware. User cannot directly access H/W. It access via kernel.

→ Kernel is heart of OS.

Medium Term scheduler – brings process ~~back~~ from wait state to suspend state (i.e M.M to S.M.)



(if very VIP process arrives & Ready queue is filled then some process can be send again by medium term scheduler to suspend state.

→ You can view all process states in Task manager of Windows.

System Calls

→ It is a way for programs to interact with the OS. A computer program makes a system call when it makes a request to the OS kernel.

→ Whenever we start a program or process we are in user mode but now if the program wants to access a file which is stored in S.M. / hard disk – H/W) then the program makes a system call to the kernel. Now the processor will switch to kernel mode. Read() system call is done to get or access data. Now for writing data Write() system call is made & system again goes to user mode. System oscillates between user & kernel mode.

→ So system calls are use to access resources.

Eg:- You have written a C program to calculate sum of 2 number. So for this no kernel is required. But if want sum to be displayed on monitor (H/W) you have to make a system call to kernel.

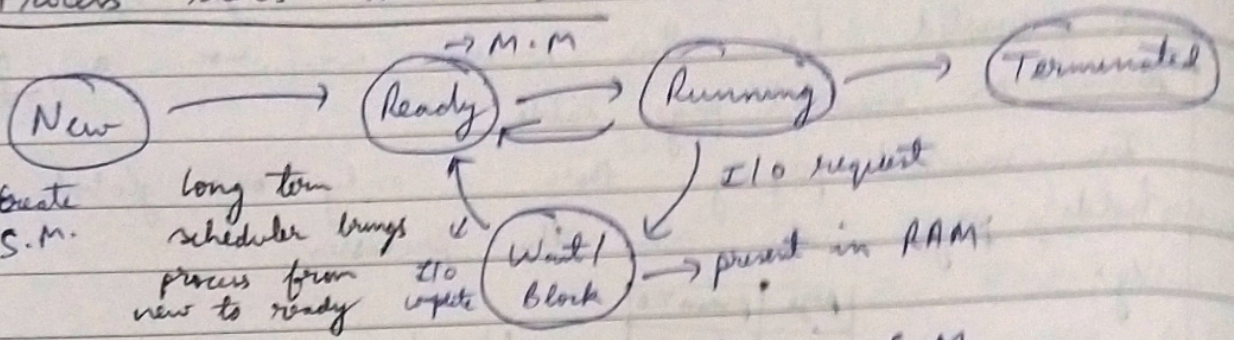
→ System calls provide service of OS to computer programs via API

Types of System calls:–

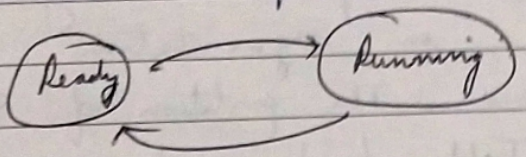
→ File related – open(), read(), write(), close()

→ Process related – fork(), load, execute, wait, & 3 more types.

Process states in OS

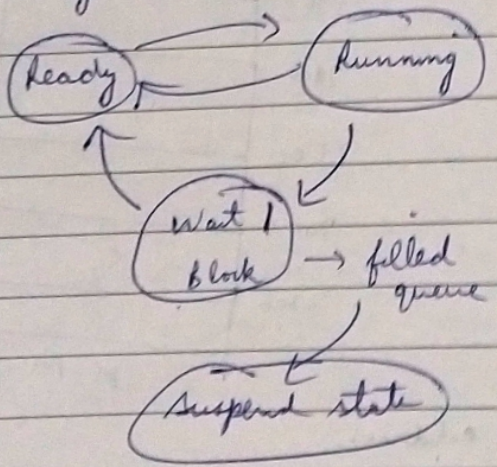


New – process is created & present in S.M.
Ready – Now process is moved to M.M
Long term scheduler – brings process from new to ready state (S.M) (M.M)
Running state – state where CPU operates on process & no of processes in running state depend on no of processor.
Terminated – State where all resources given to process are deallocated & process is not completed.



→ process is again send to ready state if higher priority process come or time quantum expires.

Short term scheduler – dispatch or schedule process from ready state to running



→ If a currently running process makes a I/O request then it is send to wait state & when it completes its I/O request it again goes to ready state as some process is already running in Running state.

Secondary state

Suspend state → If Wait / Block queue is completely filled & there is no space left then it can send a process back to S.M. This state of process is suspend state.

Que- What is interrupt ?

Sol- Interrupt is a signal emitted by H/W or S/W when a process or an event needs immediate attention.

of M.M to store them. So a large space of M.M. is wasted. For this reason we use a global P.T. known as Inverted page table

In inverted paging, there is only one global P.T. which holds pages of all the processes.

eg:-

	page	process ID
0	P_1	P_0
1	P_2	P_1
2	P_1	P_3
3	P_2	P_4
n		

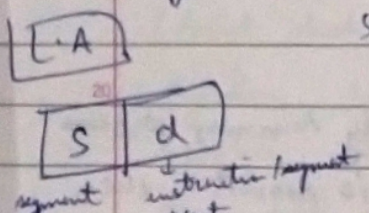
frame

No of rows in Inverted P.T. = no of frames in M.M.

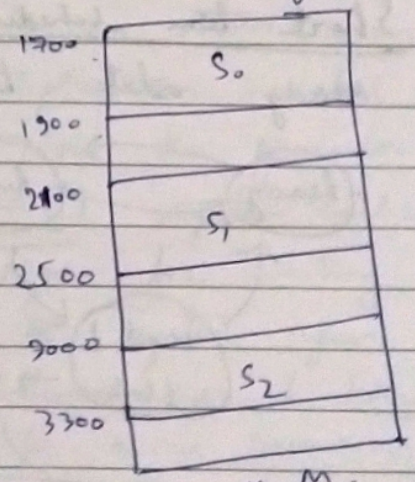
Disadvantage - Here searching is slow as linearly it has to search for required page.

→ **Segmentation**

Segmentation is another non-contiguous memory allocation technique. Unlike paging, in segmentation segments of variable size are formed. The partition of S.M. known as segments. Segmentation Table hold data of every segment.



Segment no	Base address	size
0	1700	200
1	2100	400
2	3000	300



$d \leq$ size else trap is generated indicating invalid address.

	Paging	Segmentation
1.)	Equal size partitions	1.) Variable size partitions
2.)	May suffer from Internal fragmentation	2.) May suffer from external fragmentation
3.)	Faster than Segmentation	3.) Slower
4.)	Paging is invisible to user	4.) Visible to user

→ **Demand Paging**

The process of loading the page into memory on demand (whenever page fault occurs) is known as demand paging.

Unlike normal paging where all the pages of a process are loaded into M.M before execution. Here few pages of a process are loaded & rest pages are loaded when page fault occurs & the used pages are replaced.

Process of loading new pages:-

1) If CPU try to locate a page that is not present in M.M, it generates an interrupt / trap indicating memory access fault.
2) The OS puts the interrupted process in a blocking state. For the execution, OS must bring the required page into memory.
3) The OS will search for required page in S.M. or LA space.
4) The required page will be brought from S.M to M.M. The page replacement algos are used for the decision making of replacing the page.
5) The Page Table is updated accordingly.
6) The signal will be sent to the CPU to continue the program execution & it will take place the process back into ready state.

Advantages of demand paging

1) Unlike paging where process size must be smaller than M.M. In demand page, a process larger than M.M can be executed.
2) It allows higher degree of multiprogramming by using less of the available memory for each process.

Inverted Page Table / Inverted paging

Every process has its own page table. So There P.T. are stored in main memory. So if there are 100 process we will have 100 P.T's we will use 100 frames

limited so only few pages of a process are brought into main memory. Because of this initially CPU utilization increases but after some time CPU may demand for a process page which is not present in M.M. So a page fault occurs & now time is wasted to bring page from S.M. to M.M. Thus CPU utilization starts to decrease. This condition is thrashing.

→ Thrashing – Condition or situation when the system is spending a major portion of its time in servicing the page faults, but the actual processing done is very negligible.

→ Techniques to handle Thrashing :-

1.) Big Main Memory :- But this will increase the access time.

2.) Setting Page Fault Frequency :- We know that page fault occurs very frequently when very few frames are allocated & page faults occur rarely if many frames are allocated. So we can set a upper & lower limit for no of frames of a process. So if page fault rate exceeds the upper limit, more no of frames can be allocated & if it becomes less than lower limit, then some frames of that process can be allocated to some other process.

3.) Controlling long term scheduler – long term scheduler brings processes into ready state. It brings more & more processes from S.M. to M.M in order to increase degree of multiprogramming. So we can lower the rate of long term scheduler.

→ Causes of Thrashing

1.) High degree of multiprogramming 2.) Lack of Frames of a process

Virtual Memory

Virtual memory is a storage allocation scheme in which S.M. can be addressed as though it were a part of M.M. It is a technique that is implemented using both H/W & S/W. Virtual memory is implemented using Demand Paging or Demand Segmentation.

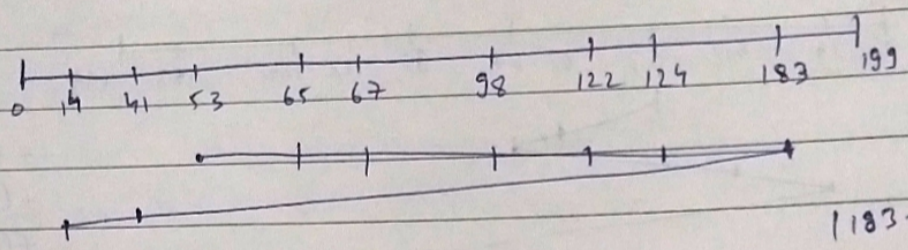
Date / /

5.) **LOOK**

→ It is same as SCAN but instead of going till last track, we go till last request, and then change direction.

Advantage - good for less load

Disadvantage - bad if more load

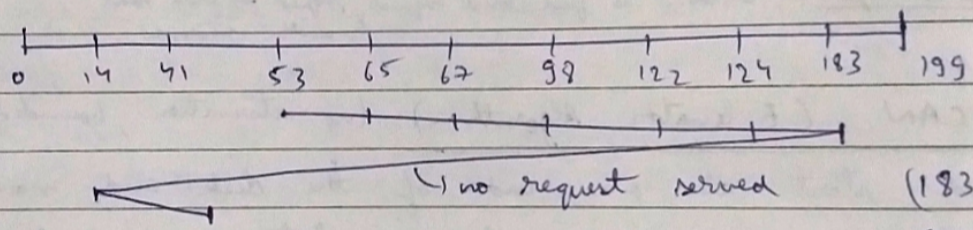


(183-53) + (183-14)

= 299

6.) **CLOOK (circular look)** (contains advantages of ~~SCAN~~ CSCAN & LOOK)

→ It takes advantage of CSCAN & LOOK



↳ serve only in one direction

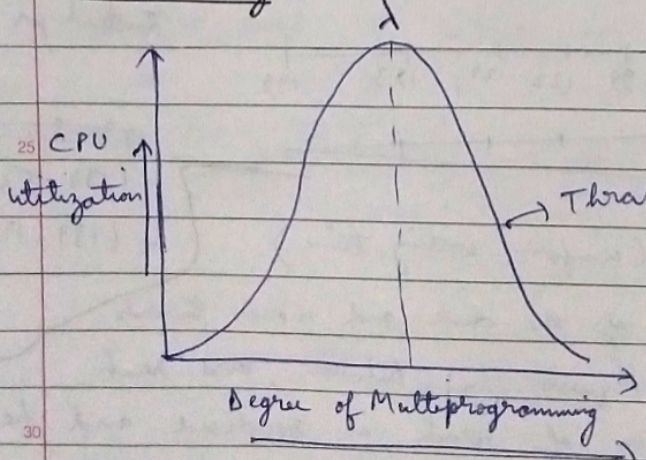
↳ no request served

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

= 326

Solves problem of starvation & waited bound is provided

Thrashing (concept of Virtual Memory) (Thrashing - Swapping pages in & out)



Note → If no of processes inside RAM is more than degree of multiprogramming is also more

In order to increase the degree of multiprogramming, we bring more & more processes into RAM. But RAM size is

Camlin Page
Date / /

Advantages & disadvantages same as FCFS in process scheduling. (no starvation, convoy effect, high seek time)

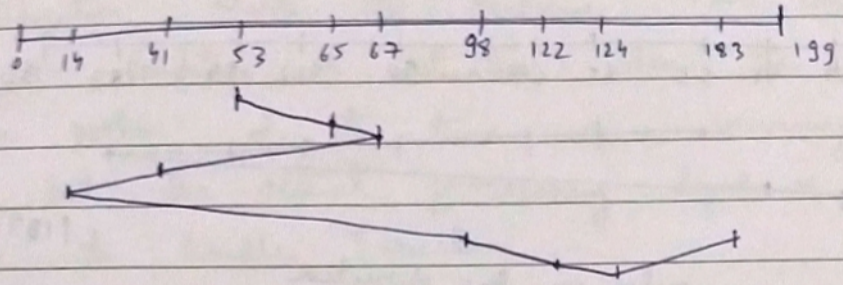
2) SSTF (Shortest Seek Time First) (best algo (most optimal)) → starvation

→ First serve that request which is closest to head

→ Tie is broken in direction of head movement.

eg:- Same question as in FCFS

Sol-



+(65-53) = 12
+(67-65) = 2
+(67-41) = 26
+(41-14) = 27
+(98-14) = 84
+(122-98) = 24
+(124-122) = 2
+(183-124) = 59
236 (Ans)

So, total track movements = 236 < 632 (FCFS)

Advantage – ↑ throughput, efficient in seek moves

Disadvantage – overhead to find closest request, request far from head will starve

3) SCAN (Elevator Algorithm) (no starvation, bounded waiting)

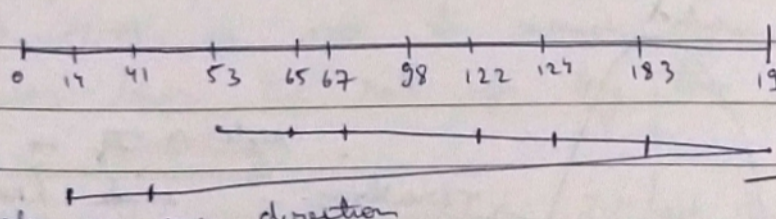
→ Head start at one end of the disk and move towards the other end, servicing requests in between, and reach other end and then direction of the head is reversed & process continues.

Head continuously scans back & forth across the disk.

eg:- Same question

Sol-

Initial pos = 53

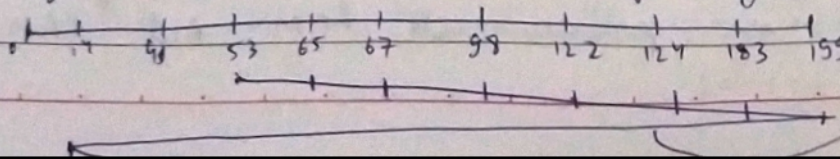


183-53 = 130
+(199-53) = 146
+(199-14) = 185
331
(Ans)

4) CSCAN (Circular SCAN) (uniform waiting time) → serve only in one direction

→ Head start at one end of the disk and moves towards the other end, servicing requests in between, and reach other end and then direction of head is reversed and head reaches first head without satisfying any request.

