VIRTUAL MEMORY

→ 32-bit processor : Range : $0 - (2^{32}-1)$ address

→ 32-bit machine ⇒ word size ⇒ 4 Bytes

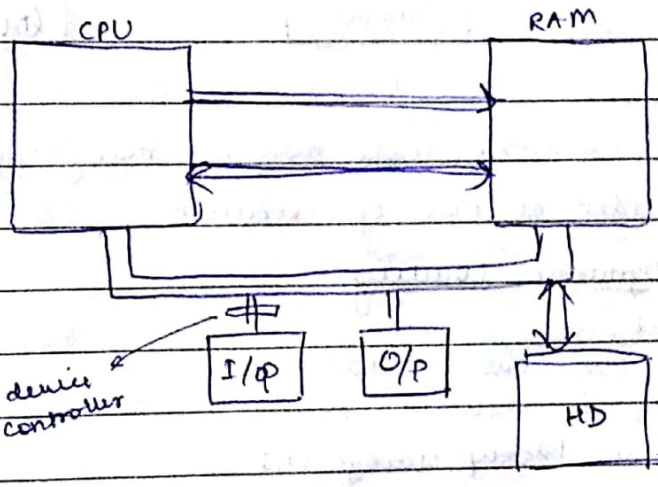
→ 64-bit machine ⇒ word size ⇒ 8 Bytes

→ 16-bit machine → ⇒ 2 Bytes

Byte Addressable ↑ because :

↳ safety space

↳ most of data types are 2 byte, 1 byte ...
more accessible

→ 

Out of Jobpool (I/p queue)

↓ Longterm scheduler (program in OS)

picks program & put in RAM.

scheduler Healthy mix of I/O bound & CPU bound process

↓

which takes no time in computation

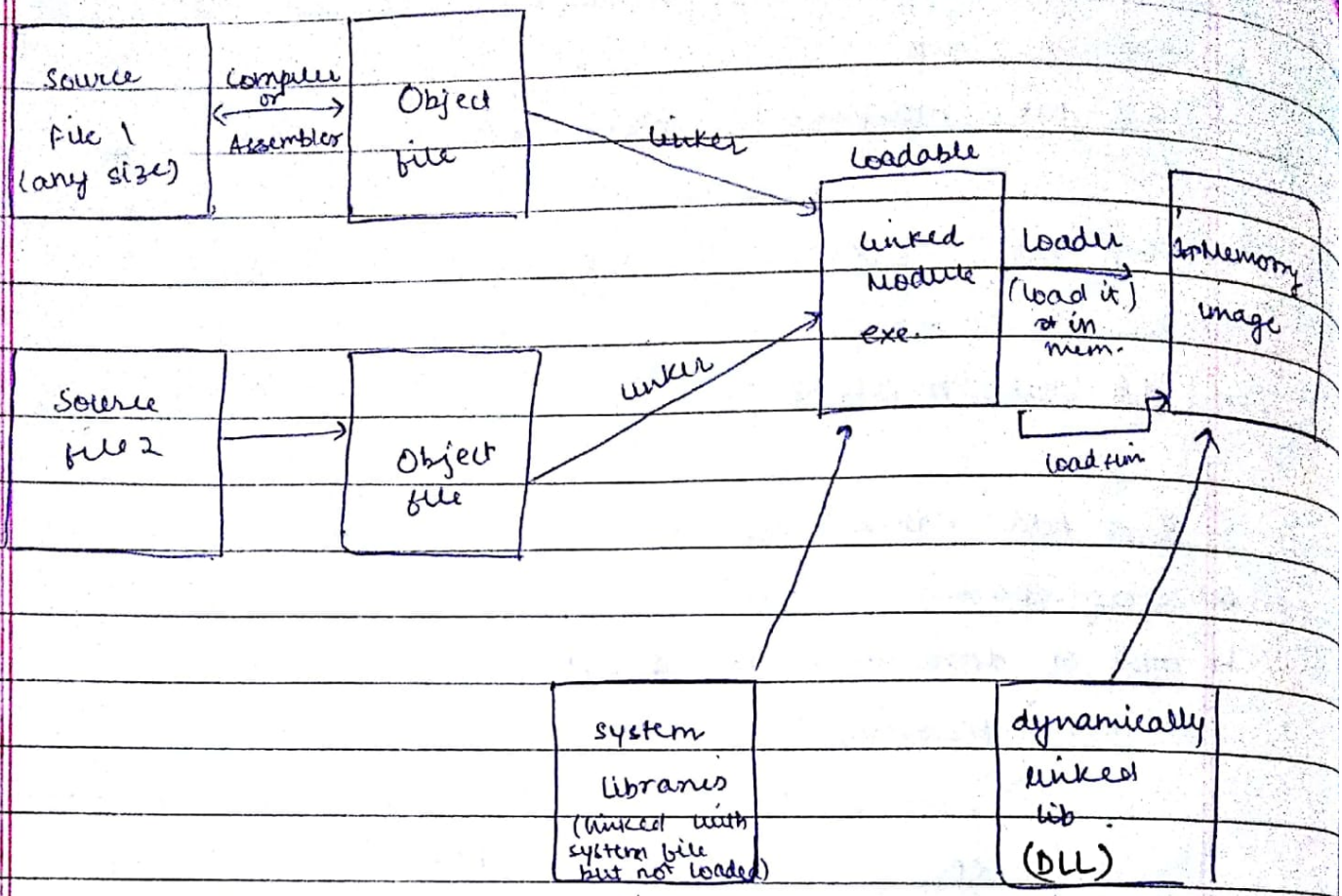
very interactive

14-03-18

→ If all i/o bound : CPU is idle, prog. all in queue for I/O

→ If all CPU bound : all in ready queue → WT ↑ TT ↑ waiting for scheduler

Teacher's Signature



→ When loading module in memory, there may be many routines which aren't needed at time of execution.

So, we use Dynamic Loading

↓

Routines / lib. will be loaded only if required. Otherwise, kept on backup storage (HD)

→ Dynamically linked lib. : small piece of code → stub (tells in module) If this lib. is to be called, then how will it be called.

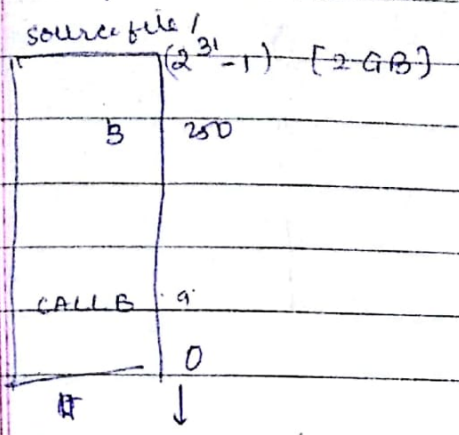
- ~~loaded~~ linked with prog. only when needed. Not initially linked.

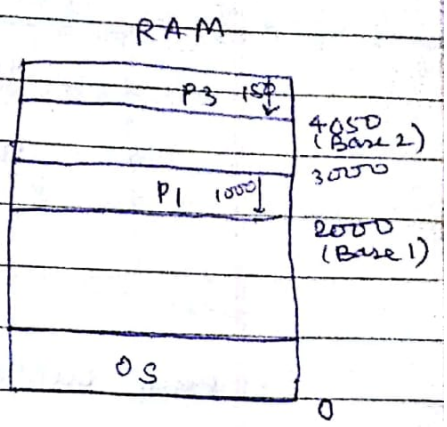
→ ILC : Increment by length of last instⁿ.

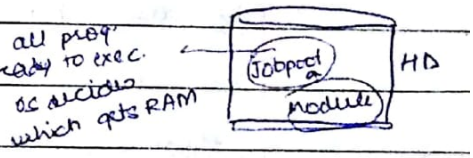
Teacher's Signature

DATE : / /
PAGE NO. :

2 - pass Assembler







Where actually int in RAM : Physical Add.

→ Add. binding : Actually putting phy. add. in process. Maybe at:

① Compile time : We know where prog. will be loaded in RAM at compile time only

Relocation const. : 2000 (let say prog. is loaded at add. 2000)

⇓

shift everything by 2000

add. binding was taken place at compile time.

Usually, this doesn't happen

② Load Time : If OS gives info. when it is being loaded. So, I know relocation const.

③ Execution Time (Run Time) : know only at runtime.

→ Talking about RAM : (above diagram)

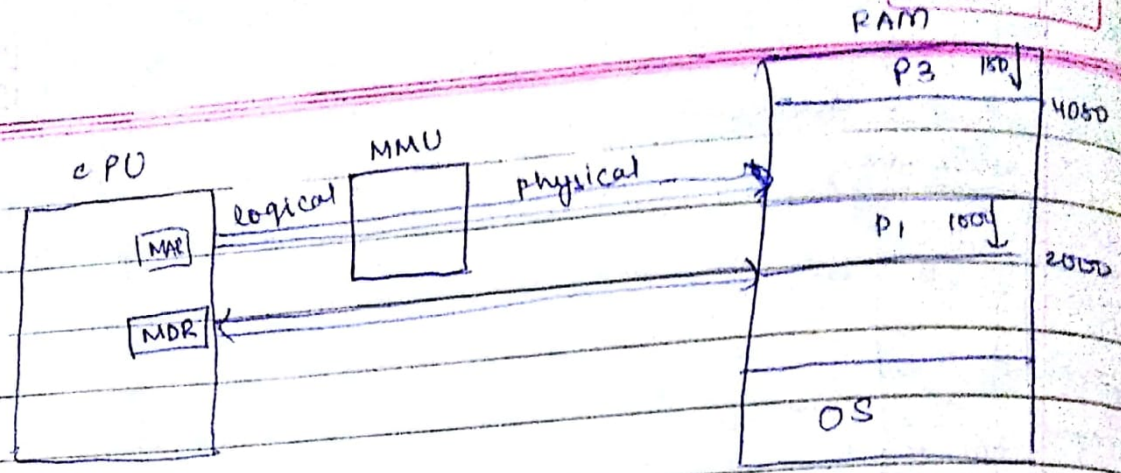
P1 & P3 loaded.

