else

use hierarchical page table

---

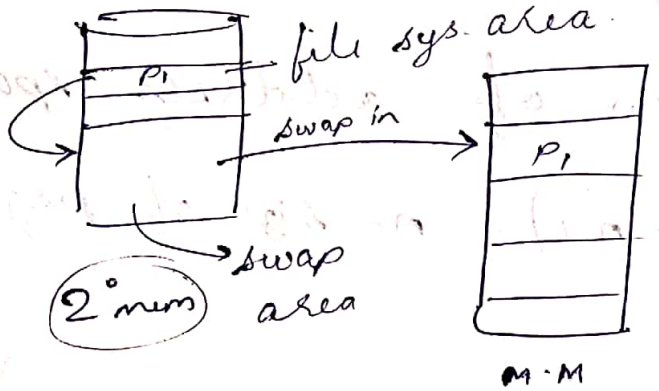
(12) swapping

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 2 mem fits stack and heap from swap area to improve swap area utilisation.

⇒ lazy swapper. (in paged mem. management) it called pager.

or page

swaps in or out one page at a time based on the current requirement of MM

Using page we create:

⊙ Virtual memory

virtual address space.

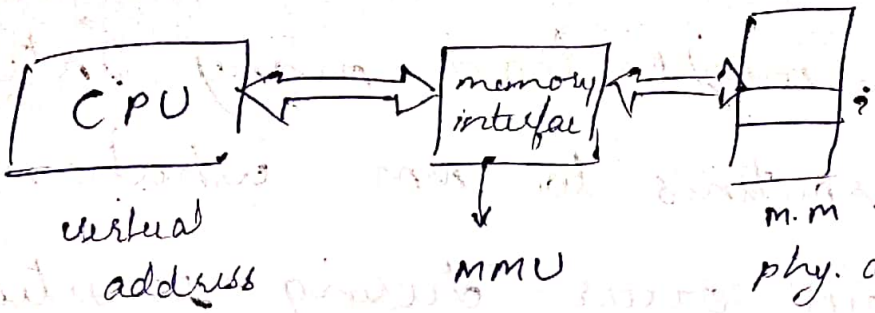
logical view of address space of a prgm (0 ... n-1) n - size of prgm

virtual address

P_1	P_2
0	0
⋮	⋮
n-1	m-1

Physical address

address of an instruction in MM.



Usual convention is that all the address spaces 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

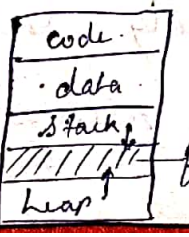
(1) limitation

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

i.e. virtual address space of a prgm 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 prgm 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

eg: 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 intie at start itself).

⊙ Demand page mem. management:

load a page only when its needed by a prgm.

Implimentation -

○ | ~~logic page#~~ frame# | R/W | V | D | R |

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

(instruction fetch)

here after execution we need to fetch the instr. again

∴ store IP after decrementing

b). for [DATA] . page fault @ data fetch.

save & load states 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)

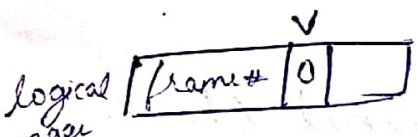
⊙ Virtual memory allows us to execute a pgm without loading the entire address space to MM.

(1) Also we can run a pgm 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?

Whenever page fault occurs, user have no control over it ∴ System generates 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?

simplified 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 $i < PTLR * 512$

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

→ for that @ first: we save gen. purpose registers

→ then initiate a disk read operation

→ change status of process to WAITING

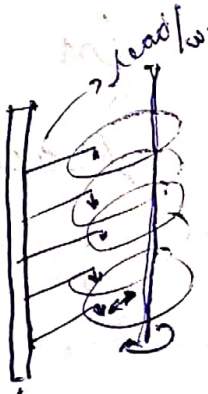
as many other process's read req. might be there in queue waiting for free memory in MM

→ When our disk read is initiated it has:

a) seek time

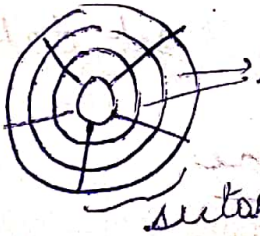
b) latency time

c) transfer time



↳ handle move to ffo using stepper motor

store info both sides of disk

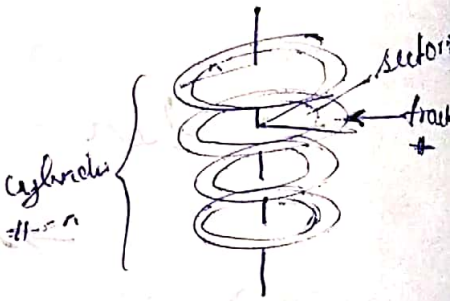


cylinder # 'a' ⇒ all a^{th} tracks in all discs

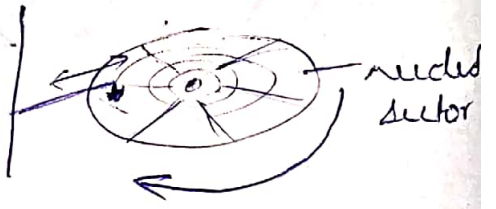
~~with track # 'a'~~

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 disc'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 disc with n sector, we need the read/write head to get data for $\frac{5}{n}$ ms time to read a sector

It's called transfer time

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

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

2° mm 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 excute 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.

