Deadlocks

In multiprogramming environment, we have limited resources that are being shared by several processes. A process requests resource and if the resource is not free/Available, the process enters a waiting state. Sometimes, the resource which has been requested is held by some other waiting process, this situation is called as Deadlock.

* Deadlock can be illustrated from a law passed by a Kansas legislature in 20th century which says "When two trains approach each other at a crossing, both should stop and start only when other is gone".

System model → A system has finite number of resources which has to be distributed among several competing processes.

⇒ Each resource may have 1 -- n instances, which should be identical i.e. the request of a process can be satisfied with allocation of any instance.

⇒ A process may utilize a resource only in the following sequence —

1. Request → Process first makes a request for the resource. Obviously, Request ≤ Resources Available. If the request cannot be granted immediately, the process has to wait.

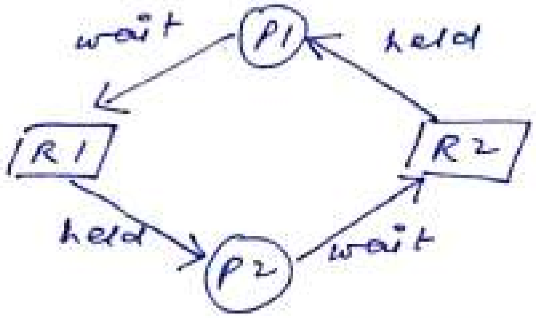
2. Use → The process can use/utilize the resource.

3. Release → The process releases the resource.

⇒ The Request() and Release() are available as system calls.

e.g. ⇒ request(), release(), open() file, close() file etc.

⇒ A set of processes are in a state of deadlock when every process in the set are waiting for an event that can only be caused by other process in the set.



Deadlock example

⇒ In the above example, Resource 'R2' is allocated to P1 and it is waiting for resource 'R1'. Similarly, Resource 'R1' is allocated to P2 and it is waiting for resource 'R2'.

Necessory conditions for Deadlock → A Deadlock may arrise if the following four conditions hold simulton-eously in a system —

1) mutual Exclusion → for this, atleast one resource in the system must be in non-sharable mode i.e. only one process can use that resource.

2) Hold and wait → A Process is holding one resource and waiting for other, which is being held by some other process.

3) No Preemption → Resources cannot be preempted i.e. Resources can only be released by the processes voluntarily.

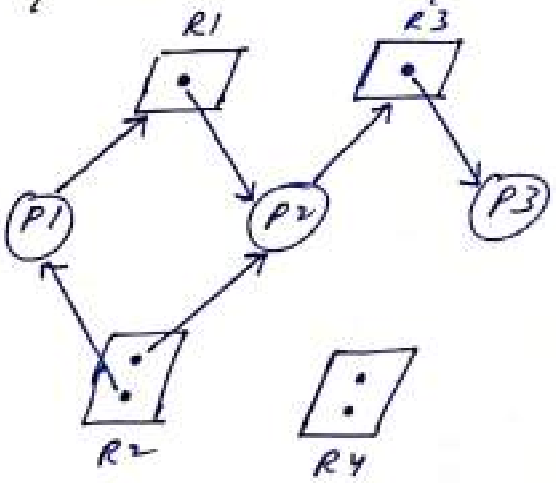
4) circular wait → A set $\{P_0, P_1, --- P_{n-1}\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2 and --- P_{n-1} is waiting for a resource held by P_0.

⇒ It is emphasized that all four conditions hold simultaneously. However, the circular-wait condition implies hold and wait.

Resource Allocation graph → Deadlocks can be described in a more precise manner using a directed graph called as Resource allocation graph.

⇒ This graph consists set of vertices V and set of edges E. The set of vertices V is partitioned into—

1) set of all active processes, $P = \{P_1, P_2, P_3 -- P_n\}$
2) set of all resources, $R = \{R_1, R_2, -- R_m\}$



Resource Allocation graph

⇒ A directed edge from 'P_i' Process to 'R_j' Resource is called a Request edge ($P_i \rightarrow R_j$). It indicates that P_i Process has requested for an instance of Resource R_j.

⇒ A directed edge from Resource 'R_j' to process 'P_i' is called a assignment edge ($R_j \rightarrow P_i$). It indicates a Resource 'R_j' instance has been allocated to process 'P_i'.

⇒ When a process makes a request, Request edge is created. When that Request is granted, this Request edge becomes assignment edge and when the process Releases this resource, assignment edge is deleted.

⇒ The RAG can be interpreted as –

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

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

$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_1, R_2 \rightarrow P_2, R_3 \rightarrow P_3\}$

⇒ Given the definition, it can be shown that if the RAG contains cycle, then a deadlock may exist.

⇒ If each resource has a single instance and the RAG contain cycle, then definitely deadlock exists.

⇒ If each resource type has several instances, then cycle does not necessarily shows that a deadlock occurred.

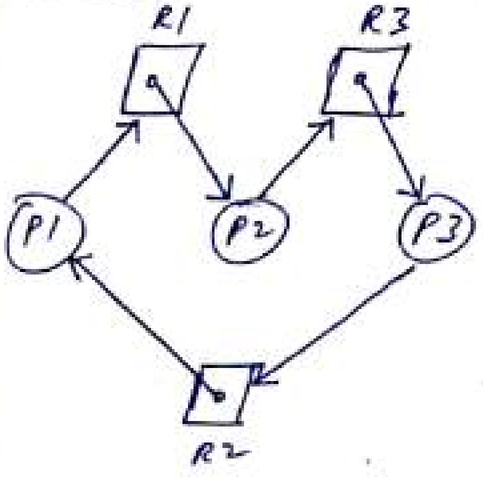


Figure – A (Deadlock)

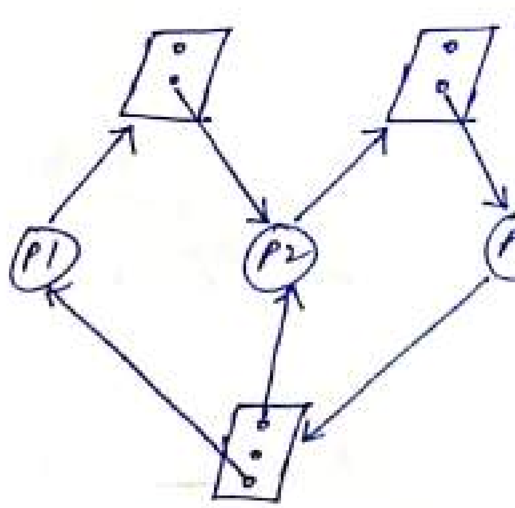


Figure – B (no Deadlock)

methods for Handling deadlocks → There are basically four methods/ ways in which we can deal with deadlocks —

1) Prevent the deadlocks (Deadlock prevention).
2) Avoid the deadlocks (Deadlock Avoidance)
3) Recover from deadlocks (Deadlock Recovery)
4) Ignore the deadlocks and pretend that deadlocks never occur in the system.

⇒ Deadlock Avoidance requires that the OS must have advance additional info. concerning which resources a process will require during its lifetime. with this info., the OS decides that whether the current request can be satisfied or must be delayed.

e.g. ⇒ RAG Algorithm, Banker's Algorithm.

⇒ Deadlock prevention provides a set of methods that ensures that atleast one of the necessary condition cannot hold.

Deadlock Prevention → we will elaborate on how we can prevent the necessary conditions for deadlocks —

1. Mutual Exclusion → sharable Resources like Read-only files must be in sharable mode and mutual Exclusion must only be imposed on non-sharable resources.

2. Hold and wait → for this, we can have a protocol that ensures when a process requests a resource, it does not hold any other resource. In this, if a process is to wait for the allocation of any resource, it should free all the resources that has already been allocated to it.

3. no preemption → this condition can be prevented by using a protocol that if a process is holding some resources and requesting other resources that cannot be immediately allocated, then resources currently allocated are preempted. the process will be started only when it can regain its old resources as well as the new ones.

⇒ this protocol is often applied to resources whose state can be easily saved like CPU Registers, memory space etc.

4. circular wait → To ensure that this condition never holds, we impose a ordering of Resources in the system and the processes can requests resources only in an increasing order of enumeration.

let $R = \{R_1, R_2, -- R_m\}$ be the set of Resources. we assign to each resource a integer, that allows us to compare two Resources.

$F(R_i)$ — Indicates/gives the number assigned to R_i.

e.g. ⇒ $F(\text{tape drive}) = 1$
$F(\text{Printer}) = 5$
$F(\text{disk drive}) = 12$

⇒ if $F(R_i) < F(R_j)$, then process will first request R_i and only then R_j.

⇒ if process requests R_i then it should free-up all the instances of R_j.

⇒ if the above two rules are used, then circular wait cond^n cannot hold.

Deadlock Avoidance → Deadlock can be avoided by having additional info. about how many maximum resources can be requested by a process.

using this info, An Algorithm can be designed that ens-ures that the system will remain in safe state and never enter in a deadlock.

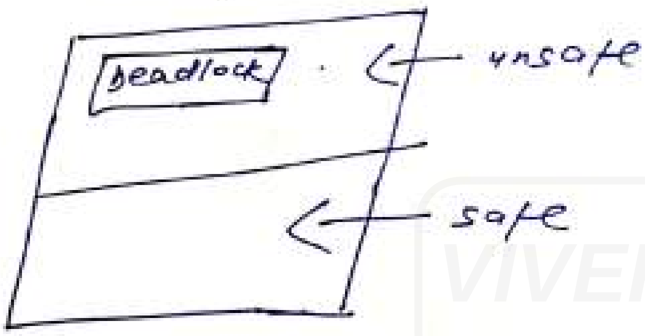
safe state -> A state is safe if the system can allocate resources in some order and still avoid a deadlock. A system is in safe state if there exists a safe sequence.

=> A sequence <$P_1, P_2, \dots P_n$> is safe, if for each P_i, the requests that P_i still can make can be satisfied by the currently available resources plus the resources held by all P_j's, where ~~P_j~~ $j<i$.

- In this, if the resources needed by P_i are not avail immediately, then P_i can wait till all P_j's finishes.

=> A safe state is not a deadlocked state.

=> An unsafe state may lead to deadlock.



- e.g =>

	MAX. Need	Allocated
P_0	10	5
P_1	4	2
P_2	9	2

suppose ~~at time~~ we have 12 magnetic tape drives. and at time t_0, ~~t_0 has~~ 5 has been allocated to P_0, 2 to P_1 and 2 to P_2. so, free Resource instance are 3.

At time t_0, the system is in safe state, the sequence is <P_1, P_0, P_2> a safe sequence.

=> suppose at time t_1, one more tape drive is allocated to P_2. now, the system is in unsafe state.

	max. need	Allocated	Avail
P_0	10	5	$\boxed{2}$
P_1	4	2	
P_2	9	3	

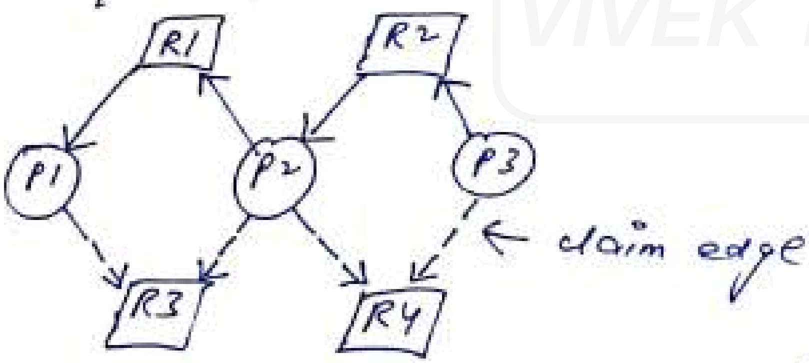
so, neither $<P_1, P_0, P_2>$ or $<P_1, P_2, P_0>$ will be a safe sequence.

