else
use hierarchical page table

---

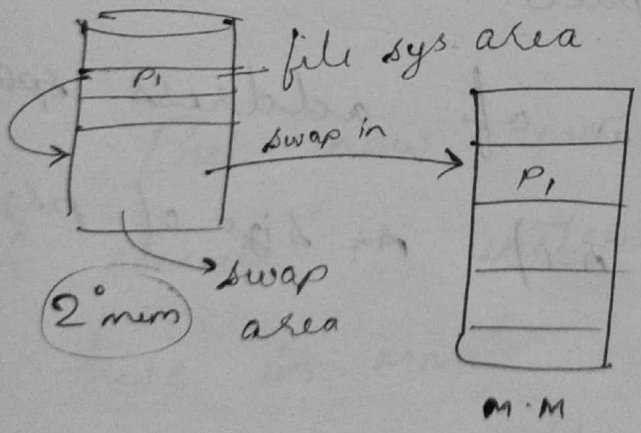
(2) swapping

data in file area of 2° mem is accessed by file system interface while the swap area is accessed by OS w/o file sys. interface for faster access

Address mechanism in 2° mem

| cylinder no | track no | sector no |

load pgm using swap area



after linking, move page to swap area and then move it to MM using swap in

Usually we swap only the stack & heap area of a process bcoz we can load new pgm to the code area P1 and later when P1 is needed we load its code area

from os mm pid stack and heap from swap area to improve swap area utilisation.

⇒ lazy swapper : (in paged mm mangiment) ## called pager

or pager

swaps in or out one page at a time based on the current requirment of mm.

Using page we create:

⊙ Virtual memory

virtual address space

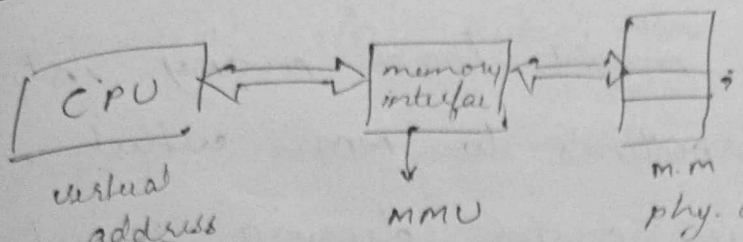
logical view of address space of a prgm (0 ... n-1) n - size of prgm

virtual address

P1	P2
0	0
⋮	⋮
n-1	m-1

Physical address

address of an instruction in mm.



Usual convention is that all the address space of a process must be loaded in M.M before its execution → von-neuman architecture

Now, LIMITATION DUE TO ALLOCATION OF ENTIRE LOGICAL ADD. SPACE

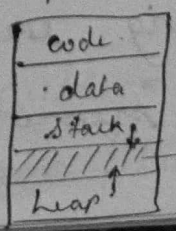
① limitation

let $\frac{P_1}{0 \\ \vdots \\ n-1}$ ⇒ n must be less than size of MM

i.e, virtual address space of a pgrm is less than MM

But now, we know for ith instruction to execute ith instruction & its data must be there in MM

② Also if we allocate both stack & heap in MM at start of pgrm itself, then large free space will be present w/o use for some time



③ Also we might load many interrupt or error routines to MM which may not always occur during execution & thus causes wastage of space

④ compile time data initialisation
e.g matrix multiplication

A[1000][1000]
B[1000][1000]
C[1000][1000]

compiler allocates almost a million bytes of memory where we actually use only a part of it. (i.e., since virtual add. space is large, for its execution large amnt of MM is allocated while we actually use only a part of it in execution

⑤ some features of a prgm may seldom get executed in which case loading entire prgm causes wastage of memory.

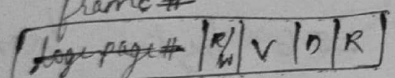
Improvement to avoid the above limitation

→ Allocate phy. mem only when its needed (not entire at start itself).

○ Demand page mem. management:

load a page only when its needed by a prgm.

Implimentation -

○ 

protection bits.

if valid bit = 1
→ corresponding page is there in MM @ frame #

if valid bit = 0
→ corresponding page is not in MM but its part of that prgm (PTLR)

load it to MM from 2° mem.