OS decides where they would be put

For P1 ⇒ 1000 : limit

P2 ⇒ 150 Bytes : limit

Teacher's Signature



- Fetch, Decode & Execute

→ (2000 is given)

- If add. binding took place at compile time → P1 will be loaded at 2000 every time because put at compile time : can't be changed. same is problem with load time

Usually, we don't prefer it.

If 2000 isn't free → P1 can't be put. (Hardbounded add.)

OS don't prefers this

→ add. binding takes place at run time.

P1 total size : 2 GB

when CPU generates add., it generates logical add.

only (int in instⁿ)

P1 is generating all 2GB add.

→ MMU : logical $\xrightarrow{\text{mapping}}$ physical

Suppose, prog. is malicious ⇒ want to access other user's prog.

In P1, write JUMP 4000

⇓

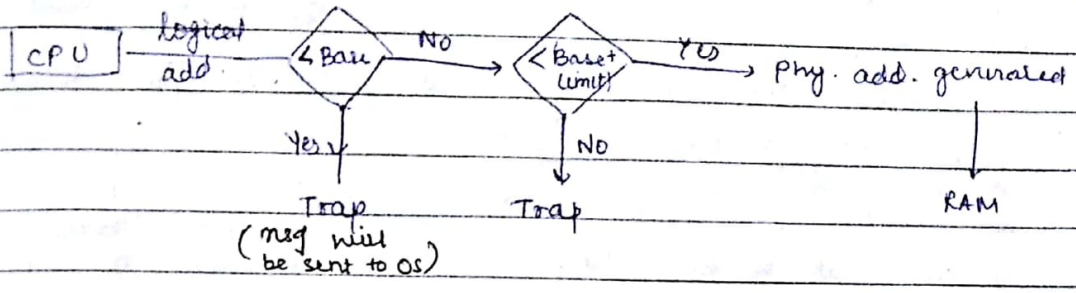
Through mapping, can go to there. But this isn't allowed

- No process can access other process's add. space (security issue)

* logical add. > Base (should be)

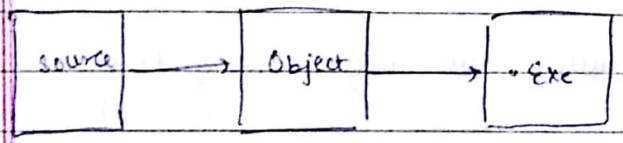
DATE: / /
PAGE NO.:

→ There should be checks at MMU to restrict this.
(2000 has been added to all inst^{n}s already)

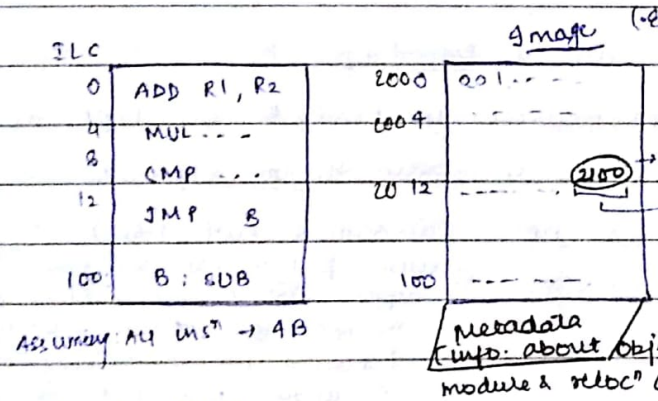


→ Base & limit are stored in some reg. for each process

16/3/18



:→ Ultimately going to be stored in HD



→ At Compile time

→ suppose, at compile time only, I know this prog. will be loaded at 2000 only. → I can already shift all add. by 2000 at time of compilation only.

→ Everything is hard coded at compile time only.

→ There's no flexibility for OS now. It always have to be loaded at same place in RAM.

Teacher's Signature

PAGE NO.:

Add. Binding → 2100 is already in program loaded

→ At load time : OS adds 2000 at time of loading
⇒ again hardcoded in Image

when CPU sends add, this is time it checks if base ≤ add < base + limit

→ At run time : Even image has JUMP 100, prog. loaded in RAM still has JUMP 100

Now, at exec. time, 100 will be sent from CPU.
It'll check it is within limit or not

Yes ⇒ MMU will take care
↓
has reloc^n const. , add it to 100 & thus, phy. add. will be generated.

→ gives us flexibility of putting the program anywhere in RAM

→ ~~Suppose 5 processes are in Top Jobpool~~

→ Backing Store : All processes in Ready Q are kept here
small, fast HD : can be in main HD or separate
OS may use it or not. (associated with RAM)
↳ either put in RAM and give CPU | Eg! suppose high pr. process come in N/W & no space is available we take 1 Process & put in Backing Store. Many OS avoid using it due to Overhead

A prog. can be
→ given max. memory
→ brought back to Backing Stor
→ swapped. ⇒ 1.) lot of Overhead (Time ↑)
2) If I/O interac^n : I/O buffer used
↓
~~somewhere in RAM~~
suppose we swap P1 with P2. ⇒ need to make sure there is no i/o performed.
→ If swapping : better to use OS memory for i/o buffer. ⇒ P1 ko life buffer abhi bhi available hoga. P2 can access it.

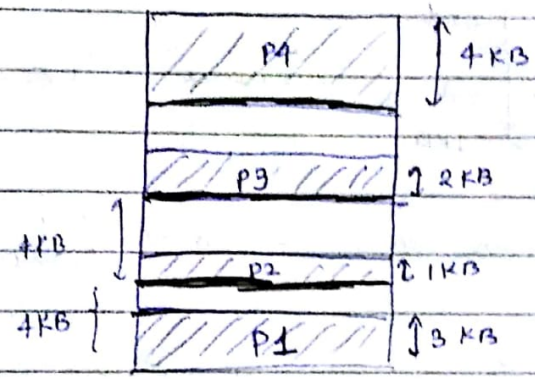
Teacher's Signature

→ Backing store is same as swapping area

→ Jobpool: New Processes which are ready to execute some of them are picked by long term scheduler & put in Ready Q

→ 2 ways of loading RAP programs in RAM: (Contiguous Mem Allocⁿ)
(whole process needs to be put together)

1) Multiple Partition scheme: 4 KB - partions :-



OS mem (ut)

→ Each time bring a process, give it a partⁿ

→ Now, another process needs to be brought in of size 6KB
(If came earlier: can give 2 partitions to it)
Now, can't bring it because no contiguous mem. space are available.

→ This leads to Internal fragmentation (Wastage of mem)
(Each block is internally fragmented to holes)

→ Even though mem. space is available, process can't be brought in RAM. (can't use it)

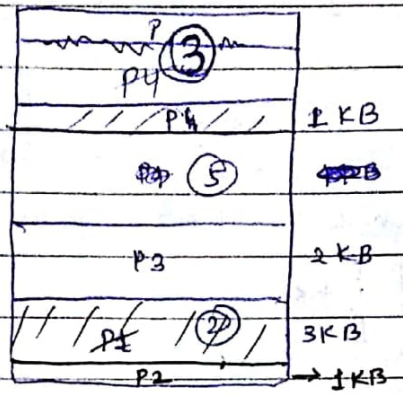
→ Wastage of Memory

→ couldn't do anything to avoid holes

Teacher's Signature

2.) Variable Partition Scheme :

not defining block as such.



whatever P1 needs, give it.

suppose P1 had to go for some I/O, ⇒ gone out of Ready Q

↳ swapped out P1

suppose P2 came ⇒ hole

P4 swapped ⇒ hole

holes created outside add. space of process

→ No block system here

→ External fragmentation is taking place

(external to process : whatever fragmentation is taking place)

↳ again can't allocate 10KB program mem.

→ If N allocated blocks ⇒ 50% rule

↓

0.5 N blocks are fragmented (not used)

→ can do : Compaction

move all processes towards 1 side & holes to another side

Teacher's Signature

Or keeps track of holes & processes

DATE : / /
PAGE NO. :

Suppose P6 comes → 2 KB size

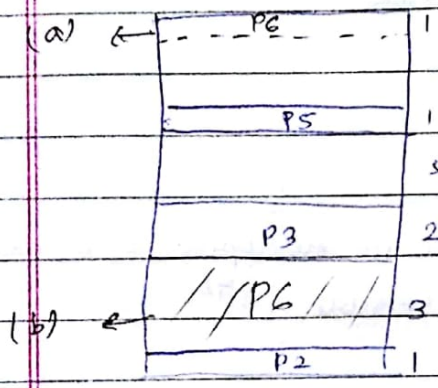
algorithms to put it :

a) First fit : scan list (from top / bottom)

1st hole where it can be put : put it over there

adv. :→ don't have to scan full list

disadv. → creating lots of small hole

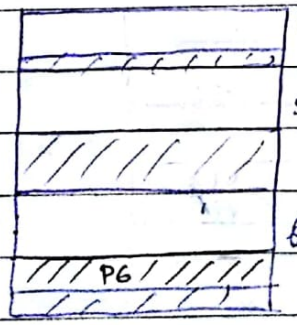


b) Best fit :

disadv. :- scan full list

adv. :- find best hole (minimizing no of holes)

c) Worst fit : giving largest hole possible



adv. : holes created are of larger size

d) In some cases, a) works better, b) works better in some

c) usually never works better.

Teacher's Signature

→ Again Compaction can be done ⇒ add. are changing again

⇓

Can do Runtime
Add. Binding Only

→ Compaction also has its own Overheads.

