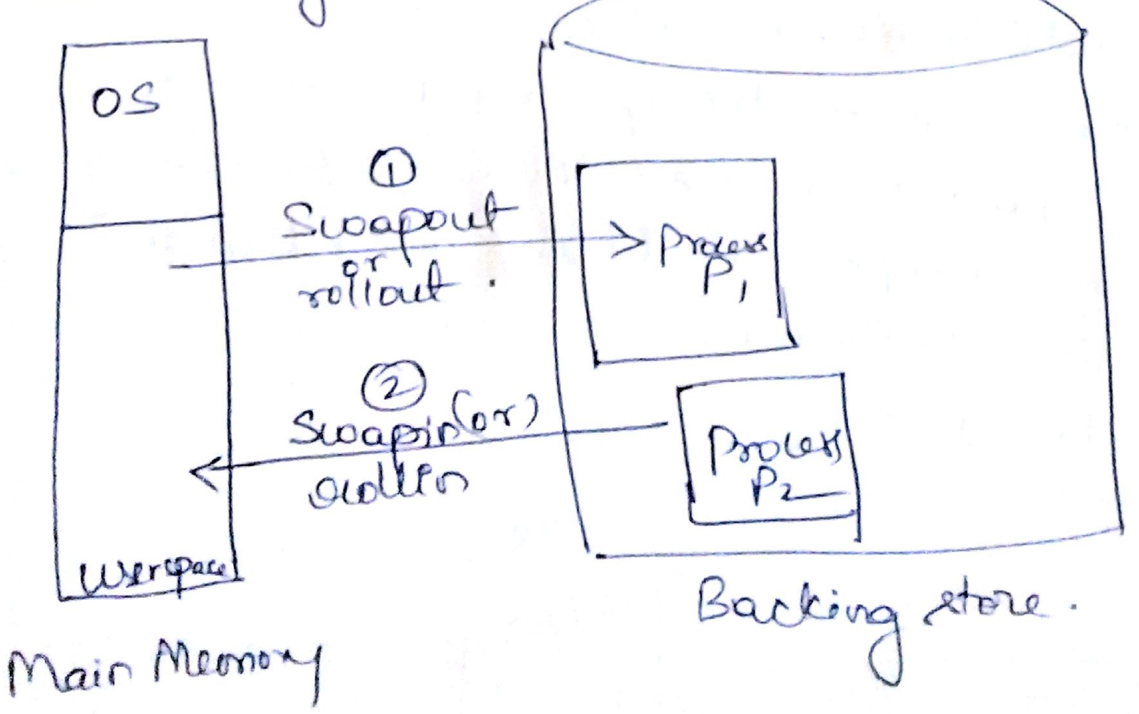
* UNIT-4 *

Q Explain about Swapping.

A process should be in Main memory to be executed. However a process can be swapped out of Main memory into backing store (secondary memory) when its time slice is completed in Roundrobin Algorithm and next process in ready Queue will be swapped into main memory. Similarly when using Priority algorithm if high priority process arrives when low priority process is executing, low priority process will be swapped out & high priority process will be swapped in. A process that is swapped out can again be swapped into main memory for continued execution from where it stopped earlier. After being swapped out & ~~later~~ again swapped process can again be swapped into same location or different location of Main Memory. If execution time address binding is done process can be loaded into different location of Main Memory else if compile time or load time address binding is done process ~~will~~ should be swapped in into previously loaded ~~ad~~ space in Main Memory.



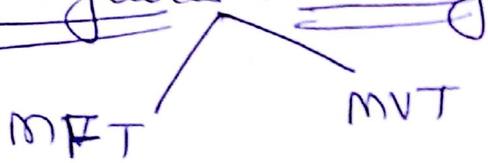
Entire main memory will not be swappedout, only one process will be swapped out of Main Memory to backing store. So to know size of processes is necessary when we want to swapout them.

only Idle processes should be swappedout.

If a process performing I/o is swapped out & another newprocess is ~~loaded~~ swapped in then I/o data of old process will now be written into new process area as it's in Mainmemory now. Solution to this is to store I/o data in OS buffers rather than into process ~~area~~ area.

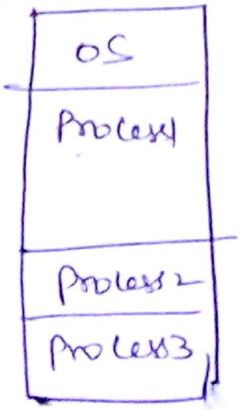
Generally swapping should be done less no. of times. If more swapping is done then context ~~time~~ switching time to load newprocess into Memory will be more than execution time of processes.

Q Contiguous Memory allocation.



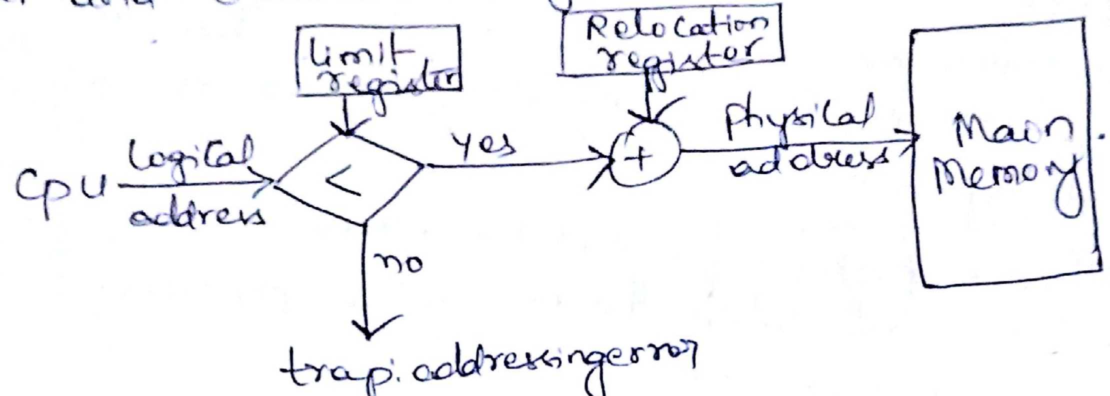
~~Memory Mapping & Memory protection~~

Generally Memory is divided into two partitions one for operating system, other for user processes. If OS is in ~~upper~~ lower part of Memory then higher memory will be used by userprocesses. Each process that ~~is~~ is waiting to come into Main Memory will be stored continuously in Main Memory



Memory mapping & protection.

one process should not write into other process allocated space in main memory & logical addresses generated by cpu should be carefully converted into physical addresses by cpu which can be done by using limit register and relocation register.



H/w for relocation & limit registers.

Limit register & relocation register will be loaded with new values for each process when that process is ~~by~~ selected for execution.

Limit register : It contains range of all logical addresses allowed for ~~a~~ particular process. (suppose prg length is 30 range of possible logical addresses = 1 to 30)

Relocatable register : It contains smallest or starting physical address for particular program.

using this we can ~~protect~~ stop a process from effecting os or other processes in main memory.

using Relocatable Register we can have OS with changing size. If OS is from 0–299 we can have user prg start at 300 by loading 300 into relocatable register. If one more module is added to os and now os is from 0 to 399 byte then we can start user Program at 400 by putting 400 in Relocatable Register.

Memory allocation using MFT (or) fixed partition scheme

Here we divide memory into many equal sized fixed partitions. Each partition can contain exactly one process. So how many partitions we have that many processes can be stored in main memory (this is our degree of multiprogramming).

When a process finishes its execution it's partition will be allocated to one of the process in Input Queue ~~location~~ of Memory.

Memory allocation using MVT (or) Variable partition Scheme.

Here all memory is treated as one large block or hole. The processes in ~~Ready~~ I/P queue of main memory can now take How much amount of memory they want. Now total memory will be divided by processes into holes or blocks of different sizes. After ~~these~~ a processes gets cpu & executes ~~Main Memory~~ it's hole will now be empty which can be allocated to next process waiting in I/P queue of main memory.

When many holes are available then a process from backing store can come into one of the holes in main memory in the following methods.

First fit: Allocate the very first hole that is sufficient for the process to load. If process size is 50mB we have holes of 70mB and 60mB then 70mB hole is alloted as it's first hole.

Best fit: Allocate the hole ~~that~~ in which process fits but wastes memory very less. i.e allocate smallest hole that is big enough. If process size is 50mB and we have holes of 70mB & 60mB then 60mB hole should be allocated ∴ it wastes only 10mB space.

- Worst fit: Allocate largest hole to process out of all holes so memory gets wasted a lot.

Q ~~Ex~~ What is Fragmentation?

In multiprogramming systems we should avoid memory waste or ~~fr~~fragmentation. ~~Single~~ p. systems with fixed sized allocation units like paging suffers from internal fragmentation. Systems with variable sized allocation units such as multiple partition scheme, segmentation suffer from external fragmentation.

External fragmentation: when there is enough total memory space to ~~so~~ allocated to a process but the available space is not continuous so process cannot be stored in memory. This large no. of small holes in memory cannot satify any process request as they are not continuous.

Scanned by CamScanner

Internal fragmentation: When total memory is divided into fixed size partitions or pages. If a process is stored in fixed size partition then size of process ~~could~~ may be little less than partition or page size. This wastage of memory is called internal fragmentation.

Q Explain about paging.

In paging physical memory is divided into equal and fixed sized ~~f~~ blocks called as frames (and Logical memory is divided into blocks of same size called Pages).

When a process is to be executed its pages should be loaded into free frames available for execution.