Sol-



no request served in return phase

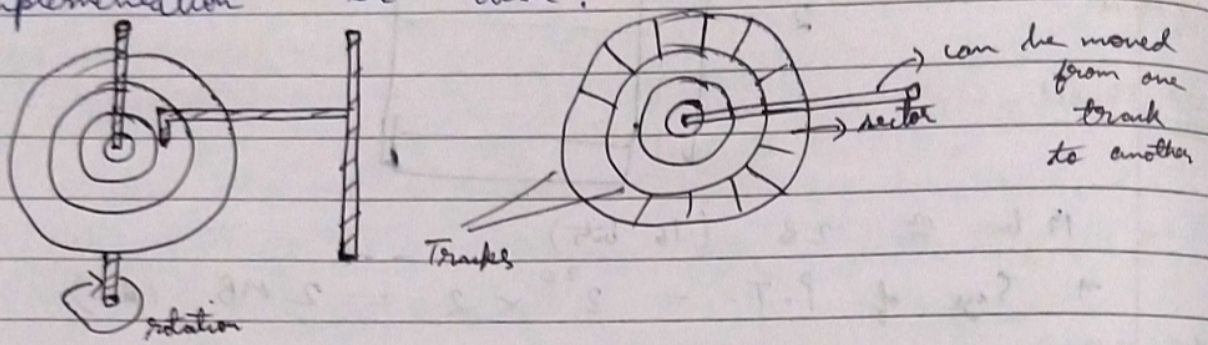
Total track = (199-53) + (199-14) + (41-14) = 336

Disk Scheduling algorithms

Secondary memory ~~disk~~ – hardware

Can be implemented in the form of optical disk, DVD drive, etc.

But we will only discuss about secondary memory implementation in disk.



→ **Seek Time** – Time required to move the R/W head on the desired track.

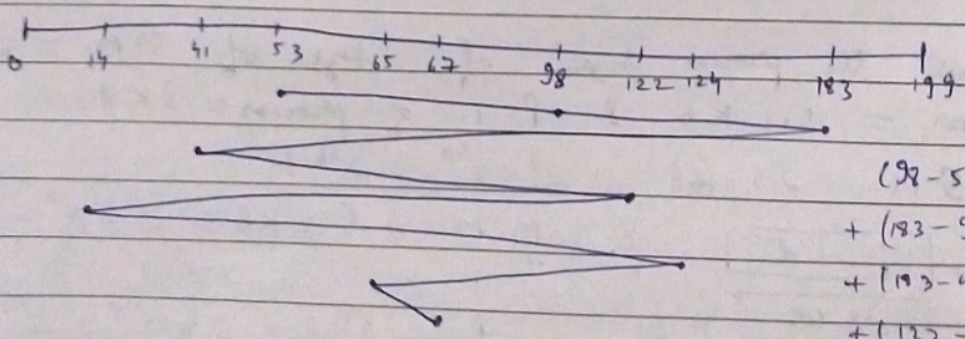
→ **Disk Scheduling** – In case of multiple I/O requests, disk scheduling algo must decide which request must be executed first. We have to insure that the R/W head moves least & all requests are fulfilled.

1.) FCFS (First Come First Serve)

Eg:- Given track numbers :– 98, 183, 41, 122, 14, 124, 65, 67

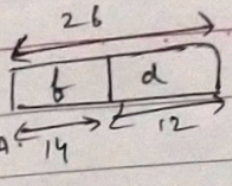
& R/W head is initially at 53. Find total track movement or ~~seek movents~~

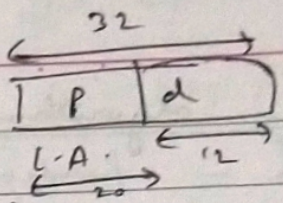
Sol–



(98 - 53) = 45
+ (183 - 98) = 85
+ (183 - 41) = 142
+ (122 - 41) = 81
+ (122 - 14) = 108
+ (124 - 14) = 110
+ (124 - 65) = 59
+ (67 - 65) = 2

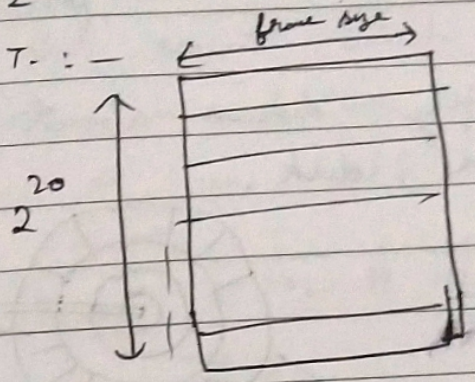
632 (Ans.)





P.S = 4KB = $2^2 \times 2^{10} = 2^{12}$ B ⇒ 12 bits for d.

Now for P.T. :-

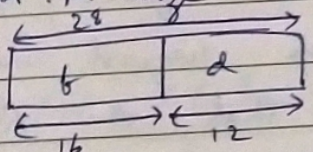


14 b ≅ 2B (16 bits)

→ Size of P.T. = $2^{20} \times 2$ = 2MB (Ans)

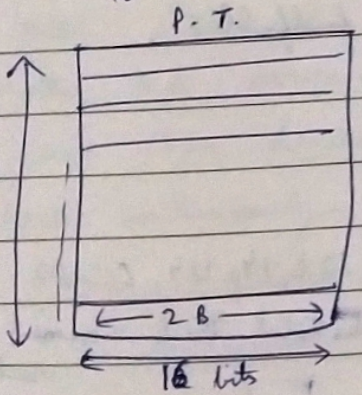
Que- Find page table size for a specific process of size = 4MB, M.M size = 256MB, P.S = 4KB & L.A = 40 b

Sol- M.M size = $2^8 \times 2^{20}$ B = 2^{28} ⇒ P.A → 28 bits



P.S = 4KB = $2^2 \times 2^{10} = 2^{12}$ ⇒ 12 bits

No of frames = $\frac{2^2 \times 2^{20}}{2^2 \times 2^{10}} = 2^{10}$

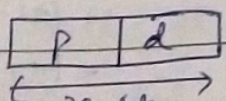


Total size of P.T. = $2^{10} \times 2$ = 2KB (Ans)

Que- Find the process size if size of S.M. = 256GB, P.S = 2KB, M.M = 512KB & P.T. of process = 8KB.

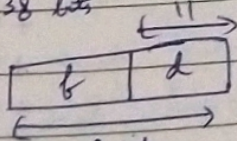
Sol- S.M. = 256GB = $2^8 \times 2^{30} = 2^{38}$ B & M.M. = 512KB = $2^9 \times 2^{10} = 2^{19}$

LA:-



P.S = 2KB = $2 \times 2^{10} = 2^{11}$ ⇒ 11 bits of d.

P.A:-



b → 8 bits or 1B

Now P.T size = 8KB = $2 \times 2^{10} = 2^{13}$ B ⇒ 2^{13} B / 2^{13} = 1 × no of pages

⇒ Process size = no of pages × page size = $2^{13} \times 2^{11} = 2^{24}$ = 16MB (Ans)

Que- Find the avg access time of a page if access time of M.M. = 400 nS, access time of T.L.B = 50 nS and hit ratio = 90% (90 out of 100 you will find a page in TLB).

Sol- If no TLB then,
avg access time = 2 × 400 = 800 nS (one for P.T. & then for M.M.)

If TLB then,
avg access time = 0.9 × (50 + 400) + 0.1 (50 + 400 + 400) = 405 + 85 = 490 nS
(0.9 ↓ for TLB; (50 + 400) ↓ Page in M.M.; 0.1 ↓ not found in TLB; 400 ↓ P.T.; 400 ↓ M.M.)

So there is considerable drop in using TLB as,
490 nS < 800 nS.

General formula :- $T_{avg\ access\ time} = h[T.L.B + MM] + (1-h)[TLB + MM + MM)$ where h → hit ratio

→ **More than one TLB. (why)**

We know that as soon as a new process comes or context switch takes place, all entries of previous process are deleted from TLB. Now pages & frames of this TLB is filled. But it may happen that after some time it needs the pages of previous process. So again instead of erasing content of this process we can use more than one TLB. Here, one TLB will store pages of one process & other will hold pages of other process. But since TLB is a hardware so using 2 or 3 TLB is very costly but search or access time is greatly improved irrespective of context switching.

Que- Find the total memory wasted because of page table if size of M.M = 64 MB, L.A is 32 bits long and Page size = 4 KB.

Sol- M.M = 64 MB = $2^6 \times 2^{20} = 2^{26}$ B
∴ Physical address is 26 bits long.

Que- Why we use TLB?

Sol- Because paging has a disadvantage that access time or speed is very less. So we use TLB which improves access time.

Camlin Page Date / /

Disadvantage of Paging (time consumption)

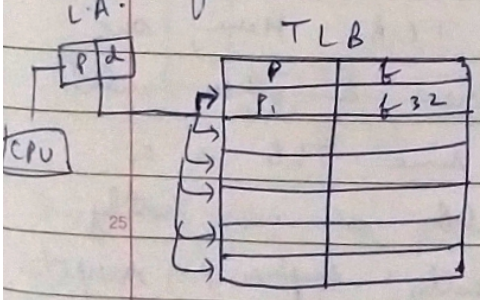
→ In Paging, main memory is accessed 2 times. One for Page Table & another for finding the frame.
So total time = 2. (MM access)
(PT) + (I) frame
↳ Page Table

Therefore, accessing a frame in M.M. is very slow because of two time access of M.M.

→ Generally a process executes in sequential form. Each page has a no of instructions.
Let a page contain 100 instructions so these instructions are executed one by one.
For 1st instruction of page 1, it will search in Page Table & then in M.M in corresponding frame number. Similarly for rest 99 inst of page 1. So this is a big problem as for same page we have to access Page Table again & again.

Translation Look Aside Buffer (TLB)

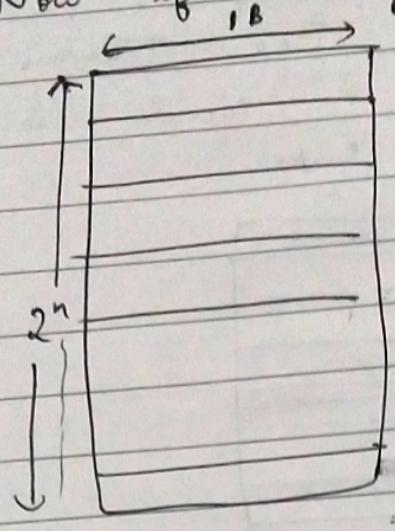
TLB is a hardware which is accessed in set associative fashion that is all the rows are accessed simultaneously



So for first instruction of page 1 it will search in Page Table & then it will move this page & frame to TLB. Now for rest (n-1) instructions of page 1. It will access TLB & search for corresponding frame number.
Searching in TLB is very fast as there is parallel searching & also TLB is smaller than M.M. But accessing TLB is very costly as it requires parallel searching.

→ Since TLB is a hardware so it is not specific to process. As soon as new process comes or context switch takes place, TLB is flushed & new empty TLB is created for new process.

Now if memory is 1B addressable then,



So with n bit address we can generate a memory of size = $2^n \times 1B$

Eg :- Find size of a memory ~~with~~ ~~which~~ is 14 bit address & is 1B addressable.

Sol- Size = $2^{14} \times 1 = 2^4 \times 2^{10} = 16KB$

Eg :- Find the bit address if size of memory is 64KB

Sol- If nothing is mentioned then we can assume it as 1B addressable.

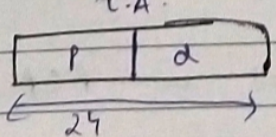
$64KB = 64 \times 2^{10} = 2^{16} = 2^n \Rightarrow n = 16$

So it is a 16 bit addressable system.

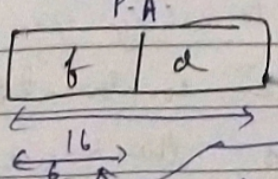
Calculating no of pages

Que- Given logical address of 24 bits & physical address of 16 bits. Page size = 1KB. Find no of pages.

Sol-



Size of Secondary memory = $2^{24} \times 1B = 16MB$



Size of M.M. = $2^{16} = 64KB$

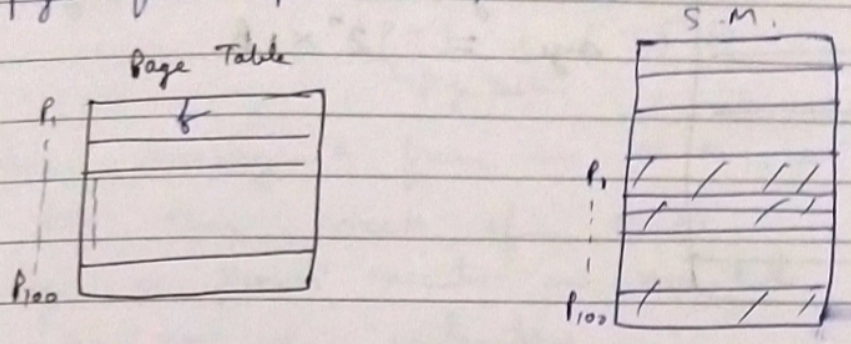
d → 10 bits

Now size of 1 Page = 1KB

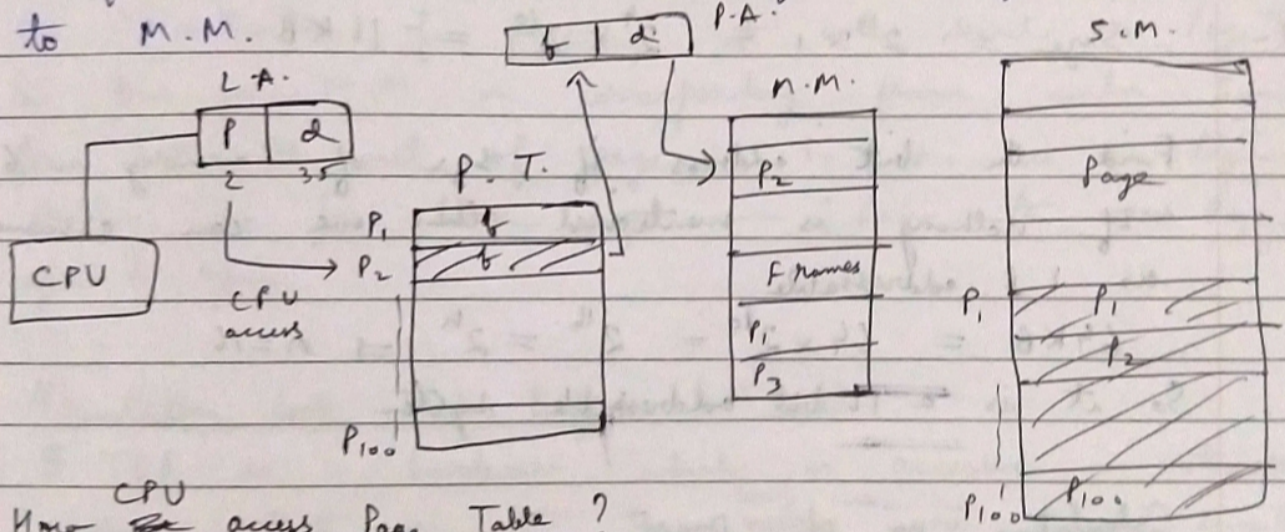
⇒ No. of pages in M.M. = $\frac{64KB}{1KB} = 64$ & No of pages in S.M. = $\frac{16 \times 2^{10} \times 2^{10}}{1 \times 10^{10}} = 16 \times 2$

→ Page Table is for a specific process. So every process will have its own Page Table.