Resource Allocation Graph Algorithm → This algorithm can be used to avoid deadlock only when we are having a single instance of each resource.

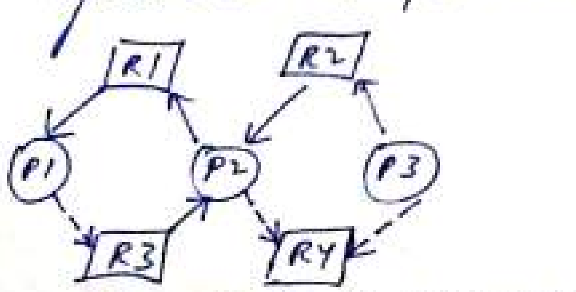
⇒ As we are having prior info. about the processes, we can make a Resource allocation graph (RAG) with a special edge called claim edge. A claim edge $P_i \rightarrow R_j$ indicates that P_i may request R_j at sometime in future.

⇒ when P_i request resource R_j, that can only be granted if converting the request edge does not result in the formation of a cycle.

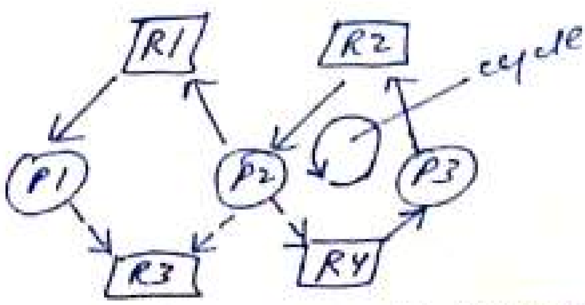
eg. ⇒



⇒ if a Request from P2 arrives for R3, R3 will not be allocated to P2 even it is free. Because it will leave the system in unsafe state.



granting R3 to P2 (unsafe)



Granting R4 to P3 (unsafe)

Banker's Algorithm → this algorithm can be used in a system where we have multiple instances of a Resource. This is called as Banker's Algorithm because it works on the same concept as bank, as bank never allocates its available cash in such a way that it could no longer satisfy the need of all its customers.

⇒ when a new process enters the system, it must declare the max. no. of instances of each resource it may need. This no. should not be greater than the total no. of resources in the system.

⇒ when a process requests a set of resources, the system must determine whether the allocation of these resource will leave the system in a safe state. if not, the process must wait for other processes to release the resources.

Data structure required → n → processes, m → Resources

⇒ Available → A vector of length 'm' indicates the no. of available resources.

⇒ MAX → A 'n×m' matrix defines the max. demand of each process.

⇒ Allocation → A 'n×m' matrix defines the no. of resources currently allocated to each process.

⇒ Need → A 'n×m' indicates the remaining resource need of each process.

need[i][j] = max[i][j] − Allocation[i][j]

Safety Algorithm →

1. let work and finish be two vectors of length 'm' and 'n'. Initialize, work = Available and finish[i] = false for i = 0, 1, -- n−1.

2. find an 'i' such that both—
   a. $Finish[i] == false$
   b. $need_i \le work$
   if no such 'i' exists, go to step 4.

3. $work = work + Allocation_i$
   $finish[i] = true$
   goto step 2.

4. if $finish[i] == true$ for all i, system is in safe state.

Resource Request Algorithm → let $Request_i$ be the request vector of P_i. when a request arrives—

1. if $Request_i \le Need_i$, goto step 2 otherwise Raise an error condition.

2. If $Request_i \le Available$, goto step 3 otherwise P_i must wait.

3. modify the states as—

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

4. find / run safety Algorithm to see if the system will be in safe state. If yes, Resources will be allocated otherwise P_i must wait.

Example → suppose we have five processes P_0 to P_4. system has three resource types A, B and C. A has 10 instances, B has 5 instances and C has 7.

suppose at time t_0, snapshot of the system is—

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2	7	4	3
P_1	2	0	0	3	2	2				1	2	2
P_2	3	0	2	9	0	2				6	0	0
P_3	2	1	1	2	2	2				0	1	1
P_4	0	0	2	4	3	3				4	3	1

⇒ suppose P_1 requests one additional instance of A and two instances of C.

so, $Request_1 = (1, 0, 2)$ or $<1, 0, 2>$

step1 — ~~work = <3, 3, 2>~~
~~finish = <false, false, false, false, false>~~

check, $Request_1 \le Need_1$

$Request_1 = <1, 0, 2>$, $Need_1 = <1, 2, 2>$

so, step1 is true, goto step2.

step2 — check, $Request_1 \le Available$

$Request_1 = <1, 0, 2>$, $Available = <3, 3, 2>$

so, step2 is true, goto step 3.

step3 — now, if we grant this request, the state of the system will be —

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

⇒ now, we will execute safety algorithm →

step1 — work = <2, 3, 0> and finish = <false, false, F, F, F>

step 2 - find a 'i' for which, $need_i \le work$, so, we have got P_1.

$need_1 = <0,2,0>$, $work = <2,3,0>$

so, finish[i] = true.

work = work + $Allocation_i$

so, work = <5, 3, 2>

so, finally after running the complete algorithm, we will get the safe sequence as—

$$\boxed{\text{safe sequence} = <P_1, P_3, P_4, P_0, P_2>}$$

---

note — when the system is in this state, a request from P_4 as <3,3,0> cannot be granted.

similarly, a request from P_0 as <0,2,0> cannot be granted because it will lead the system in unsafe state.

---

note — A system may have multiple safe sequences.

e.g. ⇒ In above example, another safe sequence is —

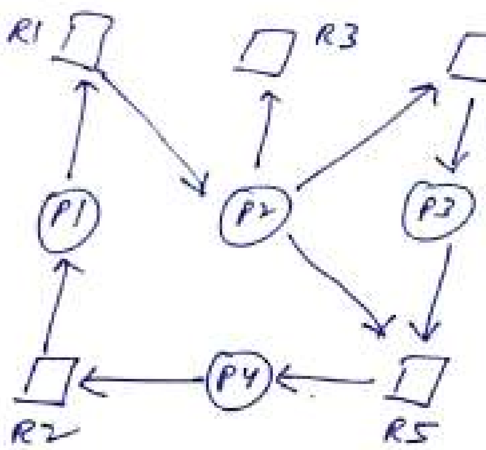
<P1, P3, P0, P2, P4>

Deadlock Detection → If the system do not have deadlock prevention or avoidance, then a deadlock may occur. so, we should have —

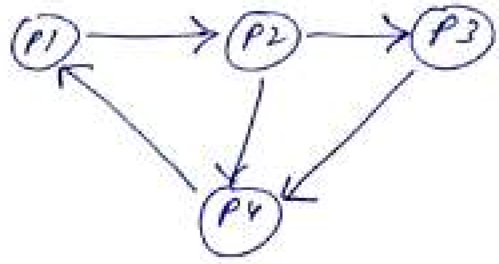
1. An Algorithm for deadlock detection.
2. A method for Recovery from deadlock.

1. single instance Resources → if all Resources have only a single instance of every resource, then deadlock can be detected using a wait-for graph.

⇒ It is an invariant of Resource allocation graph.



RAG



corresponding wait-for graph

⇒ Deadlock exists in the system if and only if wait-for graph contains a cycle.

⇒ To detect deadlocks, the system needs to maintain the wait-for graph and periodically search for the cycle.

2. Several instances of Resources → This Algorithm is a vari-ant of Banker's Algo.

1. let work and finish be two vectors. work = Available and if $Allocation_i \neq 0$, finish[i] = false.

2. find an 'i' such that

   a. finish[i] == false.

   b. $Request_i \leq work$

   if no such i exists, goto step 4.

3. work = work + $Allocation_i$.
   finish[i] = True.
   goto step 2.

4. if finish[i] = false for some i, the system is in deadlock. moreover if finish[i] == false, then process P_i is deadlocked.

e.g => five processes, P_0 to P_4.
three resources, A – 7 instances, B – 2, C – 6
suppose at time t_0, the system state is –

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

using Algo., we find the sequence $<P_0, P_2, P_3, P_1, P_4>$
so, the system is not in deadlock state.

* at T_1, if P_2 makes an additional request for an in-stance of C, after this the system is in deadlock.

Recovery from Deadlock -> There are basically two methods for Recover from a deadlock.

=> **Process Termination** -> To eliminate deadlock, we can use any of the two methods –

1. **Abort all deadlocked processes** -> This methods breaks the deadlock but at great expense. The results of the partial computation must be discarded and may have to recompute.
2. **Abort one process until deadlock eliminated** -> This method has a disadvantage that deadlock detection Algorithm has to run after Aborting one of the deadlocked process.

* selecting a process is also a trivial task. since we always wants to terminate a process whose termination incurs a min. cost. But, the min. cost is also dependent on various parameters.

Resource preemption → In this, we preempt resources from processes and give these resources to other processes until the deadlock cycle is broken.

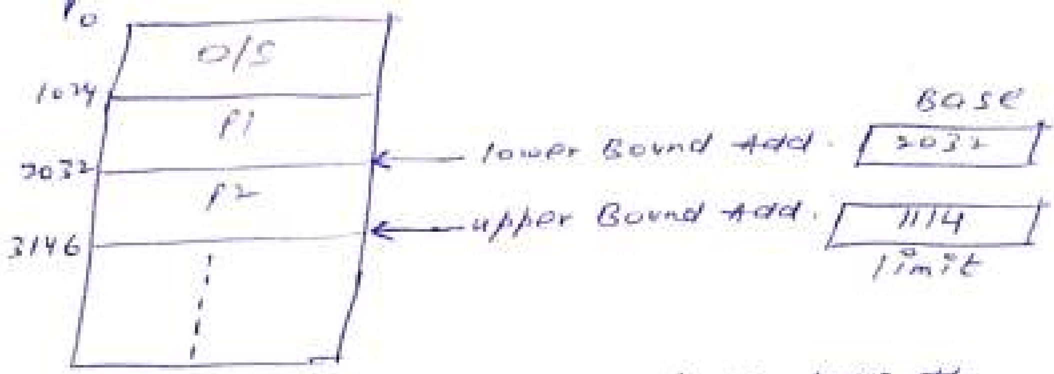
There are three issues that needs to be addressed—

1. **Selecting a victim** → In this, we must determine the order in which the resource is to be preempted, in order to incur minimum cost.
   Cost factor includes considering resources allocated & the amount of time the process has so far consumed.

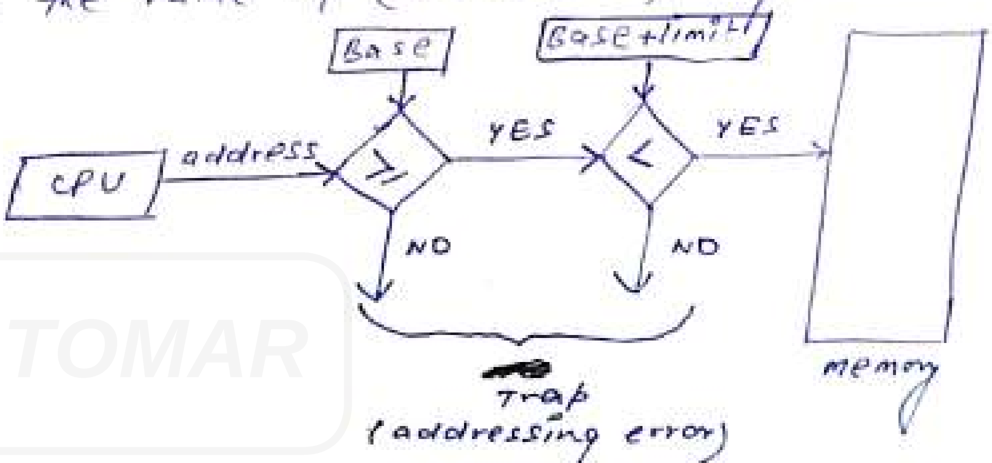
2. **Rollback** → The process whose resources has been preempted has to rollback to some safe state.
   One simplest soln is to abort the process and restart it as system will need to store very less info. about the processes.