→ here page fault exception occurs (restartable exception)

eg: i: MOV EAX, [DATA1]

max no. of page fault = 2

a) for 'i' (instruction fetch)

here after execution we need to fetch the instr again

∴ store it after decrementing

5). for [eax] . page fault @ data fetch.

save & load status of process in pcb

eg ②: MOV EAX, ECX ⟶ instr fetch fault

③ MOVSB move max 256 bytes from one mem loc to another

more than 1 page fault

④ ADD EAX, ECX ⟶ here source operand itself is modified ∴ can't perform same instr again after page fault

(h/w handles it)

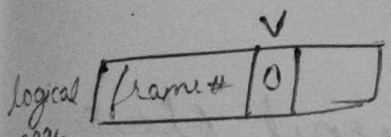
① Virtual memory allows us to execute a pgrm without loading the entire address space to MM.

① Also we can run a pgrm whose size is greater than size of MM

② can load and execute more process at a time. (as each program only uses less space

③ less I/O required to load/swap out a process. (less no. of pages need to be loaded as not all are needed at the moment.

We implement VM using demand paging



→ logical page was in 'frame#' but it got replaced later on. ∴ logical page is not current in MM.

So how is page fault implimented?

Whereas page ... system generates
no control over it ...
an interrupt:

In 80x86, CR2 stores cause and address at which an exception like page fault has occured. (present in all m/c in some way)

When interrupt occurs, CPU status, registers, IP, CS are stored and poped back after the ISR is executed to continue execution

In case of page fault exception

→ push SS, ESP (stack seg, stack pointer) → to kernel stack
→ push EFLAGS, CS, IP
→ push error code
→ JMP ISR of page fault

What does ISR do?

switching of page fault in 80x86

TSS
task stack segment
SS0 & SP0

① first OS must check if the mem. access is a valid one or not i.e. PTLR > 512

② If yes, then we need to load the page from 2° mem.

→ for that ② first we save gen. purpose registers

→ then initiate a disk read operation.

→ change status of process to WAITING.

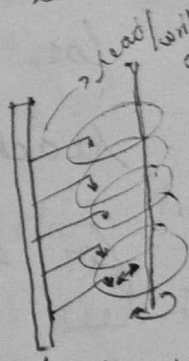
as many other process's read req. might be there in queue waiting for free memory in MM

→ When our disk read is initiate it has:

a) seek time

b) latency time

c) transfer time



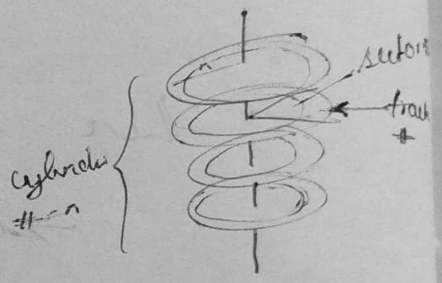
↳ handle move to ffs using stepper motor

Store info both sides of disk

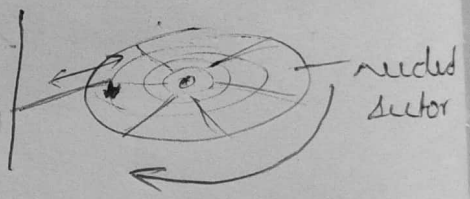
cylinder # a ⇒ all a^{th} tracks in all disks ~~with track # a~~

track # ⇒ surface # of cylinder

sector # ⇒ which sector of a surface



seek time: time need to move the read/write arm from previous state to current position (usually 5 ms)



latency time: time needed for disk's req. sector to reach the read/write head (usually 3ms for half revolution)

Now, if we need 5 ms for one revolution of a disk with n sector, we need the read/write head to get data for $5/n$ ms

It's called transfer time

time to read a sector

∴ total disk read time = seek + latency + transfer time

during this time, shedules run some other pgrm. When data is transfred, h/w generates an interrupts

2° mem Then currently running process waits. we load the page read to mm & update the page table entry of the process Then the earlier process which is in wait remain continues its execution.

When our process is sheduled some other time, we restrt the same instruction that caused page fault.