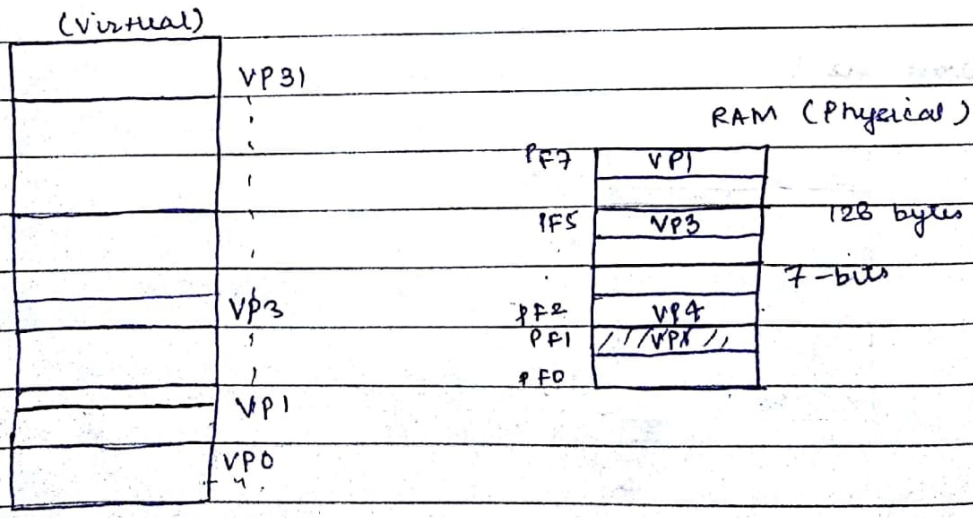
19/3/18

→ External fragmentation can take place in swapping area also doing compaction over there is not possible. (HD)

Good approach for memory management : Paging

Eg logical addres (byte addressable) : 512 bytes ⇒ 9 bit add. is needed.

RAM : 128 bytes ⇒ 7 bits needed for phy. add.



→ 32 bit processor ⇒ CPU can have $2^{32}-1$ addresses (logical)

→ OS knows about phy. add. & ~~bits~~ logical add.

Teacher's Signature

Frame / Page frame

Suppose, partition size : 16 bytes each

∴ No. of frames in RAM : $\frac{128}{16} = 8$

No. of virtual pages in V.M. = $\frac{512}{16} = 32$

(16 bytes each)

Logical add : 20 → logical add. : 9 bits

0 0 0 0 1 0 1 0 0

I can either say logical add. 20

or I can say VP 1 offset 4 (20 - 16)

32 V.P. ⇒ 5 bits needed to specify V.P.

16 bytes each ⇒ 4 bits needed to give offset

9

0 0 0 0 1 0 1 0 0

Virtual Page No. | Offset

Add. 56 : VP : 3 Offset : 8

32

32 + 16 + 8

0 0 0 1 1 1 0 0 0

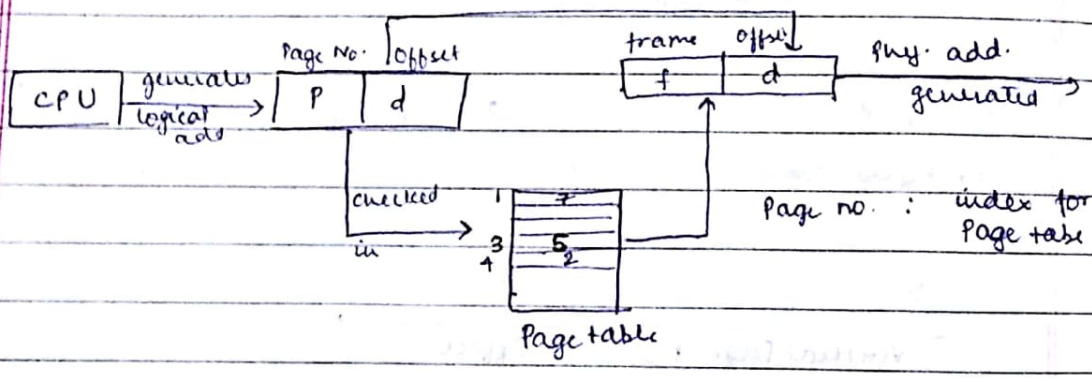
→ Many time, much part of code isn't needed. So, OS only loads that part of code in RAM which is needed.

→ Suppose it loads VP3 in PF5

Now, PC is increment & PC has add. of VP4. It 'll pick VP4 & put it in PF2 (let say), When VP4 was running, there was a func^n call which was in VP1. So, now, VP1 will be picked & put in PF7. Return To VP4. Don't know add. of VP4. ⇒ OS has to keep a

Teacher's Signature

1) frame table which has list of all available frames in RAM.
2) for each process, it needs to maintain a page table.



VP3 → check index 3 of Page table
↳ in PF3 → 5 is stored at 3rd index of Page table

Phy. add : 1 0 1 | 1 0 0 0 → Offset will remain the same
(5 → Page Frame; Offset)

Suppose add : 56 → VP : 3 Offset : 8
VP 3 maps to frame 5
10 | 1000
64 + 16 + 8
64 + 24 = 88 → phy. add. generated
→ 16 × 5 + 8 = 88

Teacher's Signature

Typical frame size : 4KB

→ There could be multiple processes in RAM.

→ If RAM is full ⇒ swapping will take place

→ At the time of context switching ⇒ state of process is stored in PCB.

similarly, here, PCB can store state here

How to decide Page size ?

→ if too less : No. of Page ↑
No. of Virtual Page ↑
⇒ Page table ↑ ⇒ lot of mem. needed

→ if too large :
No. of Page ↓
No. of V.P. ↓
Swapping will be ↑ (replacement ↑ hoga)

↳ if prog. size : 500 bytes
last page me internal fragmentation of 12 bytes lost due to

On an avg., there is a chance of 0.5 internal fragmentation on last page (jo wo use kar rha h, uska last page)

⇒ here, page size too large ⇒ avg. internal fragment" ↑↑.

→ Assuming P1 & P2 in RAM :
When P1 is going to run, all registers will store its state
when context switching : store & save registers
This is possible only when page table ↓.

If PT ↑↑ :→ RAM me har process ki page table store hogi.

→ Suppose P1 ki page table at add. 100
P2 add. 150

Teacher's Signature

In CPU (or MMU), I will have a PTBR
Page Table Base Register
↓ stores physical add.

If P1 is running → PTBR → 100 → mapping will occur with Page table
P2 → PTBR → 150
(points to add. 150)

21/3/18

size (Phy mem) > size (VM)

→ Page size = 1KB
No. of bits req. for Offset = 10 bits = d

L Add : has 4 pages → 12 bits req.
Phy Mem : " 8 " → 13 bits

→ No. of entries in Page Table = 4 (4 pages in VM)

→ Size of each ~~table~~ entry in ~~table~~ page table
↓
Frame no.
⇓
3 bits (just getting frame no.)

It can access : $2^3 \times 2^{10}$ addresses (↑ offset)

→ If Page Table is too small : can use register for each page. On context switch, values of registers are modified!
→ If Page Table is large ⇒ keep it in RAM only

Teacher's Signature

Ki page table

Eg:- P1 ~~page table~~ : 100 - 180

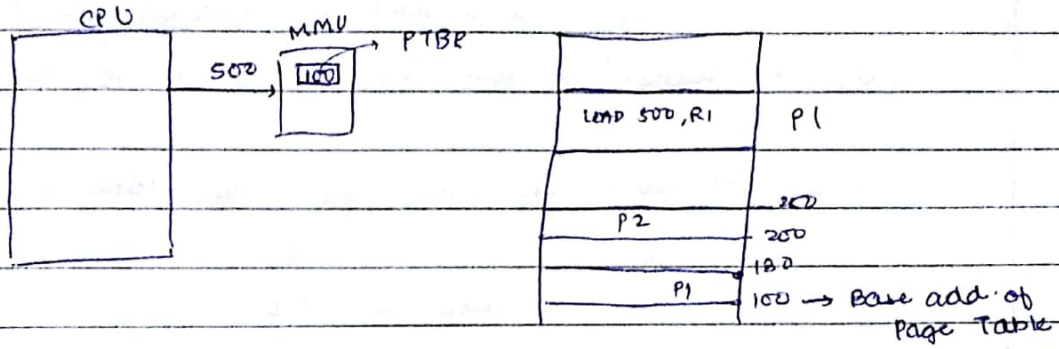
P2 " " " : 200 - 250

↳ in P1

→ LOAD 500, R1 : direct add. mode

↳ whatever is in add. 500, put it into R1 → Mem. access

→ 500 : LAdd. unit



500 is put into add. bus

MMU has PTBR. Since P1 is running, pointer for P1's Page Table is in PTBR

500 : | Page | d | : check Page table in which frame this page is in.

⇓

go to that frame, add d, access loc^n & bring data to CPU.

→ If 500 waala page isn't brought into RAM, it'll be brought from backing store & page table will be updated.

→ If a memory access takes 100 ns

→ first access memory to access page table (100 ns) & get frame & then compute f + d, access that add. in RAM (100 ns)

→ 2 memory accesses required for 1 inst^n.

Teacher's Signature

- We don't want that.

(doing that because Page Table is in memory)

To avoid this : Bring Page Table to Cache

However, won't use Data & Inst" cache here.