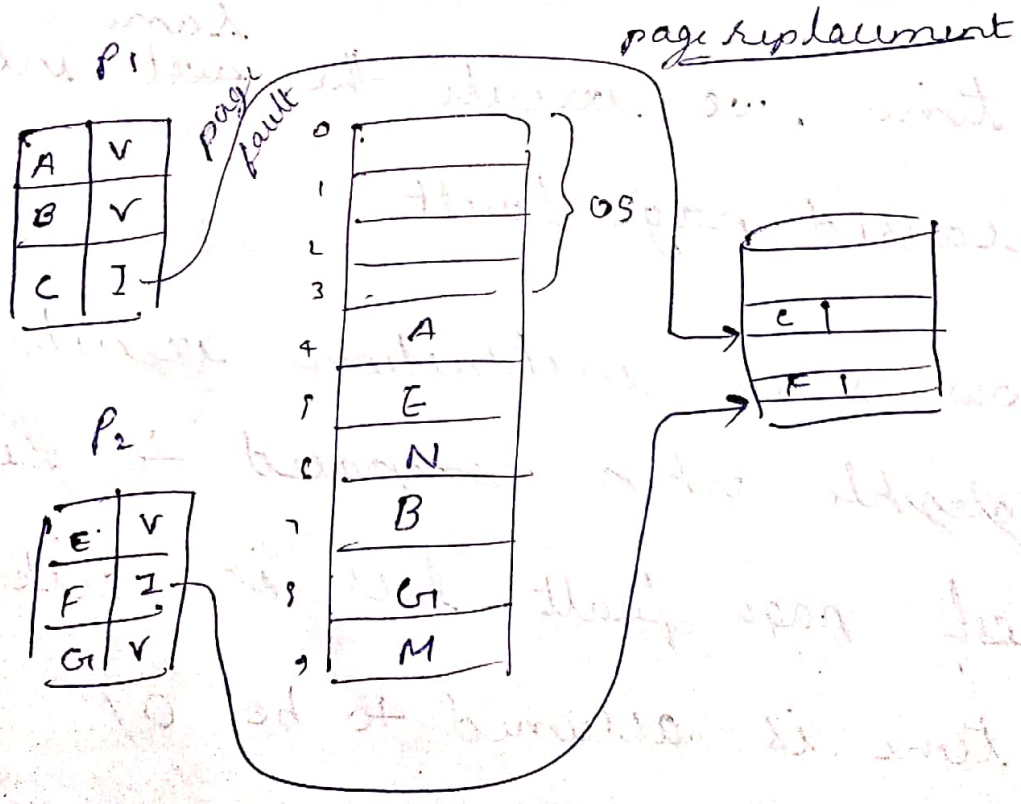
Av. mem. access time = $0.99 \times 1ns + 0.01 \times (1 + 8 \times 10^6 ns)$

$= 0.99 + 8 \times 10^4$

$\approx 8 \times 10^4 ns$

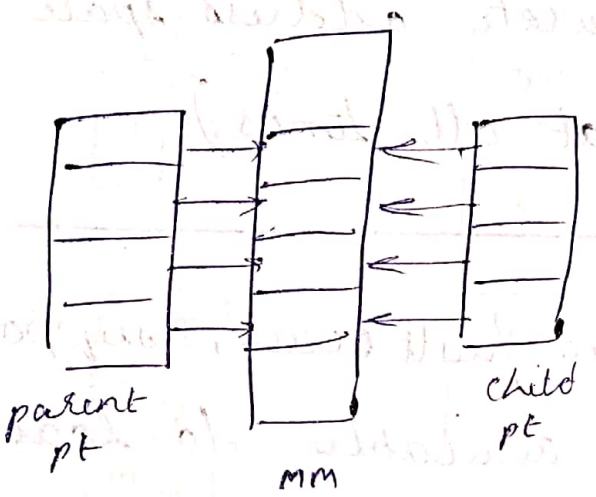
even 1% page fault causes, A.M.A becomes 80,000 times slower.

Advantage of demand paging



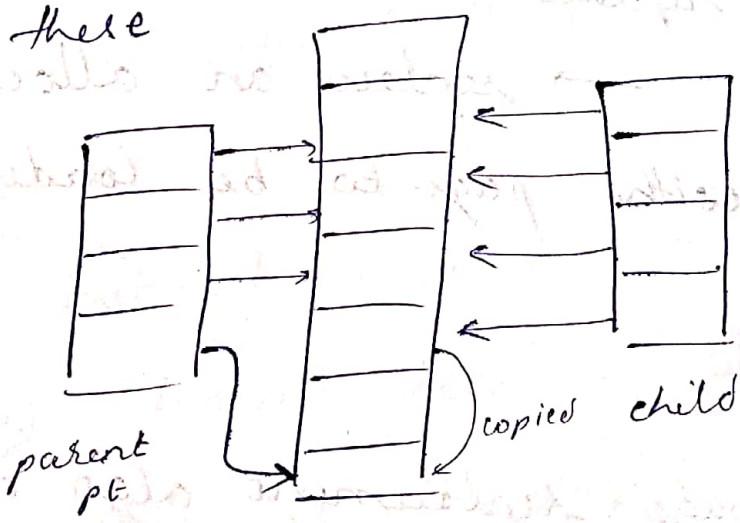
copy - on - write

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

ie.,



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)