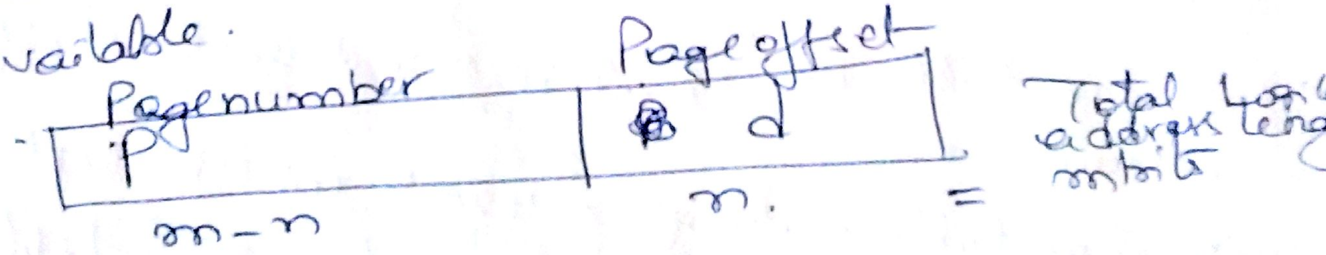
Paging makes physical address space of a process non-contiguous i.e where ever free frame is available our process will be stored there.

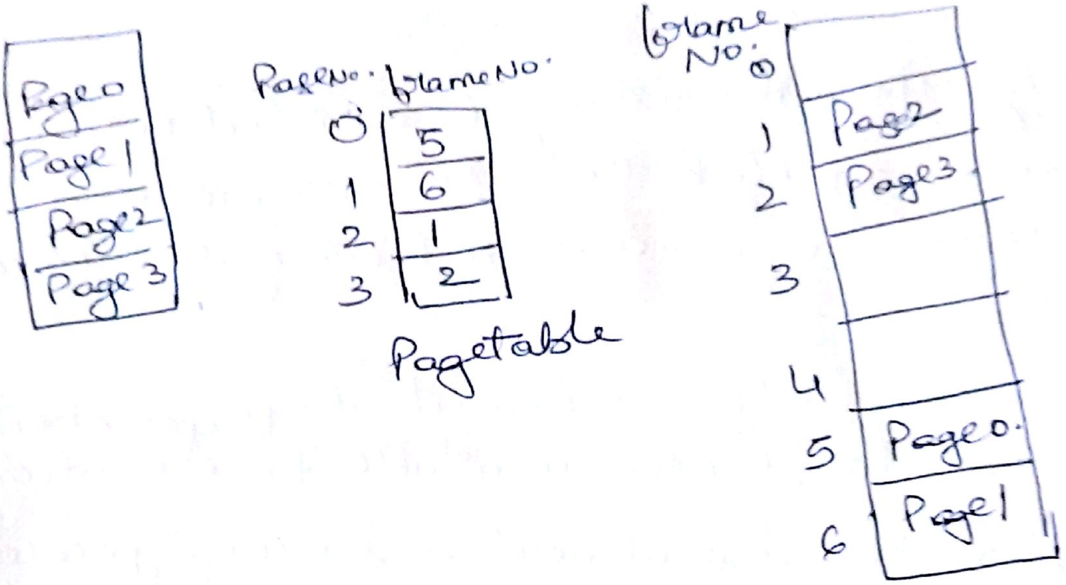
Paging avoids external fragmentation.

Paging suffers from internal fragmentation.

If we have logical address of m bits out of that if page size is 2^n then n bits are used as Page offset and (m-n) bits are used to identify one page out of 2^{m-n} pages.

Suppose logical address is 4 bits ($m=$) we have Page size 4 bytes then 2 bits $= n$ are used to identify a ~~cooo~~ byte in a page. and remaining $m-n = 4-2$ bits $= 2$ bits are used to identify one page out of four pages available.





~~In~~ Page0 is inframe 5, Page1 is in frame 6 and Page 2 in frame1, Page 3 is inframe 2.

Which ~~frame~~ Page is in which frame in physical memory is stored in Pagetable.

Paging seprate user's view of memory from actual physical memory. user imagines that his program is stored continusly ~~but~~ at oneplace, but paging divides program into several pages and stores them in different noncontinuous frame

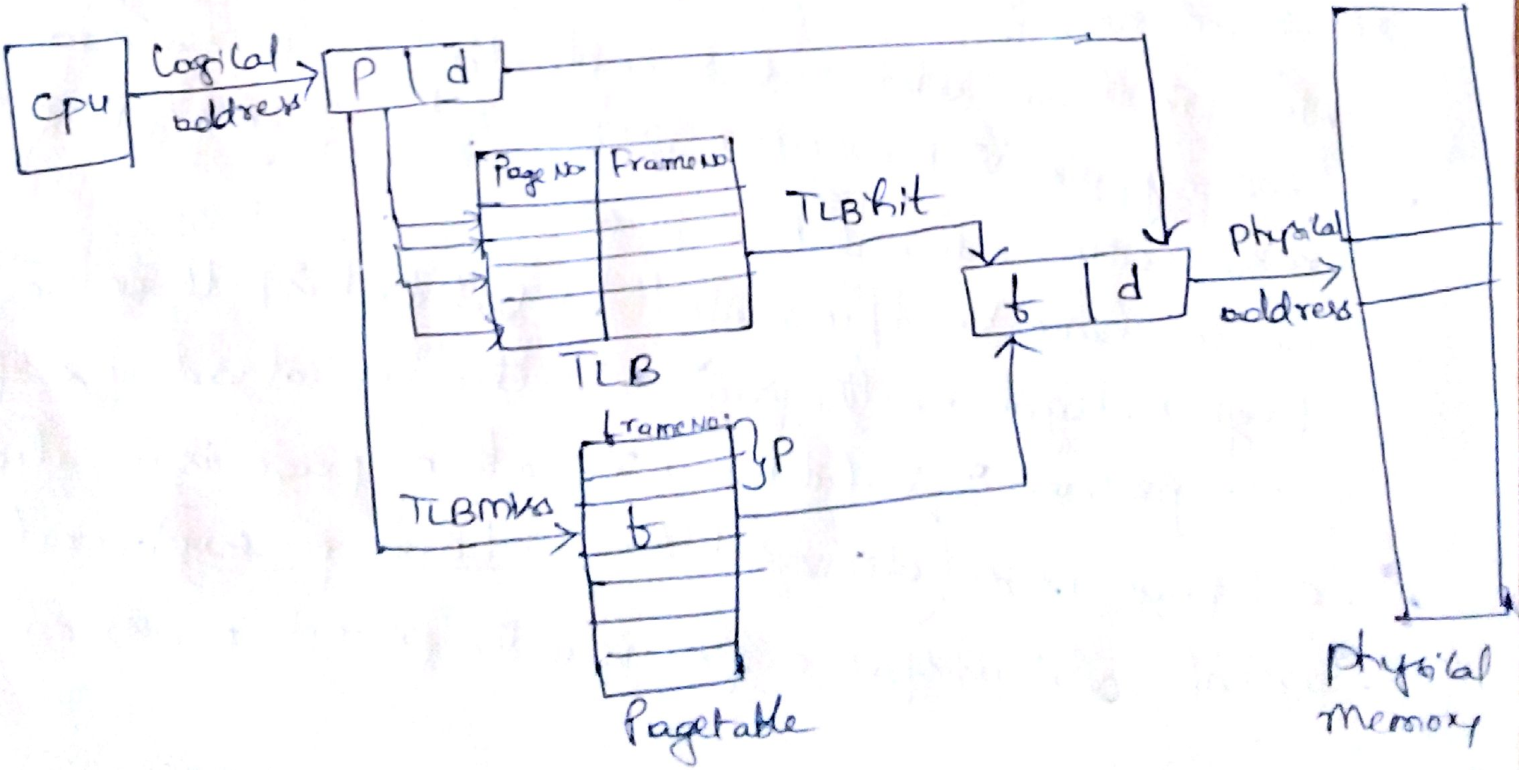
Hardware support for paging

1] Pagetable is small in size it can be stored in registers.

2] Pagetable is large in size it will be kept in main memory and a ~~p~~ address of Page table in main memory is stored in page table base register (PTBR).

But if we store pagetable in main memory it will take more time to findout frame no. for a particular page no.. So ~~a~~ solution to this is to have Transaction Look aside buffer (TLB) in highspeed associative memory which is very fast memory. TLB doesnot contain all entries of pagetable but rather ~~it~~ only few pagenos and frame nos. So it's size is less when compare to pagetable. So for a particular logical address we search for matching Page no. ~~is~~ first in TLB, if match is not found in ~~pagetable~~ TLB then we search for that pagenumber in page table & then get corresponding frame number. In that frame we will goto ~~of~~ Pageoffset d position. If TLB hit is done we get frame no. quickly else if we search for page no. in page table it takes more time.

Address Translation from logical to physical address in paging using TLB and page table



Just Read no need to byheart.

If TLB hit ratio = 80%.
TLB miss ratio = 20%.

If time needed to access TLB = 20ns
time needed to access memory or Pagetable in Mem = 100ns.

If ~~Value~~ Page No. is found in TLB, (TLB hit) then time to access that page = Time to access TLB for frame No. + Time to access Main memory for that frame
= 20ns + 100 ns = 120ns.

If page No. is not found in TLB, it's in Page table in Memory then time to access page
= Time to access TLB + Time of access Page table in Memory + Time to access Page in Memory
= 20 ns + 100ns + 100ns = 220ns

∴ Total effective access time = 0.80 x 120 + 0.20 x 220
= 140 nanosecds.