→ No of rows in Page Table is equal to no of pages for process P_i in S.M.



→ Corresponding to every page in Page Table, there is a frame no or base address of the frame it corresponds to M.M.



→ How CPU access Page Table?

→ Page Table Base ~~address~~ register holds the base address of Page Table. So using this address CPU access Page Table.

→ d or instruction offset remains same.

→ Page Table Base Register is stored in PCB.

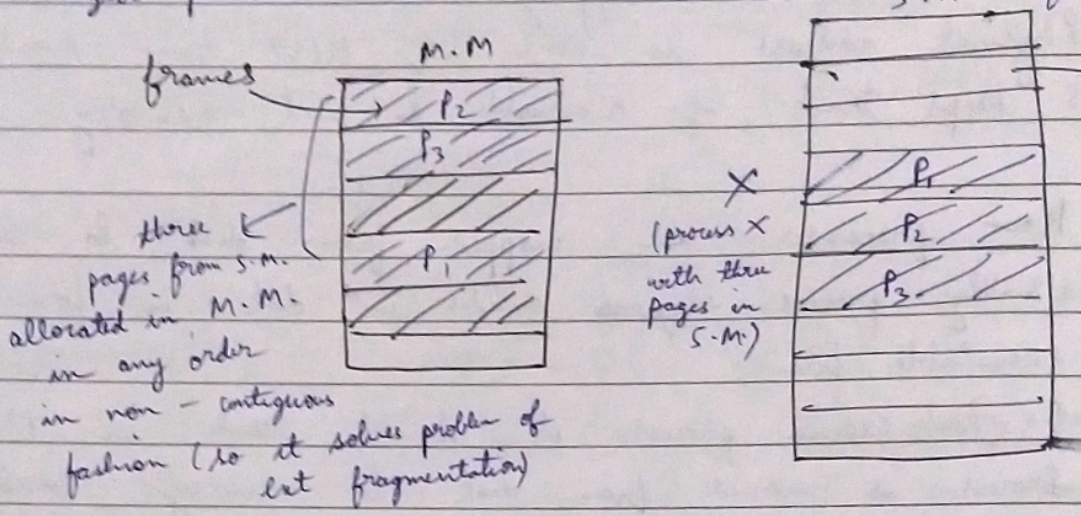
→ Page Table is itself stored in M.M.

Address system (Converting address bit to memory space & vice versa)

With n bits we can form 2^n combinations

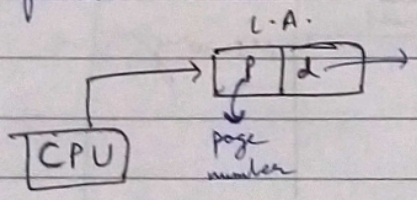
→ If you have a n bit address then we can generate a memory with 2^n locations.

Size of partition in both M.M. & S.M is equal. These equal sized partitions in M.M are known as frames.

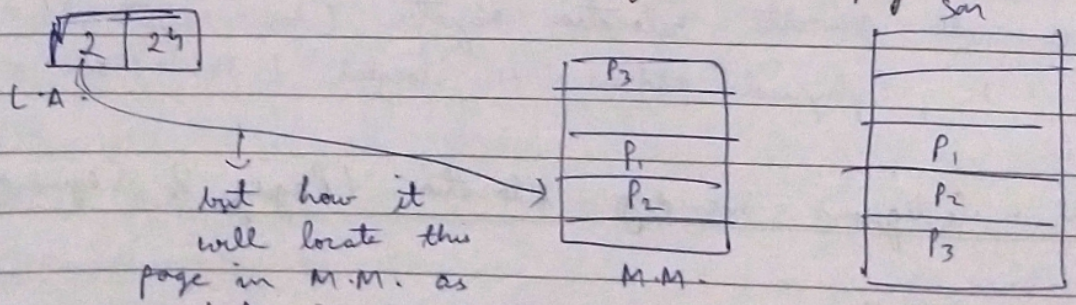


→ How physical address is generated for non-contiguous allocation?

→ Here it is different from contiguous because in contiguous we only have to remember the base address of process & then we can easily found. But here process is in the form of pages which is stored at non-contiguous fashion in M.M.



Every page contains instructions. Each page contains equal no of instructions as size of each page is also equal.



For this reason we use Page Table

Page Table is not a H/W but a data structure.

MMU (memory management unit)

→ The run time mapping between Virtual address & Physical address is done by H/W device known as MMU.

→ OS keeps track of available & used memory.

→ How processes are mapped from disk to memory ?

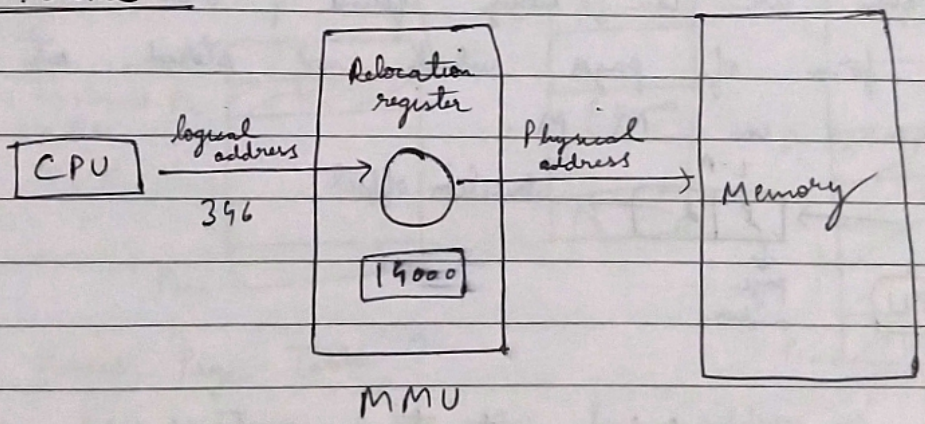
1.) Usually process ~~requires~~ resides in disk in form of binary executable file.

2.) So to execute process it should reside in M.M.

3.) Process is moved from disk to memory based on memory management in use.

4.) The processes waits in disk in form of ready queue to acquire memory.

MMU scheme



1.) CPU will generate logical address :- 346

2.) MMU will generate relocation register (base register) :- 14000

3.) In M.M., physical address is located (346 + 14000 = 14346)

Non - Contiguous memory Allocation (Paging & Segmentation)

→ **Paging**

Here we divide both S.M. & M.M. into fixed size partitions. These fixed size partitions in S.M. are called Pages. They will suffer from internal fragmentation.

Date / /

eg :- Find (limit) which (relocation) instruction number is legal.

Pno	L.R.	R.R	Instruction no
P_1	500	1200	450 ✓ (since 450 < 500)
P_2	275	550	300 ✗ (300 > 275)

Mapping Virtual Address to Physical Address

→ Address binding is the process of mapping from one address space to another address space.

→ Logical address is address generated by CPU during execution whereas Physical Address refers to location in memory unit (main memory).

← User deals with only logical address. It undergoes translation by MMU or address translation unit. The O/P of this process is the appropriate physical address or location of data in RAM.

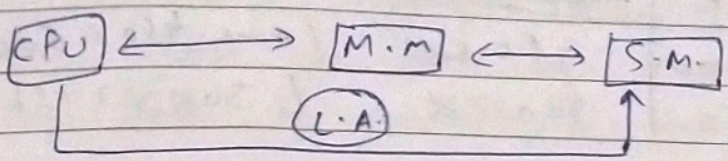
Address binding can be done in 3 ways :-

1.) Compile Time - If you know that during compile time where process will reside in memory then absolute address is generated. i.e. physical address is embeded to the executable of program during compilation. loading the executable as a process in memory is very fast. But if the generated address space is preoccupied by other process, then the program crashes & we need to recompile the program to change address space.

2.) Load Time - If it is not known at the compile time where process will reside then relocatable address will be generated. loader translates relocatable address to physical address. The base address of process in main memory is added to all logical addresses by loader to generate absolute address.

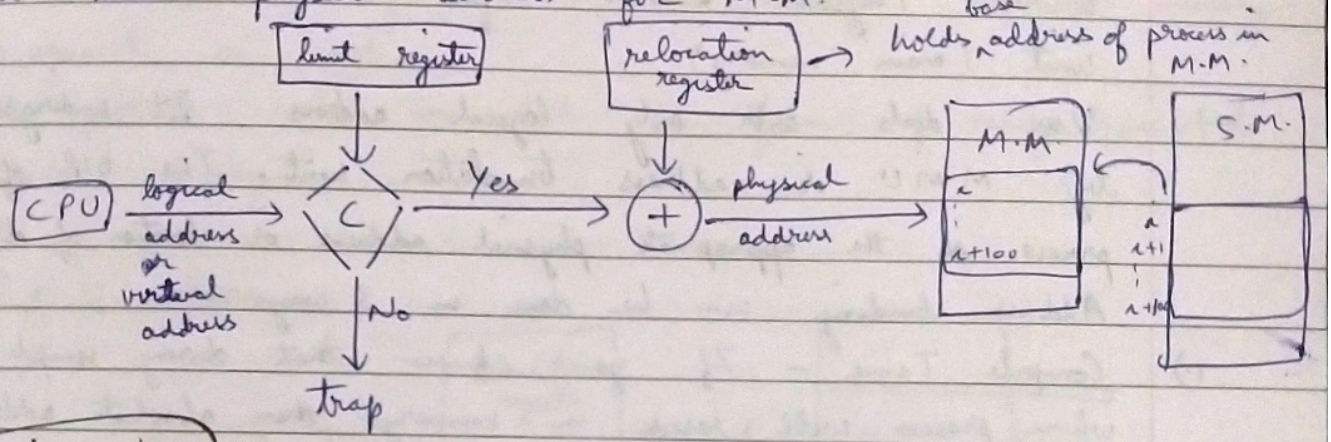
3.) Execution Time - The instructions are in memory & are being processed by CPU. Additional memory may be allocated or deallocated at this time. This is used if process can be moved from one memory to another during execution. (dynamic - linking :- Linking that is done during load time or run time).

Address Translation



→ CPU interacts directly with M.M & then M.M interacts with S.M.

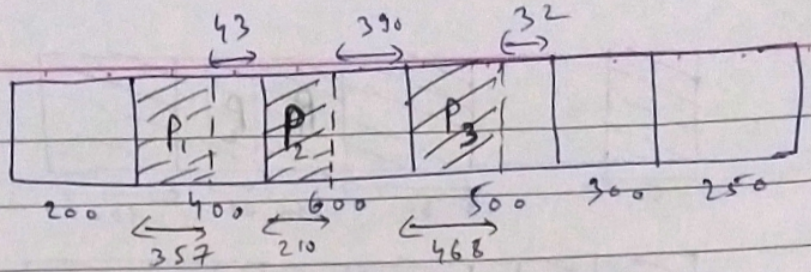
→ CPU generates a logical address for S.M. but it is difficult to access S.M. so we have to convert it into physical address for M.M.



limit register → security mechanism (checks whether logical address generated is valid or not. If invalid then it will trap the process.)

Eg:- CPU wants to access 52nd instruction of a process. Now this process is brought into M.M & stored in contiguous fashion. Here relocation register will hold the base address of process. Now it will just add 52 to base address & now this becomes the physical address. So this is how translation takes place. Now lets say that CPU generates a logical address for 120th instruction but there were only 100 instructions in process. So this is a illegal way to access data of some other process. Therefore, limit register will send it into trap.

Note:- Limit register → holds no of instructions in a process.
Relocation register → holds base address

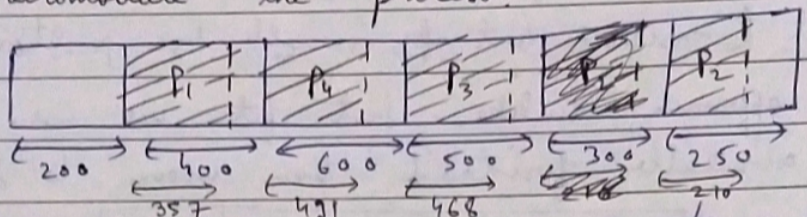


After all:- → No space left for 491 P4 process

Total internal fragmentation = 43 + 390 + 32 = ~~22~~ 465

Best fit (best algo for variable size)

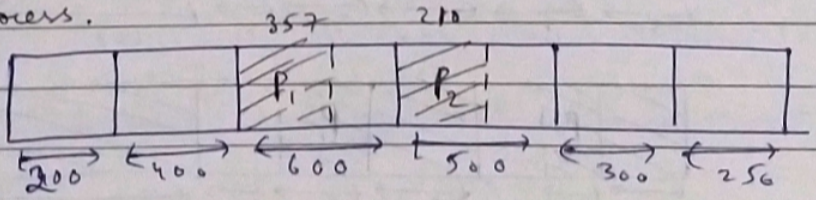
Always choose the smallest possible partition which can accomodate the process.



Total internal fragmentation = 43 + 109 + 32 + ~~90~~ = 224

Worst fit (worst for variable size)

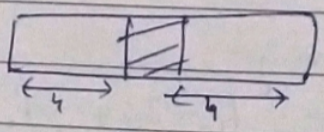
Always chose the largest possible partition which can accomodate the process.



P_3 & P_4 can be accomodated .. Total internal fragmentation = 243 + 290 = 543

Note - If available space is greater than required space but cannot be used for accomodation then it is external fragmentation else no external fragmentation.

eg :- $P_1 = 9$

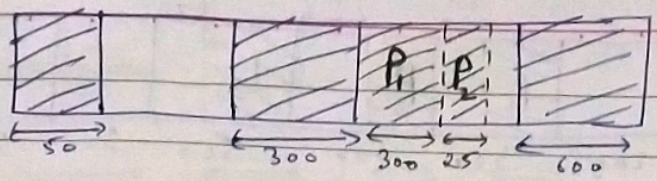


Here available space = 8 < 9 so no external fragmentation.

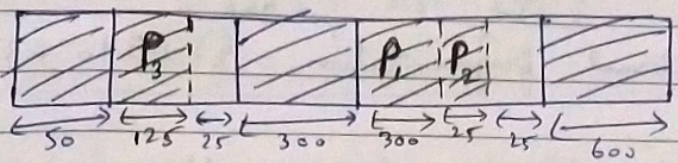
But if $P_1 = 5$ then there is external fragmentation = 8

Contiguous allocations leads to external fragmentation while fixed size partitioning leads to both internal as well as external fragmentation.