We have : Translation Lookaside Buffer (TLB)

↓

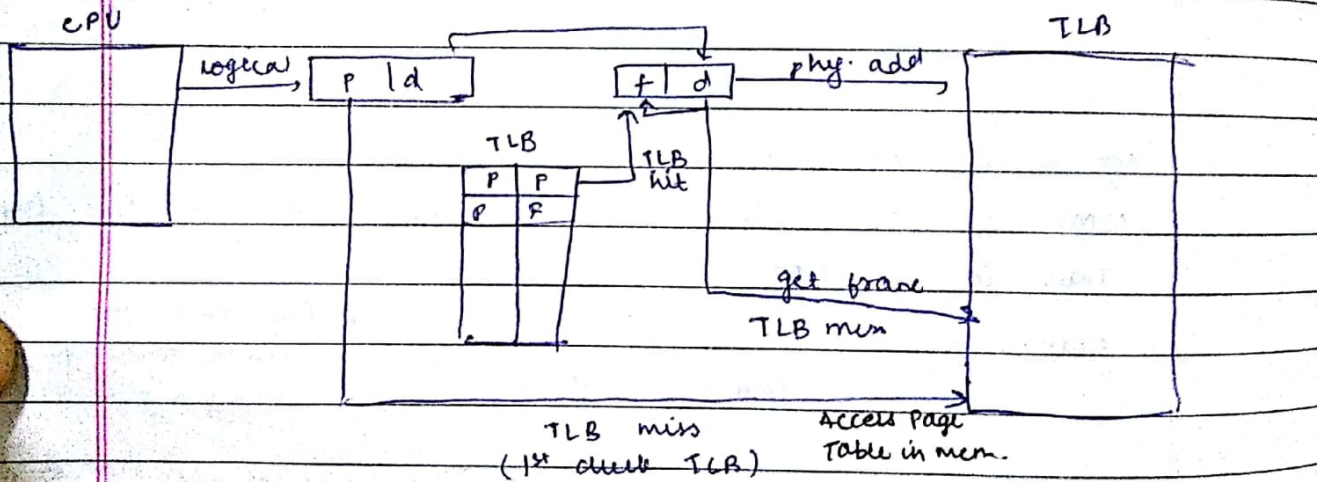
actually a cache

(could be in MMU or separate memory)

related to transl" of add. from logical to physical

It's small → can't ~~use~~ bring full Page Table to TLB.

1st time add. comes → check in TLB



If P_1 was running, 1st time there'll a miss. Once it's accessed, I'll bring some entries in TLB.

↓

has 2 values (atleast)

[Page No. & Frame No.]

* In original Page Table : It was complete : just needed frame no. (indexed into Page No.)

↓

There is no page no. here. Only has frame no. → If put into TLB → have to store page no. also.

Teacher's Signature

(found entry in TLB

TLB hit

→ Suppose, we check TLB & get frame no. → add (f+d) & access that add

→ If TLB miss : have to access Page Table in memory, take frame no. & add d
↳ put in TLB also (either only this entry or a group of entries) (parallelly)

→ If hit rate = 80% → 80% of time, entry is found in TLB

Mem access : 100 ns

TLB " : 20 ns.

hit : 100 + 20 = 120

miss : 120 + 100 = 220
(First checks in TLB → Low)

effective mem. access time

= 0.8 (120) + 0.2 (220)

= 96 + 44 = 140

23/3/18

→ Can have cache (L1) on the processor chip for faster access & L2 cache b/w processor & RAM.

① → In cache (TLB), apart from tag, we've valid / invalid bit is also present

		valid/invalid bit	Read/Write permission
0		0	0
1	f1	0	
2			
3	f	1	
4	15	1	
5	10	1	
6			
7	f2	0	

Initially, when not brought into RAM, all bits = 0

Page Table in RAM

Teacher's Signature

Suppose Page 4 was brought & put in frame 15

* valid bit = 1

At execution time:

Check Page no. 4 ⇒ also check valid bit

if = 0 ⇒ invalid ⇒ The page is not brought in RAM

or I'm trying to access an add. which isn't in my add. space

Suppose

If → In my logical add. space ⇒ bring page to RAM & modify bit = 1

If → Not in my " ⇒ Trap to OS (error cond" will be generated)

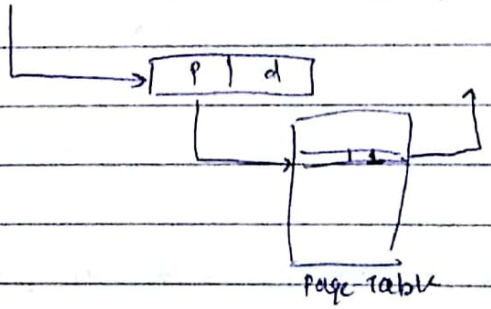
② Next is Read / Write / Execute permission

also stored in Page table

log. add.

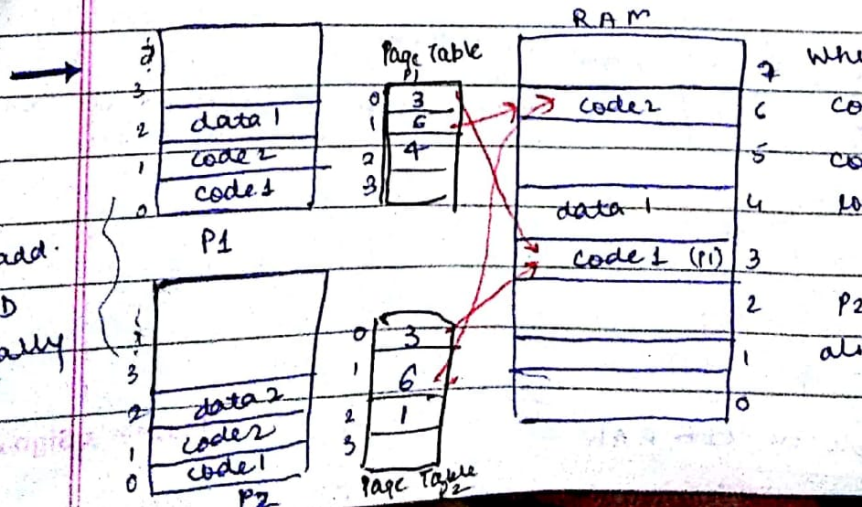
STORE R1, 2500 = store value of R1 in 2500 mem. add.

WRITE Opⁿ



→ Write bit = 0 ⇒ Only read & execute permissions

If want to write (STORE) → again an error.



Teacher's Signature

Editor → type of app^n's are shared

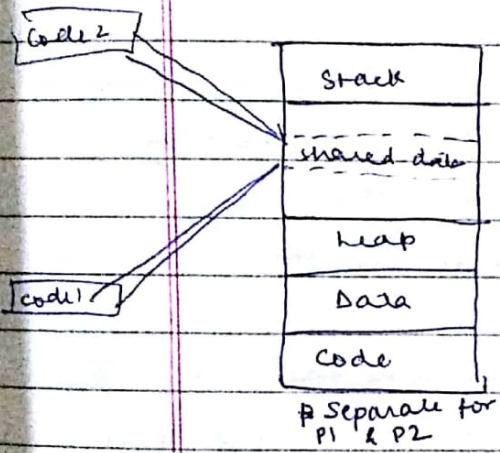
DATE: / /
PAGE NO.:

Now, P2 also needs to be loaded. But OS sees code1 & code2 ↓ shared. are already in RAM.

→ Map code 1 → 3 & code 2 → 6. Only data part'll be diff

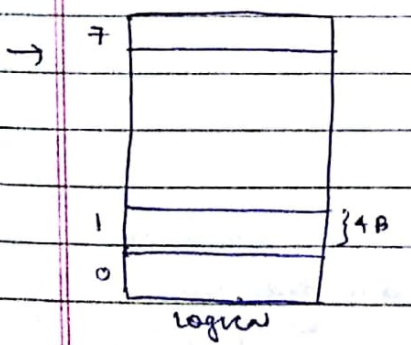
3 & 6 : Shared Pages
↓
Shared Memory

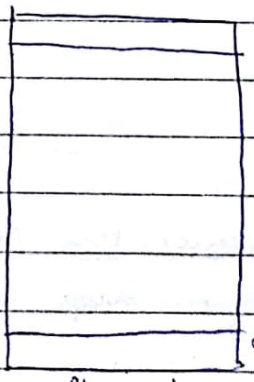
Pure Pages : can't be changed
⇓
Only Read mode (no write mode)



Separate for P1 & P2

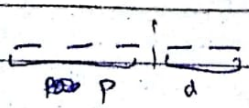
Phy. Mem. will have just 1 copy of it.
Log. ~~add. mem~~ add. will be diff for both

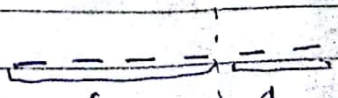




Logical add. : 8 × 4
= 32
⇓
5 bits needed for add

⇓
64 Bytes
⇓
6 bits ∞ for add.

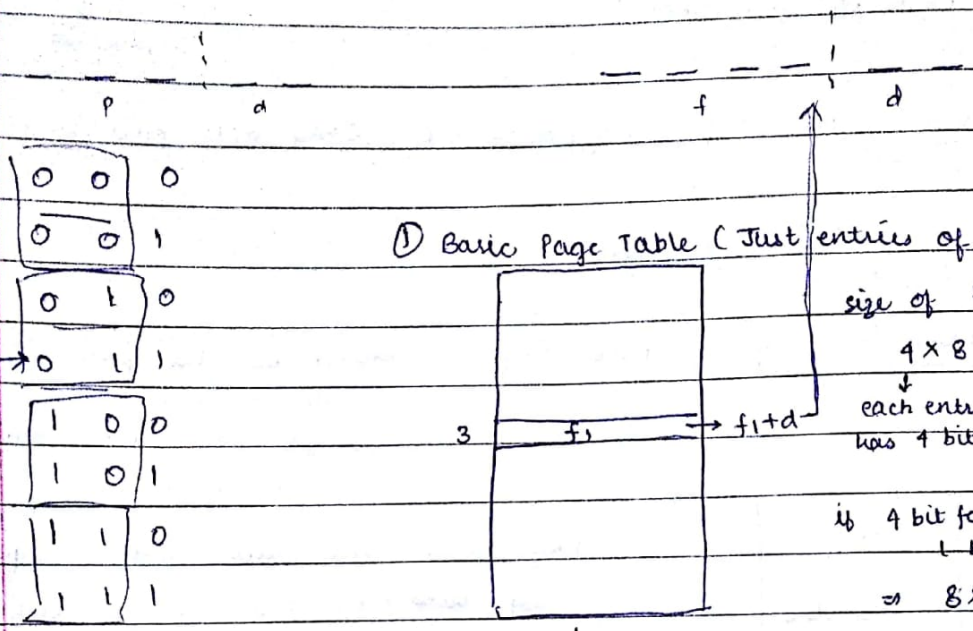




Teacher's Signature

DATE : / /
PAGE NO. :