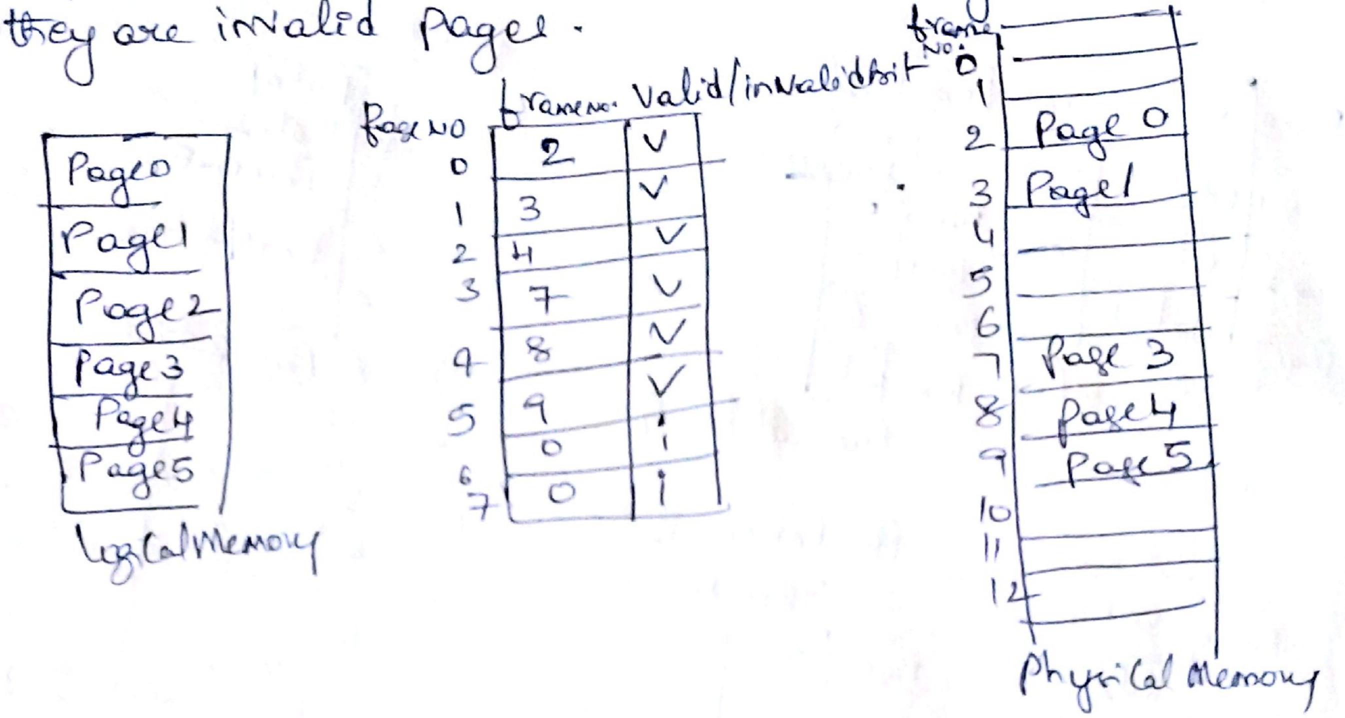
Protection in paging

-> Each and every frame has protection bits associated with it.
We can have a bit which indicates whether a page is

-> Read only page or Read write page. If it's Read only page no writes are allowed to a page.

-> We can have valid/invalid bit, if valid bit is 1 then the Page is legal i.e the page is in the Logical address space of our process. If Valid bit is Zero then page is invalid i.e page ~~is in~~ (~~doesnot belong to our process~~) is not in logical address space of process or they r not in main memory

In below program page 0,1,2,3,4,5 belong to our process & that process is in physical memory so access to them is valid, ~~p~~ but page 6 & 7 are not in main memory. ~~doesnot belong to our process~~ so they are invalid pages.



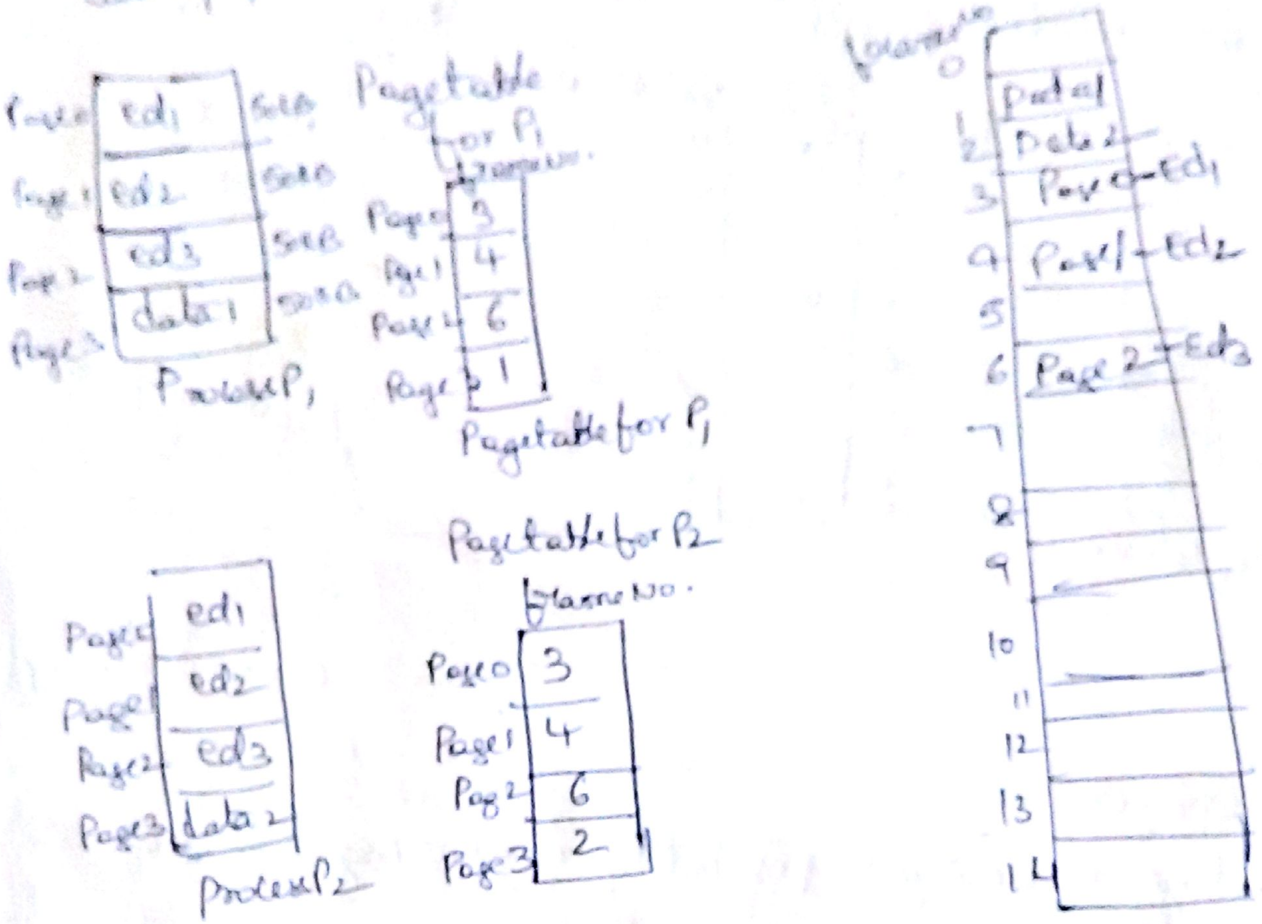
Page no	frame no.	Valid/invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

frame No.	
0	
1	
2	Page 0
3	Page1
4	
5	
6	
7	Page 3
8	Page4
9	Page 5
10	
11	
12	

Shared pages:

Advantage of paging is possibility of sharing common code. This is useful in Time shareing environment where different users use same s/w like text editor (to type & edit their Programs) with different data. Suppose if Text editor occupiee 150KB of space for code and 50KB of dataspace for 1 user for 40 users ~~we~~ we need 8000KB Memory. To avoid memory waitage code pages can be shared & data pages will be separate for each process.

Ed1, Ed2, Ed3 are Text editor code pages each of 50KB size and they can be shared pages of Process1, Process P2, Process P3 and data1, data2, data3 are seperate data pages for P1, P2, P3.



Q Structure of Page Table.

The following are Techniques for structuring of page table:

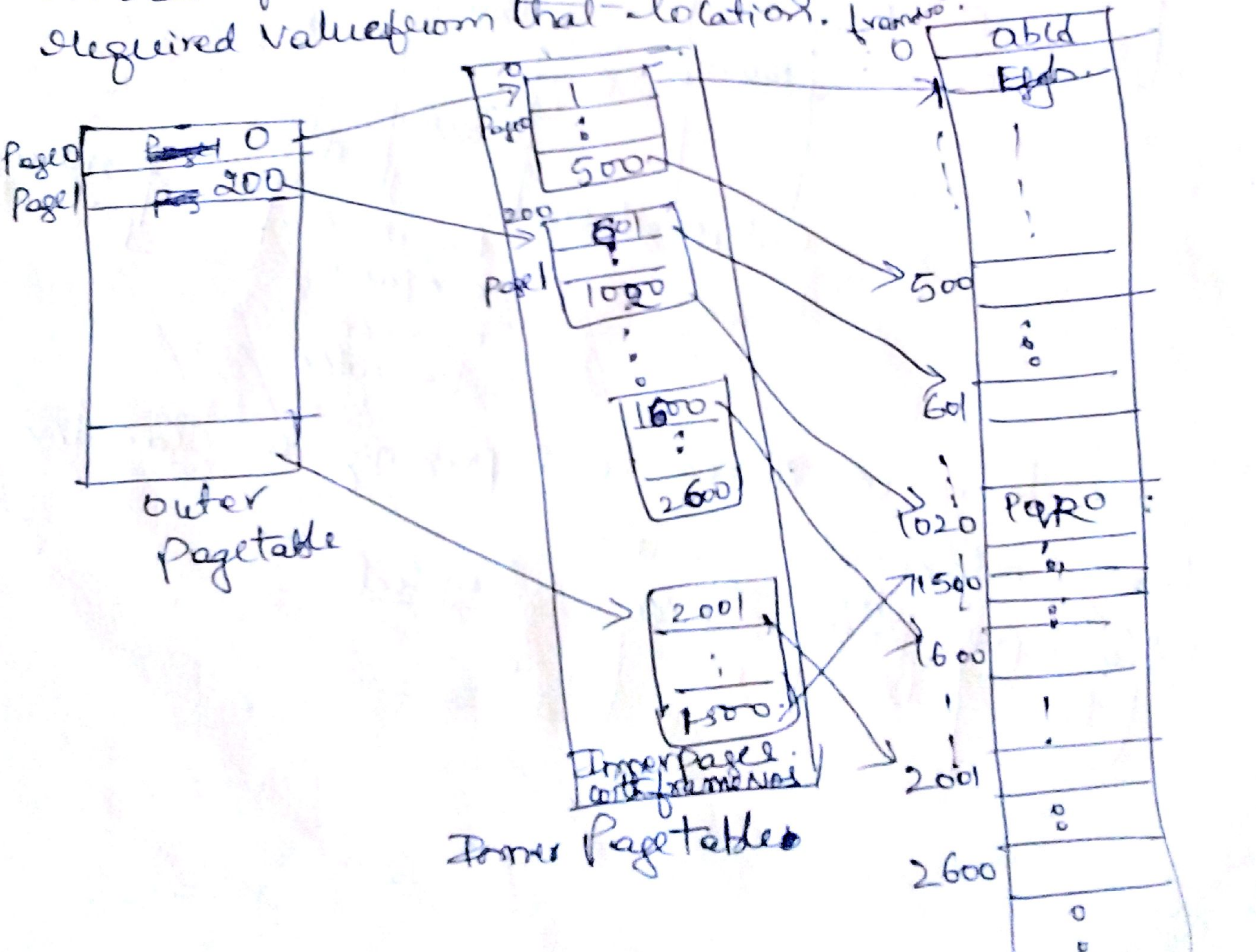
a) Hierarchical paging
b) Hashed Page Table.
c) Inverted page table.