3. **Starvation** → Using the above approach, it may happen that the same process is picked again & again as a victim. So, we must ensure that the starvation should not occur and a process is picked as a victim for a finite no. of times.

Basic Hardware → our main concern is protecting OS from user processes but also protecting user processes from one another.
As we know, Every process has a seprate memory space, we should have the ability to determine the Range of legal addresses for a process.
⇒ so, for this, we can have two addresses i.e. a lower bound address and an upper bound address.
⇒ one method is to store the lower bound address in one Register as Base Register and limit in other Register as limit Register.



⇒ so, any address generated by the CPU must be ~~less th~~ greater than or equals to the value of Base Register and less than the value of (Base+limit) Register.



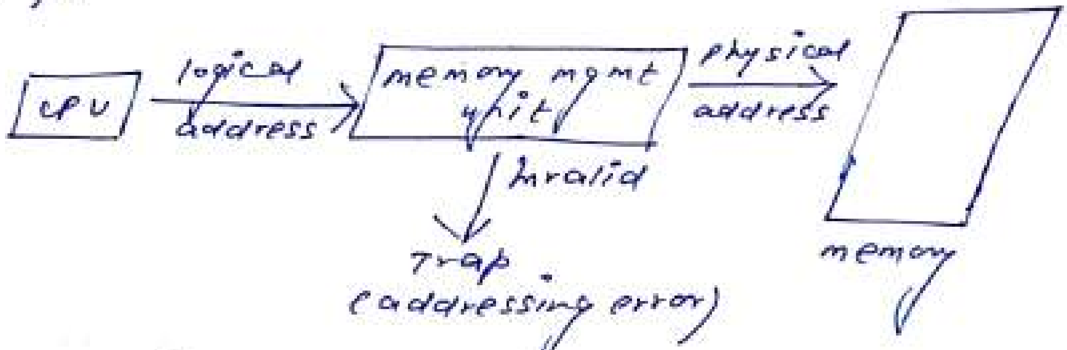
Protection with Base and limit Registers

---

logical versus physical Address space → An address generated by the CPU is Reffered as logical address, and the set of all logical addresses generated by CPU/program is called as logical address space / virtual address space.
⇒ The actual address, which is seen by main memory or loaded

into the memory address Register is called as physical address. The set of all physical addresses corresponding to logical addresses are known as physical address space.

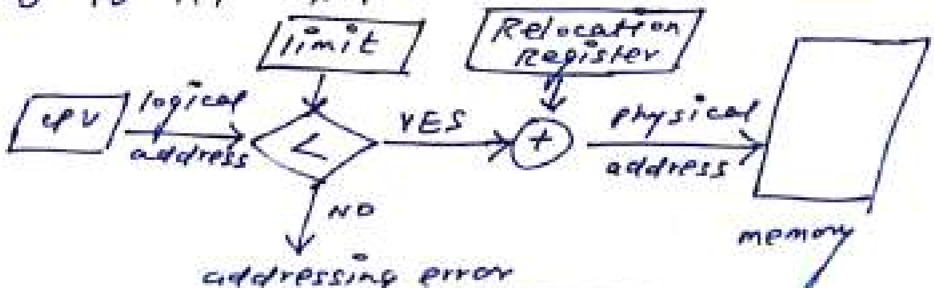
⇒ The Run-time mapping from logical to physical addresses is done by a H/w device, which is known as memory management unit (MMU).

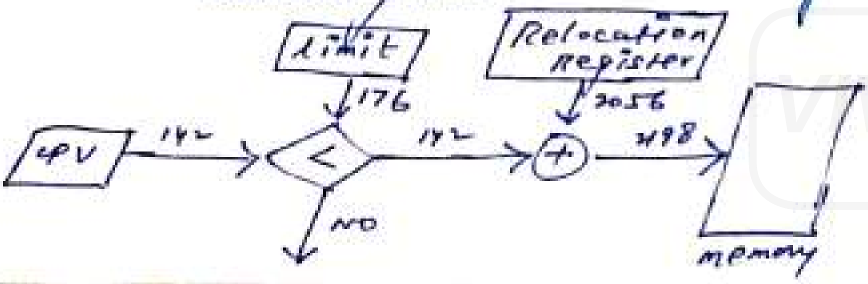


Generation of logical Addresses from CPU → CPU can generate ~~two form~~ logical address as a Relocatable factor, which can be added to the content of Base Register to get the actual physical address. If the logical address is greater than the limit defined for that particular process, then MMU generates a Addressing error as a Trap to OS.

⇒ now, the Base Register will be called as Relocation register.

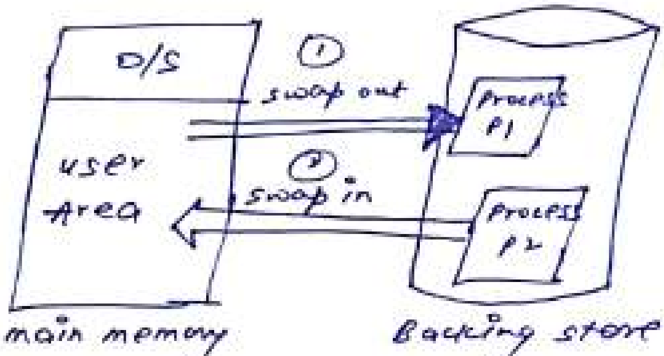
⇒ if the contents of Relocation register is 'R'. and the contents of limit Register is 'max'. then, the CPU can generate logical address in Range of 0 to max. and the physical address can be in the Range from 'R+0' to 'R+max'.





Swapping → A Process must be in main memory for execution. However, if the process is not currently attended by CPU / suspended or may be waiting for some I/O event, It can be swapped temporarily out of memory to a Backing store and then brought back into memory.

⇒ swapping is done to effectively utilize the memory in multi-programming environment.



⇒ Normally, a process that is swapped out is swapped back into the same memory space occupied previously. This Restriction is dependent on the address binding used.

⇒ If the address binding has taken place at compile or load time, then process cannot be easily moved to other memory space. But, if the Binding is at execution time, then the process can be swapped to new memory space.

⇒ The issues Related to swapping are—

1. The major part of the swap time is transfer time.
2. To swap out a process, the process must be completely idle, i.e. we never swap out a process with pending I/O.
3. new swapping schemes for swapping permits swapping only when there are many processes executing and swapping is halted, when the load on the system is reduced.

⇒ new versions of unix and windows uses advanced swapping schemes.

memory Partitioning → since we are working in multiprogramming environment, we must have environment such that multiple processes can reside in the main memory.

⇒ for this, the main memory has to be partitioned into smaller parts.

⇒ There are basically two techniques/approaches for memory partitioning—

1. MFT (multiprogramming with fixed no. of partitions)
2. MVT (multiprogramming with variable no. of partitions)

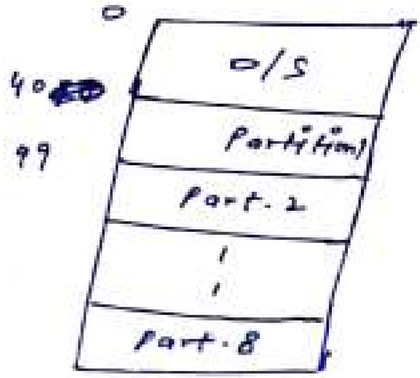
1. multiprogramming with fixed no. of Partitions (MFT) → As we know, that OS occupies some fixed portion of main memory and Rest can be used by user processes.

⇒ In this, memory is divided into a fixed no. of Partitions that may be of equal or unequal size.

a) Partitioning into fixed no. of Partitions of equal size →

⇒ In this, the memory is divided into fixed no. of Partitions of equal size.

⇒ suppose we have total of 512K of memory and O/S needs 40K. then, memory can be divided into 8 Partitions, each of size 59K.



⇒ so, any job that comes in the memory can be placed in any of the free partition.

⇒ There are basically two disadvantages of this—

1. The degree of multiprogramming is dependent on the no. of partitions.

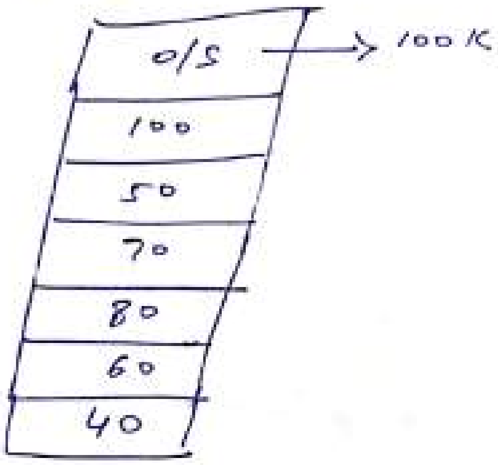
2. If a job comes which needs 70K of memory, then it cannot be placed in any of the partition. In this case, the programmer must use overlays to load the program.

Overlaying → overlaying means "replacement of a block of code with another one". It allows the program to divide into self contained object code called overlays.

b) Partitioning into fixed no. of partitions of variable size →

⇒ In this, available memory is divided into partitions of different / variable sizes.

⇒ Suppose we are having a total 500K of memory and O/S requires 100K of memory. So, we can divide it into 6 more partitions as—

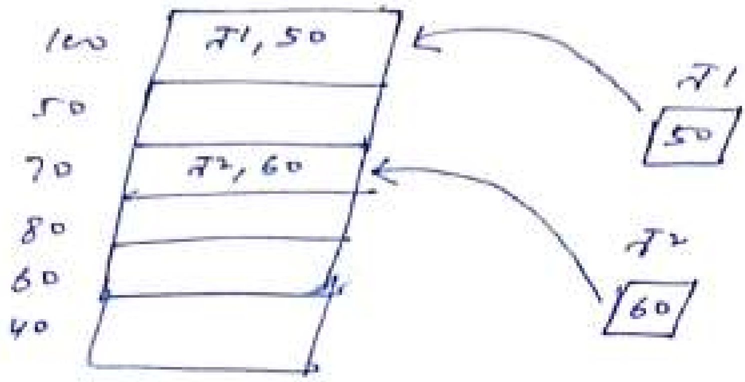


⇒ If any process arrives, then it can be placed in any block in which it can fit and the block must be free.