$P_2 = 25$:-



$P_3 = 125$:-

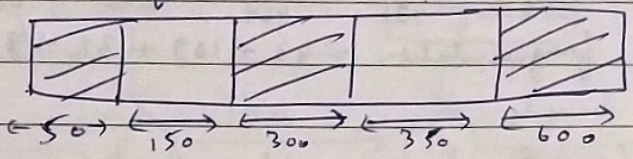


$P_4 = 50$ (now 50 space is not available together & since it is contiguous so we cannot allocate. Ext fragmentation)

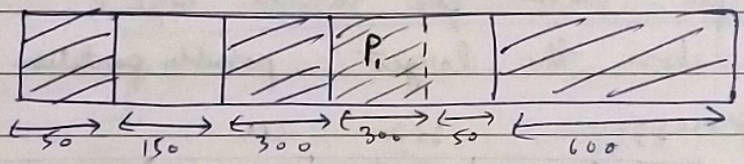
Worst Fit (considered best for variable size partitioning)

In this approach, unlike best fit we use largest possible space available first & then move to smaller.

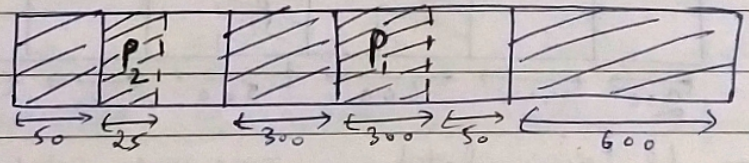
Snapshot :-



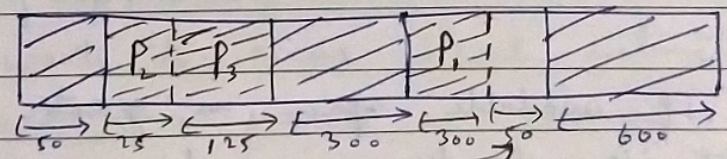
$P_1 = 300$:-



$P_2 = 25$:-



$P_3 = 125$:-

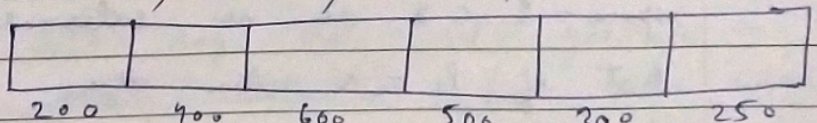


$P_4 = 50$:-

Fixed size Partitioning

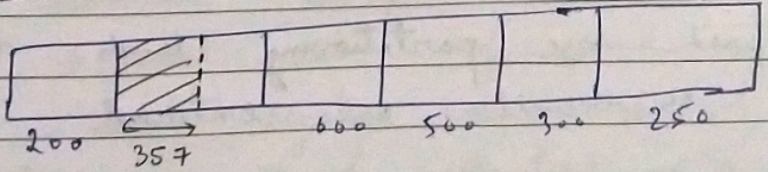
Que :- Given 4 processes $P_1 - 357$, $P_2 - 210$, $P_3 - 468$, $P_4 - 491$.

Current snapshot :-



Sol- First fit :-

$P_1 = 357$:

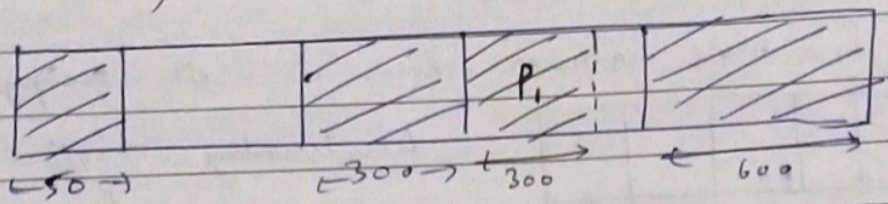


5d- Using First fit variable size partitioning :-

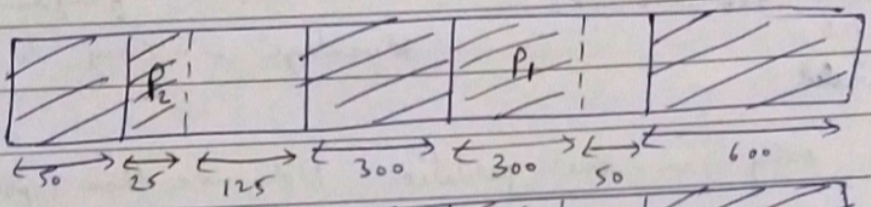
In this approach we always start from base address & find the first empty space which can accomodate it. Now since this is variable size partitioning so the left empty space can be reused.

$P_1 = 300$, $P_2 = 25$, $P_3 = 125$, $P_4 = 50$.

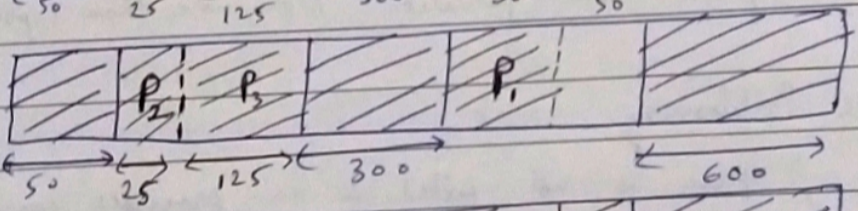
P_1 :-



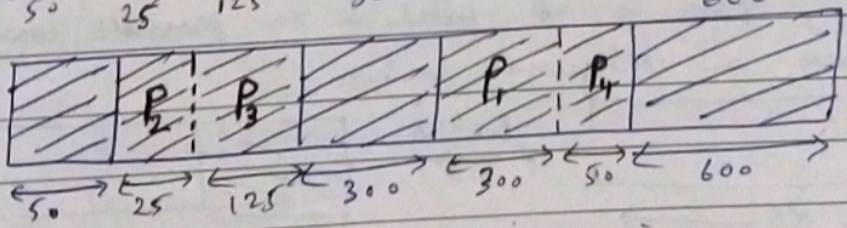
P_2 :-



P_3 :-



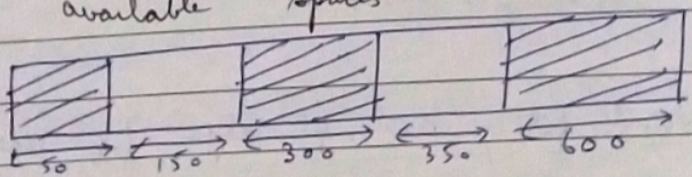
P_4 :-



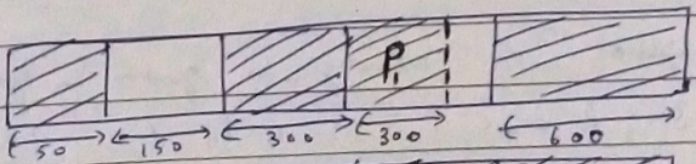
Best fit (worst for variable size)

Here we assign process that space which is smaller out of two available spaces.

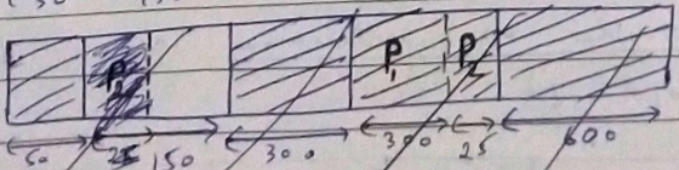
Snapshot :-



P_1 :-



$P_2 = 25$:-

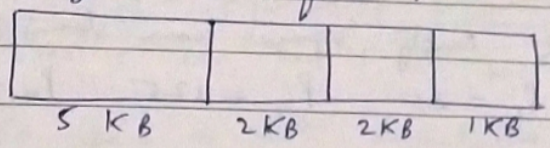


$P_3 = 125$:-

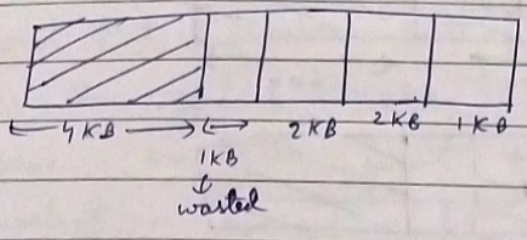
Fixed Size Partitioning

In this approach, memory is divided into partitions of fixed size. There partitions may not be equal but their size is fixed.

eg :-



Now if $P_1 = 4$ KB arrives then it will occupy 5 KB block.



Disadvantage :- Suffers from internal fragmentation

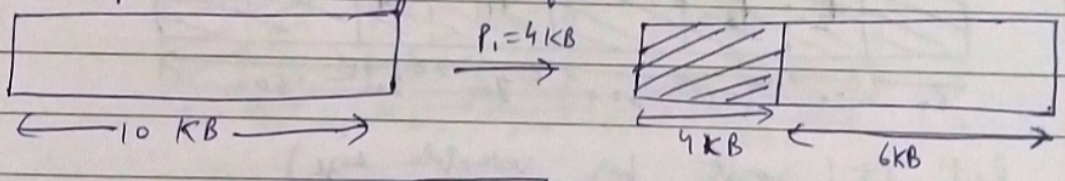
Advantage :- Easy to manage

→ One process only in one partition. Not more than 1 process in a partition.

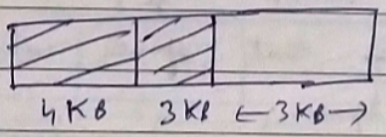
Variable Size Partitioning

Here initially, space is not filled & as processes come they are allocated spaces in contiguous fashion.

eg :-



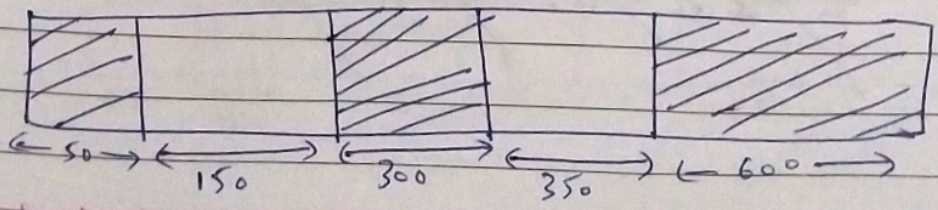
$P_2 = 3$KB ,



Advantage :- Solves problem of internal fragmentation

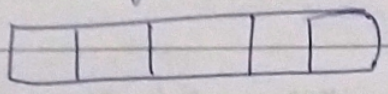
First fit, Worst fit, Best fit algo for Variable Size

Que- A current snapshot of memory is given & you need to accomodate four processes $P_1 = 300$, $P_2 = 25$, $P_3 = 125$ & $P_4 = 50$. This is to be done using variable size partitioning algos :- First fit, Best fit, Worst fit.



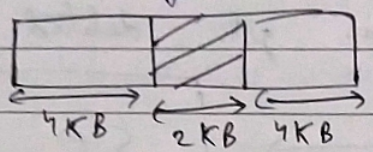
Contagious memory allocation

Ex - arrays



Advantage - Searching is fast as we can only need base address of the process.

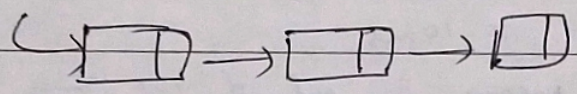
Disadvantage - External Fragmentation



A process of 5 KB need to be stored. But in contagious we cannot do although we are having 8 KB free space.

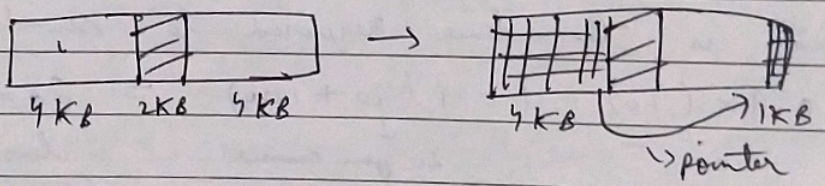
Non-Contagious Allocation

Ex - Linked list



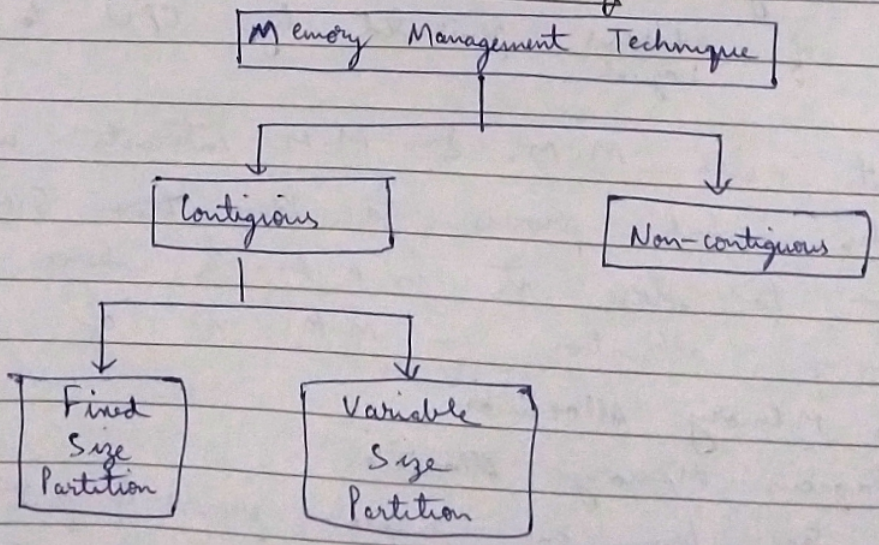
Advantage :- No External Fragmentation

$P_1 = 5KB$



Disadvantage - Slow access (no random access only sequential)

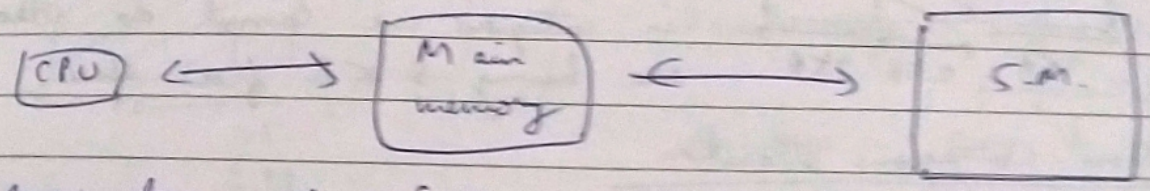
Fixed Size / Variable Size Partioning



So all the currently executing instructions or the most frequently used instructions are placed in cache memory.

→ In OS, we deal with main memory while in COA we deal with cache memory.

→ S.M. is a permanent memory while M.M. is volatile or temporary ∴ data is fetched from S.M. to M.M. After use it ⇒ gets deleted.