Now since instruction execution time is negligible when compared to seek time... Cost of page fault lies in disk read (wait time is assumed to be 0)

Now assume page fault occurs 1% of time and cost 8 ms.

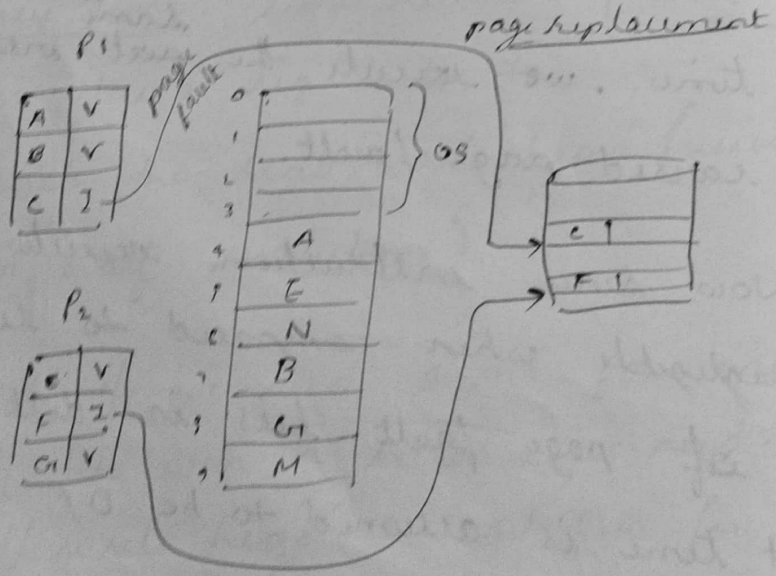
Av. mem. access time = $0.99 \times 1\text{ns} + 0.01 \times (1 + 8 \times 10^6\text{ns})$

$= 0.99 + 8 \times 10^4$

$\approx 8 \times 10^4$ ns

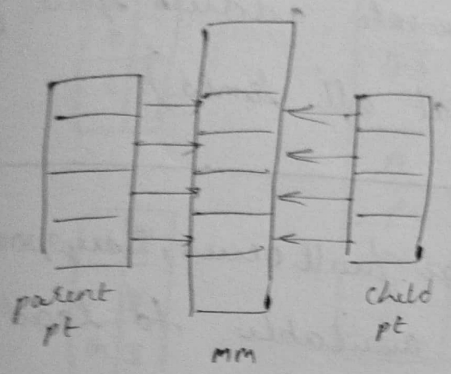
even 1% page fault causes, A.M.A becomes 80,000 times slower.

Advantage of demand paging

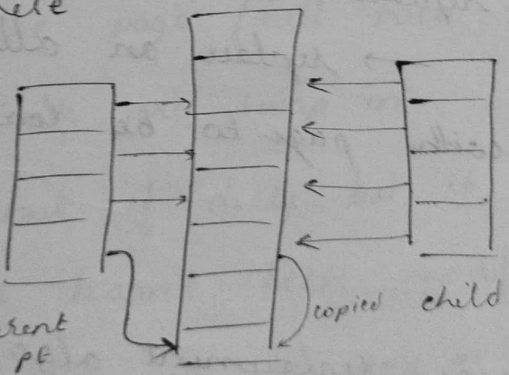


Copy - on - write

when fork() is executed parent & child will have same code space & data space



Now whenever parent or child writes something, the one who wrote copies the page to a new frame in MM and updates these



i.e.

using demand paging

if parent modifies its 4th code page then that page is copied to some other location and parent writes there

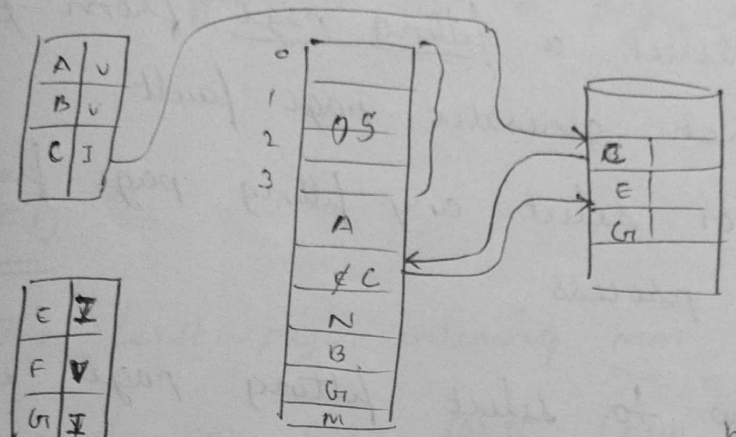
Now since in most of the cases, the code area is read only, it improves the performance of MM.
(no need for separate address space for parent & child at all times)

