VIRTUAL MEMORY

CHAPTER # 10

→ Benefits in program execution that is only partially in memory.

① A program would no longer be constrained by the ammount of physical mem that is available.

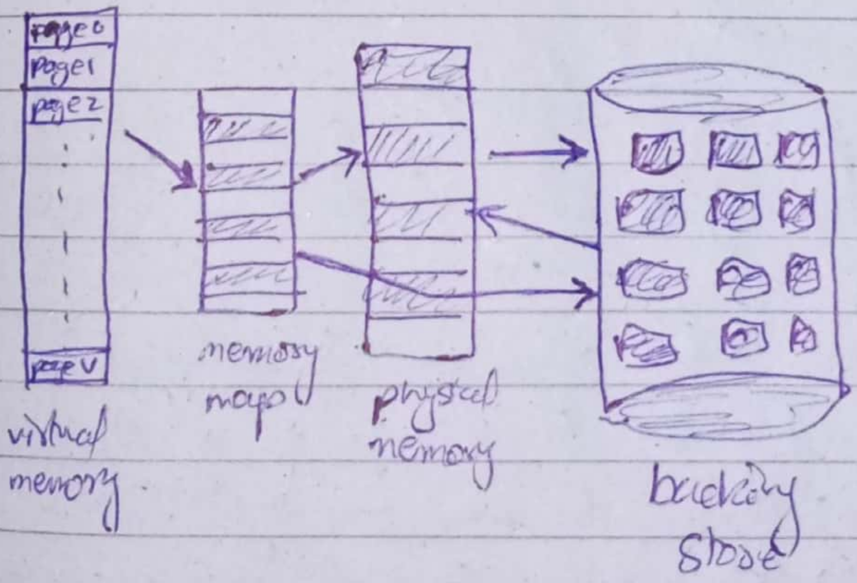
② Users would be able to write for an extreme large virtual address space

③ More programs run at same time

④ Incr CPU utilization & throuput

⑤ Less I/O would be needed to load or swap portions of program int mem.

→ The virtual address space of a process refers to a logical (or virtual view of how a process is stored in memory.

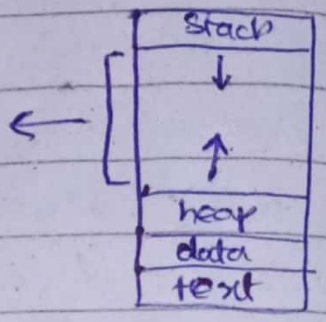


pages can be shared during process creation thru
system call.

like in chp # 09

→ Virtual memory also used in sharing memory b/w process

virtual address space that include holes are known as sparse address space. These holes can be filled when needed by a stack or heap.



"DEMAND PAGING"

→ Starting mein humein poora program memory mein load krne ki need nhi.

→ The best strategy is to load pages only when they are needed.

→ Pages that are never accessed are never loaded in main memory.

→ While a process is executing some pages may be in main memory & some in secondary memory or backing store.

→ Valid / Invalid bits are used to distinguish b/w the pages which are in memory or in secondary.