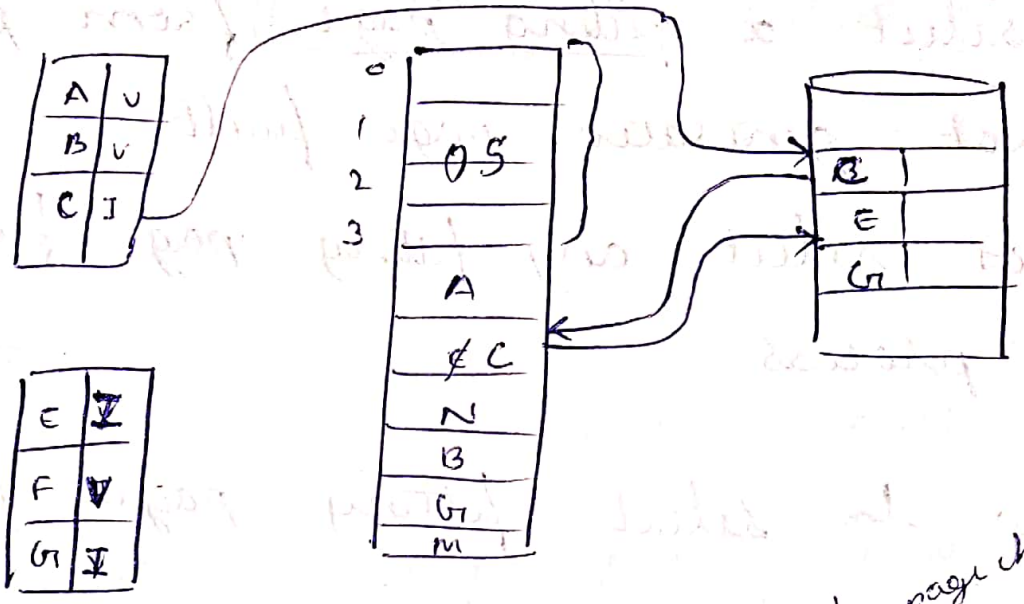
Now whenever page fault occurs, suppose if no mem. is available to load page to, then we can

1. terminate P_i (not efficient)
2. swap out P_i (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). Then, the req. page is loaded to the free frame



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's, 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

Now how a page is selected for replacement?

a). select a fitting page from process 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 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 there in MM

pages that can generate fault

Based on the no. of page fault occured using reference string & 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~~ ✓8
~~5~~ 3
~~7~~ 4
~~2~~ 7

→ (10-) page fault

→ not efficient

→ easy implementation

Now, what if we ↑ the size of MM?

Usually it ↓ the # page fault.

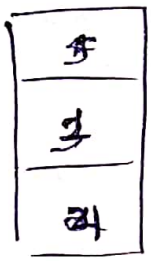
imp But FIFO algorithm actually increases the # page fault when certain seq. of page access is made even if we ↑ MM size.

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:



mm. 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.

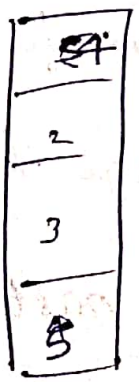
⊙ These anomalies are called Belady's Anomaly

(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 min 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

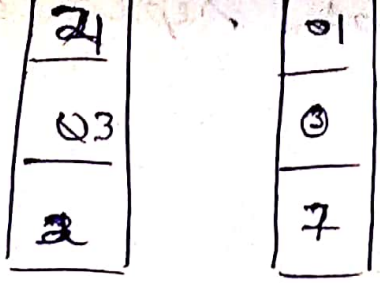
here the least recently used page is replaced.

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



⇒ 12 page faults

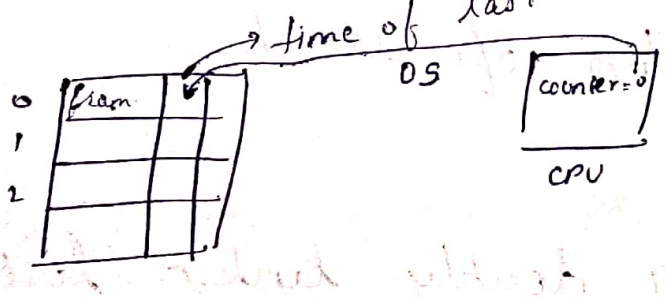
how to implint ?

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 access)

is the page to be replaced.

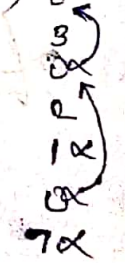
Issues:

(1) need to check all mm entries in page table to find min (counter value) - i.e., search in all pt where (v=1 & counter = min (counter))

(2) each mem access need an additional mem. access to write counter value to mm (costly!)

Now we can avoid checking all entries by using stack

i.e., LRU page is always @ bottom of stack. Whenever a page is referenced (if its present in stack, its removed) and put on top of stack

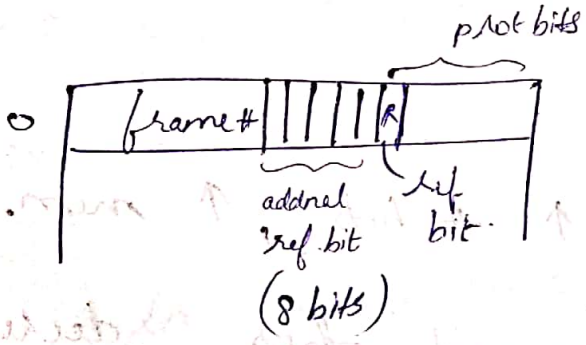


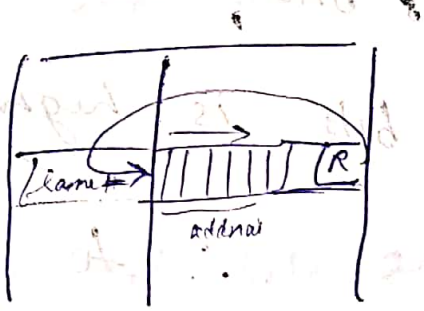
Now using doubly linked list we can implement the stack

If same mem page is accessed no updation. But for each new page we need 6 updation in memory to update linked list 4 links to remove an entry & 2 to add one to the end.