Placement Algorithm → when any process arrives, then there can be four diff. type of placement algorithms—

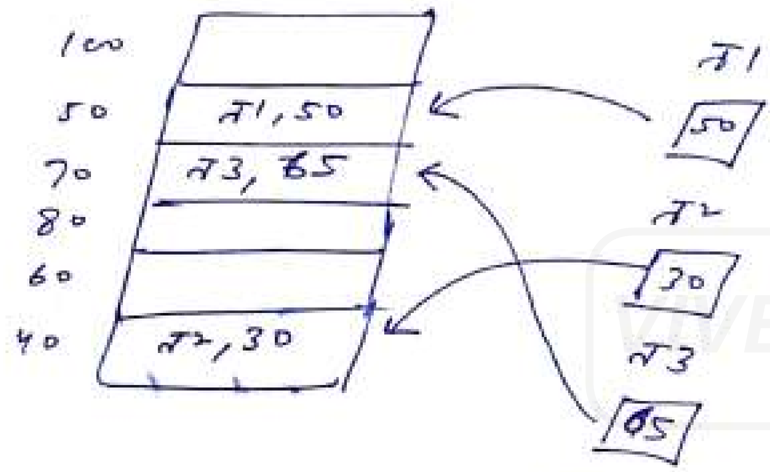
1) First Fit → It allocates the process to the first available partition, which is large enough to hold the process.

e.g. =>



2) <u>Best fit</u> -> It allocates the partition to a process, which is closest to it in terms of size.

e.g. =>

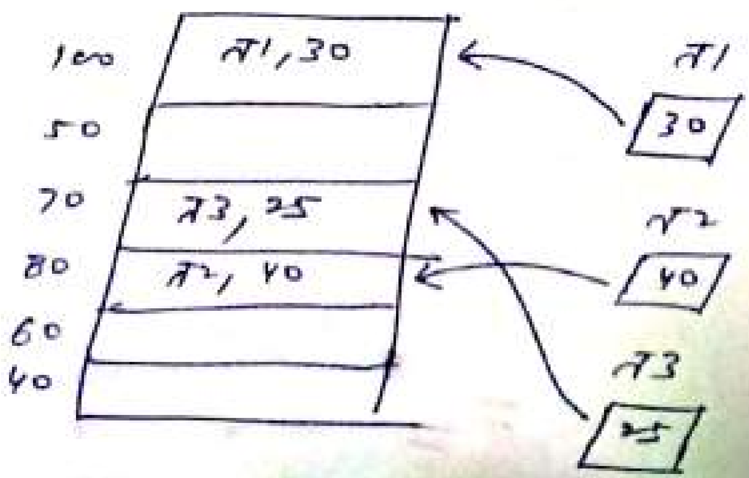


3) <u>Next fit</u> -> It allocates a partition to a process, which comes after the previously allocated partition and large enough to hold the process.

e.g. => In above case, If after प1, प2 come and need 30K of memory, then it will be allocated with third partition i.e. 70K

4) <u>Worst fit</u> -> It allocates a partition, which is the largest free partition.

e.g. =>

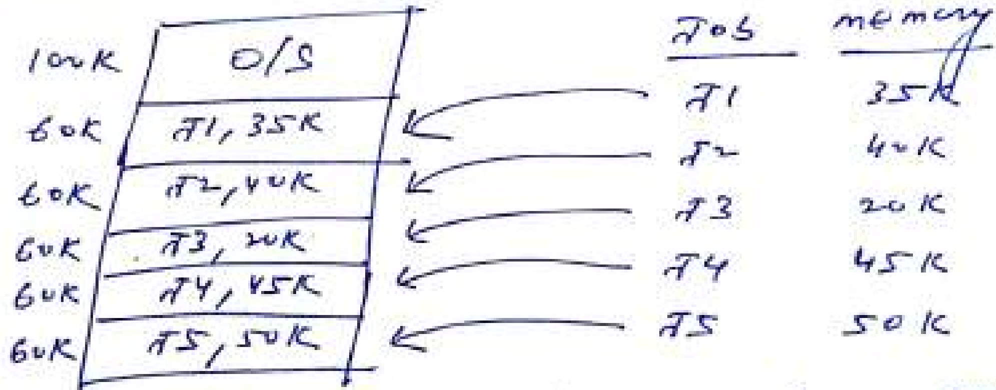


Fragmentation → Fragmentation is basically wastage of memory. fragmentation can arise in case of partition of fixed size as well as in partition of variable size partitions.

⇒ Fragmentation can be internal as well as external.

① Internal fragmentation → Internal fragmentation is the phenomenon in which there is a wasted space internal to a partition due to the fact that the allocated partition is larger than the job.

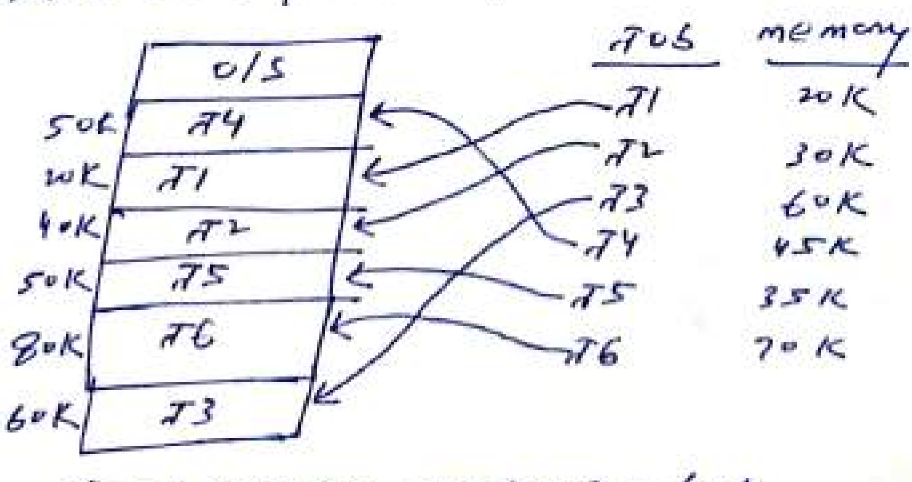
e.g. ⇒ ① suppose we are having a total of 400K of memory and OS needs 100K. Suppose we are having 5 partitions each of 60K.



So, here as the allocated partitions are larger than the processes/jobs, we are having wastage of memory.

Total wastage of memory = 110K due to internal fragmentation

② suppose we are using fixed partition of variable size with Best-fit partition.



So, total memory wastage = 40K

① External fragmentation →

② multiprogramming with variable no. of partitions (MVT) →

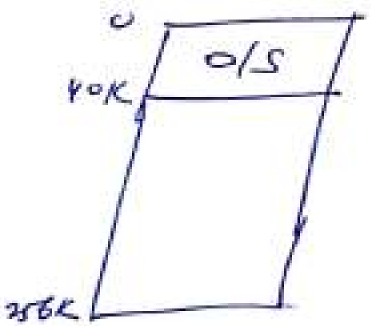
this memory partitioning is also known as dynamic partitioning because in this memory is partitioned on the basis of the requirement of job dynamically.

⇒ to overcome the difficulty in static partitioning (Internal fragmentation), this technique has been developed.

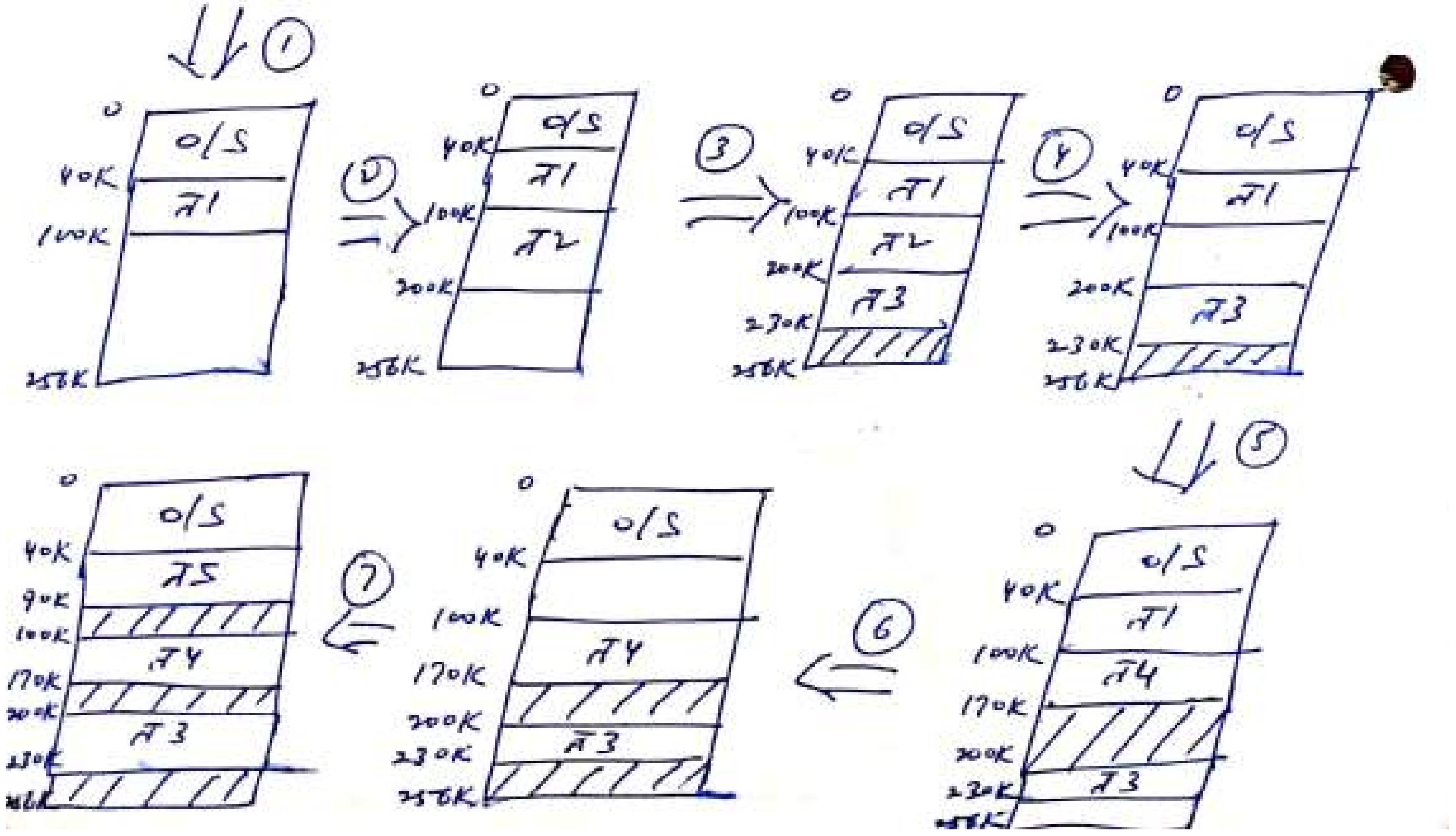
⇒ In this, the partitions are of variable length and number. when a process is brought into main memory, it is allocated exactly as much memory as it requires.

⇒ It completely removes the internal fragmentation.

e.g ⇒ suppose we are having total 256K of memory, out of which OS requires 40K and the jobs are –



Jobs	memory
J1	60K
J2	100K
J3	30K
J4	70K
J5	50K
J6	60K



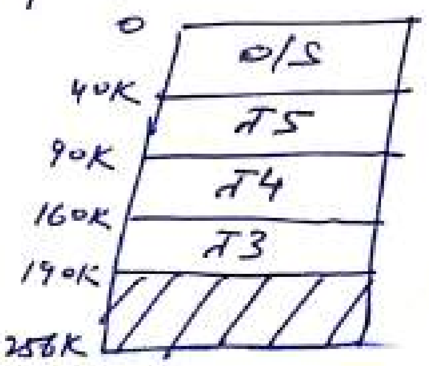
External fragmentation -> so, in the above figure, we can see that we are having total free memory as 66K but yet we cannot allocate this memory to P6. This is called as external fragmentation.

=> so, external fragmentation is the phenomenon in which we are having multiple holes in the memory but we are not able to fullfill the request of any process. As time goes on, memory becomes more and more fragmented and memory utilization declines.

=> this is called as external fragmentation because holes are external to the partitions that are allocated to jobs/processes.

Soln for External fragmentation -> one technique for overcoming external fragmentation is compaction. From time to time, the OS shifts the processes so that they are contiguous and so that all the free memory is together in one block

e.g => compaction in the previous figure results in a block of memory of 66K.



memory after compaction

=> compaction is also an overhead to the system, bcoz during the process of compaction, no task can be done by CPU.