Now whenever page fault occur, suppose if no mem. is available to load page to, then we can

① terminate P_1 (not efficient)
② swap out P_1 (costly)
③ page replacement
→ replace an allocated page in MM with page to be loaded from 2° mem.

here:
(i) the page replacement algo chooses a page in MM & swap out it to 2° mem, updates the process's page table entry to invalid and update free frame list

(ii). ... req. page is loaded to the free frame



page chosen by page replacement algo

in pt

Now, if Dirty bit is 1, before swap out we need to write back the content of swapping page to 2° mem to avoid data loss. Here page fault need 2 I/O; write back and swap in.

but if dirty bit is 0, it means that page hasn't been modified and its content is same as its copy in 2° mem (loaded at first) ∴ no need to write back we only need to load new page to that frame ∴ only 1 I/O operation needed

that generated page fault

b). or select any fitting page from any process

Now to select fitting page we use page reference string.

a) It can be a randomly generated page but that won't improve efficiency

b). we create a sequence of (page refered, offset) and using that generate a list of pages that can generate page fault called reference string

eg: (0,0), (0,10), (1,5), (1,6), (10,7), (5,7), (8,3)

0, 1, 10, 5, 8

no page fault as 0th page accessed just before ∴ it's there in MM

pages that can generate fault

Based on the no of page fault occured using different string of MM, we can evaluate efficiency of a page replacement algo.

① FIFO

The first page entering MM will be swapped on page fault. (new pages are added to tail of FIFO queue.) or add a timer field

eg: 0, 5, 7, 0, 2, 1, 3, 1, 4, 1, 7, 8 → faults

0 already in MM : no fault

1 swap 0 as it came 1st

✓ => page fault

FIFO queue: 0 5 7 ~~0~~ ~~2~~ ~~1~~ 3 4 7 8

~~0~~	~~5~~	7	~~0~~ 3
~~5~~	7	~~0~~ 4	7
7	~~0~~ 2	4	~~7~~ 8

→ 10 page fault

→ not efficient

→ easy implementation

Now, what if we ↑ the size of MM?

Usually if ↑ the # page fault

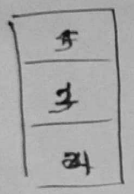
imp But FIFO algorithm actually increases the # page fault when certain seq. of page access is made even if we ↑ mm size

disadvantage of FIFO

→ It also ↑ page faults as it always replace active pages (it doesn't look recent reference)

1 2 3 4 1 2 5 1 2 3 4 5 ⇒ 9 faults

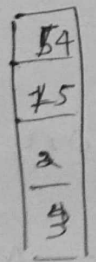
eg:



min. frame = 3.

Now if we ↑ frame does # page fault ↓?

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



⇒ 10 faults

∴ even though # frames ↑

page faults ↑

∴ not an efficient algo.

⊙ These an

Anoma

(2) Optim

replac

∴ here

refer

futu

a p

tim

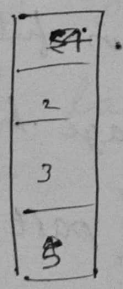
eg:

⊙ These anomalies are called Belady's Anomaly

② Optimal Algorithm

In FIFO, when 5 comes we replace 1 but immediate reference is for 1 ∴ here we don't have info about future reference. Now, if we know about the future references, we could replace 5 with a page in MM that is not referred for longest time.

eg: 1 2 3 4 1 2 (5) 1 2 3 (4) 5 = 6



replace 5 with 4 since its the page in MM that is referred after 1 2 & 3

replace 4 with 1 2 or 3 as 5 is referred immediately

But even though # page fault is min, its not possible to implement as it need info about future reference.