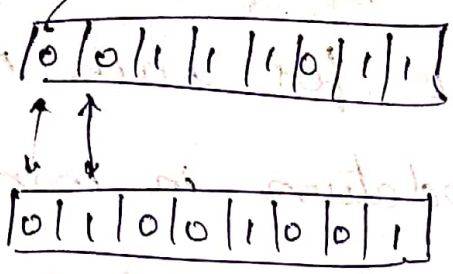
This is costly. ∴ modern system uses LRU approximation.

(1) It uses additional reference bits in page table.





Now at each time slice if 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 leftmost end of addnal ref bit & shift it right



after n time slice

LRU = page 8

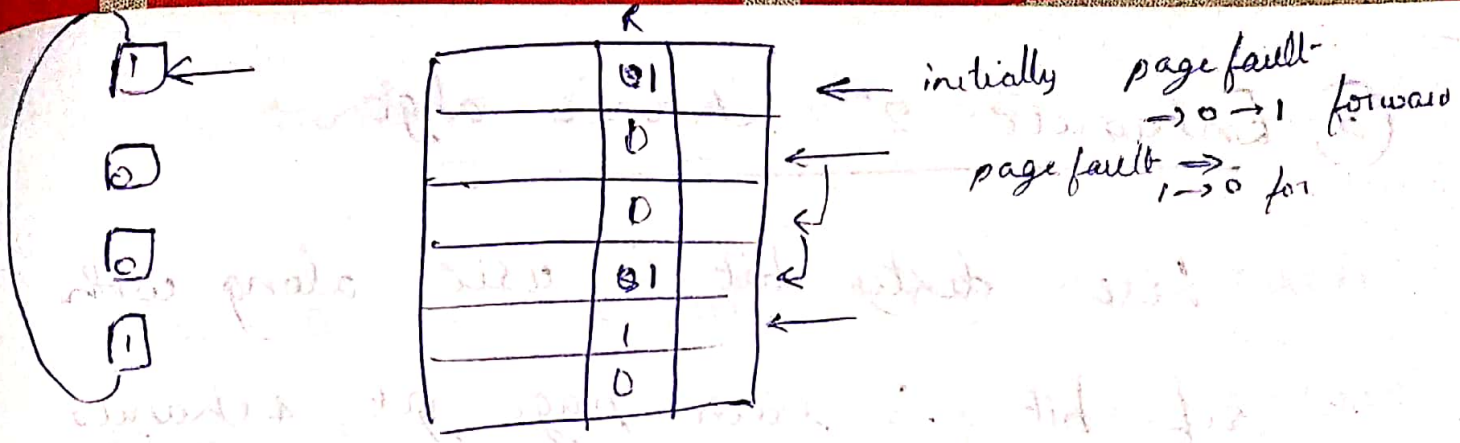
∴ we can find LRU using just integer comparison.

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

D , R

if no (0,0) choose → (0 , 0) → best page to be replaced

→ (0 , 1) → next best (no new ref & no need for write back)

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

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

→ 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 may have large no of access in coming future (recently brought to mem.)

(b) MFU (most freq. used)

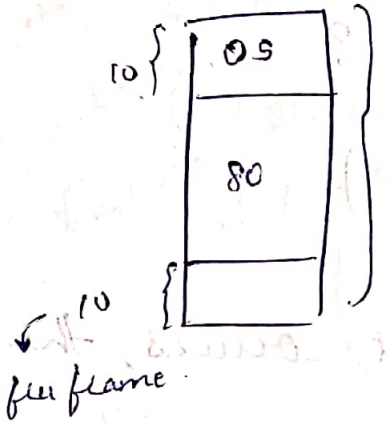
replace page with max # references.

i.e. assuming the recently brought to mem pages with low counter may have many accesses in near future.

(not compatable 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° mem 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 location. 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° mem 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 (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 mem., when size of virtual mem is larger than size of physical memory how to allocate frames?

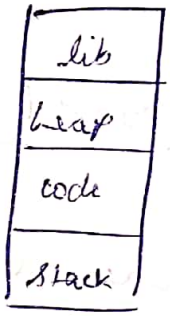
→ Depends on mem. architecture

⊙ To get min no. of frames.

Assume each instr is 1 word & 1 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)

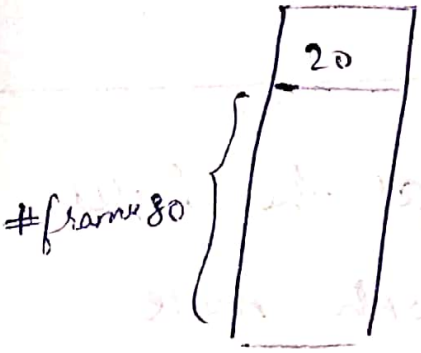
↳ mem op. has address to another location

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

but like, 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

ie 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 $\frac{m}{S_i} \times S_i$ 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.

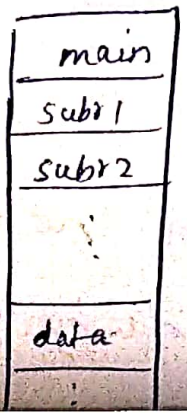
Cons: When we use swap in swap out mechanism, when heavy page faults occurs, then # processes waiting for swap in/out ↑ & as a result #process in ready queue ↓ drastically. Thinking its because of lack of enough process in mm, system adds more process, ↑ degree of multiprocessing, which adds to page

fault, due to less #frames, processes waiting for disc oper. ↑ & CPU utilisation ↓ drastically. This situation is called **thrashing**

Solution

(1) local swapping : when page fault occur, replace a page of same process (no an ideal soln)

(2) Allocate enough memory to continue execution of process.