Paging -> Paging is a memory mgmt technique that allows the physical space of a process to be non-contiguous.

⇒ In case of paged memory mgmt, every process is divided into a no. of pages.

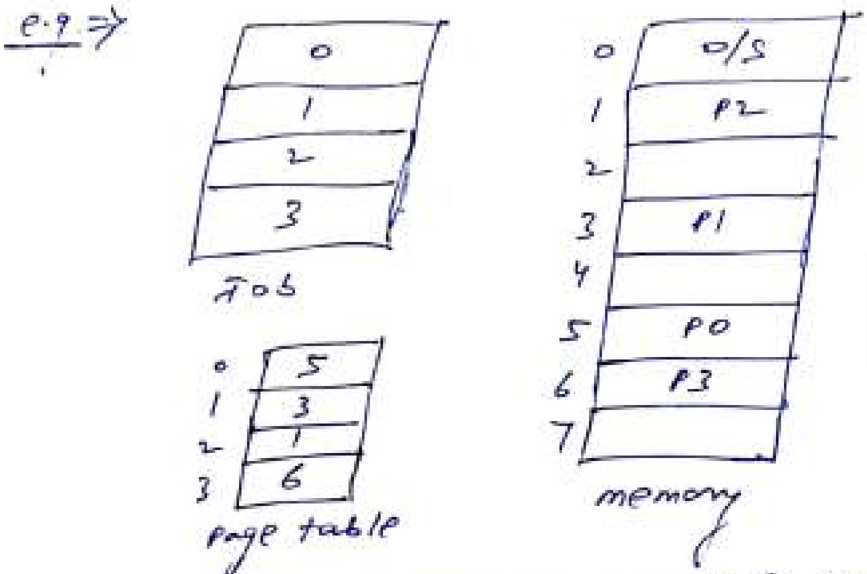
⇒ main memory is also divided into no. of partitions, whose size is same as page-size and these partitions are called frames. i.e.

$$\boxed{\text{Page size} = \text{frame size}}$$

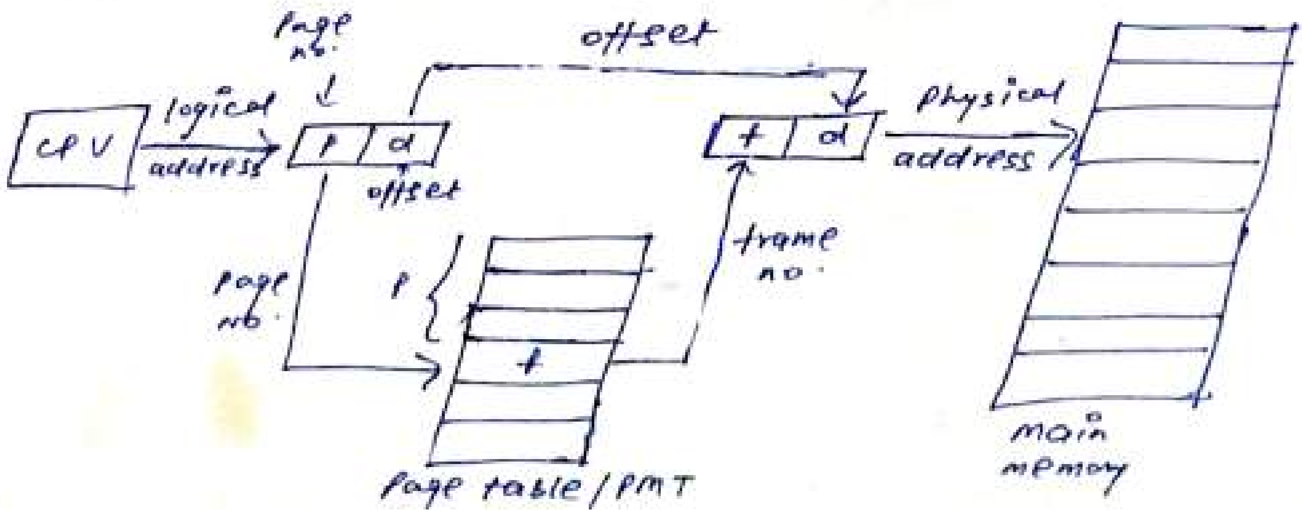
⇒ when a process is to be executed, its pages are loaded into any available memory frames.

⇒ logical memory (logical address space of process) is divided into a no. of pages, where every page is of same size except the last one, whose size can be less than the page size.

e.g. ⇒



⇒ when the CPU ~~gen~~ executes this process, it generates the logical addresses without bothering about where the page is located. so, page table provides a mapping of these logical addresses to physical addresses.



⇒ so every logical address has to be break into two components –

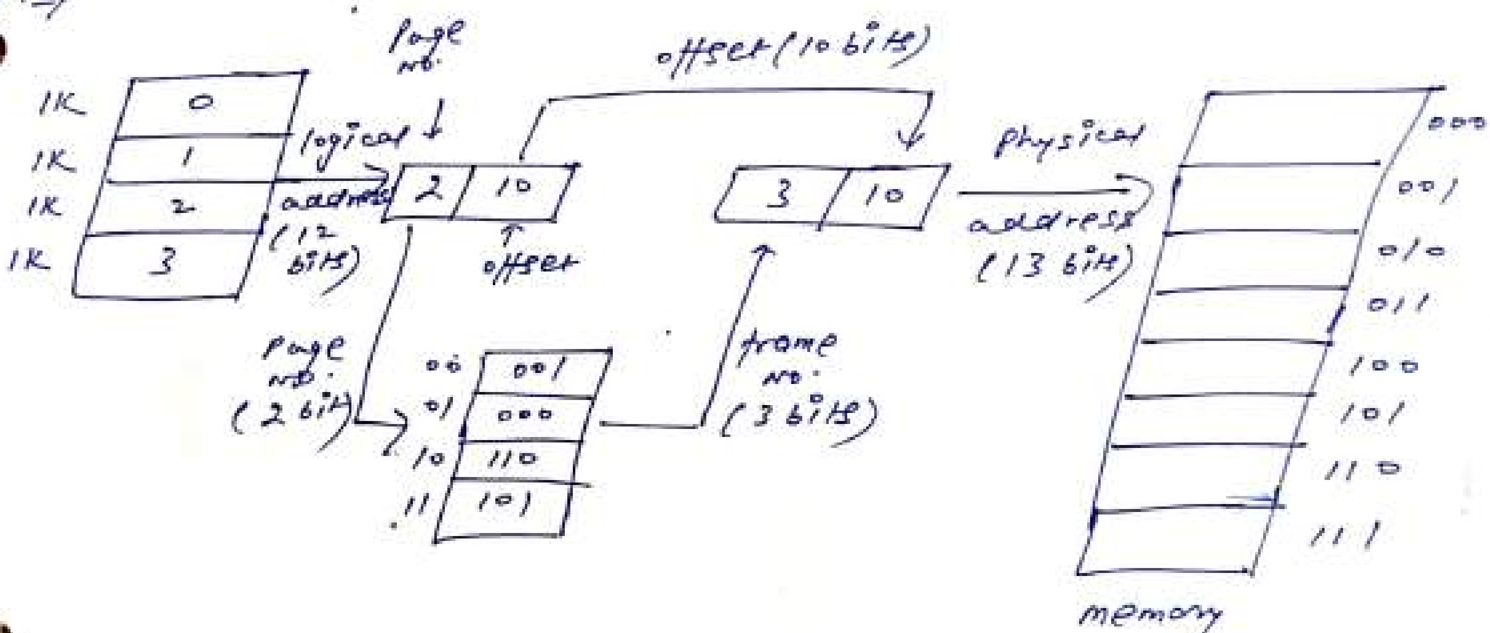
1. Page no. → It gives the index of page table, which contains base address of each page in physical memory.

2. Page offset → offset gives the displacement within an page.

⇒ If the size of logical address space is 2^m and a page size is 2^n then,

* higher-order (m−n) bits of logical address designate the page no.
* lower-order 'n' bits will designate the offset.

e.g. ⇒



Fragmentation in Paging → In case of paging, there isn't any external fragmentation. But, in paging there can be small amount of internal fragmentation. This happens when the logical address space is not a integer multiple of page size.

* when this happens, the max. internal fragmentation can be (P−1), where 'P' is page size.

e.g. ⇒ if page size is '10 kb' and logical address space is 41 kb, then we will allocate the '5' pages to the process.

so, internal fragmentation = 9 kb. (i.e. 10−1 (max.))

Technique	strengths	weaknesses
① fixed partitioning	simple to implement, little OS overhead	degree of m.p. is fixed, internal frag-mentation
② Dynamic partitioning	no internal fragme-ntation	External fragmentation or inefficient use of processor to counter External fragmentation
③ simple paging	no External fragmentation	A small amount of internal fragmentation
④ simple segmentation	no internal fragme-ntation	External fragmentation
⑤ virtual memory paging	no external fragmenta-tion, higher degree of multiprogramming	complex memory mgmt overhead
⑥ virtual memory segmentation	no internal fragmentation, higher degree of multi-programming, large virtual address space, protection and sharing	complex memory mgmt overhead

Hardware support → The main concern of paging is that how to store the page table. most system allocates a page table for each process.

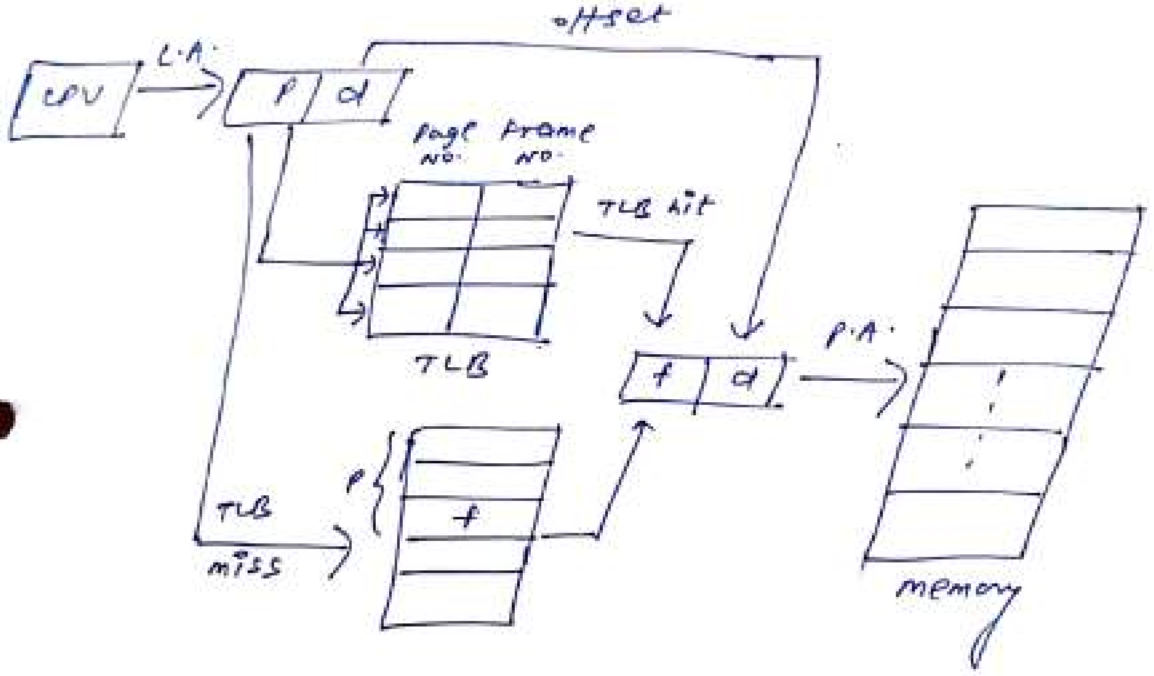
⇒ Page table can be stored in three ways -

① using set of Registers → In this, page tables are stored in Registers. The cpu dispatcher loads and reloads these registers. This method is satisfactory but can only be effecient, if page table is small.