a) Hierarchical paging.

If size of page.table is large then we cannot store page table completely continuosly at particular place in main memory. so solution to this is to divide pagetable into smaller pieces.

one way to do this is to use two level paging algorithm

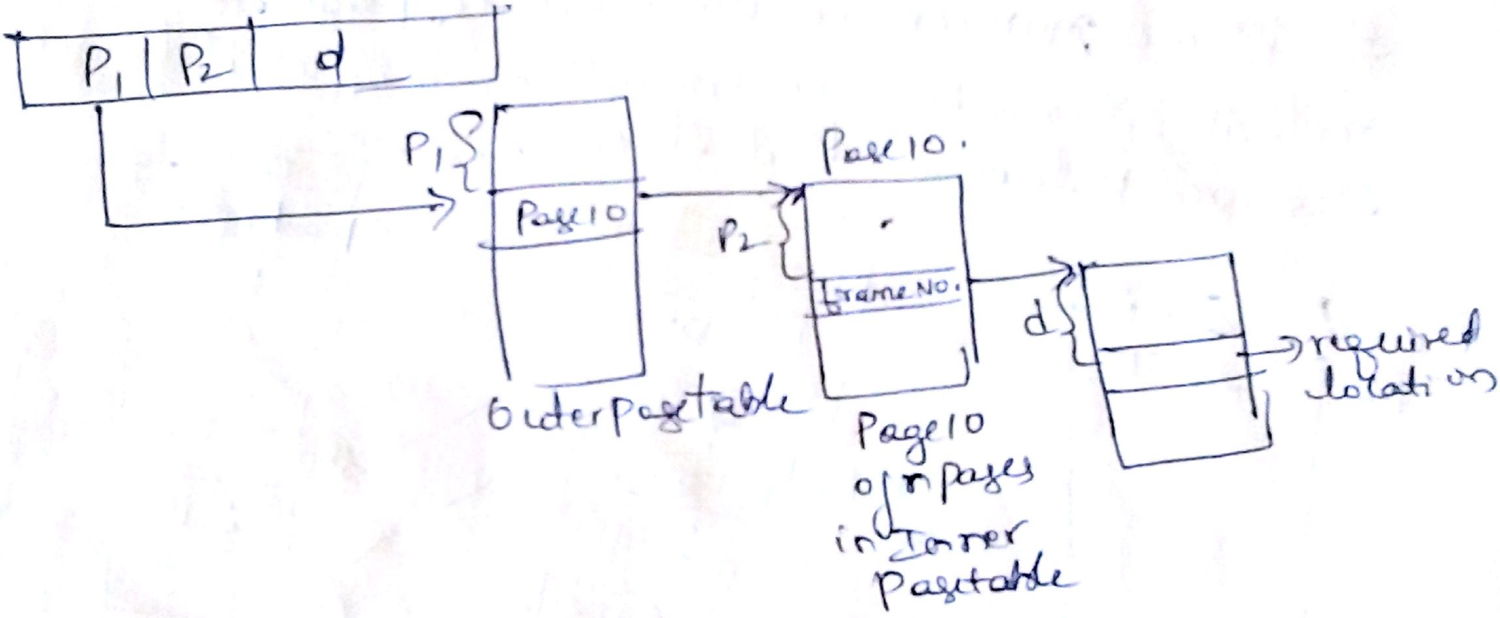
~~In~~ Here we have outer page table which gives address of one inner page ~~table~~ out of n inner pages ~~tables~~ at second level Pagetable. In this inner page ~~table~~ we get ~~address~~ ~~of~~ frame number for our required page. ~~to which~~ on that frame Number we will go to offset d to get required value from that location.



Suppose we have 32 bit address if Pagesize is 4KB $= 2^{12}$ then 12 bits will go for offset, out of remaining 20 bits → 10 bits for outer page table to identify one row in it, 10 bits to find out an entry in page identified by outer Page table.

Page Number		Page offset
p_1	p_2	12
10	10	12

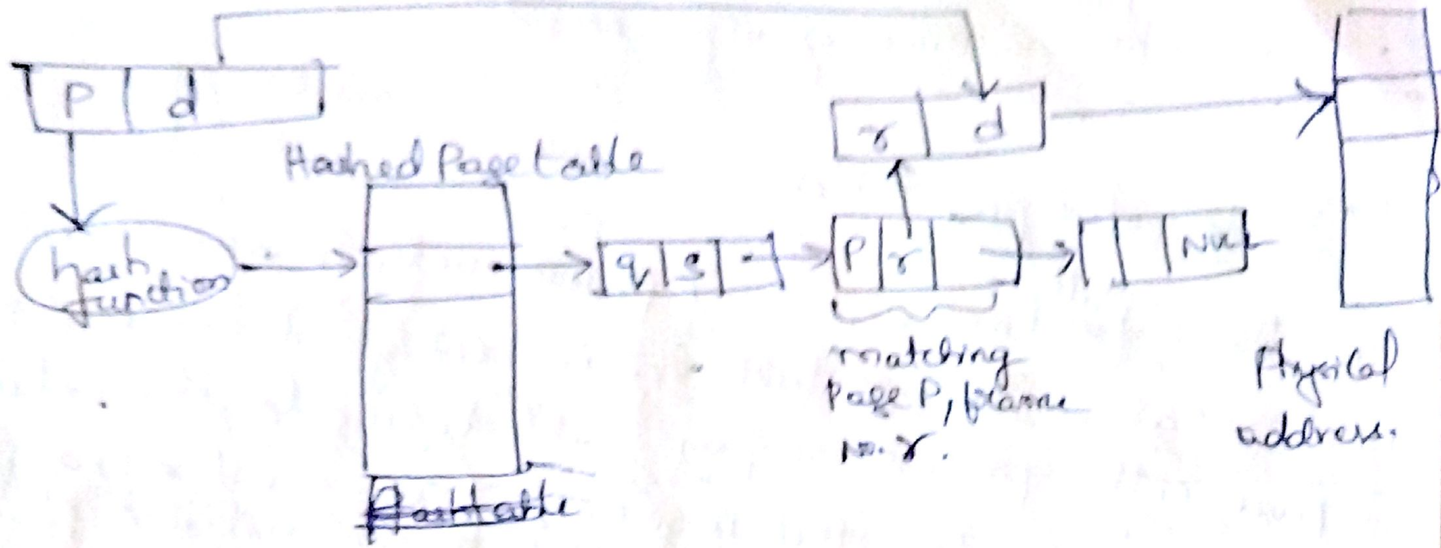
Total address 32 bits = 10 10 12



Similarly in three level paging we can divide logical address into 3 parts.

2nd outer page	outer page	Inner page	offset
P_1	P_2	P_3	d

(b) Hashed pagetable



Here we use hashfunction on page number p which produces a virtual pageNumber. We search for this virtual page number in hash table where matching is found we find a linked list of ~~items~~ elements. Here each node of linked list contains following fields

a) virtual page no.
b) Frame no. for mapped pageno.
c) pointer to next node.

Different pagenos map to same location of hashtable, where we will find linked list of pagenos & corresponding framenos. (Suppose all page nos which produce remainder 6 when hash function is applied will goto 6th location in hashtable).