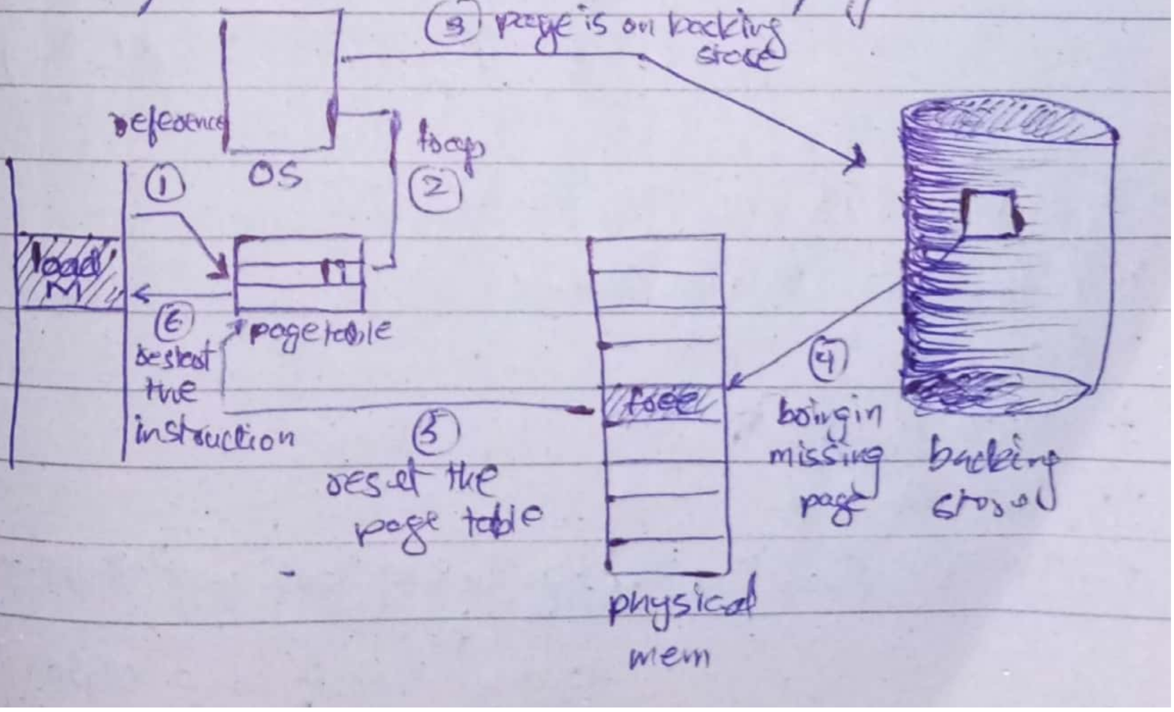
Valid → Page is in main memory.

Invalid → Either the page is not in address space of process or it is valid but currently in ~~secon~~ backing store (secondary storage).

→ Access to page marked INVALID causes page fault.

⇒ PAGE FAULT HANDLING:

→ First check a valid or invalid memory aceess.
↳ if invalid, then terminate the process
↳ else if valid then we page it in
→ Find a free frame (from a free frame list)
→ We now read the desired page from the backing store
→ The page is now allocated to the free frame.
→ The pagetable is then updated, to indicate that the page is now available in memory it is set to VALID.
→ We restart the instruction that was interrupted ~~to~~ due to page fault.!
→ Now process can access that page.



Thrashing : a phenomenon where the perfo[illegible] of a system significantly degrades due to excessive paging or swapping activity. It occurs when OS spends majority of its time m[illegible] pages b/w RAM & Secondary storager rather than executing actual proc[illegible]

- pure demand page : Never bring a page into memory until it is required.
  - ↳ We can start executing a process with no page
  - ↳ When OS sets the IP on the 1st instruction of process that page would not be available in main memory.
  - ↳ This will result in page fault.
  - ↳ Then page is brought into memory & continues it's execution.
  - ↳ Page faulting occurs unless every page is loaded in main memory.

- Hardware Support :

① Page Table

② Secondary Memory → This memory holds those pages that are not present in main memory. A part of SM i.e swap space is specialy allocated for this pur[illegible]

- Restarting Instructions on Page Fault :

1. Fetch and decode (ADD)
2. Fetch A
3. Fetch B
4. Add A & B
5. Store the Sum in C → Agr is step par page fault aata hai (for example c memory mein load nhi tha) to hame step 1 se restart krna parega.

Reasons of Thrashing:

→ (1) High degree of multiprogramming
(2) Large working sets
(3) Insufficent physical mem

- Another Scenario: For example a ~~program~~ instruction is executing & during it's execution it modifies or overwrite several data. After changing multiple data ~~it~~ a page fault occurs. Now when we restart the instructions how can we get the initial values prior to modification

Solution #01: The OS checks the source & destination operands whether they are loaded or not, if not a page fault occurs before an modification.