② main memory → page tables can be stored in main memory and a PTBR(page table base Register) points to the page table. This approach is not much feasible bcoz for accessing a byte/word, we need two memory access. i.e. one for finding frame nu. and other for accessing actual data.

(2) using Translation look-aside buffer (TLB) → In this, we use a small, fast-lookup, associative and high speed cache called TLB. When the TLB is presented with an item, the item is compared with all the keys and returns the corresponding item.

⇒ TLB is fast, but the H/W is expensive.

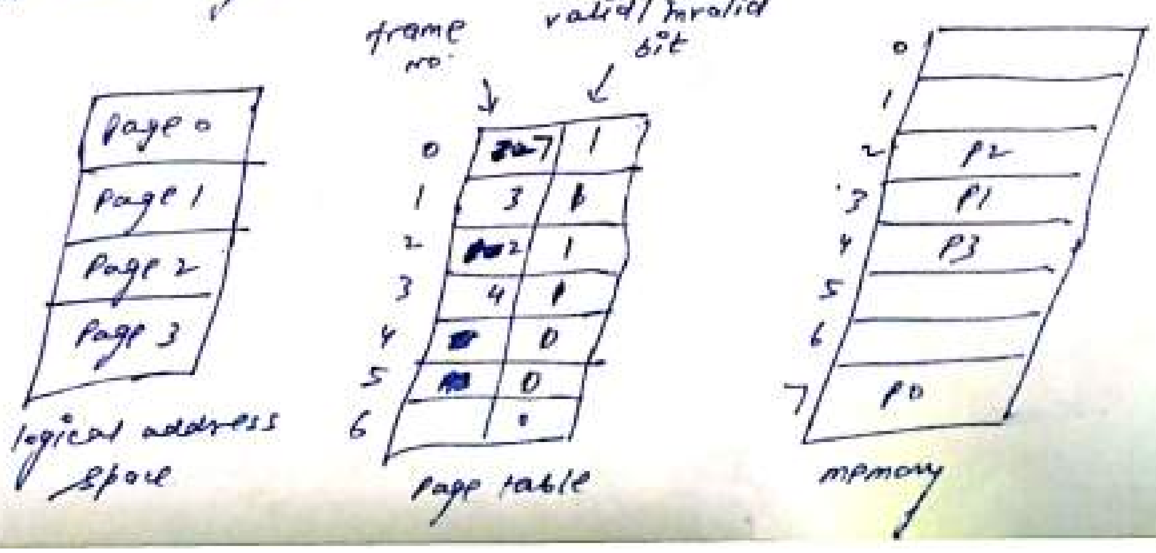


Paging H/W with TLB

Protection → memory protection in paging environment can be achieved by having extra bits in page table.

one additional bit is generally attached to each entry in the page table i.e. a valid-invalid bit. When this bit is '1' (valid) the associated page is of process logical address space.

one extra bit can be defined for controlling the Read only or Read write pages.



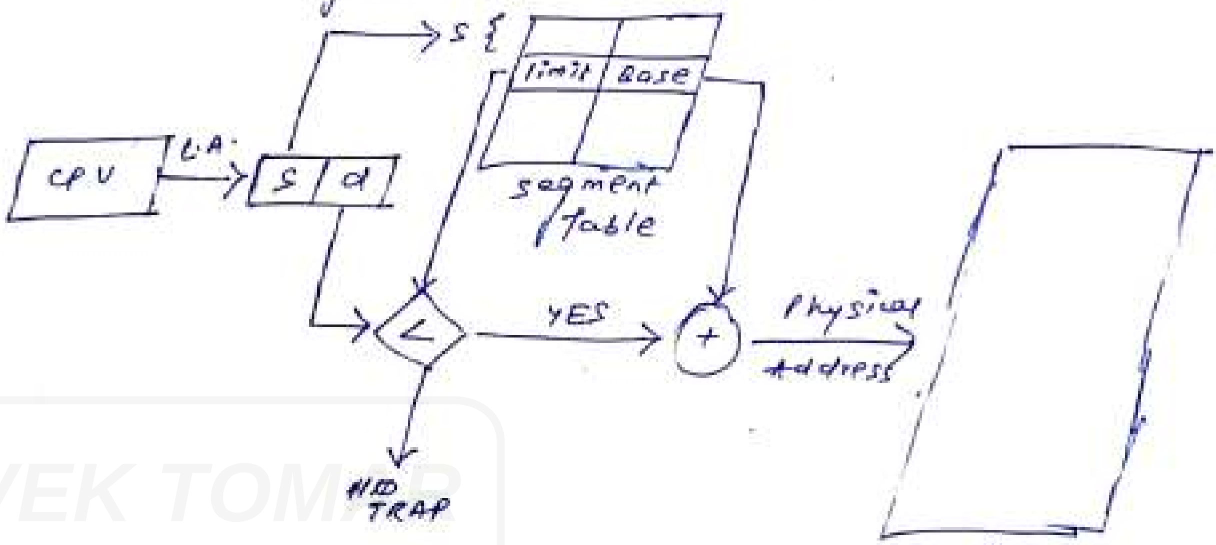
Segmentation → segmentation is a memory mgmt scheme which supports user view of memory.

In this, the logical address space is a collection of no. of segments, and it is not required that all segments are of same length i.e. segments can be of different length. However, there is a maximum segment length.

⇒ Because of use of unequal size segments, segmentation is similar to dynamic partitioning, However the difference is that the diff. segments can occupy non-contiguous memory locations.

⇒ In this, a logical address generated by CPU has two tuples–

$\langle \text{segment-number}, \text{offset} \rangle$



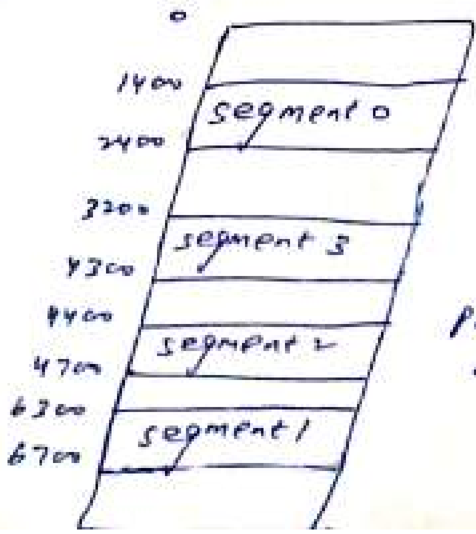
⇒ The segment no. from logical address is used as an index to segment table. The offset of a segment must be between 0 and limit of segment. When an offset is valid, it is added to the base of that segment.

segment 0	segment 1
segment 2	segment 3

logical address space

	limit	Base
0	1000	1400
1	400	6300
2	300	4400
3	1100	3200

segment table

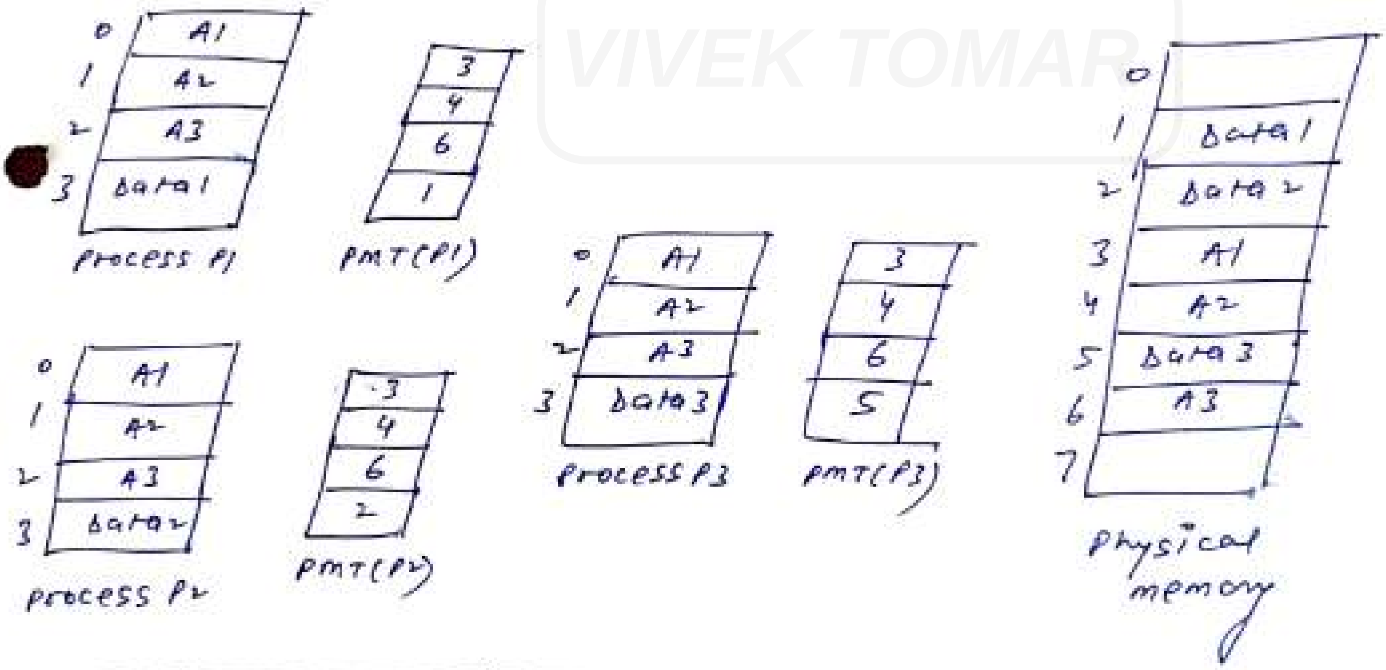


Physical memory

Sharing of Pages in Paging technique → An Advantage of Paging is the possiblity of sharing pages. so, the code can be shared among processes.

If we are having two or more processes that can share the code, then they can have shared pages however they will have their different data.

e.g. ⇒ Heavily used programs like Text editor, compiler, database systems, run time libraries etc. can be shared.