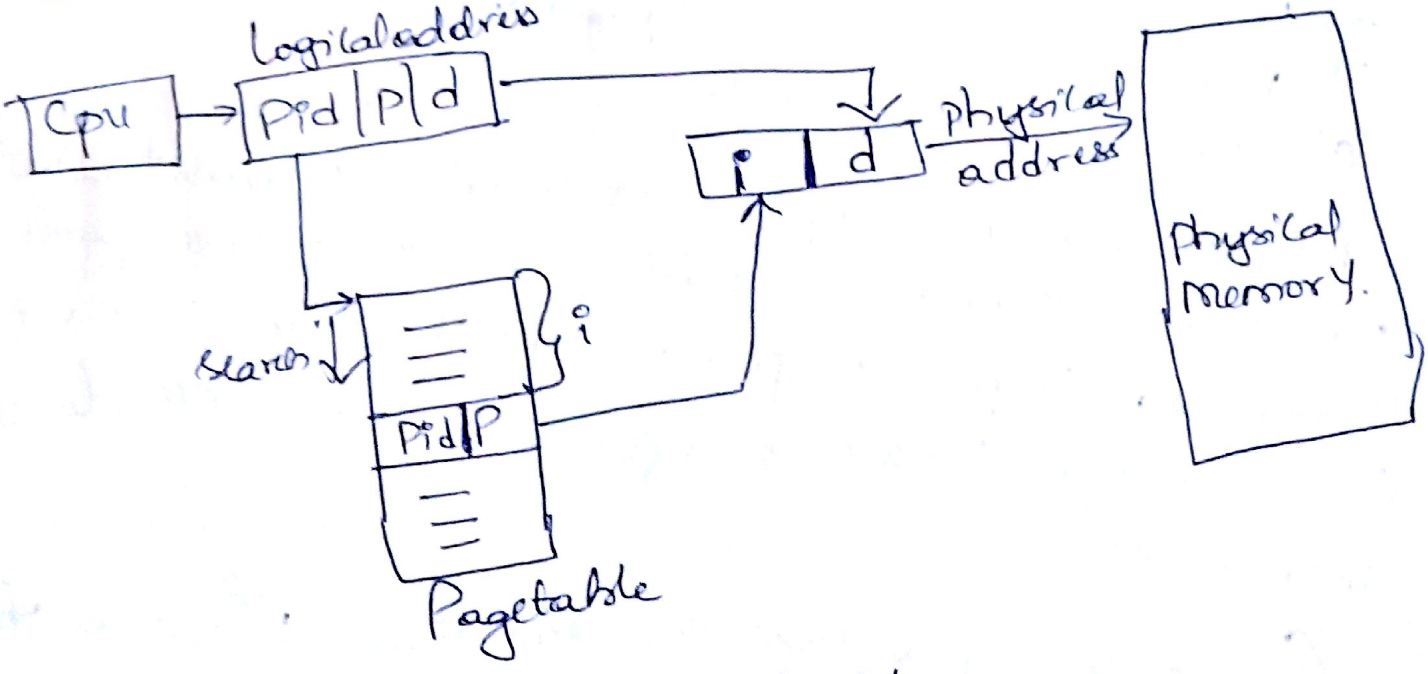
Generally if pagetable has more than 2^{32} entries hashing is used.

clustered pagetables are similar to hashed pagetables,
~~I~~ In clustered pagetables each matching entry of page

table has so many pages & corresponding frame nos.
Clustered pagetables are used for sparse address spaces i.e noncontinuous memory references.

(C) Inverted page tables :-

~~Here for~~ Generally each and every process we have associated pagetable. The page table has one entry init for each & every page inthe process. So page table willbe lengthy. Solution to thisis to use Inverted page tables (Even if all pages of process are not in frames of main memory entry for them is created in page table)
In inverted page table only those pages which are loaded into frames of main memory will be given an entry in page table
Each entry of page table contains virtual address i.e pagenumber and process id of process that owns that page. So we don't have one page table for each process, for all processes we have one page table.

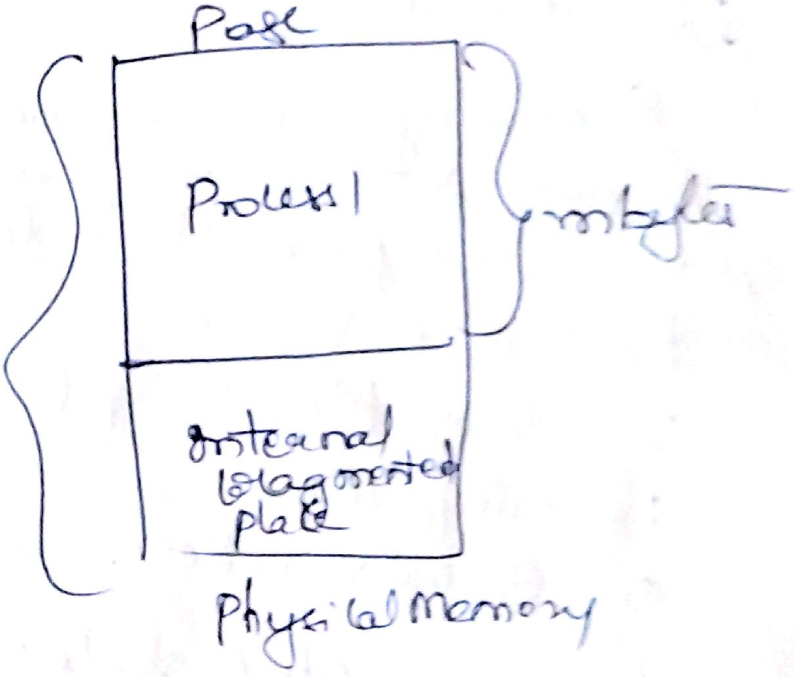


Inverted page table.

we search pagetable for matching pid and p if match is found in pagetable at entry i then physical address is Calculated using framenoi and offset d.

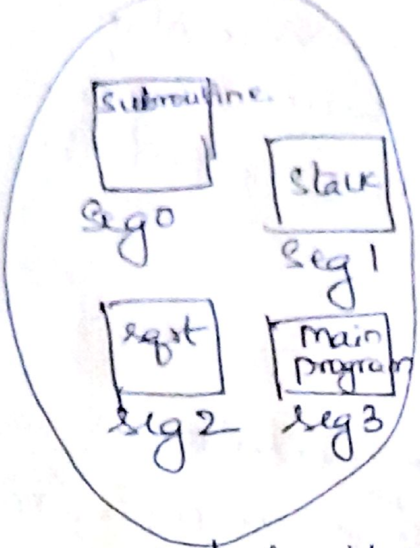
Inverted pagetable reduces amount of memory needed for pagetables because only one Pagetable is maintained for all processes, but it increases searchtime

Paging suffers from internal fragmentation. if page size is mbytes and process size is mbytes (m<n) then (m-n) bytes of physical Memory gets wasted.

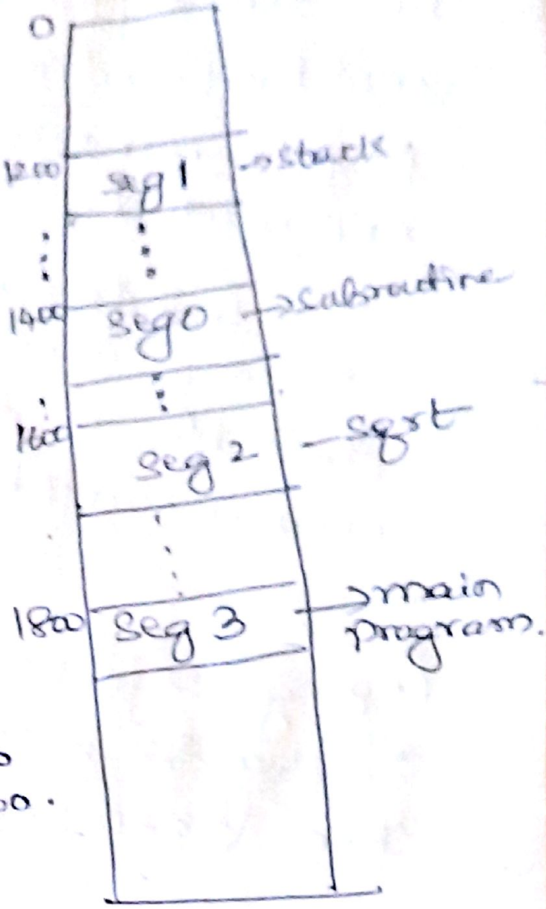


Paging doesnot have external fragmentation.

Q Segmentation:



segno	limit	base
0	100	1400
1	50	1200
2	200	1600
3	100	1800



Seg 0 starts at address 1400 & has length 100
seg 1 starts at address 1200 and has length 50
seg 2 starts at add 1600 and has length 200.

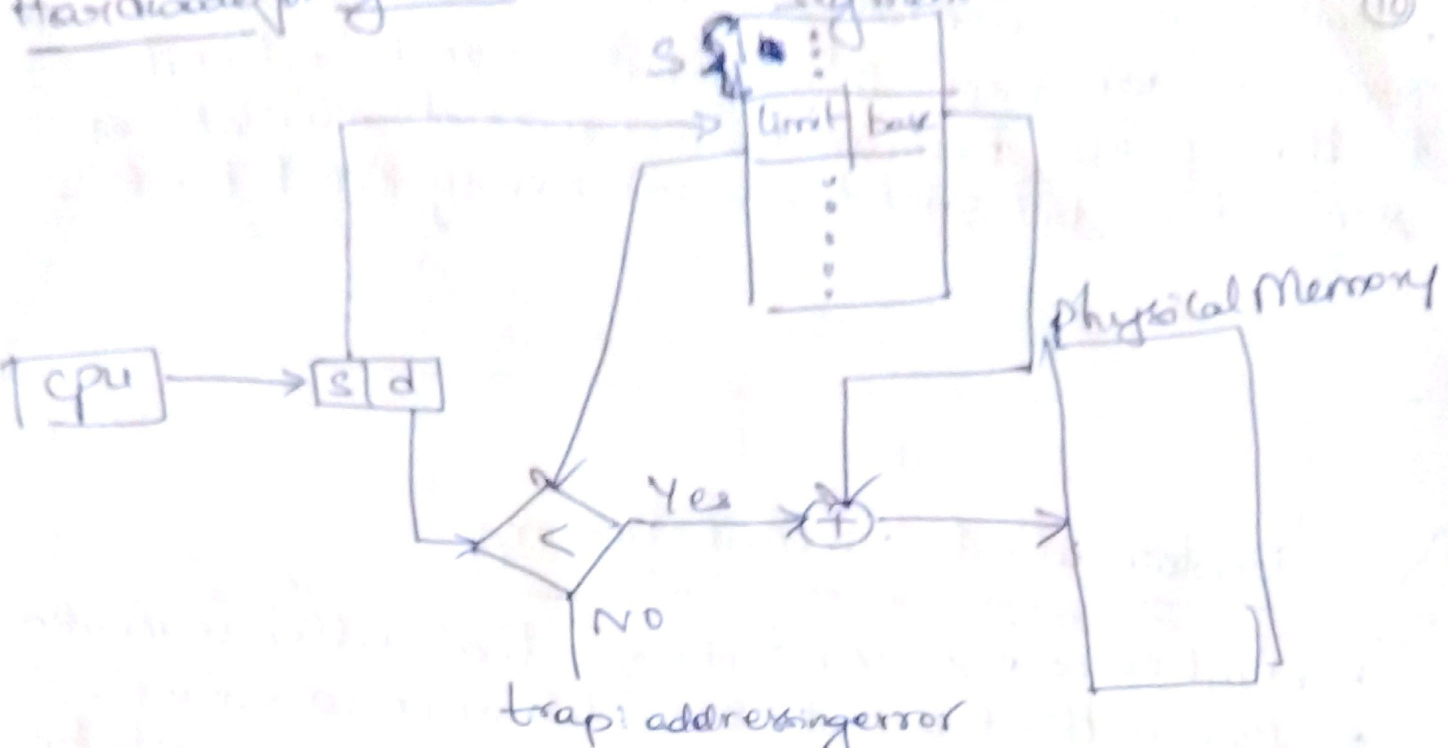
Generally when we write any program, the Program contains ~~main~~ many segments as follows.