(can be used to compare efficiency)

(3) LRU

here the least recently used page is replaced.

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

In optimal algo, we look for a page that is not used for longest time in future

In LRU algo, we look for a page that is not used for longest time in past.

(based on locality

21	01
03	0
2	7

⇒ 12 page faults

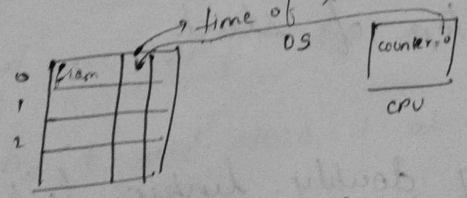
how to implent ?

we need an additional field to say time of most recent access.

① use counter

② stack.

① After each mem. access, counter - a h/w - is incremented, and counter value is copied to the field corresponding to each page (in page table) that shows time of last access



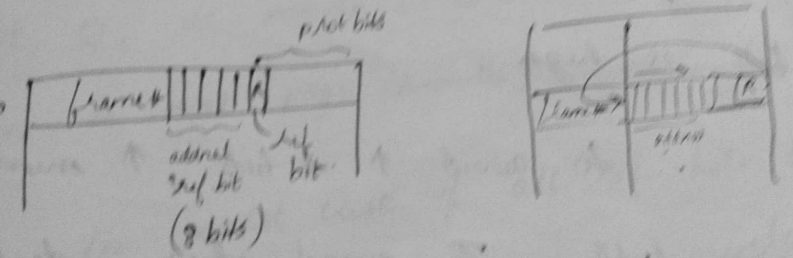
The page with min (time of last auss)

Scanned by CamScanner

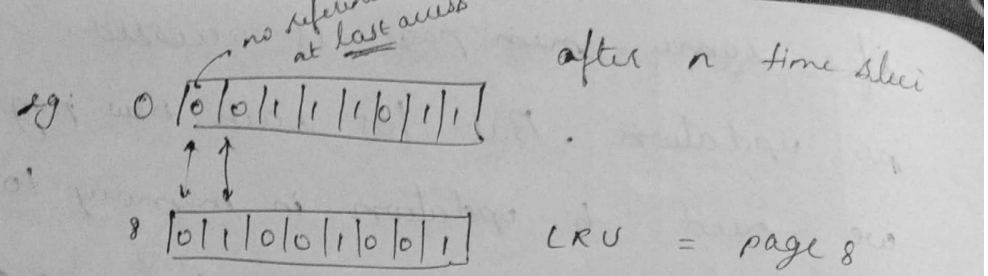
If same page is accessed no updation. But for each new page we need to updation in memory to update linked list & links to remove an entry & to add one to the end.

This is costly. ∴ modern system uses LRU approximation.

① It uses additional reference bits in page table.



Now at each time when a page is referenced we shift the addnal ref bit to right by 1 and store 1 at its leftmost bit else just shift (i.e. we add ref bit value to left end of addnal ref bit & shift it



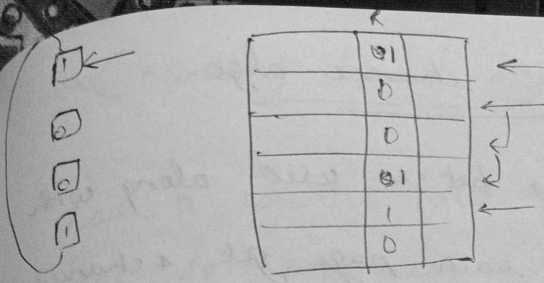
∴ we can find LRU using just integer comparison.

But if 2 have same addnal ref bits use FIFO or some other algo to select.

But if there are a 1000 pages & only 8 bits, the no. of pages with same ref. bits is high.

∴ to avoid ↑ # bits, ↑ mem. of page table. To avoid this decrease the # reference bits.

② If # addnal ref. bits = 0, that algo is called second chance / clock algo. i.e., it only has a single ref. bit (no addnal bits)



Here whenever a page fault occurs, we look for a page to be replaced by looking at its ref. bits. if its 1 then its made 0 (gives a second chance) and look the ref. bit of next page.