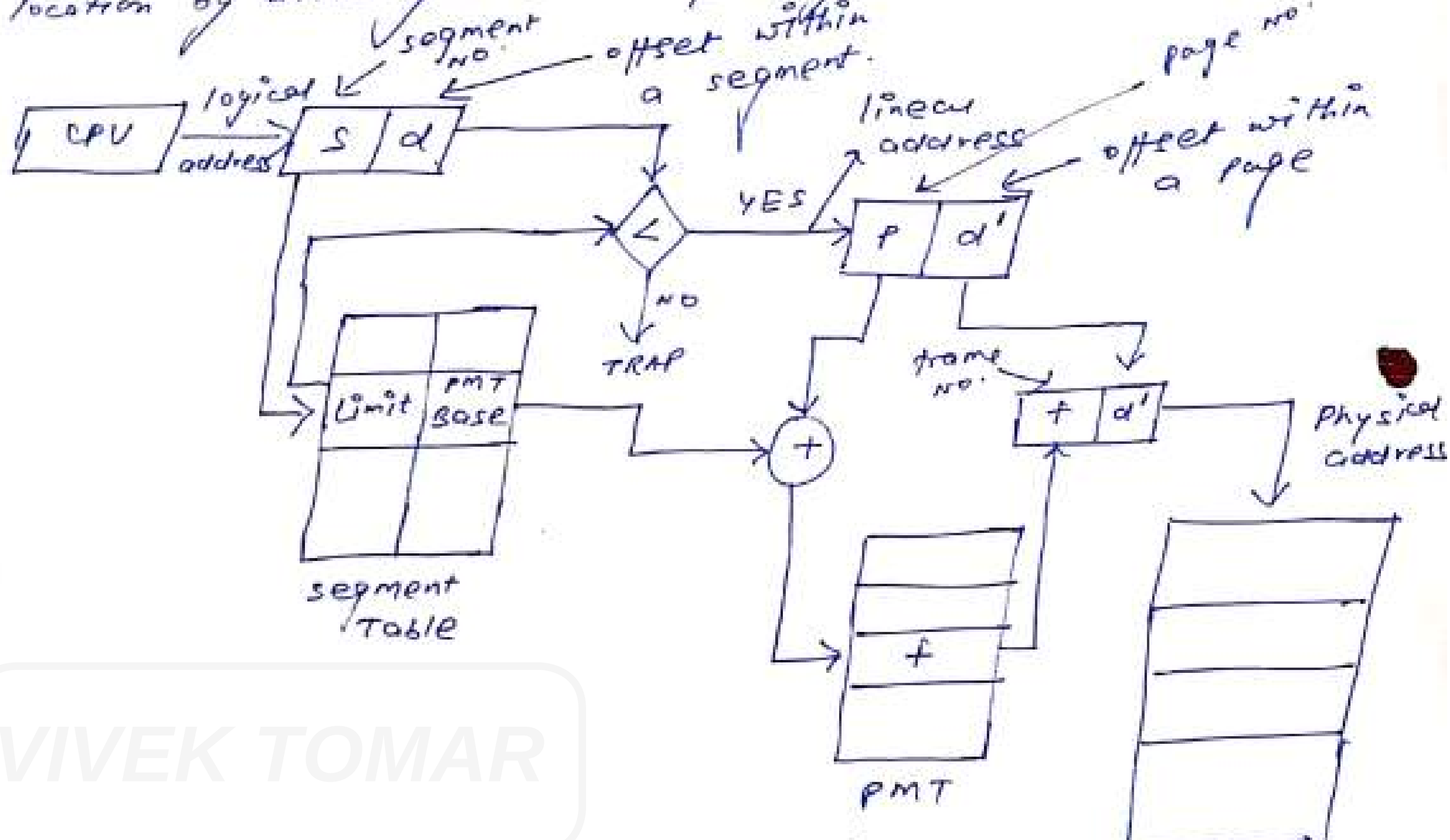


Paged segmentation → Paged segmentation is a memory mgmt technique that combines both paging and segmentation technique.

* In this, we exploit the advantages of both paging and segmentation.
* In paging, every user program is broken into number of pages of same size.
* In this, the user Program / logical address space is broken into a number of segments, and then the segments will be broken into no. of pages.

CPU $\xrightarrow[\text{address}]{\text{logical}}$ segmentation unit $\xrightarrow[\text{address}]{\text{linear}}$ Paging unit $\xrightarrow[\text{address}]{\text{Physical}}$ Main memory

* This technique is useful because if we are dividing a logical add. space into no. of segments, then it may be possible that for a particular segment, there isn't contiguous memory available. So, this technique first divides the logical add. space into segments and then ~~these~~ a segment can have non-contiguous location by dividing into no. of pages.



paged segmentation

1. CPU generates the logical address, which has two fields i.e. a segment no. and offset within the segment.
2. we have a PMT for each segment of logical add. space.
3. the segment table has two entries i.e. limit of the segment and base add. of PMT for that segment. The segment no. will be used to corresponding segment from segment table.
4. if the offset is less than the limit, then offset 'd' is nothing but logical address within the segment. so, this 'd' can be broken into page no. and offset within a page.
5. Since, we can have multiple PMT within a segment, the PMT base will be added to 'p' to get actual page no.

6. finally, after getting the actual frame no. of main memory, this frame no. will be combined with the offset within a page to get the actual/physical address.

---

virtual memory concepts ->

=> virtual memory is a technique that allows the execution of processes that are not completely in memory. using this technique, A illusion is created that user have more memory at disposal than the physical memory.

=> VM basically abstracts main memory into an extremly large memory, seprating the logical memory from the physical memory.

=> vm concept can be implemented through Demand paging.

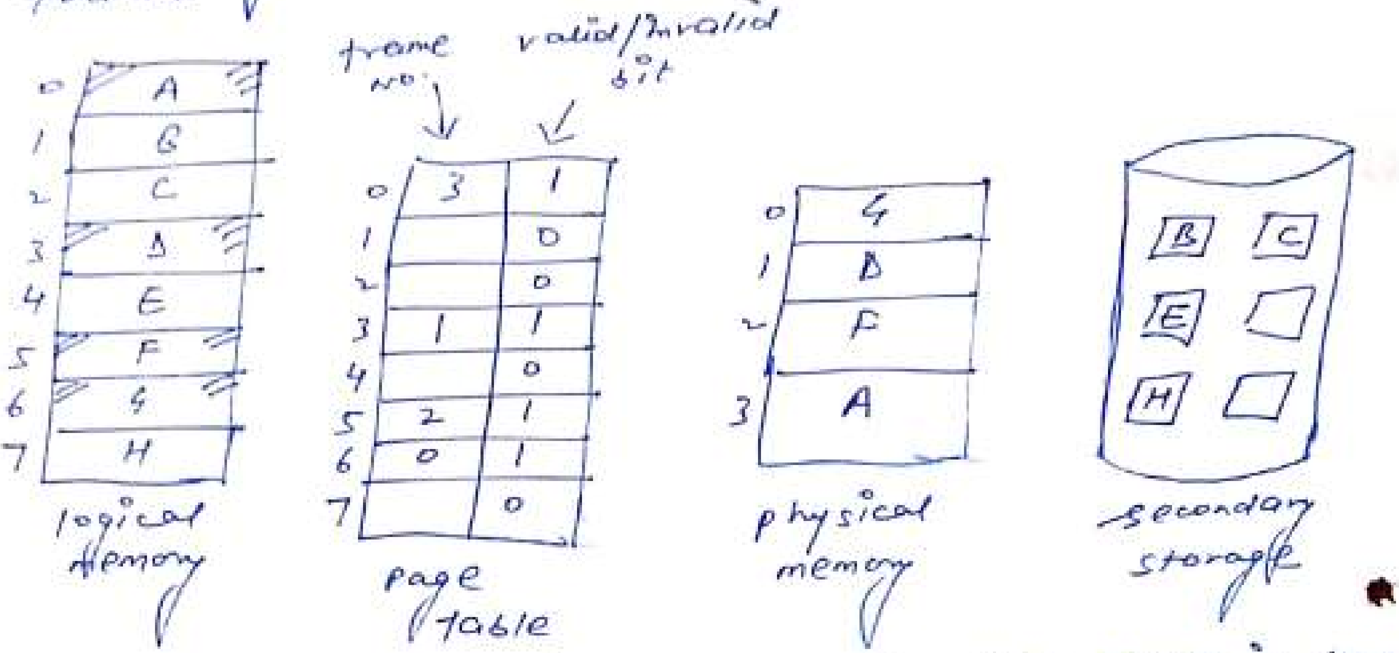
=> using vm, the whole process needs not to be in main memory completely. the pages/segments that are required can be in physical memory and others can reside on the backing store / secondary storage.

=> vm can be implemented using the concept of paging/segmentation with the concept of swapping.

Demand paging -> Demand paging a strategy to initially load pages only as they are needed/Demanded. with demand paged technique, pages that are demanded are loaded and pages that are never demanded are never loaded into physical memory.

=> In this, the logical address space of processes are divided into pages and physical memory is divided into frames. when the process comes to memory, it is loaded on secondary storage. when we want to execute the process, we swap it into memory. for this, we use a lazy paper, which swaps the required pages from Backing store to the physical memory.

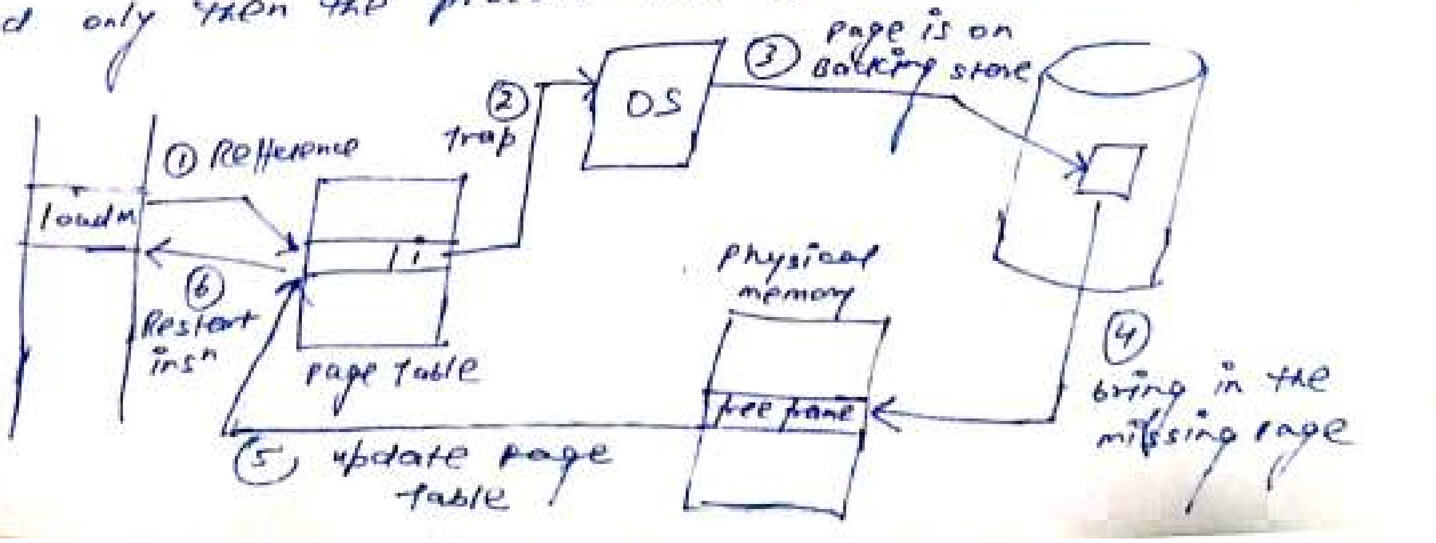
⇒ Suppose the logical address space has '8' pages and physical memory has '4' frames, then it can be indicated as follows.



⇒ The valid/invalid bit indicates whether the page is legal and in physical memory or not. If it is '0', then either it is a invalid page or it is currently on the disk.

⇒ At any time, the pages that are in the physical memory are called as memory Resident / Resident set.

Page-fault → If the cpu generates a address (logical address) which is not currently in the main/physical memory, this situation is called as page fault. In this case, there will be trap generated and the process will be suspended. Then, the OS reads this page from disk to the physical memory and only then the process will be resumed.



Performance of demand paging → let us find the effective access time of demand paging.

⇒ let 'P' be the probability of occurence of page fault ($0 \le P \le 1$). we would expect 'P' to be as much close to zero as possible.

so, $\boxed{\text{effective access time} = (1-P) \times T + P \times \text{page fault time}}$

where, $T \rightarrow$ memory access time.

⇒ Page fault time is the time required to service a page fault.

Page Replacement → If there occurs a page fault, then that requested page needs to be brought from secondary storage to main memory. If there isn't any free frame in main memory, then OS has to use some sort of technique to find the victim frame. this technique is page replacement.