1. main function() (code)
2. Stacks
3. Library files (Ex Sqrt(), Power())
4. Subfunctions.
5. Global variables.

for each of above parts of program a separate area of memory is allocated as needed and they are called as segments.

Segmentation is memory management scheme where user's view of memory is same as how segments are stored in memory
In Paging user's view is different from how prg is stored (in form of Pages in memory).

Hardware for Segmentation



Address generated by cpu contains seg number and segment offset d $\boxed{s \mid d}$.

Segment number is used as ~~offset~~ index into segment table where we will find segment starting address base and length of segment limit. If Generated offset is less than segment limit then to ~~get~~ segment starting address base, offset d is added to get Physical address. If ~~segment limit~~ offset d is not less than segment limit then ~~its~~ we are trying to access ~~ou~~ memory out of our segment so trap to OS or addressing error occurs.

Segment table contains starting address of segments base and length of each segment limit.

Segmentation does not have internal flagmentation.

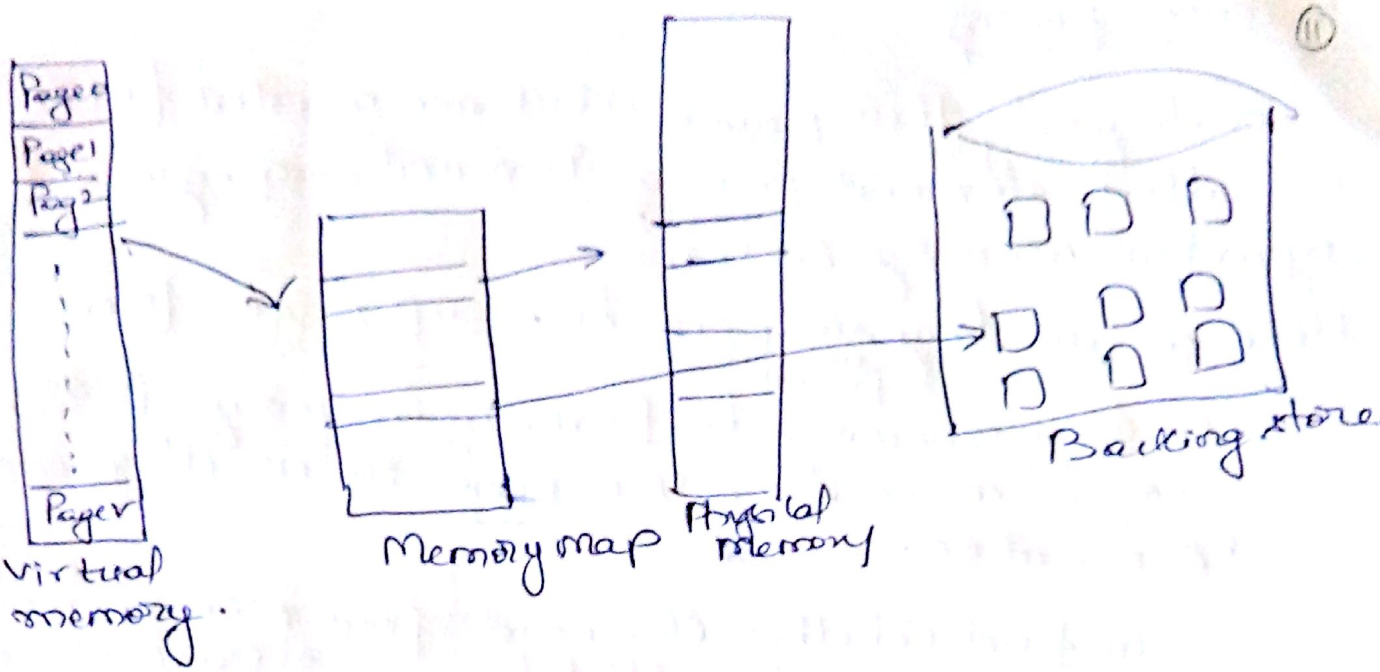
Segmentation has external flagmentation problem because ~~total~~ physical memory is divided into No. of small segments and when a new process needs memory we have free Memory but can't allocate to that process because Memory is not available as single block but partitioned into many small blocks.

Q Explain about Virtual Memory.

Virtual memory is a technique that allows execution of process that is not completly in memory. Advantage of virtual memory is program can be larger than Memory.

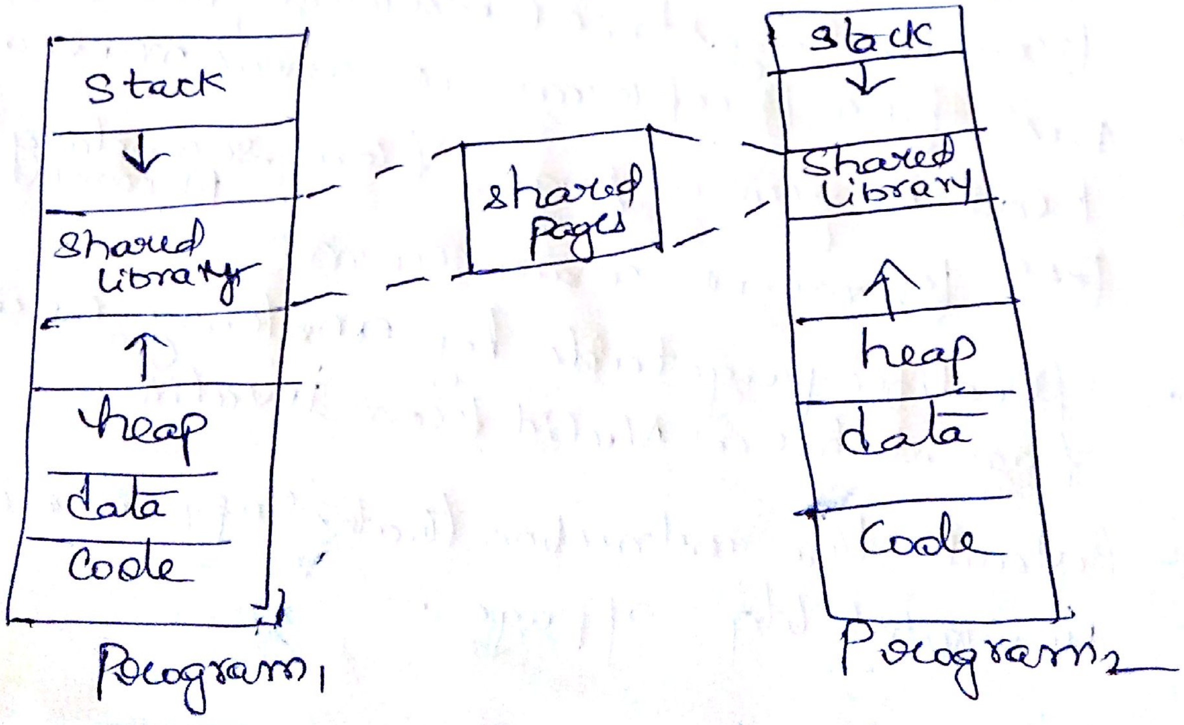
In general entire program nead not be in Main memory for execution. Some parts of program described below are not at all needed by program for its execution.

- Code to handle unusual error conditions is not used all times.
- Generally if we need 10 elements in an array we declare array with 100 elements which waste memory.
- Depending upon users option some parts of a program may not be used at all to load those parts in memory is waste.



Virtual Memory advantages are

- ∵ only few pages of program or part of program are loaded into main memory many programs can be stored in main memory.
- ~~To~~ ~~With~~ Less I/O will be needed to load part of program into main memory & swap it out to secondary memory.
- Library files can be shared by different programs.
- If two programs have same coding part then same memory can be shared by ~~[illegible]~~ them.