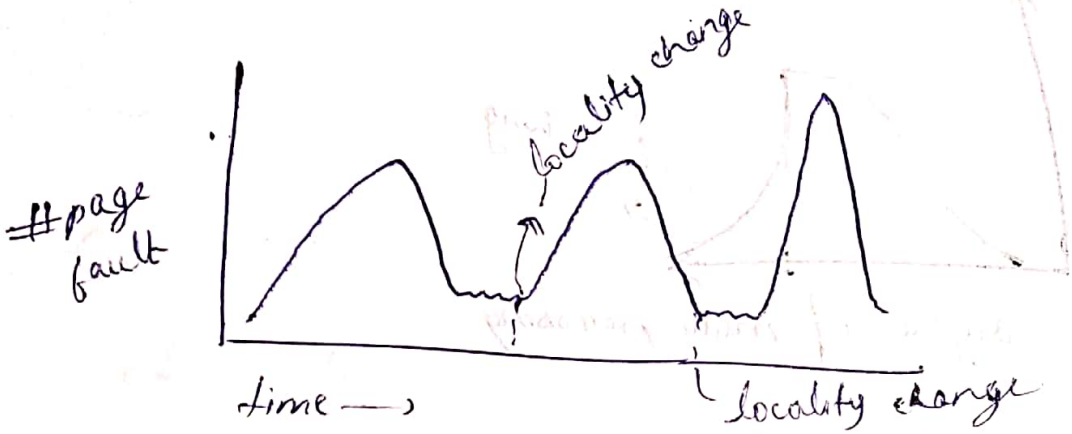
P1 

— It is based on locality of reference

→ initially to run P, we need pages to load main & data associated locality of ref.

— when sub routine 1 is executed its locality of reference changes as it needs another set of data

— III^rd for sub 2 and rest.



→ i.e., initially there will large # of page fault but later it ↓ & reaches a steady state But one locality change, # page fault again ↑

Based on this, we have 2 model;

Working set model.

Δ working set window = the last 'n' no. of mem access made by process.

$ working set : Actual # diff pages that are accessed in the working set window.

1 2 5 7 4 3 5 2 1 4 1 3 7 1 8 10 12 17

window after some time

w.s = 1 2 5 7 4 3

If Δ = 8

WS = 3 7 1 8 10 12

Now as window changes, working set also changes.

Let $D = \sum_{i=1}^{n} WSS_i$ → working set of i^{th} process.

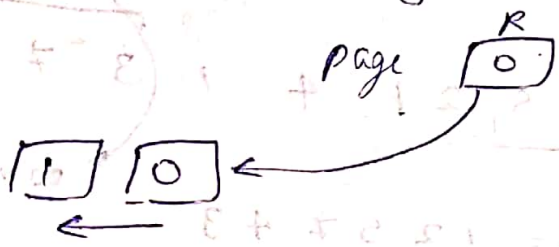
i.e. i^{th} process use WSS_i # pages

If # frames = $n < D$, then # page fault is high

else if $n > D$, then all pages can be accommodated in mm ∴ less page fault

IMPLEMENTING WORKING SET MODEL

Now let timer = 100 for every page there is an additional in memory reference bit.



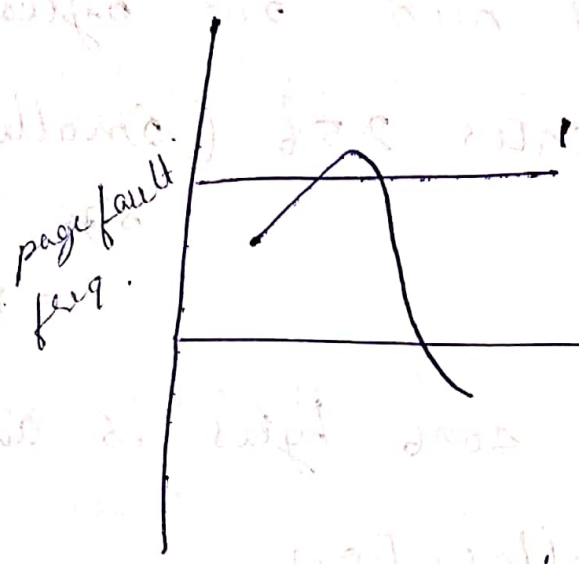
when timer int occurs we copy ref bit of page to its own page ref.

Now after the Δ (working set window) is passed if there is atleast 1 reference to that page in all three ref. bit, we ~~don~~ say that, that page is in the working set.

If all three ref bit is 0, no ref in last > 300 time,

∴ it's not in working set & hence can be removed.

Page fault frequency



page fault upper bound

page fault lower

based on local page replacement algo.

when # page fault is above threshold 1, OS adds frames to the process and if its less than threshold 2, it reclaims frames from that process to keep # page fault b/w the upper & lower bound.

All processes replace its own pages.