Different structures of Page Table



Basic Page Table : Linear table (starting se end tak)

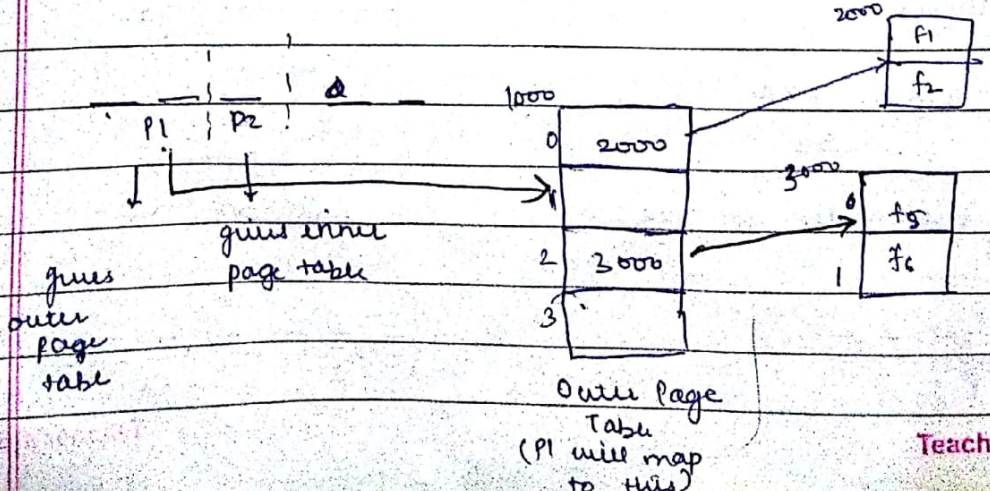
→ suppose I have 1 million pages → Page Table has 1 million pages
→ contiguous space needed in RAM : very large space

↓

Break Page Table in RAM

② Hierarchical Page Table

Break page part (here : 3 bits) into two parts



Teacher's Signature

DATE: / /
PAGE NO.:

Now, PTBR will store base add. of outer Page Table (1000)

If 000 : 0 of Outer will store 2000

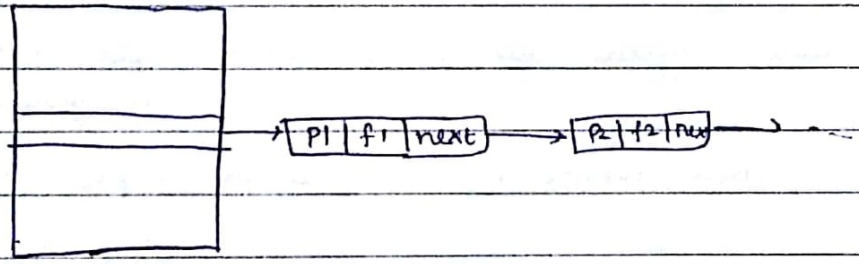
→ have put 4 entries, 2 entries, 2, 2, 2 somewhere else & have ptr to ptr.

→ If 101 : 2 of Outer ⇒ f4 → f4 + d

Add. is still larger ⇒ can make 1 more layer

③ Hashed Page Table

→ Can have Page table in form of linked list in case of Hash func^n (using page no.)

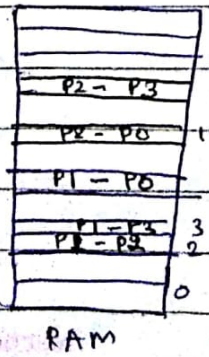


④ Inverted Page Table

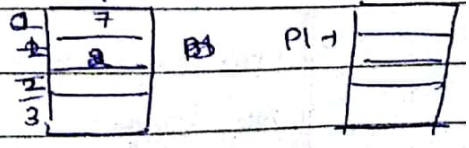
Right now, we've 1 Page Table per process.

Now, we'll have 1 Page Table for complete memory.

Mapping will be from Frame → Page



Page table for P1



Teacher's Signature

→ Earlier, had different page table for different processes.
→ Rather than keeping track of pages, we will keep track of frames
→ Other way : have 1 common table
↓
will have all frames in RAM

Mapping : Frames → Pages

	pid	page	
0			
1	pid_1	3	
2	pid_1	1	
3	pid_1	2	
			7
	pid_1	0	
	pid_2	0	10

→ Inverted Page Table

Now, when logical add. is generated : pid, page & offset will be generated
↓
check whole table, find pid & page ⇒ index will give me frame

frame + Offset
↓
gives physical add.

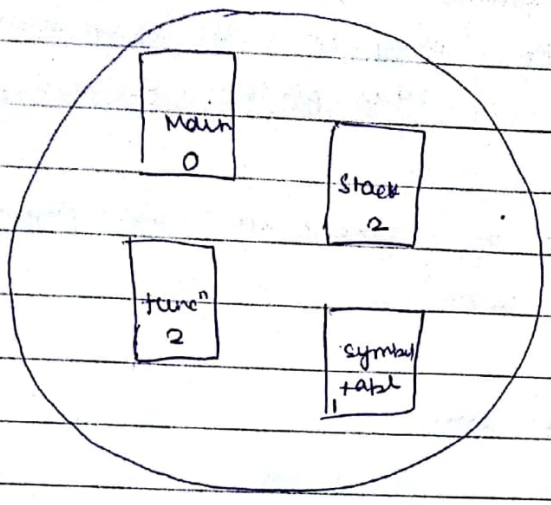
* This can't be used in case of shared pages. (1 index can't have 2 entries)
→ In this case, if you've shared code, you'll have to keep multiple copies of the code.

→ Searching will become complex if no. of frames is too large

Teacher's Signature

DATE: / /
PAGE NO.:

User view : (Synonym to User view of a particular ~~vvvv~~ appⁿ)



→ Segmentation

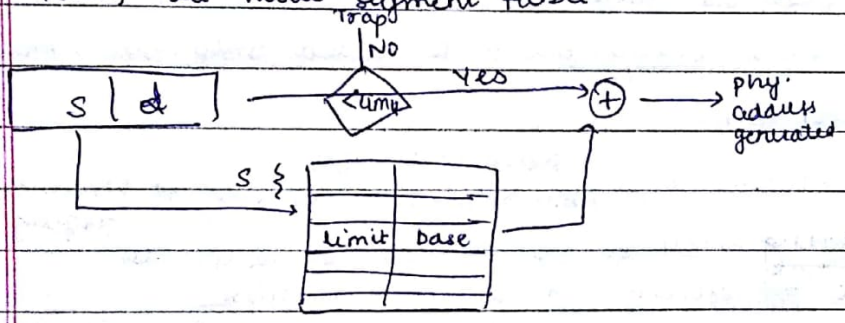
Logical add. generated is of form

s	d

→ it's not linear form of addresses (like pages) ↳ fixed size

→ very similar to paging : all segments can be of diff. size.

Here, we have segment table



Segment Table

Each segment is going to be loaded somewhere in memory.

→ Have to keep track of base & limit of that segment.

if d > limit ↓ trap to OS ↑ d will be checked by limit

phy. add = base add. + d

→ If I used paging ⇒ all segments will be in pages.

(½ main in some other frame, ½ main somewhere else)

Here, all inst's in main are available together

Teacher's Signature

→ Not have to jump again & again from 1 frame to another.

Disadvantage: If segments are too large : may have to swap with some other process (Replacement Algo)

→ Many systems use comb" of Segmentation and Paging.
(Divide segments into pages)

Advantage: more synonym of user view

→ Each approach will have its own advantages and disadvantages.

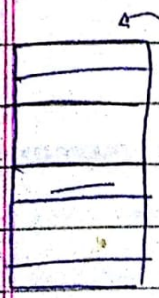
⇒ Suppose 25 Pages are there in ~~Your~~ VM. → P1

Many times, some routines are to be called only when there is some exception.

Should be brought in RAM only when required ⇒ Demand Paging

Demand Paging → all pages available in swap area in Backing store.
P1 is about to execute. → (in swap area → in Ready Queue)
↓
processes selected to be in readyQ

Jobpool : who want to go to RAM & get CPU



156 pe → 1st instn : checked Page table : Valid bit = 0 → Page not in RAM
& → It is needed now.
↓
No frame entry available

↓

Page fault has occured : when required mem. reference is not available in RAM.

Teacher's Signature

→ Demand Paging : may cause Page fault

DATE : / /
PAGE NO. :

Now, what will happen ?

→ Disk has to be accessed

→ Page has to be brought into RAM.

Page Fault : give msg to OS.

→ OS will give request for Disk access. (device access : Process can't use it directly)

⇓

this Process goes in device queue for HD ⇒ waiting for some I/O.

→ while this process is not running, CPU will be assigned to some other process

→ Meanwhile, page will be located in disk & trans

→ look for free frame in RAM & bring page.

→ Update Page Table for that process.

→ If no free frame → some frame has been taken out ⇓ update its page table also.

→ ~~PT~~ Table maintained by OS (of all processes info.) also needs to be updated

→ Process 1 is again in Ready Q with updated PT

→ TLB ki PT also needs to be updated.

→ ~~May abs~~ CPU scheduling also occurring

→ May be some replacement algorithm is also working.

→ Agar ~~koi~~ iss process ka No need of demand paging
↳ directly pages ko RAM me load karde.

→ Global PT which OS is maintaining is also updated.

→ has to decide which page has to be replaced if RAM is full.

Only In Demand Paging :

* All Pages are in backing store : bring required page to RAM.