Solution #02: Use temporary registers to hold values of modified data. If there is a page fault the initial values are written back in memory.

Free Frame List: head → $\boxed{7}$ → $\boxed{8}$ → $\boxed{20}$ → ... $\boxed{98}$

→ To resolve page fault most OS uses a free-frame list.

→ Each frame is "zeroed-out" before being allocated, thus erasing their previous content. This is called 'zero fill on demand'.

- Pre-Paging: OS guesses in advance which pages will be needed and pre-loads them into memory.
  - ↳ If the guess of OS is wrong → page fault
  - ↳ Errors may occur in removing useful pages
  - ↳ Difficult to get right guess ~~&~~ due to branch code.

"Performance on Demand Paging"

→ let ma = memory access time
p = probability of page fault.

effective access time = (1-p)ma x (p) x page fault time

↳ means page is in memory
- may be in TLB
- or TLB miss
- Then page table entries loaded from main mem

↳ page fault probability

TLB mi
page n
in them
read fr
swapsp

→ With an avg page fault service time of 8ms and a memory acess time of 200ns,

$$= (1-p) \times 200 + p \times 8ms$$
$$= (1-p) \times 200 + p \times 8,000,000$$
$$= 200 + 7999,800 \times p \quad ns$$

effective access time ∝ page fault.

→ if want performance degradation to be less than 10%,

$$220 > 200 + 7,999,800 \times p$$
$$20 > 7,999,800 \times p$$
$$\boxed{p < 0.0000025}$$

I/O to swap space is usually faster than filesystems.

→ The use of swap space also affects the performance

Option#01: Copy entire process to swap space at program's startup than perform demand paging from swap space.

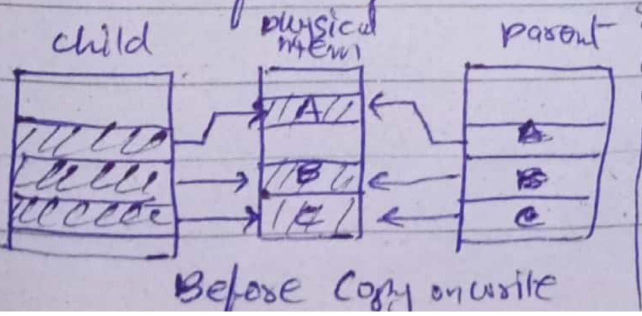
Option#02: (Better) Demand page from file system initially but to write the pages to swap space as they are replaced.

→ The demand paging of binary executable file is done from file system. As these files are not modified so they need ~~to~~ not be ~~be~~ replaced again in secondary memory, they are simply replace by another frame. And when needed again they are accessed from filesystem.

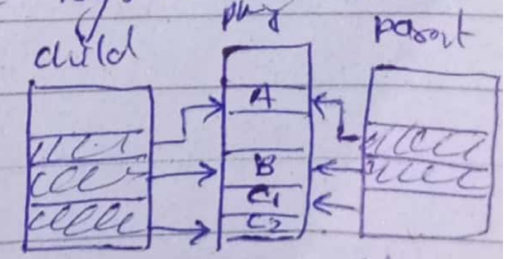
"Copy On Write"

→ Copy-on write allows the parent & child to share the same address space.

→ These shared pages are marked as Copy-on-write, means make a copy of the page when ever either of the process ~~write~~ trys to write on it.



Before Copy on write



After copy on write.

→ Only pages that can be modified should be marked as COW.

→ Read-only or executable code can be shared by two process.

Vfork(); Virtual memory fork.

→ During execution of child parent is suspended.

→ Child uses address space of parent

→ Doesnot use copy on write.

→ Any changes made by child will be visible to parent once it continues.

→ Vfork() is intended to use when child calls exec() immediately after creation.

→ Vfork() extremly efficent because of no copying of pages.