---

kernel memory allocation

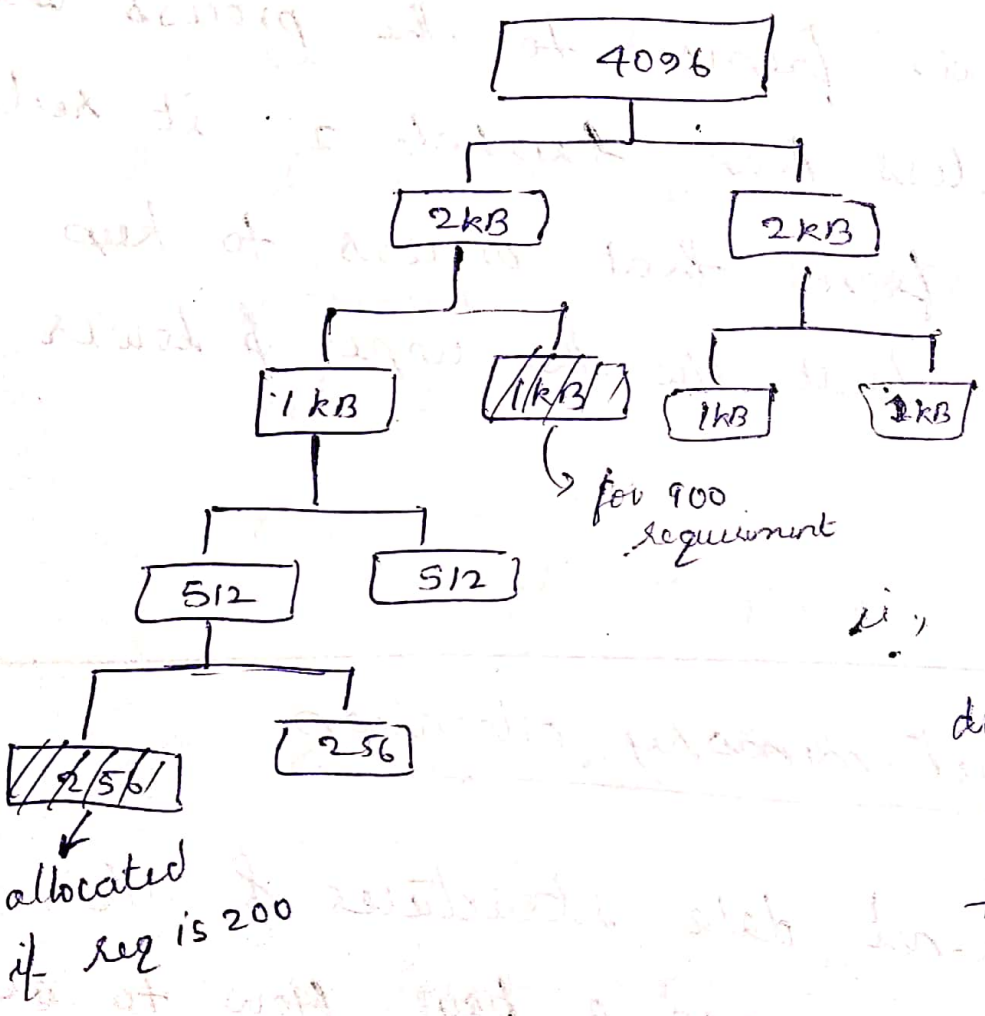
For kernel data structures & I/O buffer may need a page. How to allocate page?

(1) Buddy system

If kernel need 200 bytes.

then it allocates 256 (Smallest memory size ≥ req size = 2^n)

suppose initially 4096 bytes is available for kernel allocation



i) large block is divided into smaller blocks till least memory is wasted

Now, when [256] block is freed, it combines blocks till largest sized free block is obtained.

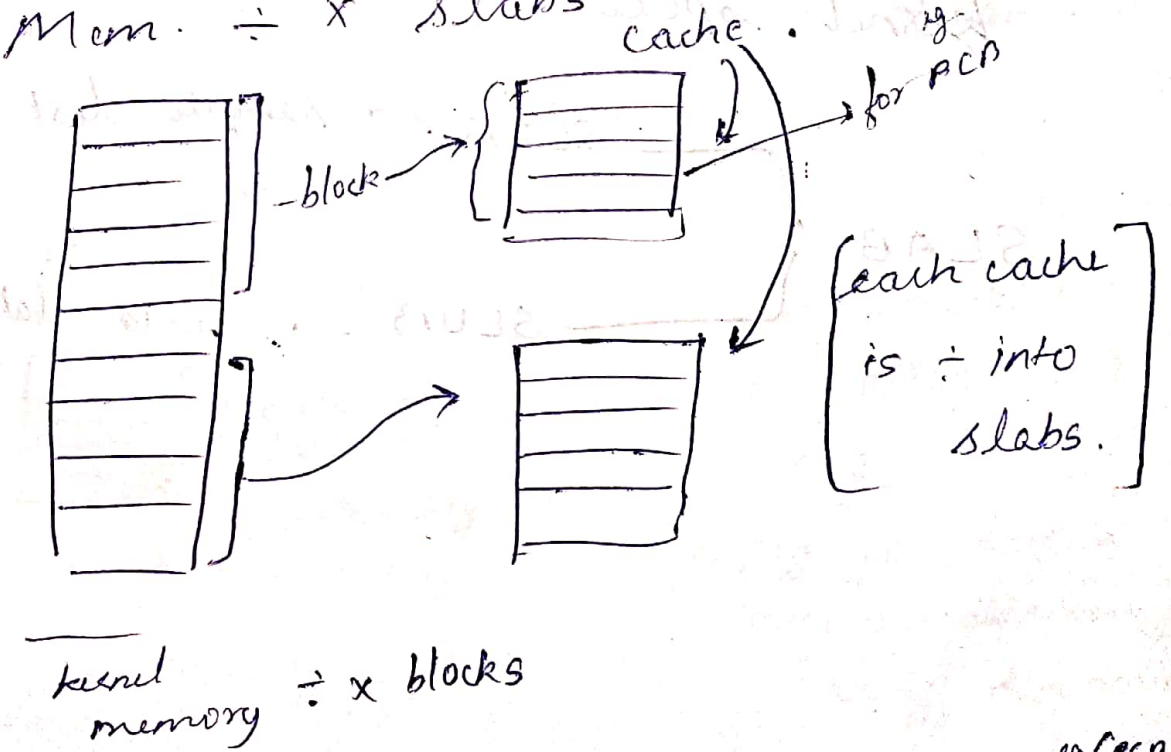
Cons

→ memory can be allocated as a power of 2. ∴ internal fragmentation occurs

---

(3) SLAB Allocation → one or more contigous pages

Mem. ÷ x slabs



kernel memory ÷ x blocks

eg (PCB)

if any internal kernel data structure is stored in a cache, all PCBs are stored in

that cache.