Demand Paging :

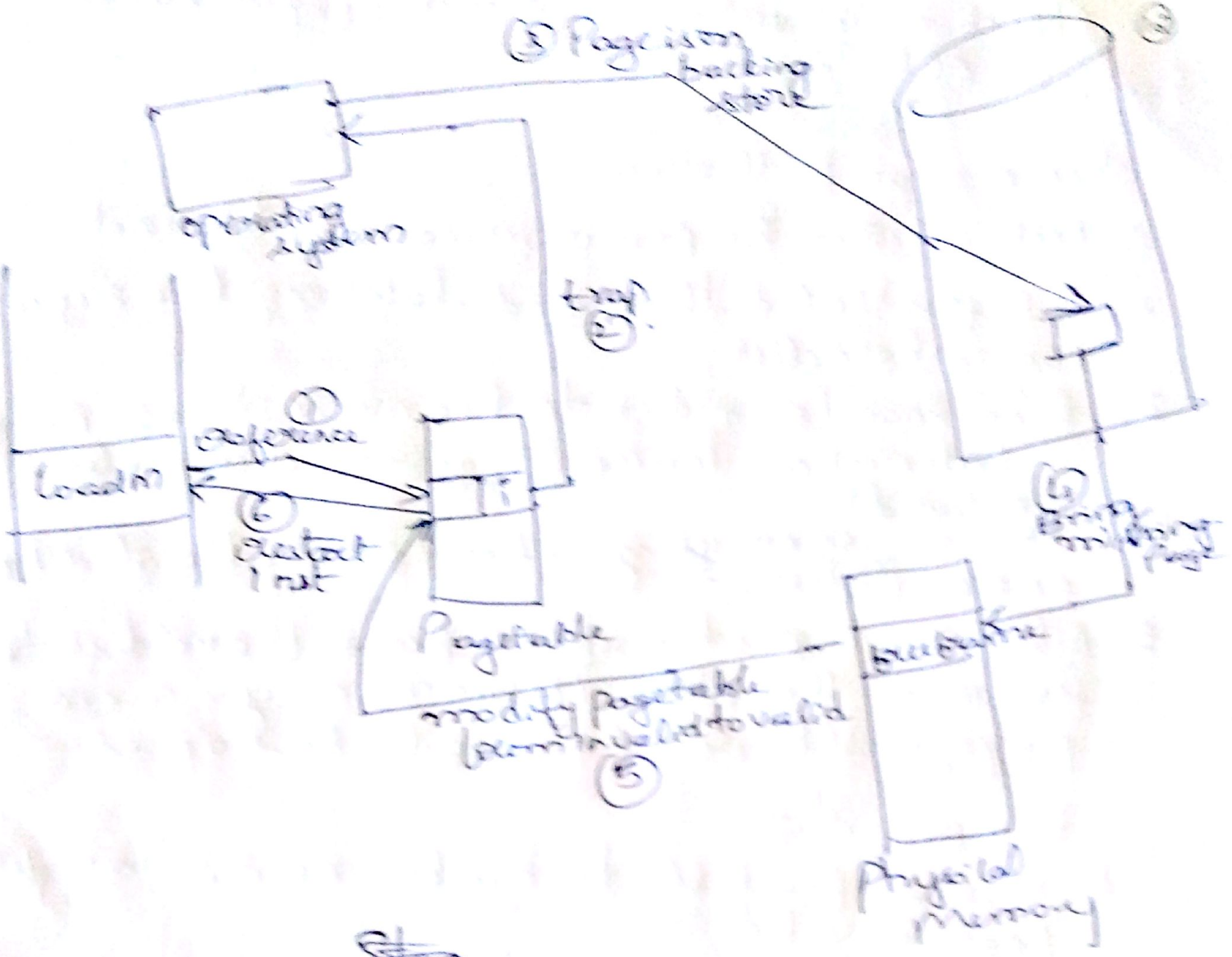
Loading only those pages which are needed for execution is Called demand paging. Demand paging is used by virtual memory systems.

Here we use lazy swapper or Pager to swap pages from secondary memory to Primary memory. A lazy swapper never swaps in a page unless it's needed for execution.

To find whether required page is in main memory or not we use Valid/invalid bit. If page is in memory Valid bit is set. If ~~invalid~~ Valid bit is 0 then page is invalid (i.e page doesnot belong to our program or Page belongs to our process but is not loaded in to memory).

If page is not in main memory page fault ~~occ~~ occurs. The following are steps to handle pagefaul

1. we check pagetable to see if page ref is valid/ invalid
2. If page reference is invalid trap to os is raised. if Page reference is valid then we should bring that Page into main memory from backing store (~~to~~ secondary memory)
3. First find freeframe in main memory
4. Read required page from secondary memory to ~~Pre~~ frame in main mem.
5. Modify Pagetable by making new Page status as Valid from invalid.
6. Restart the instruction that stopped due to unavailability of page.



~~Step~~

Steps in handling Pagefault.

Performance of demand Paging

Let's Calculate effective accesstime for demand paged memory.

Let P be probability of page fault. $(0 \le P \le 1)$

effective access time = $(1-P) \times ma + P \times$ pagefault time

ma = memoryaccesstime (when required page is in primary memory)

Pagefaulttime = Time to bring page from secondary to primary memory & excute (i.e Page is not in primary memory).

Effective accesstime is directly proportional to Pagefault rate P.

When a page fault occurs

1) Check whether the page reference is Valid or not.

2) If Page is Valid then find locationg the required Page in harddisk.

3) Read Page from disk to frame in physical Memory (Latencytime to access Page from disk is also included)

4) Now ∴ we are using disk cpu is allocated to another new process.

5) After our, Page is transfered from Hard disk to frames in ~~Ma~~ physical / Main memory new process state will be saved & it will be stopped temporarily

6) Change Pagetables to mark newly bought Page as Valid.

7) ~~Cont~~ Get cpu & start our stopped process again

=

Q Page Replacement & Page Replacement Algorithms.

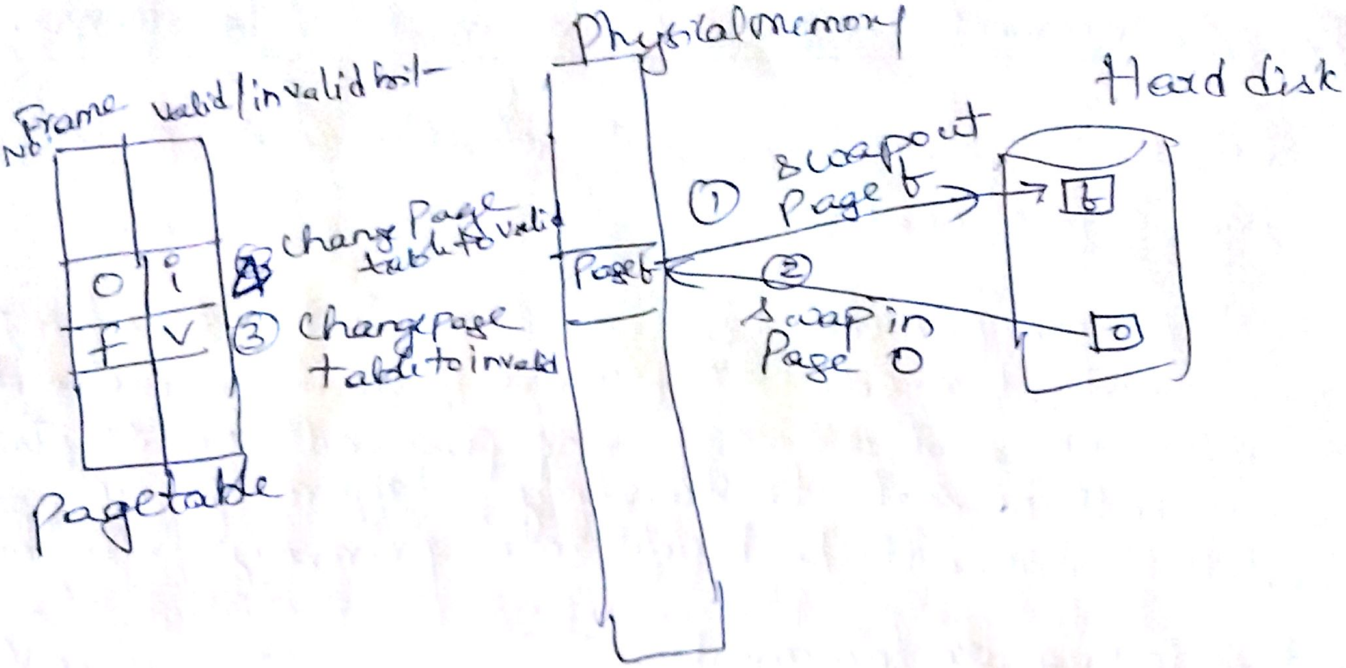
What is page replacement? What are steps in Page replacement?

Suppose a process is allocated 3 frames for its execution and it needs 4th frame/page then out ~~out~~ of 3 frames in physical memory one page will be replaced i.e it will be sent to secondary memory and needed page will be bought into primary memory

Steps in page replacement :-

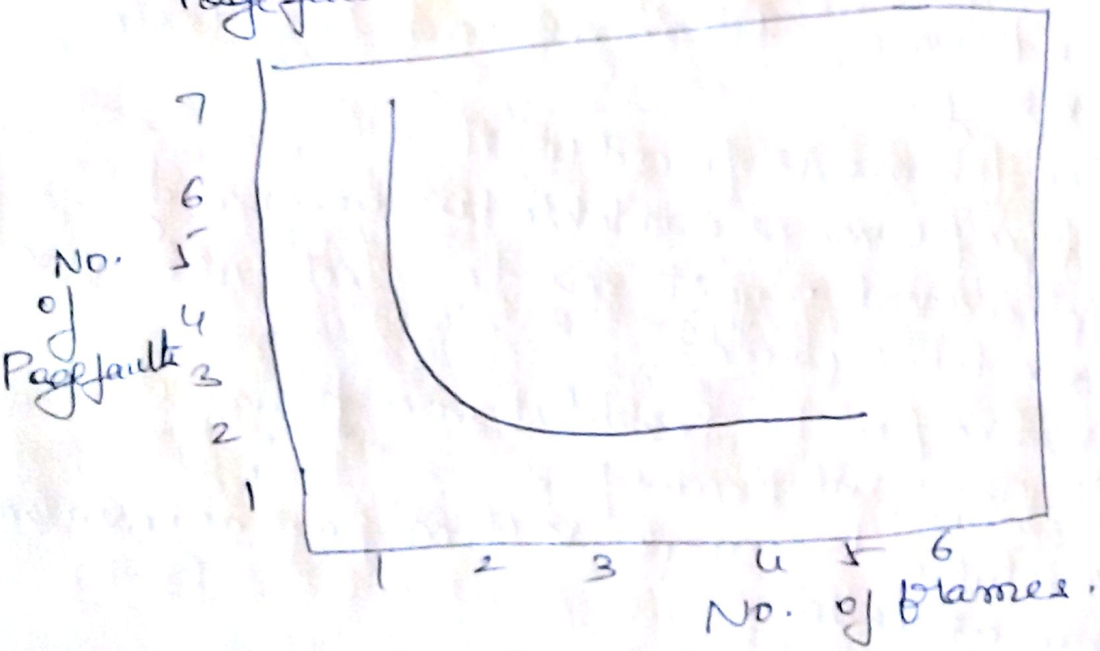
When we need a new page for our process execution and that page is not in ~~Main~~ physical memory then the following steps occur.