"PAGE REPLACEMENT"

→ Frame Allocation problem refers to the challenge of allocating limited physical memory frames among multiple process or pages in a virtual memory system.

↳ It involves deciding how many physical memory frames to allocate to each process or page & which frames should be allocated for efficent usege.

→ The page replacement problem refers to selection of a page to be evicted from main memory when a page fault occurs. This is done to add a new page ~~to~~ from secondary storage to main mem.

→ Following steps are taken when a page fault occurs. ~~(modified~~

① Find the location of the desired page on secondary storage

② Find a free frame:

↳ if there is a free frame use it

↳ else if there is no free frame, use a page-replacement algo to select a victim frame.

↳ write victim frame to secondary storage (if necessary); change the page & frame table

③ Read the desired page into the newly freed frame; change the page & frame table

④ Restart the process.

→ The swapping of pages increase the effective access time.

Dirty Page: A dirty page in a memory is the one which has been modified (written to) since it was last loaded.

↳ So if a dirty page is replace it is needed to be written back to secondary storage

Clean Page: A page which isn't modified since it was loaded. like executable or readonly fil

↳ These pages arent required to be written back, they are simply discared.

→ A modify bit (or dirty bit) is used for this purpose if this is set means dirty page if clear means clean.

→ These scheme significantly reduces the time required to service a page fault & it reduces I/O to a secondary storage by one-half for a clean p

① FIFO Page Replacement :

→ The first page brought in is replaced.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7
7	7	7	2	2	2	2	4	4	4	0			0	0			7
	0	0	0	0	3	3	3	2	2	2			1	1			1
		1	1	1	1	0	0	0	3	3			3	2			2
PF	PF	PF	PF 7 is replaced		PF	PF	PF	PF	PF.				PF	PF			PF

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

→ For the above reference string if we perform FIFO for 3-frames we get
hit = 3 & page-faults = 9

But if we do with 4-frames we get,
hit = 2 & pagefaults = 10

→ The page faults should have decreased by increasing no. of frames. This is called Belady's anomly.

⇒ Hit ratio = $\frac{\text{No. of hits}}{\text{total no. in refrence string}} = \frac{9}{18.6} = \frac{9}{18}$

→ Page fault ratio = $\frac{\text{No. of faults}}{\text{No. of refren}}$ =

② Optimal Page Replacement :

"Replace the page that will not be used for the longest period of time"

→ This algo gurantees the lowest possible page faul rate for a fixed no. of frames.

now 1 is replace

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

7 replace hoa q k we sabse last mein ayega 7, 0, 1 mein se

→ This algo cannot be implemented as it require futur knowledge.

⇒ Hit ratio = $\frac{11}{20}$, Page fault ratio = $\frac{9}{20}$

→ If we reverse this refrence string we will get same no of page faults

③ LRU Page Replacement :

" Replace the page that has not been used for longest period of time. "

→ Least Recently Used (LRU) algo.

→ It associates a timestamp with each page when it was last used.

→ Ismen hum peeche dekhenge k kon sabse last use hoa hai.

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		
x	x	x	x		x		x	x	x	x			x		x		x		

→ Hit ratio = $\frac{8}{20}$, Page fault = $\frac{12}{20}$

→ if we reverse the string we get same no. of page faults

→ Optimal & LRU are called Stack Algorithms; they never exhibit Belady's anomaly.

→ Timestamp, doublylinklist & stack.

→ Stack Implementation: Jo bhi page hoga recently refrenced wo stack k top par rhega. Agr wo middle of stack mein hai to waha se utha kr ~~uper~~ top par rakh denge. → To expensive as we need to change all pointers

→ Timestap also costly as OS needs to look at all pages &
[illegible] for all pages.

(4) Second Chance Algorithm:

→ A modified form of FIFO page algo.

→ Har page k saath ek refence bit hogi jo initialy zero hogi.