A slab can be in free state, partially filled or ~~com~~ fully allocated state.

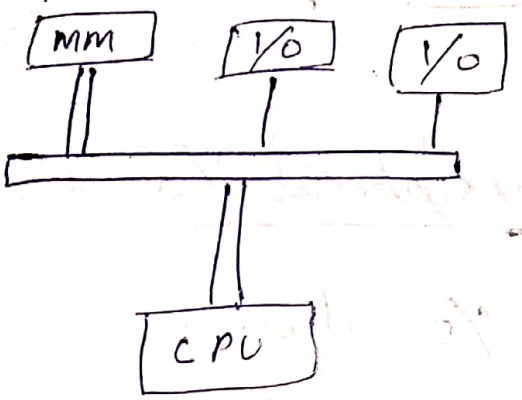
Now, when a new process's PCB comes, it stored in a partial or free slab in the cache allocated for PCB.

If no free slabs are there, OS requests an additional slab from kernel space.

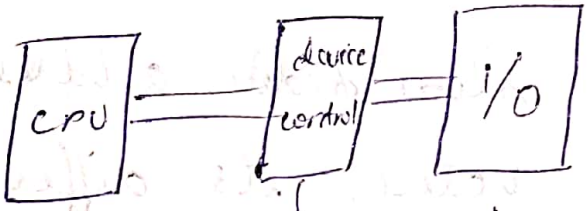
or part-filled

SLAB
- SLOB → simple list of block
- SLUB → meta data for SLOB ?

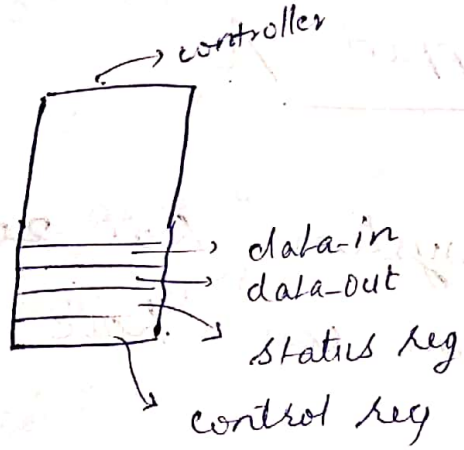
Device management



I/O device H/W & I/O controllers



interface H/W that allows transaction of data b/w CPU & I/O device



all device controller has atleast 4 reg.

reg for data-in out & status may vary depending on complexity of I/O device

status reg has the status & control reg has control info used to control the I/O device

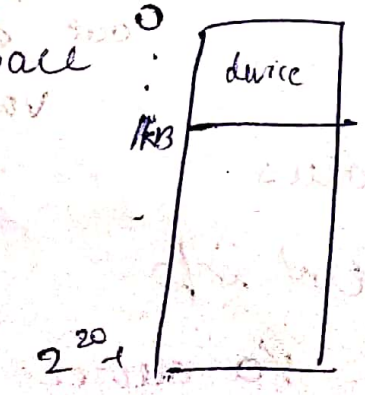
Device addressing

(1) We have peripheral I/O i.e., seperate address for I/O devices
(64 kB address for 80x86)
↳ diff from mem. address

Also called isolated I/O
(even though mem & dev. address are equal in value, its different in accessing mechanism)

(2) Memory mapped I/O

Suppose our system has 20 bit add. bus, then MM can have $2^{20}-1$ add. space



Now a portion of this, is reserved for device addressing

ii, each address in the 1st 1KB (in above example) is associated with an I/O device.

eg. mov add., 10

if add is in 1st 1KB, then 10 can be given as I/P or O/P data to a device

[In isolated I/O, IN & OUT instruction are used for mem. related operations]

In isolated, only 64KB is there for I/O but in memory mapped I/O we can have as many as our requirement within $2^{48}-1$ address space as per architecture.

WAYS TO ACCESS TO I/O

① polling: CPU always reads status reg which indicate whether the I/P or O/P

is ready in data_in or data_out reg.
If ready, then cpu transfer data
to/from I/O.

i.e., continuous read of status reg
& status of I/O

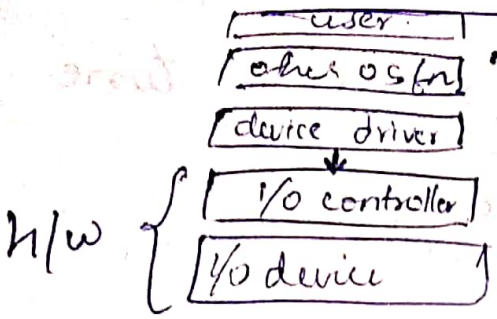
Cons: If speed of I/O is comparable to cpu, we can use this

(2) interrupt driven I/O: When cpu req. I/O, it initiates it & controller gives it to I/O. once the I/O is done an interrupt is given back to cpu to says its done & do further instr:-

(3) DMA : I/O can directly access MM

If its a block transfer (4096 bytes) interrupt driven & polling is ineffient (I/O → cpu reg → MM)

1) Device driver



device driver is a s/w (part of OS) that manage device using I/O controller (In linux its ldm - loadable device module)

(Uniform interface for device driver:

The device driver provide a uniform interface for OS to interact with diff types I/O devices & their controllers

eg: common stand. to interact with drivers
u(read, n, device#)

Types of I/O device ~~drivers~~

character device & block device

→ block devices - reads & write 1 block at a time

eg: magnetic disk

character ~~by~~ device : reads & write a stream of bytes at a time

eg: 1 byte @ a time

Linux I/O implemented based on file system ~~I/O~~ interface

suppose we want to read from /dev/disk

OS converts it to its inode # ("11" to a file)
for device also

inode # has major # → type of device
it allows OS to load the driver based on the device corresponding to major number with minor number as parameter (to know instant of that device)

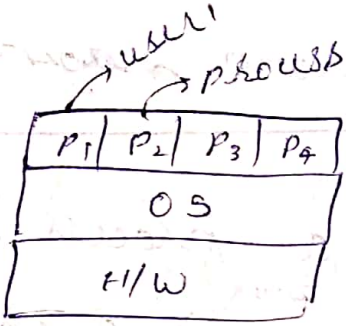
eg: major # says to write to a printer
minor # " " " " printer1

writing & reading to/from file / I/O device has common interface in OS level.