Ex- 

Access time of S.M. = 100ms

Access time of M.M. = 10ms

Hit ratio = 90% (means 90 out of 100 times you will find data in main memory)

Q- So what is the time required to search for data)

Sol- $0.9 \times (10) + 0.1 (10 + 100) \doteq 20$ ms (which is very less but if there was no m.m. then it would be (100 ms)

↓ we go in search m.m.

Two main duties of OS:-

1.) ~~how~~ to bring data from S.M. to M.M.

2.) translation of address generated by CPU (Logical) to Physical address

→ CPU interacts with M.M. & M.M interacts with S.M.

→ OS decides which process to bring from S.M. to M.M. and how to store it in M.M. & where to store.

So memory allocation in M.M. is of 2 types:-

1.) Contagious Memory Allocation

2.) Non Contagious Memory Allocation

Contagious - We bring one complete process & put it together in M.M.

Non Contagious - Process is broken into parts (parts) & arranged in non-contagious fashion on main memory.

Memory Management

→ Basics of memory management

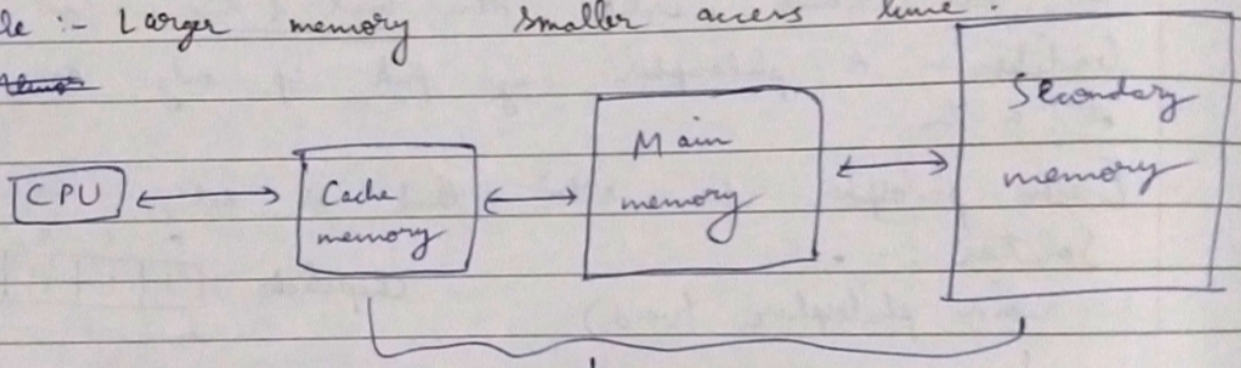
Criteria for good memory

1) Size (↑) 2) Access Time (↓) 3) Per unit cost (↓)

But we know that if we increase size then access time increase. So we want a memory which is bigger, less access time & less per unit cost.

So to achieve above factors to some extent we have a hierarchy of memory. Since using a single memory all 3 factors cannot be achieved.

Rule :- Larger memory smaller access time.



↓

Memory Hierarchy

→ Programs run in sequential form (generally).

I_0

I_{50} (if I_{50} instruction is running then most probably ≈ next instruction will be I_{51}

↓

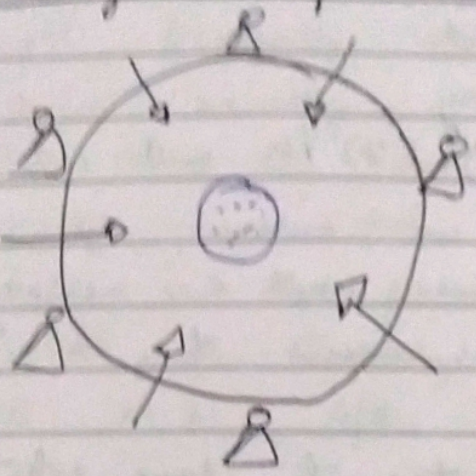
I_{51}

I_{200}

So what happens is that future instructions which may be executed are pre-fetched from S.M. to M.M. Now CPU will take less time to have this instruction because it has to search in smaller space. To make accessing more fast this is further pre-fetched from M.M. to Cache memory which is even smaller.

Camlin | Page
Date / /

Dining Philosopher Problem (Not very Imp for Gate)



Consider 5 philosphers & 5 chopsticks. They are seated on a round table. Now we are having a bowl of rice in the centre of table. But to eat rice using chopsticks we require 2 sticks. So we need to find order of execution using Semaphores.

Here chopsticks are shared resource or can be treated as crit critical section. Here bowl of rice is not C.S.

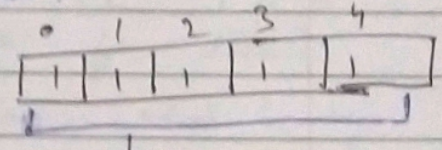
Condition - A philosopher may pick up only one chopstick at a time.

Each philosopher can either think or eat.

Solution :-

```
void philospher (void)
{
    while (T) {
    Thinking ();
    wait (chopstick [i]);          } for 5 chopsticks
    wait (chopstick [(i+1) % 5]);
    Eat ();
    signal (chopstick [i]);
    signal (chopstick [(i+1) % 5]);
    }
}
```

chopstick



array of semaphores initially set to 1.

Date / /

→ solving using semaphores

Reader – Writer problem (process Synchronization)

Here, we have a text document with 2 processes :- Reader & Writer. Reader reads data from document while writer writes the data. But here we can have more than one reader at a time as read-read don't clash but read-write & write-write conflict in database. So writer must operate separately.
So, document is C.S. for reader where more than one reader can operate but no writer.
Similarly for writer also it is a C.S where no other reader & writer can operate. Only one writer at a time.

Here we use 2 semaphores :-

1) wrt 2) mutex

Variables used :- readcount

Code :-

For writer :-

```
wait (wrt)
  write operation
signal (write)
```

→ because only one writer can operate

For reader :-

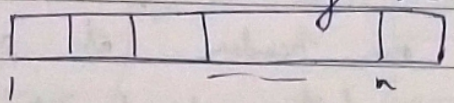
```
wait (mutex)
readcount ++
if (readcount == 1)
    wait (wrt)
signal (mutex)
  read operation
wait (mutex)
readcount --
if (readcount == 0)
  signal (wrt)
signal (mutex)
```

→ solving using semaphores

Producer - Consumer Problem (Process Synchronization)

Producer will produce something & consumer will consume.

So, we take an array or buffer of n blocks



Now producer can produce only if there is atleast one empty slot in the array. For consumer, there must be atleast one filled slot. Also at a time only producer or consumer can operate on array. So they cannot act together. Thus, array or buffer is treated as critical section.

To solve this problem we use 3 semaphores :-

1.) $S = 1$, for ensuring that only one will operate that is when producer is producing then consumer will not consume & vice versa.

2.) $E = n$, for checking no of empty slots. Initially all slots are empty so $E = n$.

3.) $F = 0$, for storing no of filled slots. Initially all slots are ~~filled~~ empty so $F = 0$.

Code :-

Two different processes :- producer & consumer

```
void producer ()
{
  while (T)
  {
    produce ()
    wait (E)
    wait (S)        → C.S.
      append ()
      signal (S)
    signal (F)
  }
}
```

```
void consumer ()
{
  while (T)
  {
    wait (F)
    wait (S)        ← C.S.
      take ()
      signal (S)
    signal (E)
    use ()
  }
}
```

→ If there are multiple ~~deadlocks~~ semaphores X & Y with different order then there may be a deadlock.

P	Q
P(X)	P(Y)
P(Y)	P(X)
C.S.	C.S.
V(X)	V(X)
V(Y)	V(Y)

→ order different so there may be deadlock

X = Y = 1

In P, P(X) → X = 0 now content switch made to Q where P(Y) → Y = 0. Now both X = 0 & Y = 0 so no one access C.S. thus deadlock.

Que- There concurrent process X, Y and Z access and update a shared variable. They uses four semaphores a, b, c, d. Such that X uses (a, b, c), Y uses (b, c, d) & Z uses (a, c, d). Which of the following is dead lock free order.

~~ans~~	X	Y	Z
a.)	P(a) P(b) P(c)	P(b) P(c) P(d)	P(c) P(d) P(a)
b.)	P(b) P(a) P(c)	P(b) P(c) P(d)	P(a) P(c) P(d)
c.)	P(b) P(a) P(c)	P(c) P(b) P(d)	P(a) P(c) P(d)
d.)	P(a) P(b) P(c)	P(c) P(b) P(d)	P(c) P(d) P(a)

Sol- We can directly eliminate those cases where order of execution is different as in a.) P(a), P(b), P(c), ~~b.)~~ c.) P(b), P(c), P(a) all different, all different

Now in b.) Lets start from X,

P(b) → b = 0 now we cannot do content switch to Y as same P(b). So content switch to ~~P(a)~~ Z P(a) → a = 0. Then P(c) so we can go into C.S. Thus (b.) → ans.)

Camlin | Page
Date / /

Que- Suppose we want to synchronize

Process P	Process Q
while (1)	while (1)
{	{
~~P(S)~~ W	Y
print '0';	print '1';
print '0';	print '1';
X	Z
} P	} Q

O/P :- String should not be of form $1 0^n 1$ or $0 1^n 0$ where n is odd

	W	X	Y	Z
a.)	P(S)	V(S)	P(T)	V(T) S,T=1
b.)	P(S)	V(T)	P(T)	V(S) S,T=1
c.)	P(S)	V(S)	P(S)	V(S) S=1
d.)	V(S)	V(T)	P(S)	V(T) S,T=1

Sol- a.) P(1) → S=0 P(1) → T=0

print '0' → now context switch
print '0';

print '1' → 1 now back to

→ so generates 010 which should not be generated

b.) Similarly as a.) generated

c.) Here it is not possible to generate 010, 101 type so ans is c.) (Ans-)

Imp Deadlock related questions approach using Semaphores

→ Whenever there is only one semaphores X then there is no deadlock.

P	Q
P(X)	P(X)
C.S.	C.S.
V(X)	V(X)

→ no deadlock

→ When there are multiple semaphores but their order is same then also no deadlock.

P	Q
P(X)	P(X)
P(Y)	P(Y)
C.S.	C.S
V(X)	V(X)
V(Y)	V(Y)

→ no deadlock

Note :- Wait is also represented by P( )
signal (S) → V(S)

Semaphores (for deciding order of execution)

$P_2 \rightarrow P_1 \rightarrow P_3$ execution

wait (S_1)		wait (S_2)
P_1	P_2	P_3
signal (S_2)	signal (S_1)	

$S_1 = 0$, $S_2 = 0$

(So for executing 3 processes in this order, we need 2 semaphores)

Semaphores (for managing resource)

If we are having duplicates of resources then only one thing we have to change that is set the value of S to the number of parallel processes you want.

Eg :-

```
Pi
{
  wait (S)
  C.S.
  Signal (S)
  R.S
}
```

you have 5 printers & you want them to access resources in C.S. together then set value of S = 5.

Que- Suppose we want to synchronise processes P & Q, using two binary semaphores S, T
O/P :- 00 11 0011 ... (alternate 00 followed by 11)

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

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

	W	X	Y	Z	
~~a.)~~					
a.)	P(S)	V(S)	P(T)	V(T)	S, T = 1
b.)	P(S)	V(T)	P(T)	V(S)	S=1, T=0
c.)	P(S)	V(T)	P(T)	V(S)	S, T = 1
d.)	P(S)	V(S)	P(T)	V(T)	S=1, T=0

Sol- (b.)

Note :- Starting value of Semaphore is always :— $S = 1$;

```
S = 1
do
{   wait (S);
    critical section;
    exit (S): signal (S);
// remainder section;
} while (T)
```

Eg :- $P_1, P_2, --, P_n$

For P_1, $S = 1$
So wait (1) → move into C.S.
now $S = 0$.
So if P_2 makes a context switch but wait (0) → will loop in & can't go into C.S. until P_1 comes out & using signal makes $S = 1$.

So it is M.E. & also maintains progressive as one process can change only value of S & other process which wants to enter can again change S.

So in semaphores, M.E. & progress are ensured but not bounded weight (optional).

Very Imp Qn A shared variable x, initialized to 0, is operated by four process W, X, Y, Z. Process W & X increment x by one, while process Y, Z decrement x by 2. Each process before reading 'wait' on a semaphore 'S' and signal on 'S' after writing. If semaphore 'S' is initialized to 2. Find max possible value of x after all processes complete execution.

a.) 2 b.) −1 c.) 1 d.) 2

Sol-

W	X	Y	Z
wait (S)	wait (S)	wait (S)	wait (S)
R (x)	R (x)	R (x)	R (x)
x += 1	x += 1	x -= 2	x -= 2
W (x)	W (x)	W (x)	W (x)
signal (S)	signal (S)	signal (S)	Signal (S)

Ans :- Max possible value = 2 & Min possible value = −4 for S = 2 (content switch is done)

Camlin Page
Date / /

→ analogus to Peterson Solution

Que- Fill in the while condition:-

```
flag [i] = T
turn = j
while ( ___ );
CS
flag [i] = F
```

Options :-
a) flag [i] = T && turn = i
b) flag [i] = T && turn = j
c) flag [j] = T && turn = i
d) flag [j] = T && turn = j

In these type of problems, always use Turn Solution, Flag Solution & Peterson Solution. (always analogus to these)

Semaphores

We have seen in race condition that when processes access shared resources in different orders then result changes. While this is not true for private resources. So to solve this problem there are 3 major criterias which must be fulfilled :- mutual exclusion, progress and Bounded weight. So to solve this problem we used three solutions :- Turn solution (only M.E.), flag solution (deadlock), petterson solution (both ME & progressive but only for 2 processes).

→ Therefore for n processes, we use Semaphores

Three major applications of Semaphores :-

1.) Critical Section Problem (solves)
2.) Decides order of execution of processes
3.) Resource Management

Defintion — Semaphores — A Semaphore is an integer variable that apart from initialization is accessed only through 2 standard atomic operations wait (S) & signal (S).

```
int S;
wait (S)                    signal (S)
{                           {
  while (S <= 0);             S++;
  S--;                      }
}
```

3.) Dekker's Algo or Peterson Solution is consistent as it ensures mutual exclusion, progress & solves problem of deadlock.
But Peterson Solution is only true for 2 processes.
For n-process, we use Semaphores.

Ques- Consider method used by process P_1 & P_2 for accessing ~~C.S.~~ Critical Section. Initial value of shared boolean variable S_1, S_2 is randomly assigned.

P_1	P_2
while ($S_1 == S_2$);	while ($S_1 != S_2$);
critical section	critical section
$S_1 = S_2$	$S_2 = not(S_1)$

Check whether method follows :-

1.) M.E. & Progressive
2) M.E. & ~~P~~
3.) ~~M.E.~~ & ~~P~~
4.) ~~M.E.~~ & P

Sol- S_1 & S_2 assumes boolean value 0.
It is clear that only one process can enter C.S. at a time so it is M.E. But there is no initial value of S_1 or S_2 which can decide whether P_1 or P_2 want to be a part of C.S. or not. It happens alternatively & thus it is not progressive. Option (2.) → Same to flag solution.