→ No. of Page Fault v/s degree of multiprogramming :
This is calculated

Teacher's Signature

accessing RAM : 100 ns

cache 10-15 ns

HD : ms

DATE: / /
PAGE NO.:

→ If P1 → 25 pages & RAM size = 50 pages

Degree of multiprogramming : 2

28/3/18

→ Suppose Page fault rate = p

Memory access time = ma

How many times page will be available in RAM? 1-p

Eff. Mem. access Time = (1-p) ma + p x page fault time

(1-p) → ns

page fault time → accessing the HD. (ms)

If page fault rate ↑ ⇒ Eff. Mem. access Time ↑↑.

⇒ have to minimize p.

1) Can ↑ no. of frames : Page size ↓↓

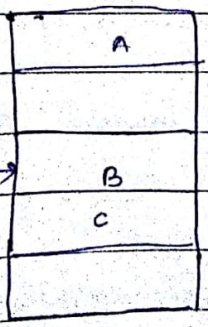
⇒ Size of Page Table ↑

⇒ Internal fragmentation

↑ Can cause lot of Page faults.

↳ If a frame alloc^n algo.

↳ If all frames are full → need frame replacement policy



B → goes back into Backing store

⇒ whole page is to be written into Backing store

↳ will take some time.

(Disc access time + writing back in HD)

Teacher's Signature

To minimise this access time

⇒ can use Dirty bit / Modified bit :

with every page : have " , any write opⁿ

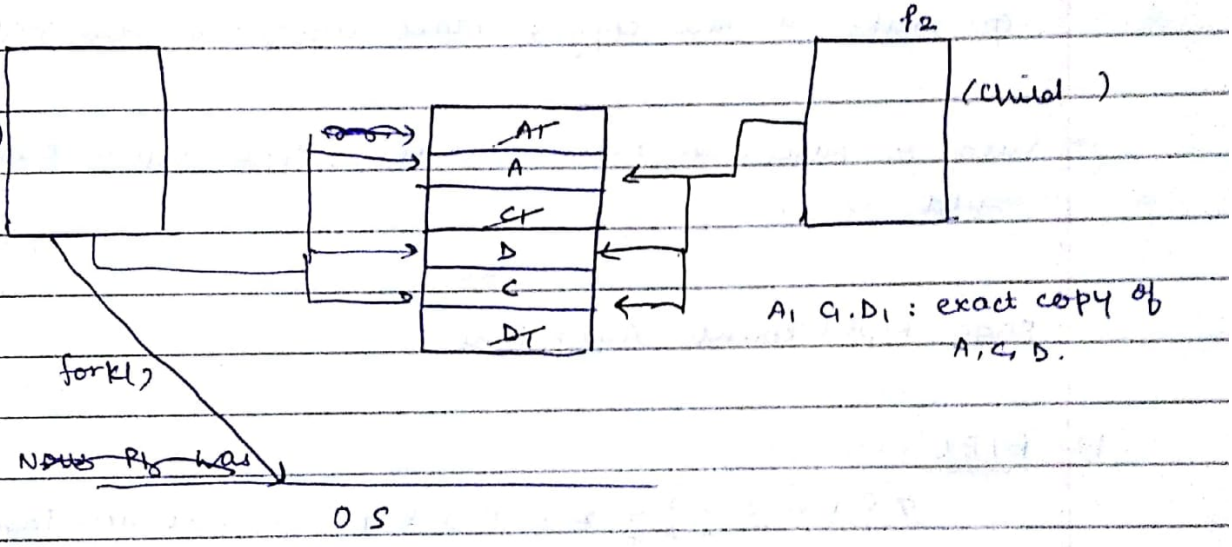
If I've made any change in page in RAM ⇒ dirty bit = 1.

when replacing, check bit

if bit = 1 ⇒ need to write into backing store

bit = 0 ⇒ Original copy is already in HD.

simply overwrite it by bringing new page.



Now P1 has given fork() system call.

↳ child process is created with exact replica.

→ when new code is written in P2 : new pages 'll come in RAM at A1, B1, C1.

→ If not ⇒ we're wasting frames putting 2 exactly same pages

⇓

It asks child also to share the same pages.

Teacher's Signature

P_1 & P_2 both are running at same time.

→ when P_2 wants to change a page, then only OS will make a copy of that page & P_2 will make change in that so page of parent process doesn't change.

This technique is called: **Copy-on-Write** (Imp)

child will get its own copy of pages only when it needs to make a new copy & make changes in that only.

→ Even if parent is making change, copy will be created for child.

Page Replacement Algorithms:

1) **FIFO**

7 0 1 2 0 3 0 4 2 3 0 3 2 1 → Virtual Page no.

Only 3 frames in RAM. Count page faults

	7	F (Fault)	
	7, 0	F	7 0 1
	7, 0, 1	F	
7 ✗	2, 0, 1	F	
	2, 0, 1	✗	0, 1, 3 F
0 ✗	2, 3, 1	F	
	2, 3, 0	F	
	4, 3, 0	F	
	4, 2, 0	F	
	4, 2, 3	F	
	0, 2, 3	F	
	0, 2, 3	✗	
	0, 2, 3	✗	

3 frames → 11 F

1 frame → 14 F

Teacher's Signature

If ↑ frame no. ⇒ no. of F ↓

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

3 frames : F F F F F F F X X F F ✓ ⇒ 9 F

4 frames : F F F F X X F F F F F F ⇒ 10 F

↳ Seen after ↑ frame no. ⇒ no. of F ↑ : Belody's Anomaly.

2.) Optimal Page Replacement Algorithm :

Replace that page which will not be used for longest period of time ⇒ looking into future.

→ same eg :

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

7	F
7, 0	F
7, 0, 1	F
2, 0, 1	F
2, 0, 1	X
2, 0, 3	F
2, 0, 3	X
2, 4, 3	F
2, 4, 3	X
2, 4, 3	X
2, 0, 3	F
	X
	X
	F

→ 7 isn't used for longest period of time

→ Replace 1

3 frames : 8 F

⇒ can't look into future. ⇒ need to know complete reference string.

Teacher's Signature

int a[100]

a[0] - 1

a[99]

a[50][2]

DATE: / /

PAGE NO.:

→ this is just used for comparison with other algo (reference string)

3.) LRU :

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

7	F
7, 0	F
7, 0, 1	F
2, 0, 1	F
2, 0, 1	✗
2, 0, 3	F
2, 0, 3	✗
4, 0, 3	F
4, 0, 2	F
4, 3, 2	F
0, 3, 2	F
0, 3, 2	✗
0, 3, 2	✗
0, 3, 1	F

7 0 2 2 2

3 frames : 10 F

→ most commonly used → in PT

→ Implement : 1.) Put time of use for each page. When full → min(time of use) → replace that page.

(use counter)

2) use stack : Bring a page : push (doubly linked list)

ToS : most recently used.

already had 0, again got 0 : remove 0 & push on top

Page at BoS : least recently used.

use doubly linked list.

Teacher's Signature

→ causes a lots of overheads.

2/4/18

4) LRU Approximation :

→ can have LRU approximation : keep reference bit
if 1 ⇒ used recently (following LRU)
0 ⇒ not " "

→ may use Additional bits (Ref. bits)
In page table : have ref. bits

Let → 3 bits 0 0 0 → Initially
when pages is brought 1 0 0 → 0 1 0 → 0 0 1
page 3 1 0 0 → 0 1 0
page 5 1 0 0

whichever value is least ⇒ it is LRU ⇒ has to be replaced
(reference bit)

2 3 (2) 5 (3) 0

2	X	0 0	→ 0 1 0	2 → 0 0 1	5 → 0 1 0	3 → 0 0 1
3	1	0 0	2 → 0 1 0	5 → 0 0 1	3 → 0 0 0	
5	1	0 0	→ 0 1 0			

because 2 has come

if referenced, put 1 here.

Page replacement : check value of reference bit)

DATE : / /
PAGE NO. :

↓ which page will be replaced ?

→ 23524 when 4 comes

↳	2	100 → 010 → 001 → 000 ⇒ 4
	3	~~4~~100 → 010 → 001 ⇒ ~~4~~
	5	100 → 010 ⇒ 2

⇒ Page 3 will be replaced when 4 comes.

5) Second chance Algorithm :

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

F F F ✗ F ✗ F ✗ ✗ ✗ F ✗ F ✗ F ✗ F ✗ F ✗ ⇒ 9 F : 18 references

↳ I'm going to attach a ref. bit (2nd chance bit) to each page.

↳ Everytime new page is brought bit = 0

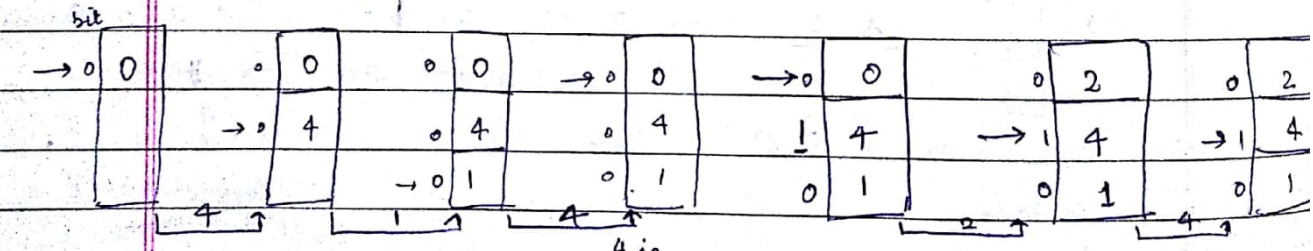
↳ " page is referenced, ref. bit = 1

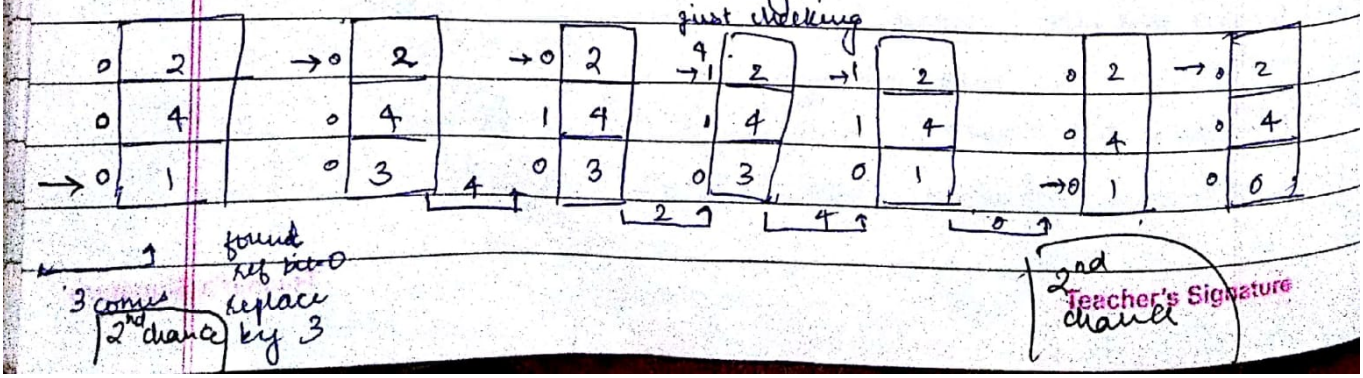
↳ If a page is to be replaced, we will go in Round Robin manner & check ref. bit (starting from ptr)

If bit = 1 → make bit = 0 : got a second chance.