Virtualisation

Gives as an intution of seperate m/c with h/w, kernel and address space

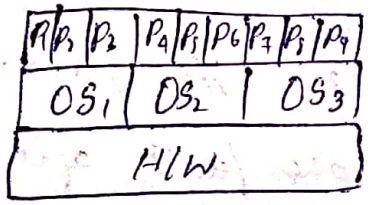
Till now



user feels that the entire h/w is dedicated for him (even though time sharing is happening) in same OS.

This type of virtualisation is called process virtualisation (on a process level)

In VM



request level parallelism

multiple OS needs same H/W.

simi OS has same priority & if all try to access H/w at same time

chaos happen. So we need some mechanism to manage this.
(no time so lets finish here! :))

Other considerations in VM managem (demand paging)

(1) Pre-paging : Whenever a process starts (for 1st/after times) execution, there is a chance of more # page faults initially

so to avoid this, if these pages are already loaded into mm before a page fault occurs, by paging h/w this is called pre paging

if P_1 need 1 3 7 10 & 13, its loaded even before P_1 generates page fault after its scheduled each time..

(2) Page_size: Average intral fragmentation
= ½ page size

⇒ ∴ ↑ page size ↑, int frag ↑.
if ↓, " " ↓

⇒ Also if page size ↑, I/O time ↑ & vice versa.

⇒ # page fault rate ↑ with ↓ with page size

But can we ↑ page size at h/w level or OS_level?

A: OS cannot ↑ page size, it depends on h/w resources available

(modern system provide 4-8kB pages as well as > 4MB page size)

∴ can't modify page size much. But affect AMA time

→ (3) TLB reach

page #	frame #

if TLB can store 16 page info.

& size of page = 4096.

then TLB reach = 16 × 4096.

If locality of a process is < TLB reach, all the memory access is through TLB & AMA ↓

But if process locality > TLB reach memory access may cause TLB miss & hence AMA ↑

∴ how to ↑ TLB reach?

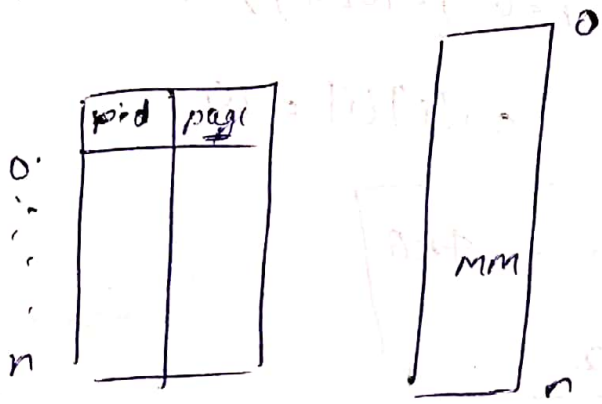
(a) increase size of TLB (# entries in TLB)

then TLB hit ↑, & AMA ↓

(b) ↑ size of page: If our system allow 1MB or 4MB pages, then its difficult to handle TLB in hardware level (amnt of info to be stored ↑)

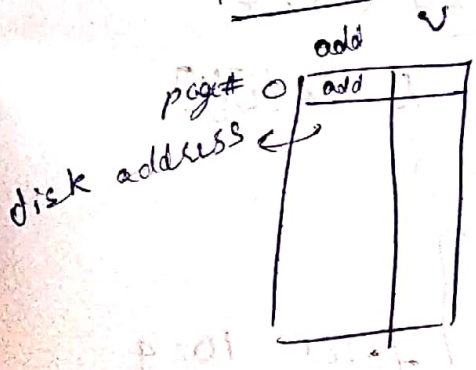
It needs to be handled by s/w.
But this ↑ AMA.

④ Inverted page table:



usually if [pid | page#], we check all entries in inv. page table. But how can we say if page fault occurred, were that page stored in 2° mem.

To have this, we have an external page table per process



if [pid | page] is not in inv. page table, it looks in external p.t & if the page is invalid, the address of the area where that page is stored in 2° mem is obtained. Then its load to MM & external & inv. page table is updated.

(5) Program structure

```
float A[1024][1024]

for i=0; i<1024; i++
    for j=0; j<1024; j++
        a[j][i] = 10;
```

If page size = 4kB

page = 512.

now, since matrix is stored in row major

A[0][0] ⟶ 1 page fault

A[1][0] ⟶ page fault as A[1][0] is after 4×1024 bytes from A[0][0]

A[2][0] ⟶ 1 (float = 4kB.)

A[512][0] ⟶ page fault & page replacement as mem has only 512 frames.

∴ no. of faults = 1024 × 1024

Now a[j][i] is changed to A[i][j]

A[0][0] ⟶ 1

A[0][1] ⟶ no fault

A[1][0] ⟶ 1

A[1][1] ⟶ no fault

A[1023][0] ⟶ 1

[1] ⟶ no fault

} total 1024 faults.