Ques-

Process X	Process Y
while (T)	while (T)
{	{
var P = T;	var Q = T;
while (var Q == T);	while (var P == T);
C.S.	C.S.
var P = F;	var P = T;
}	}

Options :-

	M.E.	deadlock
a)	X	X
b.)	✓	X
c)	X	✓
~~d)~~	✓	✓

flag [0 | 1] [F | F] → initial

P_0	P_1

while (1)
{
flag [0] = T;
while (flag [1]) ;
critical section
flag [0] = F;
}
	while (1)
	{
	flag [1] = T;
	while (flag [0]);
	critical section;
	flag [1] = F;
	}

This solution is not consistent as it satisfies both conditions of mutual exclusion ~~& progress~~. but for progress here deadlock may occur because of content switch

Peterson Solution (Dekker's Algo)

Initially flag [0 | 1] [F | F]

P_0	P_1

while (1)
{ flag [0] = T;
turn = 1
while (turn == 1 && flag [1] == T);
critical section
flag [0] = F;
}
	while (1)
	{
	flag [1] = T
	turn = 0
	while (turn == 0 && flag [0] == T);
	critical section;
	flag [1] = F;
	}

This solution is consistent as it ensures both mutual exclusion & progress. It also satisfies max bound condition.

Note- 1.) Turn Variable is not consistent as it lead to mutual exclusion but not progress.

2.) Flag Variable solution is not consistent as it lead to mutual exclusion but not progress at it can lead to deadlock.

Initial turn = 0

P_0	P_1
while (1)	while (1)
{	{
while (turn != 0); (i)	while (turn != 1);
critical section;	critical section;
turn = 1;	turn = 0
remainder section;	remainder section;
}	}

If turn = 0, now P_0 checks while (turn != 0); It is false so it will move to critical section. Now turn is 0 only If P_1 tries to enter then while(turn != 1); which is true as turn is 0 but because of (i) at end it will not go inside & will come again check until turn becomes 1 & while (turn != 1); is false.

→ So it satisfies mutual exclusion condition as only one process can enter ~~out~~ critical section at a ~~time~~.

→ But this solution do not follow progress as P_1 will always follow P_0 then again P_1 & so on.

$P_0 \rightarrow P_1$ (so no matter whether P_1 wants to enter or not it has to enter therefore not a progressive approach)

⇒ So turn variable solution is not consistent as it is mutual exclusion but not progressive. Therefore, it is not consistent as both are mandatory conditions.

Using Flag Variable Two Process Solution for Critical ~~Solution~~ Section Problem

In previous solution, we did not ask the process that whether it wants to enter into critical section or not. So here we use a boolean array flag which we set as :- flag

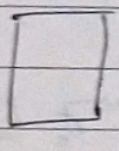
0	1
F	F

Critical Section Problem

P
{


← Primary Resource

Question :- How to solve critical section problem such that no inconsistency arises?


← Shared Resource

critical section (part of code where process access shared resource)
}

Criteria to be satisfied to solve a critical section problem with consistency :-

1.) Mutual exclusion (mandatory criteria) → access critical section in mutual exclusion fashion that is not in a shared form

2.) (mandatory) Progress (only those process should enter into critical section which wants the resources, although even if all processes are considered system will be consistent but effecincy will be quite low).

3.) (optional but good) Bounded weight — There must be max bound upto which a process can wait for entering into critical section. So a time limit must be set. It avoids the problem of starvation.

So two criterias (1 & 2) are mandatory while Bounded weight is optional.

Using Turn Variable Two Process Solution for Critical Section Problem

```
P0 {
    entry section
    C.S. (critical section)
    exit section
    remainder section
}
```

Deadlock Ignorance (~~or~~ Ostrich Algorithm)

It means that the deadlock is simply ignored & it is assumed that it will never occur.

→ When storm approaches, an ostrich puts his head in the sand & pretend that there is no problem at all.

Process Synchronization (required because many processes access same resources in a multiprogramming system)

Critical Section → Area in process execution where a process access shared resources is known as Critical section.

```
P
{   critical section
|
|  [ C.S. ]
|
}
```

Problem – when two processes access shared resource at same time.

Only one process can enter into critical section at a time.

when two different processes execute in different order or fashion & give different results then it is known as **Race Condition.**

Critical Section Que – Given 2 processes P_1 & P_2

```
P1 ( ) {                    P2 ( ) {
① —  C = B - 1;             D = 2 * B;  — ③
② —  B = 2 * C;             B = D - 1;  — ④
   }
```

B is a shared variable with initial value 2.

How many different values of B exists?

Sol –

C_1 (1, 2, 3, 4)	C_2 (3, 4, 1, 2)	C_3 (1, 3, 4, 2)	C_4 (3, 1, 2, 4)	C_5 (1, 3, 2, 4)	C_6 (3, 1, 4, 2)
C = 2 - 1 = 1	D = 4	C = 1			
B = 2 × 1 = 2	B = 3	D = 4			
D = 2 × 2 = 4	C = 2	B = 3			
B = 4 - 1 = ③	B = ④	B = ②	B = ~~5~~ ③		

Pessimistic Approach (Process Termination)

Here we have 2 approaches:-

1.) Abort all deadlocked processes

2.) Abort one process at a time and decide next to abort after deadlock detection.

→ costly process (like P_1, P_2, P_3 are in deadlocked state but P_1 has completed its execution 90%, now if we abort it, it has to start from all over again)

→ In this approach we abort one process & check for deadlock using deadlock detection algo then again it ~~check~~ abort one process & check for deadlock. So this process involves overhead of ~~two~~ checking again & again.

Factors to be considered by a system before killing a process :-

1.) Priority of process
2.) How long the process has computed?
3.) How much longer a process will compute before execution
4.) How many & what type of resources process has used
5.) How many resources the process needs to complete its execution.

Optimistic Approach (Preemption of Resources & Processes)

Take some resources from a process & give it to other process until the cycle gets broken.

3 Questions :-

1.) Which process will give its resources? (based on previous factors)
2.) Process rollbacks to safe state & starts its execution again.
3.) Starvation - If same process is always pre-empted then it may lead to starvation of this process. So we can set a limit to no of rollbacks for every process. After this limit, no process can rollback.

Deadlock detection & Recovery

→ Allow the system to enter into deadlock state.

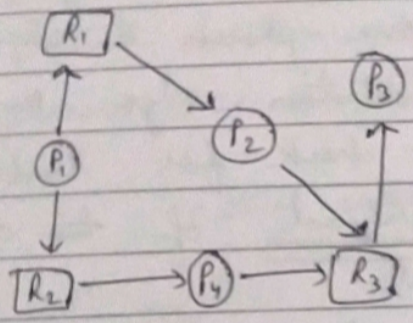
→ Deadlock detection algorithms are of 2 types :-

1.) ~~Single~~ Wait for graph (if resources are of single instance)

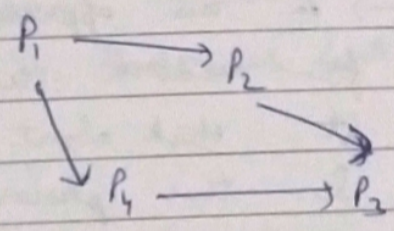
2.) ~~DA~~ Banker's Algorithm (if resources are of multiple instance)

Wait for graph

eg:- 1.)

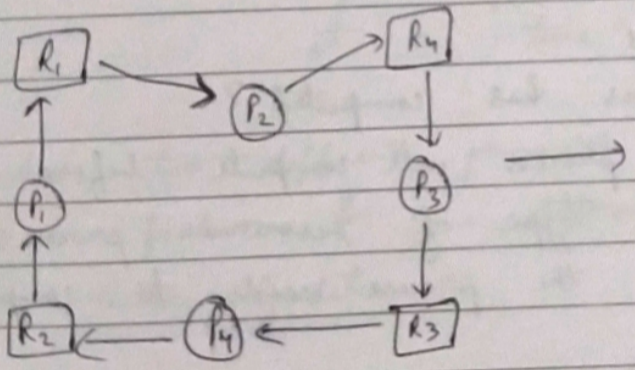


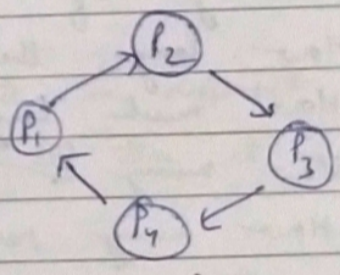
Resource allocation graph



Wait for graph
(here no cycle formed so no deadlock at this instant of time)

2.)





(here cycle is formed so system is in deadlock at this instant of time)

So in order to detect deadlock we can call deadlock detection algorithm after definite intervals of time.

Deadlock Recovery

1.) Pessimistic Approach
↳ abort all deadlocked processes
↳ abort one process at a time and decide next to abort after deadlock detection

2. Optimistic Approach
↳ Pre-empt some resources from process & give these resources to other processes until the deadlock cycle is broken.

Camlin Page
Date / /

Que-2.) Find :-

1.) Need Matrix

2.) Is system in safe state? If yes find safe sequence.

3.) If request from P_2 arrives for (1, 1, 0, 0) can request be immediately granted.

4.) If request from P_5 arrives for (0, 0, 2, 2, 0) can it be granted?

Sol -

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_1	2	0	0	1	4	2	1	2	3	3	2	1	2	2	1	1
P_2	3	1	2	1	5	2	5	2	5	3	2	2	2	1	3	1
P_3	2	1	0	3	2	3	1	6	7	4	2	6	0	2	1	3
P_4	1	3	1	2	1	4	2	4	8	7	3	8	0	1	1	2
P_5	1	4	3	2	3	6	6	5	9	11	6	10	2	2	3	3

So order :- P_1 P_4 P_5 P_2 P_3

Sol-3.) In this type of question we first update the table

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_1	2	0	0	1	4	2	1	2	2	2	2	1	2	2	1	1
P_2	4	2	2	1	5	2	5	2	4	3	3	2	1	0	3	1
	2	1	0	3					3	1	2	1				
									+	1	1	0	0			
									=	4	2	2	1			

Now check for safe state using Banker's algo.

If in safe state then request can be granted else not.

So OS actually performs futuristic approach in deadlock avoidance that is it first checks if the request can be granted or not. If it is not possible to ~~grant~~ mantain safe state with that request then it doesn't grant that request.

Time complexity = $O(n^2 m)$ (n → no of process & m → no of resources)

Deadlock Avoidance (Banker's Algorithm)

→ Handles multiple instances of some resources, for only one instance of each resource we use resource allocation graph algorithm.

Three things which need to be known before using Banker's are:-

1.) how many instances of each resources each process can request (max)
2.) how many instances of each resource each process currently hold (Allocation)
3.) how many instances of each resource is available in the system (Available)

Safe state – If we are able to execute all the processes in system without going into unsafe state (deadlock) then safe state.

Safe sequence – Order in which processes must be executed to ensure safe state.

Que – Find the need matrix & check whether if system is in safe state or not. If yes then find safe sequence

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_1	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_2	1	0	0	0	1	7	5	0					0	7	5	0
P_3	1	3	5	4	2	3	5	6					1	0	0	2
P_4	0	6	3	2	0	6	5	2					0	0	2	0
P_5	0	0	1	4	0	6	5	6					0	6	4	2

Sol – Need matrix (max – Allocation)

	A	B	C	D
P_1	0	0	0	0
P_2	0	7	5	0
P_3	1	0	0	2
P_4	0	0	2	0
P_5	0	6	4	2

A B C D

But available is 1 5 2 0

So P_1 needs 0 0 0 0 & available is 1 5 2 0.

⇒ P_1 can execute. Now available = 0 regined available + allocation (after we released)

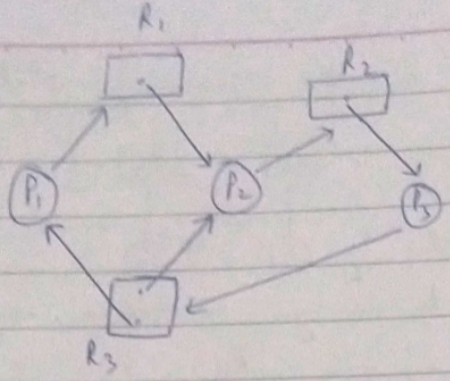
= 1 5 3 2

With 1 5 3 2 we can execute P_3 as need of P_3 is 1 0 0 2.

⇒ New available = 1 3 5 4 + 1 5 3 2 = 2 8 8 6

Now we can execute P_4 ⇒ available = 2 14 11 8. Now P_5 & finally P_2. Safe Sequence :- P_1 P_3 P_4 P_5 P_2 (there can be multiple sequences)

Eg :-



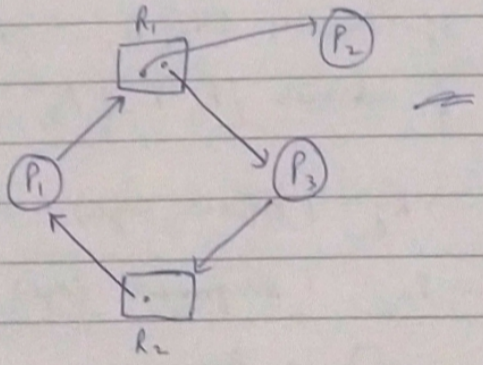
Que - Check whether process is in deadlock ?

Sol - Always check if there is any process which is not waiting for a resource. Here a cycle is formed so in deadlock

Another method (using matrix)

	Allocation			Request			Availability		
	R_1	R_2	R_3	R_1	R_2	R_3			
P_1	0	0	1	1	0	0	0	0	0 → So deadlock
P_2	1	0	1	0	1	0			
P_3	0	1	0	0	0	1			

Eg :-



Sol - Here P_2 is not waiting for any resource so instead of cycle this is not in deadlock.

Another method (using matrix)

	Allocation		Request		Availability		
	R_1	R_2	R_1	R_2	R_1	R_2	
P_1	0	1	1	0	0	0	→ So we can fulfill P_2 request & then it will release R_1 & then it can be used
P_2	1	0	0	0			
P_3	1	0	0	1			

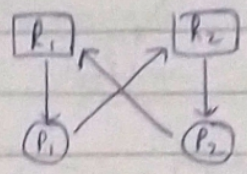
victim instead of process in the running state.

Circular wait → Violating circular wait

Circular wait can be eliminated by first giving a natural number to every resource.

$f : N \rightarrow R \quad (R_1, R_2, \ldots, R_n)$

→ Allow every process to acquire resources only in increasing or decreasing order of the resource number



Initially

P_1	P_2
R_1	R_2
R_2	R_1

→ leads to deadlock as R_2 acquired by P_2 & R_1 by P_1

solution →

P_1	P_2
R_1	R_1
R_2	R_2

→ only increasing or decreasing order

→ if a process requires a lesser number (in case of increasing order) than it must release all the resources larger than required number.