1. find the location of desired page on Hard disk.
2. Find free frame to bring desired Page into physical memory.
   a) If free frame is found use it.
   b) If no free frame is available use any one of Page replacement algorithms to select victim page (i.e page to be replaced)
   c) send victim Page to harddisk and bring desired Page to physical Memory
3. change Pagetables to mark victim page as invalid desired page as valid.
4. Restart halted process



Page replacement

Generally as number of frames increase, No. of Pagefaults decrease.



Different Page replacement Algorithms are

a) FIFO Pagereplacement Algorithm
b) optimal Page " "
c) Least Recently used (LRU) Pagereplacement Algorithm.
d) LRU approximation PageReplacement Alg
e) Counting based "
f) Page Buffering Algorithms

a) FIFO Page replacement Algorithm:

Here each page bought into frames of mainmemory arrivaltime is noted. Now when all frames are over, if we need frame to load Newpage the frame ~~with~~ Containing page with early arrival Page is removed to place new page there ∴ Page which first is ~~Com~~ allo cated frame will be emptied ~~first~~ (or out) first.

Implement FIFO algorithm for the reference string using 3 frames. 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

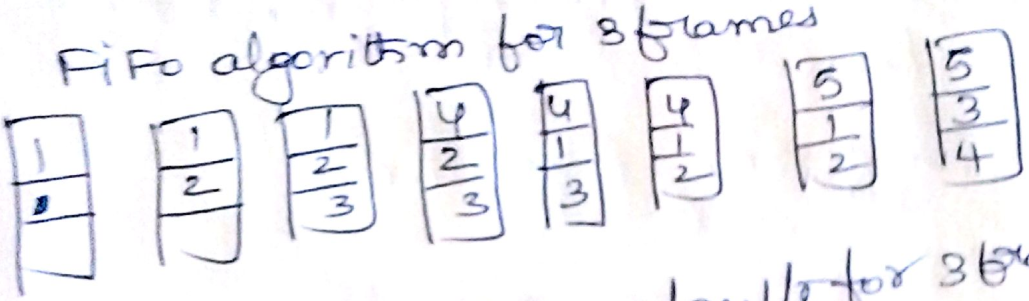
7	7	7	2	2	2	4	4	4	0	0	0	7	7	7
	0	0	0	3	3	3	2	2	2	1	1	1	0	0
		1	1	1	0	0	0	3	3	3	2	2	2	1

No. of pagefaults = 15.

FIFO page replacement ~~&~~ suffers from Belady's Anamoly .i.e if we increase no. of frames alloted for a process then no. of Page faults increase instead of decreasing. Below example shows this.

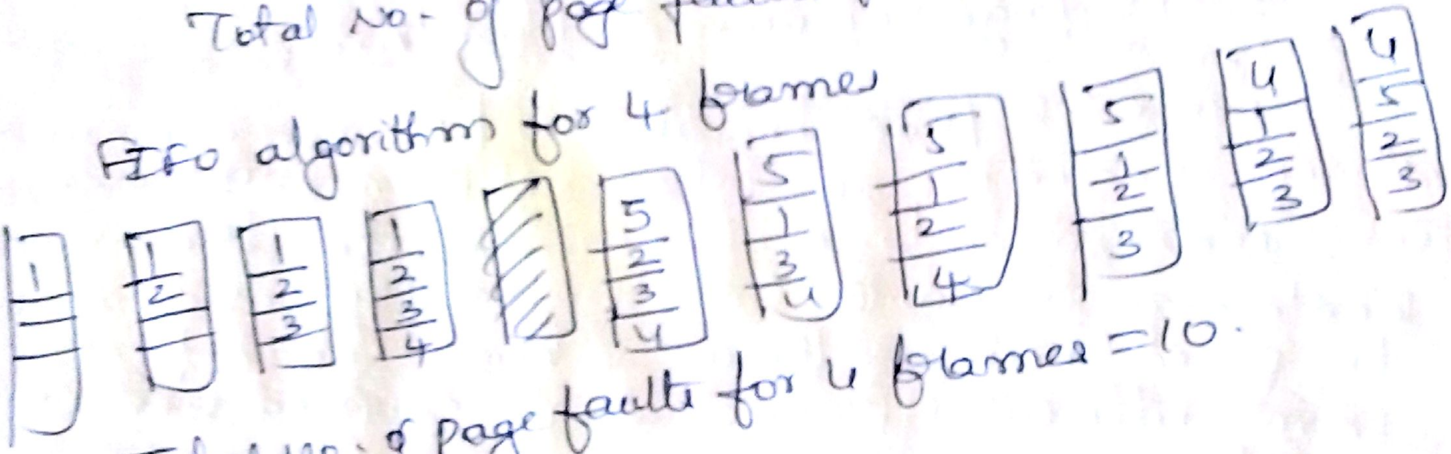
reference string is 1 2 3 4 1 2 5 1 2 3 4 5

FIFO algorithm for 3 frames



Total No. of pag faults for 3 frames = 8

FIFO algorithm for 4 frames



Total No. of page faults for 4 frames = 10.

∴ as no. of frames increased Pagefaults increased instead of decreasing, This is Called Belady's Anamoly.

(b) Optimal Page replacement Algorithm:

Here we replace a page that will not be used for longest period of time.

This Algorithm is best page replacement algorithm as it has lowest page fault rate.

It's difficult to implement this algorithm because it requires knowledge of ~~future~~ pages that will be used in future (i.e the "next time" when a page will be used)

Implement optimal page replacement algorithm for the below string with 3 frames

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

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

Total number of Page faults = 9

(c) Least Recently used Page replacement Algorithm (LRU)

Here page that is not used for longest period of time will be replaced.

Implement LRU algorithm for the below reference string with 3 frames

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

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

Total No of page faults = 12

(9) LRU Approximation Page Replacement Algorithm. They r four.

(1) Additional reference bits algorithm.

(2) Second chance (clock) Algorithm.

(3) Enhanced ~~Pag~~ second chance algorithm.

~~(4)~~

(1) Additional reference bit algorithm: ~~For~~ Steps

(i) To store history of each page n bits are used. (in our example 4 bits ($u_3 u_2 u_1 u_0$) where all are 0's.

(ii) when a page just allocated frame make left most bit u_3 to 1 for that page

(iii) After completion of ~~an~~ each interval all bits ~~[illegible]~~ $u_3 u_2 u_1 u_0$ are shifted right by one position.

(iv) If a page is to be replaced the page with lowest value for $\boxed{u_3 u_2 u_1 u_0}$ is replaced first

Implement additional reference bit Algorithm for the following string ~~& show~~ using 4 bits to store histo of page & use five frames. (T is end of interval)

3 2 3 T 8 0 3 T 3 0 2 T 6 3 4 7.

P	U3	U2	U1	U0
-	0	0	0	0
-	0	0	0	0
-	0	0	0	0
-	0	0	0	0
-	0	0	0	0

3 2 3 T

P	U3	U2	U1	U0
3	1	0	0	0
2	1	0	0	0
-	0	0	0	0
-	0	0	0	0
-	0	0	0	0

Right shift 1time abovetable $u_3 u_2 u_1 u_0$ for each page

P	u_3	u_2	u_1	u_0
3	0	1	0	0
2	0	1	0	0
-	0	0	0	0
-	0	0	0	0
-	0	0	0	0

——— End of Interval1—

8 0 3 T

P	u_3	u_2	u_1	u_0
3	1	1	0	0
2	0	1	0	0
8	1	0	0	0
0	1	0	0	0
-	0	0	0	0

Right shift onetime abovetable $u_3 u_2 u_1 u_0$ for each page

P	u_3	u_2	u_1	u_0
3	0	1	1	0
2	0	0	1	0
8	0	1	0	0
0	0	1	0	0
-	0	0	0	0

——— End of Interval 2 —

3 0 2 T

P	U3	U2	U1	U0
3	1	1	1	0
2	1	0	1	0
8	0	1	0	0
0	1	1	0	0
e	0	0	0	0

Right shift ontime above table.

P	U3	U2	U1	U0
3	0	1	1	1
2	0	1	0	1
8	0	0	1	0
0	0	1	1	0
-	0	0	0	0

— End of Interval3 —

6 3 4 7

P	U3	U2	U1	U0
3	1	1	1	1
6	1	0	0	0
4	1	0	0	0
0	0	1	1	0
7	1	0	0	0

Total No. of Page faults = 7.

(e) ~~LRU~~ Counting Based Page Replacement Algorithms. (18)

(i) Most frequently used (MFU) Page replacement Algorithm

Here ~~page count~~ for each page no. of times it is referenced till now is calculated as count. Page which has largest count will be replaced first.

Implement MFU algorithm for below string with 3 frames. 7 0 1 2 0 3 ~~0~~4 2 3 0 3 2 1 2 0 1 7 0 1

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

Total No. of Page faults = 13.

First 7, 0, 1 have occured for 1 time so any of them can be replaced to bring 2 page in.

(ii) Least ~~Recently~~ frequently used (LFU) Page replacement Algorithm

Here For each page No. of times it's referenced till now is calculated as Count. Page which has smallest Count is replaced first.

Disadvantage of this method is if a page is used during starting stage of process more times ~~than~~ it's Count will be more. So even if the ~~process~~ Page is not used during later stages of process it will not be replaced as it's Count is more.

7 01 2 03042 303 212 01701

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

Total no. of Page faults = 13.

If two pages have same least count then any one of them Can be replaced.