When we encounter a page with ref. bit 0, its made 1 and replaced, and move to next page (handle)

In worst case if all ref. bits are 1, then since the pages are in circular queue, after all R are made 0, it again checks the first page, since it was made 0 in last check, its replaced

(3) Enhanced 2nd chance algo.

Here dirty bit is used along with ref bit. ∴ each page get 4 chances

R , D

if no (0,0) choose → (0 , 0) → best page to be replaced (no new ref & no need for write back)

if no (0,1) → (0, 1) → next best (need to write back)

if no (1,0) → (1, 0) → next class (no write back ∴ give second chance)

(1, 1) → next class

mostly recently referred & need to write back

replace the first non-empty class pages

∴ to implement this we need multiple traversal through circular queue.

1st check if any (0,0) page exist. else check for subsequent classes.

Counting based algorithm:

has a counter field along with each page so whenever a page is accessed its counter is incremented

Now how to choose page?

(a) LFU (least frequently used)

Whenever a page fault occurs, the page with least no of access is replaced

problem:

The page with least no. of access (recently brought to mem.) may have large no. of access in coming future

(b) MFU (most freq. used)

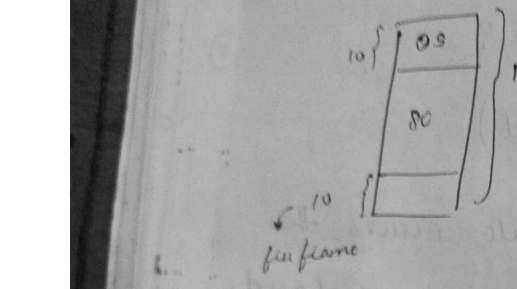
replace page with max # references.

i, assuming the recently brought to mem pages with low counts may have max accesses in near future

Cost comparable to optimal algorithm

Page buffering algorithm

→ here OS keeps a free frame pool.



i.e., after the 80 frames are filled, process generate a page fault.

here when the OS is executing the page replacement algorithm, pager moves the page to be access from 2° mm to free frame pool. ~~if the victim then~~ the victim page is moved to free frame and the page brought to mm to the victims locate if the victim has dirty bit 1, then its written back & finally free frame is freed

Again if we keep track of pages in free frame table, we can avoid moving page from 2° mm to mm if its already there in free frame pool (when it was victim, it was moved to there)

Additional enhancement

Whenever page is empty (it I/O is free), pager writes back a page with dirty bit 1 in mm & make it 0 so that it can avoided when this page is selected as victim in the future.

Allocation of frames.

In virtual mm, when size of virtual mm is larger than size of physical memory how to allocate frames?

→ Depends on mem. architecture

⊙ To get min no. of frames.

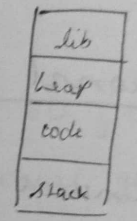
Assume each instr is 1 word & mem op

how many min. no. of page is required

to execute this?

2 pages — instr ↪ in code page

↳ mem access ↪ in stack/ heap.



eg ADD EAX, DATA1 → 2

ADD [DATA1], DATA2 → 3 (worst) (mem op)

ADD DATA1, [DATA2] → 4 (worst)

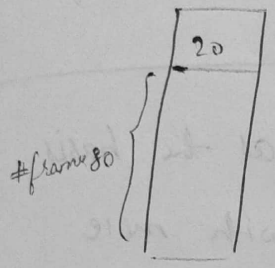
↳ mem op has address to another location

∴ We must allocate min. ~~#~~ pages

How to allocate?

Allocation algo ⌐ fixed algo.

① fixed allocation



let # process = 10

then allocate $\frac{\#\text{frames}}{\#\text{process}}$ to e

process. $= \frac{80}{10} = 8$

but here, for some process, it may have 1000 pages & some have only 1 page Then the one with 1000 page generate more page fault.

② proportional allocation

i memory allocated proportional to size of pages of process.

let s_i be size of process.

$S = \sum_{i=1}^{n} s_i$ (total size of n process)

∴ process P_i gets $\frac{m}{S} \times s_i$ pages

m - # frames (size of mm

Also we can use priority as the basis of allocating mem. One with more priority gets more pages. Or we can use a combination of size & priority.