Resource Allocation graph (in deadlock)

Directed graphs use to find whether deadlock is there or not

Set of vertices

V → Set of Processes $\{P_1, P_2, \ldots, P_n\}$

→ Set of resources $\{R_1, R_2, R_3, \ldots, R_m\}$

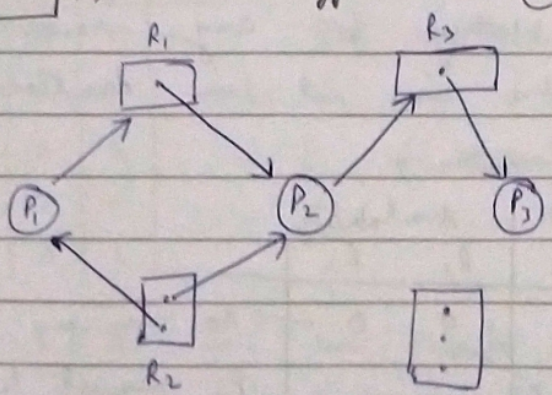
Edges

E → $P_i \rightarrow R_j$ (Request edge)

→ $R_j \rightarrow P_i$ (Assignment edge)

▭ ⇒ Resource type ○ ⇒ Process

eg:-



$P = \{P_1, P_2, P_3\}$

$R = \{R_1, R_2, R_3, R_4\}$

Note – If there is no cycle in graph then no deadlock. If cycle then may be deadlock.

4.) Ignorance / Ostrich algo - ignore the problem as if it does not exist.

Prevention

Mutual exclusion → It will insure that no deadlock occurs. But the cost of prevention is high. So we use in prevention in only those systems where consequences of deadlock are very severe, Even if one condition violates then no deadlock can occur. Prevention violates one of the necessary conditions

In order to remove mutual exclusion, we have to make a resource shareable which is impossible as for printer it can be used at a time only by one process.

Conclusion - We cannot violate mutual exclusion.

Hold & Wait condition → Methods to violate Hold & Wait

1.) Conservative approach :- Process is allowed to start execution if and only if it has acquired all the resources (Less effecient, not implementable, easy, deadlock independent)

2.) Do not hold - Process will acquire only desired resources, but before making any fresh request it must release all the resources that it currently hold. (effecient, implement)

3.) Wait timeout - We place a max time upto which a process can wait. After which process must release all the holding resources.

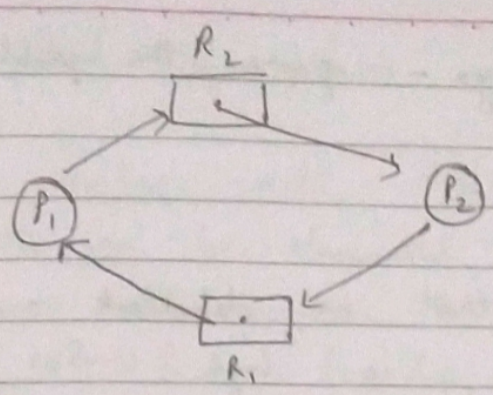
No-pre-emption → Method to avoid no-pre-emption

Forcefull pre-emption - We allow a process to forcefully preempt the resource holding by other process.

→ This method may be used by high priority proc or System process.

→ Process which are in waiting state must be selected as a

Resources - Sharable & Non-sharable (types)



R_1 is used by P_1 & P_1 needs R_2 which is used by P_2. P_2 needs R_1 So no process is leaving its resources & waiting for other process to leave. Thus deadlock.

System model

→ Every process will request for the resource.
→ If entertained then process will use the resource.
→ Process must release the resource after use.

Necessary conditions of Deadlock (all conditions must occur simultaneously)

→ Mutual exclusion - At least one resource type in the system which can be used in non-sharable mode. (one at a time / one by one) eg :- Printer.
→ Hold & Wait :- A process is currently holding at least one resource & requesting additionally which are being held by other process.
→ No-pre-emptive :- A resource cannot be pre-empted from a process by any other process. Resource can be released only voluntarily by the process holding it.
→ Circular wait condition - Each process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource.

Deadlock Handling Methods

1.) Prevention - means design such a system which violate at least one of four necessary conditions of dead lock & ensure independence from dead lock
2.) Avoidance - System maintains a set of data using which it takes a decision weather to entertain a new request or not, to be in safe state.
3.) Detection & Recovery - Here we wait until deadlock occurs & once we detect it we recover from it.

→ best in terms of avg response time

Round Robin (~~non~~ pre-emptive)

→ It is designed for time sharing system where it is not necessary to complete one process and then start another, but to be responsive and divide time of the CPU among the process in ready state.

→ Here ready queue will be treated as circular queue

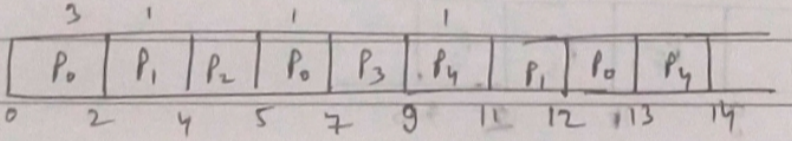
→ We use a time quantum

eg :-

P_id	A.T.	B.T.	T.A.T	W.T.
P_0	0	5	13 - 0 = 13	8
P_1	1	3	12 - 1 = 11	8
P_2	2	1	5 - 2 = 3	2
P_3	3	2	9 - 3 = 6	4
P_4	4	3	14 - 4 = 10	7

Time quantum = 2

Queue = P_0 P_1 P_2 P_0 P_3 P_4 P_1 P_0 P_4



Advantages :- Performs best in terms of avg response time, works well in case of time sharing system, client server architecture, interactive system, kind of SJF implementation.

Disadvantage :- long process may starve, depends heavily on time quantum, no idea of priority

Deadlock

In a multiprogramming system, a number of process compete for limited no of resources and if a resource is not available at that instance then process enters into waiting state.

If a process is unable to change its waiting state indefinitely because the resources requested by it are held by another waiting process. This system is said to be in deadlock.

Camlin Page
Date / /

→ suffers from starvation

Priority Algorithm (non-pre-emptive)

→ Here a priority is associated with each process.

→ At any instance of time out of all the available processes, CPU is allocated to the process which possess the highest priority (number may be higher or lower)

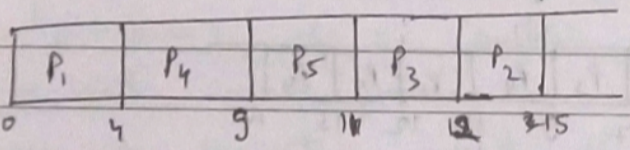
→ Tie solved FCFS order.

→ no importance to A.T. or B.T.

→ supports both pre-emptive & non-pre-emptive.

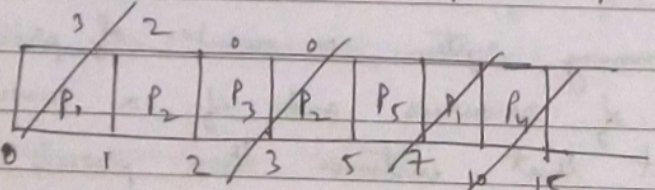
Non-pre-emptive (Note- Number with higher priority is given preference)

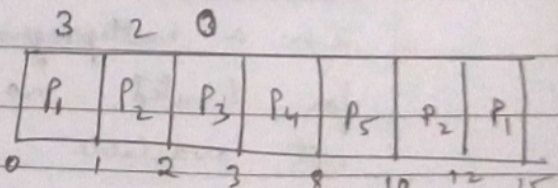
P_id	A.T.	B.T.	Priority	T.A.T.	W.T.
P_1	0	4	2	4-0=4	0
P_2	1	3	3	15-1=14	11
P_3	2	1	4	12-2=10	6
P_4	3	5	5	9-3=6	1
P_5	4	2	5	11-4=7	2



Pre-emptive version

P_id	A.T.	B.T.	Priority	T.A.T.	W.T.
P_1	0	4	2	15-0=15	11
P_2	1	3	3	12-1=11	8
P_3	2	1	4	3-2=1	0
P_4	3	5	5	8-3=5	0
P_5	4	2	5	10-4=6	4





Advantage of Priority scheduling

Provides a facility of priority specially for system processes, allows to run imp process first even if it is a user process

Disadvantage of Priority scheduling

Process with smaller priority may starve for CPU, no idea of response time or waiting time

How to save from starvation :- By using aging technique which gradually increase the priority of process that wait for long time.

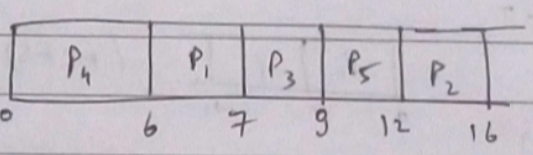
SRTF (pre-emptive) is optimal as it guarantees minimal avg waiting time.

Shortest Job First (non-pre-emptive) / Shortest Remaining Time First (SRTF) (Pre-emptive) → leads to starvation (some case)

→ Out of all available process, CPU is assigned to the process having smallest burst time requirement (no priority, no seniority)
→ If there is a tie, FCFS is use to break the tie.
→ Can be used both with non-pre-emptive & pre-emptive approach
→ Pre-emptive version (SRTF) is also called as optimal as it guarantee minimal avg waiting time

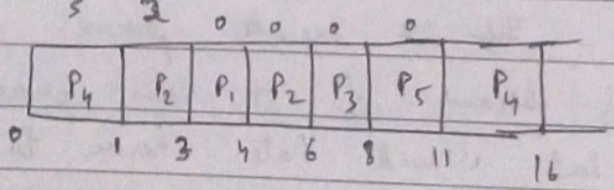
Non-Pre-emptive approach

Pid	A.T.	B.T.	T.A.T	W.T.
P_1	3	1	7-3=4	3
P_2	1	4	16-1=15	11
P_3	4	2	9-4=5	3
P_4	0	6	6-0=6	0
P_5	2	3	12-2=10	7



Pre-emptive approach (Greedy approach)

Pid	A.T.	B.T.	T.A.T.	W.T.
P_1	3	1	4-3=1	0
P_2	1	4	6-1=5	1
P_3	4	2	8-4=4	2
P_4	0	6	16-0=16	10
P_5	2	3	11-2=9	6



Advantage :- SRTF (pre-emptive) gaurantees minimal avg W.T., better avg response time than FCFS

Disadvantage :- Cannot be implemented as there is no way to know the burst time of a process, Process with longer CPU burst time will go to starvation, No idea of priority, process with long burst time have poor response time.

→ suffers from starvation

Priority Algorithm (non-pre-emptive)

→ Here a priority is associated with each process.

→ At any instance of time out of all the available processes, CPU is allocated to the process which possess the highest priority (number may be higher or lower)

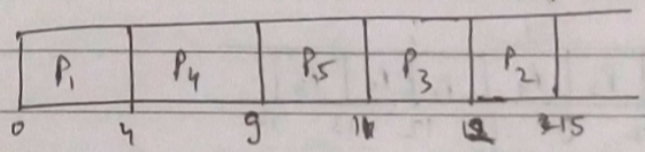
→ Tie solved FCFS order.

→ no importance to A.T. or B.T.

→ supports both pre-emptive & non-pre-emptive.

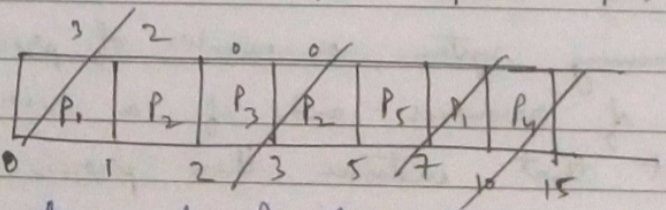
Non-pre-emptive (Note- Number with higher priority is given preference)

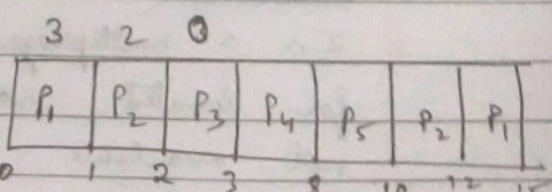
P_id	A.T.	B.T.	Priority	T.A.T.	W.T.
P_1	0	4	2	4-0=4	0
P_2	1	3	3	15-1=14	11
P_3	2	1	4	12-2=10	6
P_4	3	5	5	9-3=6	1
P_5	4	2	5	11-4=7	2



Pre-emptive version

P_id	A.T.	B.T.	Priority	T.A.T.	W.T.
P_1	0	4	2	15-0=15	11
P_2	1	3	3	12-1=11	8
P_3	2	1	4	3-2=1	0
P_4	3	5	5	8-3=5	0
P_5	4	2	5	10-4=6	4





Advantage of Priority scheduling

Provides a facility of priority specially for system processes, allows to run imp process first even if it is a user process

Disadvantage of Priority scheduling

Process with smaller priority may starve for CPU, no idea of response time or waiting time

How to save from starvation :- By using aging technique which gradually increase the priority of process that wait for long time.

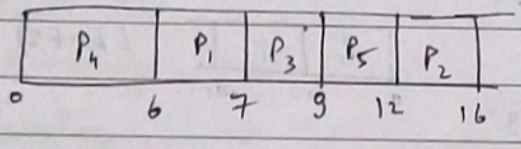
SRTF (pre-emptive) is optimal as it guarantees minimal avg waiting time.

Shortest Job First (non-pre-emptive) / Shortest Remaining Time First (SRTF) (Pre-emptive) → leads to starvation (some case)

→ Out of all available process, CPU is assigned to the process having smallest burst time requirement (no priority, no seniority)
→ If there is a tie, FCFS is use to break the tie.
→ Can be used both with non-pre-emptive & pre-emptive approach
→ Pre-emptive version (SRTF) is also called as optimal as it guarantees minimal avg waiting time

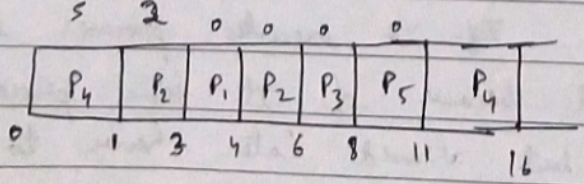
Non-Pre-emptive approach

Pid	A.T.	B.T.	T.A.T	W.T.
P_1	3	1	7-3=4	3
P_2	1	4	16-1=15	11
P_3	4	2	9-4=5	3
P_4	0	6	6-0=6	0
P_5	2	3	12-2=10	7



Pre-emptive approach (Greedy approach)

Pid	A.T.	B.T.	T.A.T.	W.T.
P_1	3	1	4-3=1	0
P_2	1	4	6-1=5	1
P_3	4	2	8-4=4	2
P_4	0	6	16-0=16	10
P_5	2	3	11-2=9	6



Advantage :- SRTF (pre-emptive) gaurantees minimal avg W.T., better avg response time than FCFS

Disadvantage :- Cannot be implemented as there is no way to know the burst time of a process, Process will longer CPU burst time will go to starvation, No idea of priority, process with large burst time have poor response time.