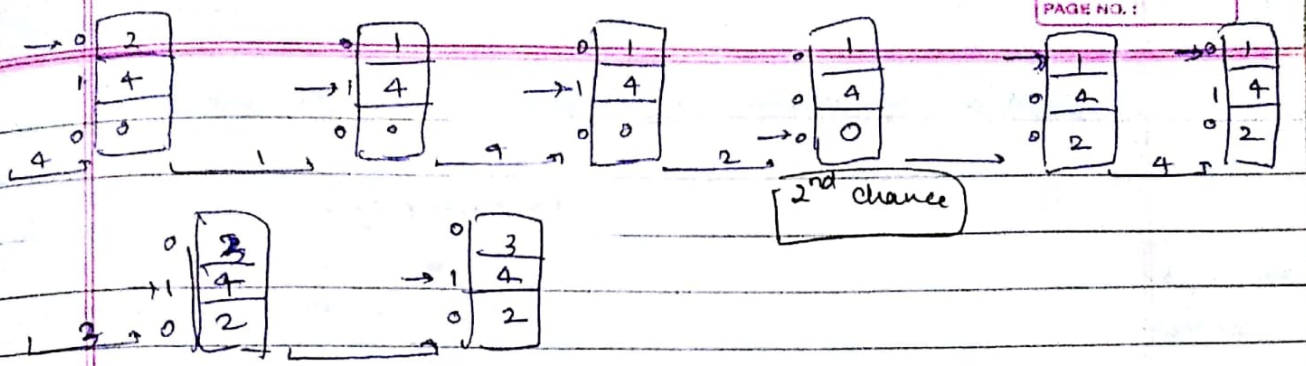
= 0 → replaces that page

1 means it's recently used





Teacher's Signature



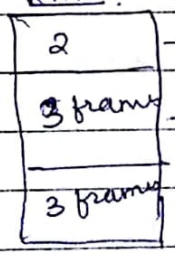
→ If it had been a regular algorithm, 1 would've replaced 4 so many times

Backing store : In Disk (sometimes, called Paging Disk)

→ When a page is replaced (1 by 3) → 1 has to be written back into disk assuming dirty bit = 1,
↳ First write 1 in disk, then only transfer 3.

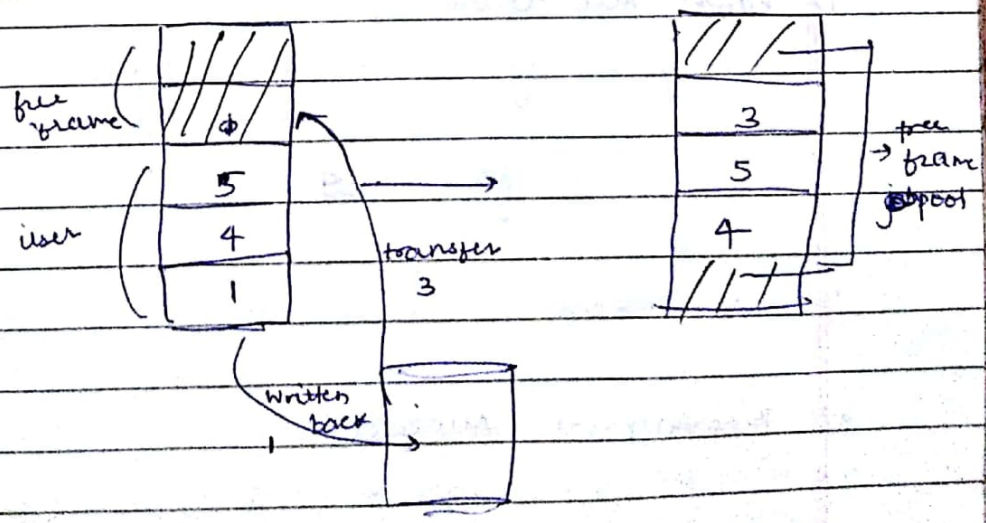
OS keeps pool of free frames

RAM.



28 frames

Now,
while 1 is being written in paging disk, transfer 3 in free frame pool
↓
doing parallely



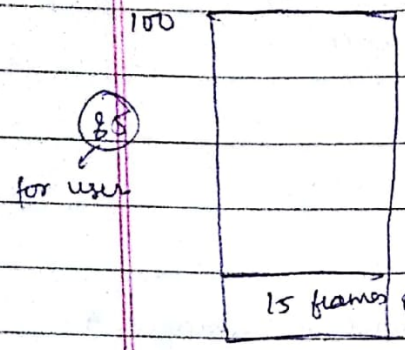
→ Free frame table / is also to be maintained regularly

Page table

Teacher's Signature

DATE: / /
PAGE NO.:

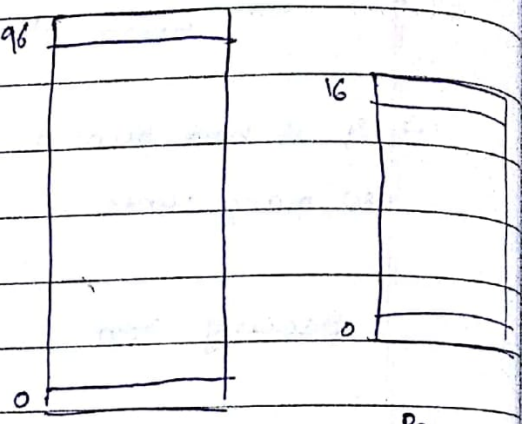
Allocation of frames (Assume no free frame pool)



No multiprogramming

⇓

1 process in RAM at any time



P_1 is running & demand page is executing.

→ Till 85 references : no problem as diff. pages

→ After that : Page Replacement will occur.

↳ If Multiprogramming allowed ?
How to allocate frames to processes ?

1) Equal Allocation :

$$\eta = \frac{m}{p} \quad \begin{matrix} \longrightarrow \text{ no. of frames} \\ \longrightarrow \text{ no. of processes} \end{matrix}$$

$$= \frac{85}{2} \simeq 42$$

↳ Not sensible

2) Proportional Allocation :

$$S = \sum s_i = 97 + 17 = 114$$

Teacher's Signature

$$\eta_i = \frac{s_i}{S} \times m$$

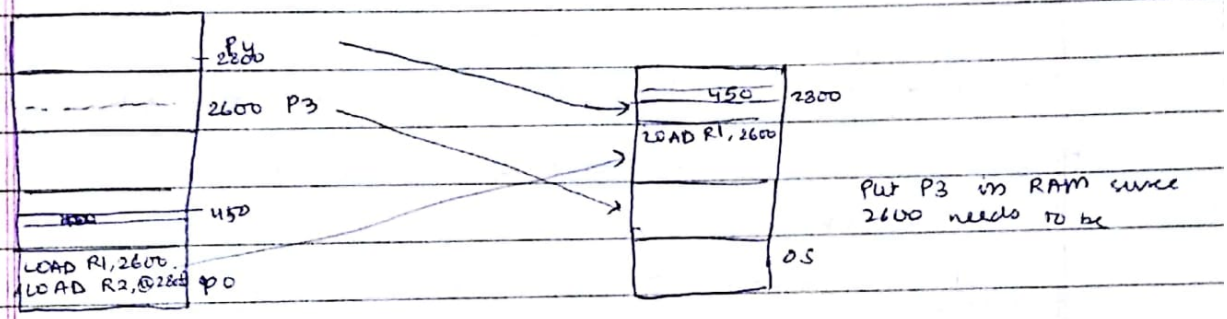
$$\eta_1 = \frac{97}{114} \times 85 \quad , \quad \eta_2 = \frac{17}{114} \times 85$$ (get approximate value in int)

(Total can't exceed 85)

→ The no of pages allocated in RAM > min no. of pages required by that process

(If not so ⇒ ↑ many P.F.) ↓ how do we know this

A/4/18



LOAD R2, @2800 → Indirect :→ Whatever is in 2800, go to that add, & put value + nt there in R2

→ Now, it has to bring P4 next.

Now, I 'll to bring 450

→ Every time I access memory locⁿ, I've to bring that page into RAM.

⇒ Min No. of frames in RAM depends on ISA.

If no indirect add. ⇒ no need of all this.

⇒ no. of frames req. ↓

Teacher's Signature

DATE : / /
PAGE NO. :

Eg: LOAD R4, @@@2600 → 3-4 levels of indirect add
↓
It'll ↑ no. of frames required by this process

* No. of levels of indirect add. $\propto$ min no. of frames required by a process

→ Let say : 15 frames in the RAM

3 processes
min no of frames
P1 → 7

∆ we've allocated only 5 → lots of Page Faults will occur. CPU utilization ↓↓
↳ If OS finds ∆ a cond^n like this, it'll suspend some pgs & free the pages

* CPU utilization is monitored regularly to know if lot many PF or not.