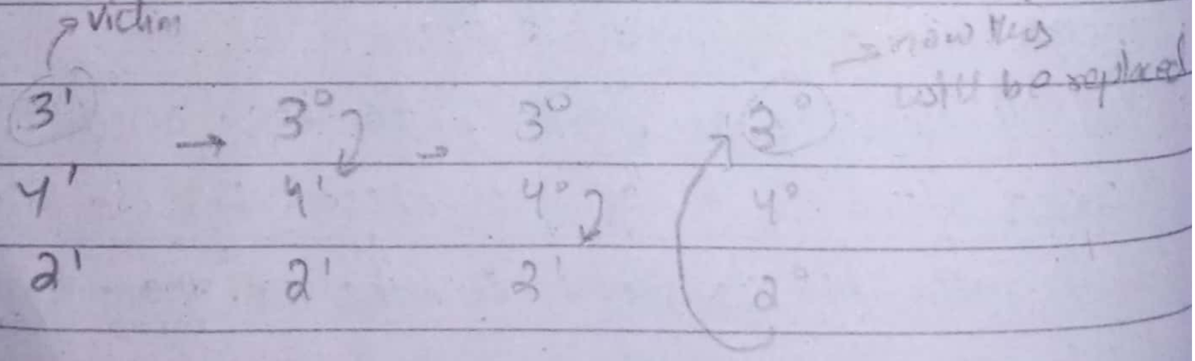
→ If the page is referenced again we set the bit.

→ The page which has a turn to replace and has a zero refence bit, is replaced.

→ ~~Agr saari hi reference bit zero 1 hai to sabki~~

→ Agr jis page ko replace krna hai uski ref bit 1 hai to usko hum zero krdenge or jo next replace ~~karne~~ hone wala hoga uski check krenge. Agr wo 0 hoa to replace agr 1 ho to next waale ko. Ye circular fashion mein chalega.

2	3	2	1	5	2	4	5	3	2	3	5
2^0	2^0	2^1	2^1	2^0	2^1	2^0	2^0	3^0	3^0	3^1	3^1
	3^0	3^0	3^0	5^0	5^0	5^0	5^1	5^1	5^0	5^0	5^1
			1^0	1^0	1^0	4^0	4^0	4^0	2	2	2
x	x		x	x	x	x					



(5) Enhanced Second Chance Algorithim :

→ Here we consider the modify bit along with the reference bit. With these two bit we can have four possiblities:

(Refrence bit ,	Modify bit)	
(0 ,	0)	neither recently used nor modified → best page to replace
(0 ,	1)	not recently used but modified → not quite good, as page needs to be written
(1 ,	0)	recently used but clean, probably will be used again soon
(1 ,	1)	→ recently used and modified, prob will be used again & also need to be written again.

→ Each page is one these four classes.

→ The OS goes at most three times searching for the (0,0) class.

→ (1) Page with (0,0) ⇒ replace

(2) Page with (0,1) ⇒ clear modify bit & continue search

(3) For pages with reference bit set, the reference bit is cleared.

(4) ~~Ao~~ Ek poora turn leliya or koi (0,0) wala nhi mila to

↳ On 2nd pass a page that was originally (0,1) or (1,0) might have changed to (0,0) ⇒ replace

↳ Agr koi (0,1) ya (1,1) wala hai doobra to modif clear

↳ By third pass all pages will be at (0,0)

⑥ Additional Bit Algorithm:

→ For each pages, OS maintains a group of reference bits 4 or 8 bits. Initially all used bits for each page are set to 0.

→ Every time a page is referenced in a time interval.
- ↳ Put 1 in place of MSB
- ↳ then Shift the bits to right after each interval.

→ The page with the lowest reference bits value is the one that is Least Recently Used, thus it will be replaced.

eg:
- 00000000 → The page which is not reference even once
- 11111111 → The page " " referenced at least once
- The page x with $\overset{(194)}{11000010}$ is more recently used than page y with 01110111 (119)
  ↳ jiski decimal value kam wo LRU.