If a process has to wait for its turn because processor was biased then it is starvation. If processor is unbiased then it is convoy effect not starvation

Camlin Page Date / /

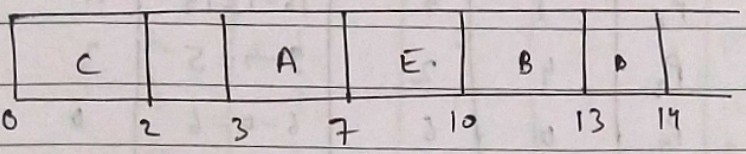
FCFS (first come first serve) (no starvation)

→ Simplest scheduling algorithm, it assign CPU to the process which arrives first
→ easy to understand & uses queue data structure
→ always non-pre-emptive in nature

eg:-

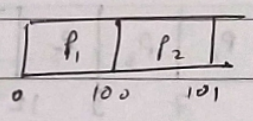
Pid	A.T.	B.T.	T.A.T (E.T - A.T.)	W.T. (T.A.T - B.T.)
A	3	4	7 - 3 = 4	0
B	5	3	13 - 5 = 8	5
C	0	2	2 - 0 = 2	0
D	5	1	14 - 5 = 9	8
E	4	3	10 - 4 = 6	3

Gran chart



Convoy Effect

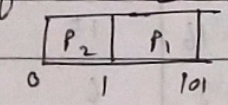
Pid	A.T.	B.T.	W.T.
P_1	0	100	0
P_2	1	1	99



(FCFS)

Convoy effect - Smaller process have to wait for long time before for bigger process to release CPU.

So in above example W.T. is very large for P_2 this can be avoided if we reverse the sequence of execution that is first P_2 & then P_1.



Disadvantage of FCFS:- It executes processes on the basis of FCFS but because of it some processes with small burst time but arrived later have to wait for a long time. For this reason some modefications are done

Advantage of FCFS:- Simple, easy to use, easy to understand

Where to use:- must be used with background processes where execution is not urgent.

Conclusion - If you run a smaller process before a bigger process then avg waiting time is reduced.

CPU scheduling

- Non-Preemptive
- Pre-emptive

Non-Pre-emptive – If CPU is given to one process then it cannot be taken back unless the process completes or process leaves CPU voluntarily to perform some I/O operations or to wait for an event.

Pre-emptive – Here process can be given CPU from other process if it has higher priority. Also if a process switches from running state to ready state because time quantum expires.

CPU scheduling terminology

→ Burst time / execution / running time – time process require for running on CPU.

→ Waiting Time – time spend by a process in ~~CPU~~ ready state waiting for CPU.

→ Arrival Time – when a process arrives in ready state

→ Exit Time – when process completes execution & exit from system.

→ Turn Around Time – Total time spend by process in CPU

$$T.A.T = E.T. - A.T. = B.T. + W.T. \quad (Typ)$$

→ Response Time – Time between a process enters ready queue and get scheduled on the CPU for first time.

~~CPU~~ Criteria for CPU scheduling algorithm

→ Avg Waiting Time – ↓ (good algo) or T.A.T ↓ (good)

→ Avg Response Time – ↓ (good algo)

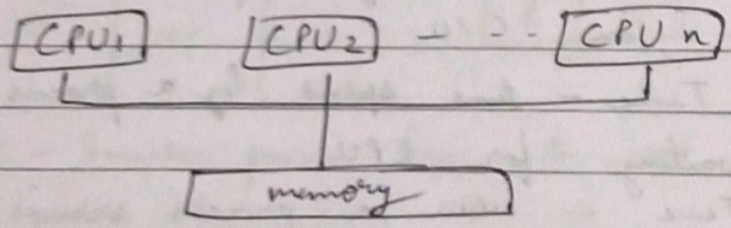
→ CPU utilization – ↑ (good)

→ Throughput – no of process executing per unit time ↑ (good)

→ Good system efficiency means both CPU & peripherals should be used effectively.
→ Scheduler should chose both CPU as well as I/O devices process so that no one waits.

Multiprocessing OS / Symmetric Assymetric Processing

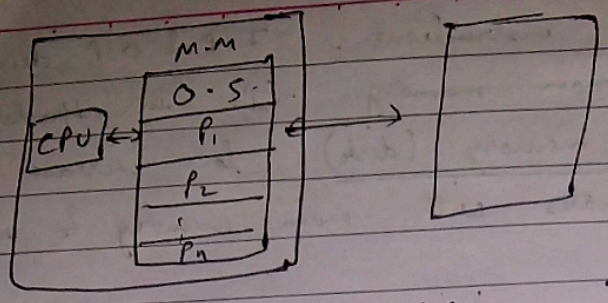
→ Two or more CPU with on a single computer, in close communication sharing the system bus, memory & other I/O devices.
→ Different processes run simultaneously, true parallel execution.
→ Two types of multiprocessing OS:-
Symmetric – one OS control all CPU, each CPU has equal rights
Assymetric – Master slave architecture, system task on one processor and application on other processor or one for I/O devices. They are easy to design but less effecient.
Advantage – Increased throughput, increased reliability, cost saving, battery effecient, true parallel processing
Disadvantage – more complex, large main memory, overhead of coupling reduce throughput



CPU scheduling

→ A process execution consist of a cycle of CPU execution & I/O execution.
→ Normally every process begins with CPU burst that may be followed by I/O burst, then another CPU burst & then I/O burst & so on, eventually in last will end up on CPU burst.
CPU bound process – These are the processes which require most of the time on CPU.
I/O bound process – These are the processes which require most of the time on I/O devices or peripherals.

S.M.



So here content switching is done until there is no process left.

→ Maximise CPU utilization

→ Multiprogramming means more than one process in main memory which are ready to execute.

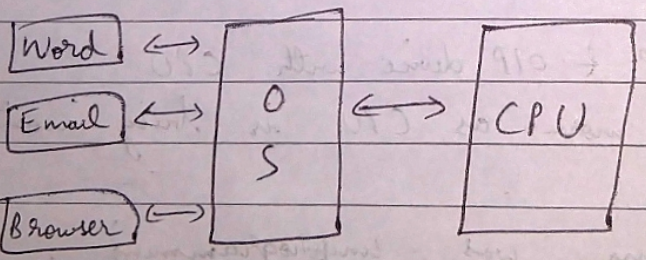
→ Process generally require CPU time & I/O time.

→ CPU is never idle unless there is no process ready to execute or at time of content switch.

Advantages - High CPU utilization less waiting time, response time, etc, may be extended to multiple users, now-a days useful when load is more.

Disadvantages - difficult scheduling, main memory management is required, memory fragmentation, paging (non-contiguous memory allocation)

Multitasking OS / Time Sharing / Fair share / Multiprogramming with Round Robin

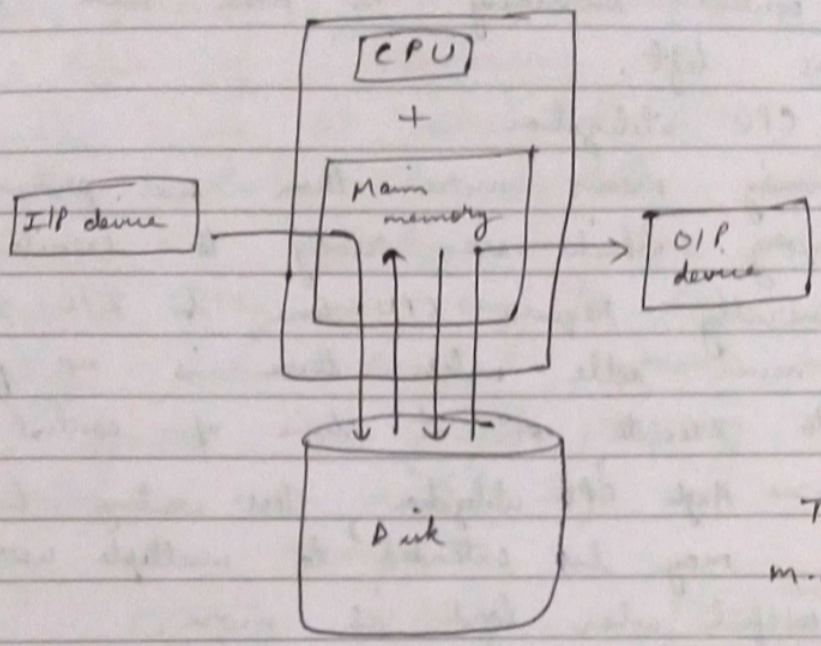


→ Multitasking is multiprogramming with time-sharing

→ Only one CPU but switches between process so quickly that it gives illusion that all executing at same time.

→ Better response time & executing multiple processes together.

to wait for other instructions. I/P & O/P devices are much slower than main memory. In spooling we introduced secondary memory (disk) to increase CPU utilization. CPU directly communicates with main memory & not with secondary memory.



So I/P data is first stored in disk & then brought back to m.m. CPU processes it & then again transfers it to disk & then finally O/P is obtained from disk.

Transfer of data between m.m & disk is very fast.

→ In spooling data is stored first onto the disk & then CPU interact with Disk via m.m.

→ Spooling is capable of overlapping I/O operations of for one job with CPU operations of other jobs.

Advantages

1-) no interaction of I/P & O/P device with CPU

2-) CPU utilization is more as CPU is busy most of time.

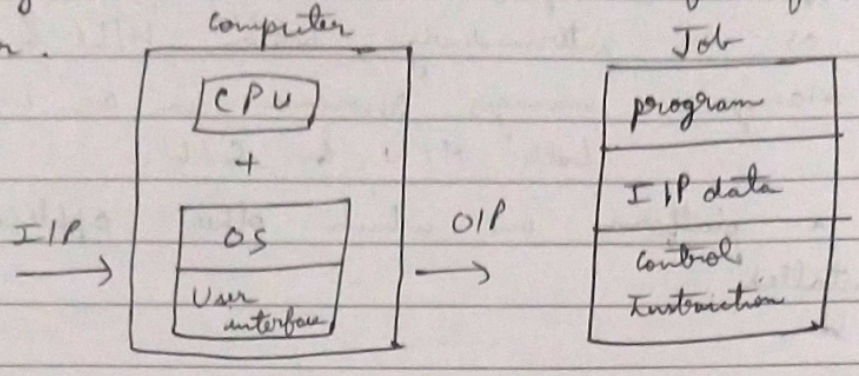
Disadvantage

→ In starting spooling was uniprogramming.

Multiprogramming OS

In previous OS, CPU waits until one process gets completed. Process requires CPU time as well as I/P, O/P operations. So even if process is in I/P, O/P CPU waits instead of taking another process. But in multiprogramming CPU keeps on working & moves on to next process.

form of punch cards after processing.
So in mainframes OS was very simple, always present in memory & main task was to transfer from one job to another.

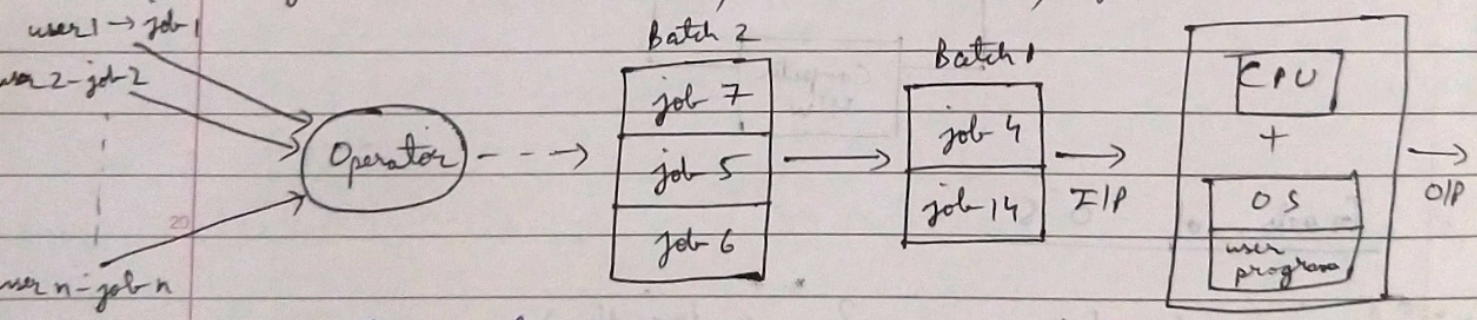


Batch processing (improves time required in mainframe)

→ jobs with similar needs are batched together & executed through the processor as a group.

→ operator starts jobs as a deck of punch cards into batch with similar needs.

→ e.g. FORTRAN batch, COBOL batch, python batch, etc.



Advantages of Batch OS

1.) in a batch job executes one after another saving time from activities like loading compiler
2.) during a batch execution no normal intervention is needed.

Disadvantages of Batch OS / Scope of Improvement

1.) memory limitation
2.) interaction of I/P & O/P devices directly with CPU.

Spooling (Simultaneous peripherals operations online)

In spooling, CPU utilization is better as CPU don't have

Operating System

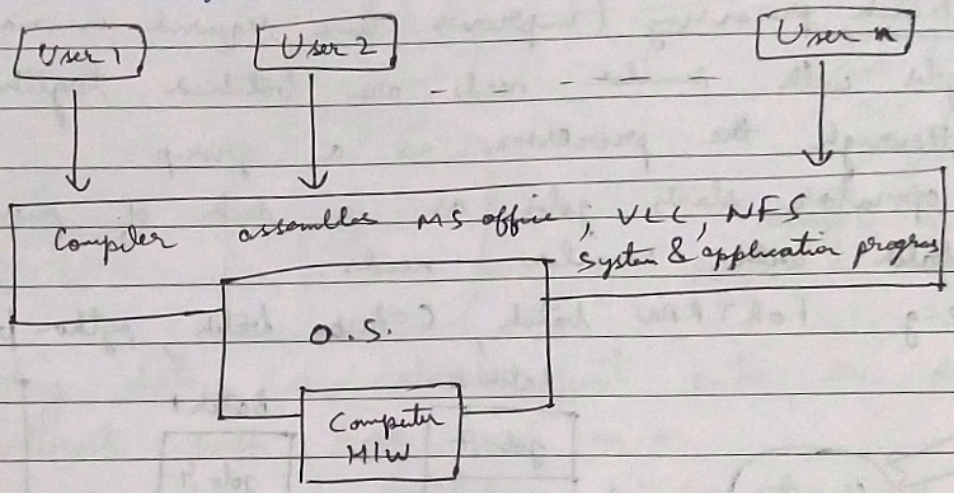
Operating system is system software.

→ It acts as an intermediary between H/W & user

→ Resource Manager - manages resources in an unbiased fashion both H/W & S/W.

→ Provides a platform on which other application programs are installed.

Abstract view of System



Goals of O.S.

→ Primary goal (convenience & user friendly)

→ Secondary goal (effecency)

Functions of OS

1.) Process management 2.) Memory management 3.) I/O device management
4.) File management 5.) Network management 6.) Security & protection

Batch Operating System

B for Batch OS there we mainframe computers

Mainframe :- Common I/P & O/P devices were card readers & tape drivers, user prepare a job which consisted of the program, I/P job is given in the form of punch cards & O/P is also obtained in the