⇒ if no frames are free, then there will be two page transfers i.e. one out and one in. we can reduce this overhead by using a modify bit. If a page has been modified, then its modified bit can be set to '1' and only then it will be swapped out otherwise just replaced by new page.

⇒ we will see following three page replacement algo.—

1) FIFO (first in first out) page replacement.
2) optimal page replacement.
3) LRU (least recently used) replacement.

⇒ we evaluate these algorithms by running it on a particular string of memory references and computing the page faults. the string of memory refferences is called as refference string.

⇒ To evaluate a page replacement algo', we should know the no. of frames available.

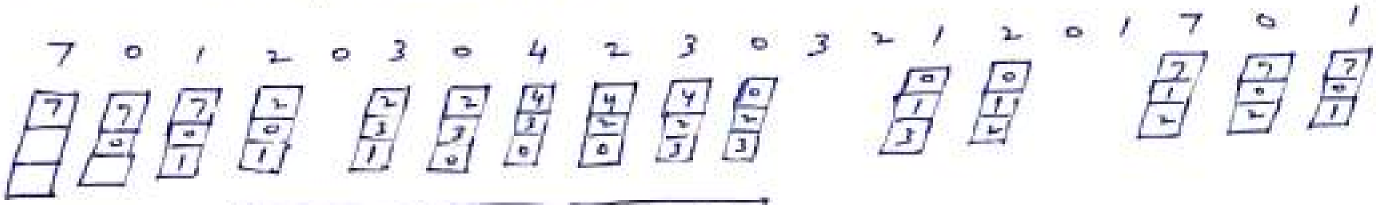
1) **FIFO Page Replacement** → In this, A time is associated with every page when it was brought into memory. when the page is replaced, the oldest page is choosen.

⇒ we can simply create a FIFO queue to hold the pages, where pages are replaced from head and inserted from ~~head~~ tail.

⇒ It is simplest Algo. but its performance is very low, when certain pages are heavily used.

e.g. ⇒ ① Refference string → 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

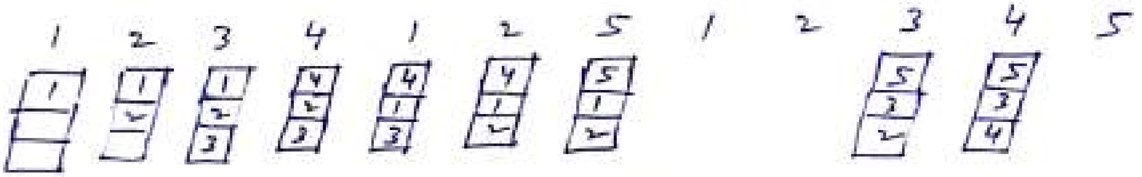
no. of frames → 3



so, $\boxed{\text{Total page faults} = 15}$

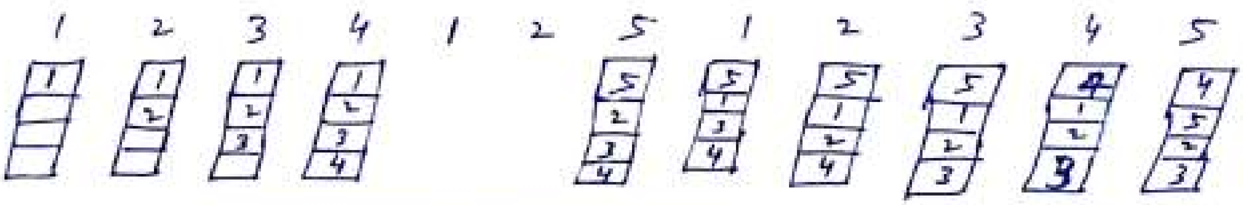
② Refference string → 1 2 3 4 1 2 5 1 2 3 4 5

ⓐ no. of frames = 3



$\boxed{\text{no. of page faults} = 9}$

ⓑ no. of frames = 4



$\boxed{\text{no. of page faults} = 10}$

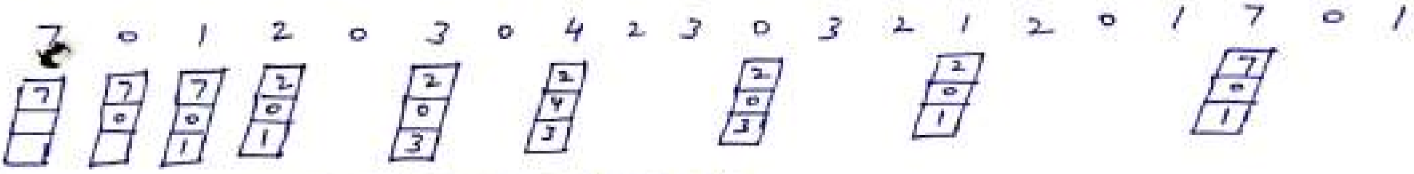
Belady's anomaly → In the above example, the no. of page faults for four frames is greater than the page faults for three frames. this unexpected result is Belady's anomaly

2) optimal page replacement → this algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly.

⇒ In this, the page will be replaced that will not be used for the longest period of time.

⇒ the optimal page-replacement algo. is difficult to implement bcoz it requires future knowledge of the refference string. As a result, it is mainly used for comparision studies.

e.g ⇒ Refference string → 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
frames → 3



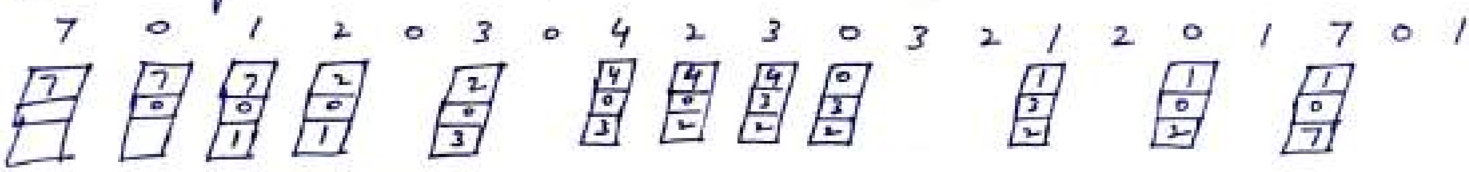
No. of Page faults = 9

---

3) LRU Page Replacement → the FIFO Algo. uses the time when a page was brought into memory whereas the optimal Algo. uses the time, when a page is to be used.

⇒ In this, we use recent past as an approximation of near future then we can replace the page that has not been used for the longest period of time.

e.g ⇒ using the above alg example —



No. of Page faults = 12

⇒ this Algo. is often used as a Page-replacement Algorithm and is considered good.

⇒ like optimal, LRU also don't suffer from Belady's anomaly.

Allocation of frames -> there are basically two things to be consi-dered while allocating frames to processes—

a) we cannot allocate the more frames than the available.

b) we must allocate a minimum no. of frames to a process.

=> As the no. of frames allocated to each process decreases, the page fault increases, slowing process execution.

=> But, if we increases the no. of frames allocated to each process then degree of multiprogramming decreases which decreases the CPU utilization.

Allocation Algorithms -> there can be two possible ways of allocating frames to processes—

1) Equal Allocation -> if there are 'n' frames and 'm' processes, then each process will get 'n/m' frames.

2) Proportional Allocation -> As diff. processes needs diff. amount of memory, there can be proportional allocation.

=> let the address space for process p_i be s_i and total logical address be 's'. then,

$$\boxed{S = \Sigma s_i}$$

if the total available frames are 'm', then no. of frame allocated to 'p_i' can be 'a_i' which is—

$$\boxed{a_i = (s_i / S) \times m}$$

=> we must adjust a_i as— $\boxed{min \leq a_i < m}$

e.g. =>

P_1 — 10 kb
P_2 — 127 kb } logical add. space

$s_1 = 10$ kb, $s_2 = 127$ kb

$S = s_1 + s_2 = 137$ kb

m = 62 frames

$a_1 = (10/137) \times 62 = 4$ frames

$a_2 = (127/137) \times 62 = 57$ frames

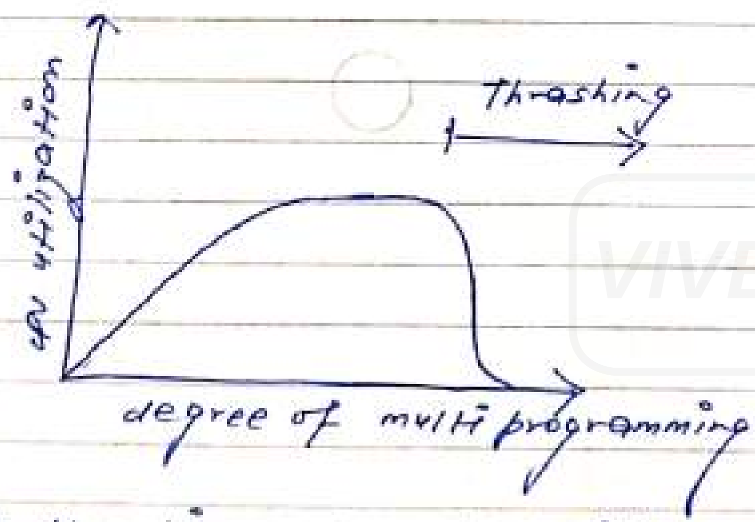
Thrashing -> If a process which is executing, do not have sufficient no. of frames, then obviously there will be more no. of page faults. So, for a new requested page, it must replace a page and again there can be new page fault for repla-ced page.

So, this process in which a Process P_i is spending more time in paging activity than execution is thrashing.

cause of thrashing -> OS monitors the CPU utilization. If CPU utilization is lower than a certain limit, then OS increases the degree of multiprogramming by introducing a new process.

As new processes are introduced, they also needs frames. so, OS takes the frames from other processes by using a global replacement algo. and give these frames to new pro-cesses. This increases the page-fault rate for those processes from whom ~~page~~ frames have been taken.

So, ~~As if~~ OS tries to increase the degree of multiprogra-mming to increase CPU utilization, but decreases the CPU utilization because frames are limited.



=> To prevent thrashing, we must provide a process with as many frames as it needs. An approach to prevent thrashing is working-set strategy, that looks at how many frames a process is actually using. This approach defines a locality model for process execution.

⇒ the locality model is based on the fact that as a process executes, it moves from locality to locality.

⇒ A locality a collection of pages, that work together actively for the execution of process.

1) <u>working set model</u> → this model is based on concept of locality. This model uses a parameter Δ, which is the most recent Δ page relferences.

⇒ the set of pages in the most recent Δ page relferences is the current working set for a process.

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

$\Delta = 10$ t_1 $\Delta = 10$ t_2

$WS(t_1) = \{1, 2, 5, 6, 7\}$ $WS(t_2) = \{1, 2, 3, 4\}$

⇒ If some page is in active state, it will be in the working set otherwise, it will be dropped from working set.

⇒ the accuracy of this model depends on the selection of Δ. if Δ is too small, it will not cover a entire locality but if it is too large, it may overlap several localities.

⇒ once Δ has been selected, the OS monitors the working set of each process and allocate enough frames to a process. If there are extra frames, then a new process can be initiated i.e. degree of multiprogramming can be increased.

2) <u>Page fault frequency</u> → working set approach is quiet effective for preventing thrashing but it has a cost involved in it, which is maintaining working set for processes time to time.

⇒ other approach is basically monitoring the page fault frequency for processes. we can define page fault upper bound and lower Bound for processes.

⇒ If the page fault rate for a process is more than the upper bound, then it is obvious that process needs more frames. so, allocate more frames to process.

⇒ If the page fault rate is less than lower bound, then there are extra frames and now, degree of multiprogramming can be increased.