CPU utilization ↓↓ : 2 possibilities :
↳ lots of i/o → lots of page faults happening
OR
↳ degree of multiprogramming is low

→ Suppose all frames are full & P2 need another pg → some replacement should occur

Global Page Replacement Policy | Local Page Replacement Policy

1) Global Page Replacement : It can take frame for from other process also

2) Local Page Replacement : Replace pages which are in frames allocated to that process only

Teacher's Signature

DATE: / /
PAGE NO.:

RAM

Eg

P3	10
P3	9
P1	8
P2	7
P1	6
P2	5
P1	4
OS	1, 2, 3

If FIFO P1 : 1 2 3 4 1 2 3

⇓

Continuous P.F.

↓

Process starts "thrashing"

P1 needs another frame.

will run only for more pages

1.) LPRP ⇒ Possible Pages : 4, 6, 8 (can use any replacement algo within 4, 6, 8)

2.) GPRP ⇒ All of them 4, 5, 6, 7, 8, 9, 10

Global better → have more frames

quadr. → scan whole list

↳ If replaced ~~Pages~~ pages of P2 ⇒ Pages of P2 ↓

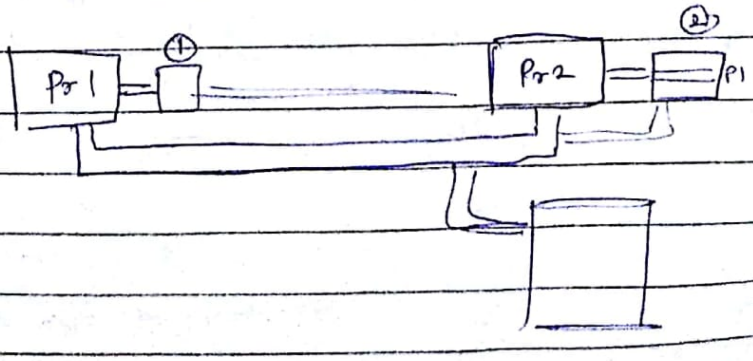
→ P.F. ↑

⇒ P.F. behavior depends on other processes also

Local → P.F. behavior depends only on inst^{n}s of that process.

Multiprocessor :

→ have some local memory attached to each processor



Teacher's Signature

DATE: / /
PAGE NO.:

P1 Time taken by P1 to access (1) < access (2) ∴ Non-Uniform memory Access (NUMA)

Allocation

→ Algo. should be s.t. if P1 goes to Pr1 ⇒ It should be allocated frames in Mem. which is near to Pr1 (①)

⇒ If UMA ⇒ can allocate frame anywhere

→ If I follow Global RP ⇒ Thrashing may ↑

Local RP ⇒ limits Thrashing to that process only

↓

CPU utilization ↓

↓

OS thinks may be I should ↑ degree of multiprog.

↓

Bring more Processes

↓

Frames allocated to each Process ↓

More thrashing ←

→ How to see how many frames are required by a process?

1) Working Set Model

Δ - working set window

Let Δ = 5 (maybe, after some time (ms))

Reference String : 1 4 3 4 1 2 7 8 2 1 3 1 5 2 1

Need of frames: {1, 3, 4} (check after 5 references) {2, 7, 8, 1} {3, 1, 4, 5, 2}

③ ④ ④

At this moment, working set size = 4

Let — — for P2 = 4

" — for P3 = 5

Teacher's Signature

DATE: / /
PAGE NO.:

working set size for all processes

Demand = $\sum wss_i$ = 15

If D > m ⇒ Thrashing will occur

→ Maybe, OS should try to fulfill demand of 1 process & check if it can fulfil Δ of others also or not

2) Page Faults Frequency (PFF)

PFF > Threshold ⇒ needs more pages Suspend processes

PFF < Threshold ⇒

If 3 P ⇒ monitor PFF for each

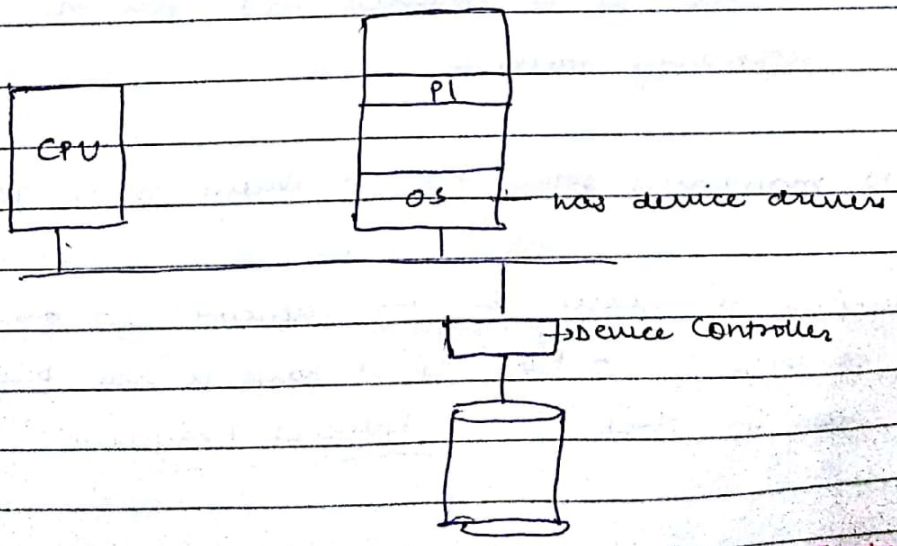
PFF (P_1) & (P_2) < Threshold Δ P_3 > T

→ OS will take 1 or 2 pages & give it to P3

→ All are above Threshold ⇒ Suspend the processes

6/4/18

→ Everytime I want to R/W on a file : have to access Disk.



Teacher's Signature

To minimize this disk access,

↳ can use a memory

⇒ Whenever a file is needed, we map it to virtual memory of that process file.

Just like other pages, this page will also be brought to RAM → also be part of PT of process

(Instead of system call & accessing disk)

This is called, Memory mapped files

(attached to add. space of process.

→ Memory-mapped I/O :

I/O can be given certain add. space in memory & communicating through this memory only.

↳ eases access to I/O device

↳ time ↓

→ File : A collection of related info.

for user, it is smallest unit you can store in secondary memory.

→ OS maintains some d.s. → Queue, Stack, Block etc.

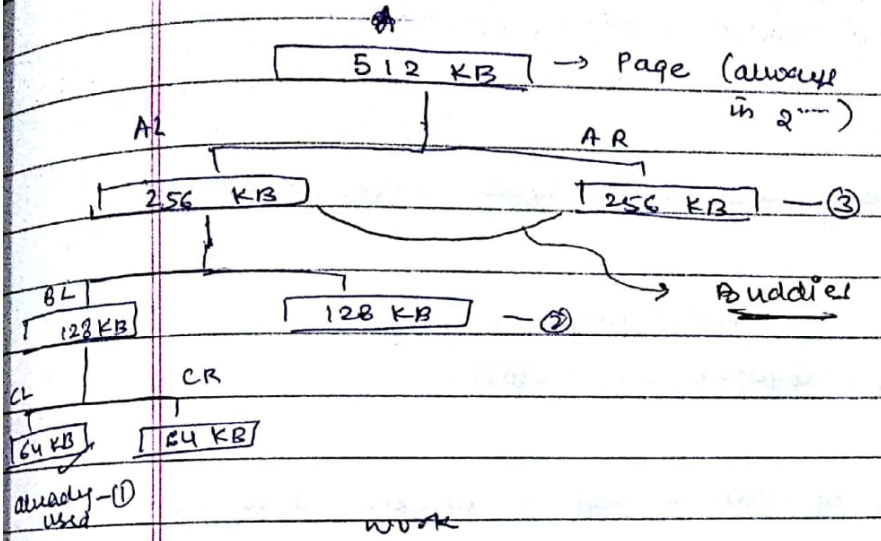
→ Suppose it creates ds ⟷ capacity 500 KB. & page size = 4 KB. It'll have to use page to store data in disk → Internal Fragment

Teacher's Signature

DATE : / /
PAGE NO. :

Allocating Memory to Kernel

1. Buddy System :



After de work is completed, there will start combining : Coalesce

→ If somewhere in b/w, 127 KB came → can assign ②

→ ___ ___ 200 KB → — ③

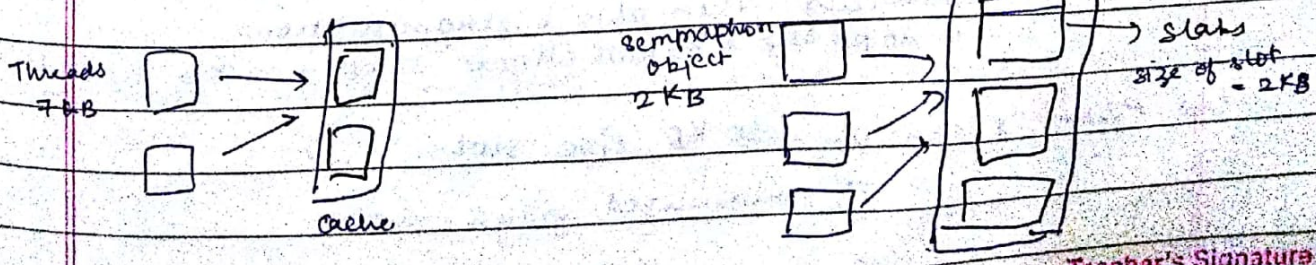
① & ② finished → again coalesce to form AL & later, combine

→ Kernel requires so many d.s. of diff. size → that's why we allocate memory.

→ Keeping track of buddies is little difficult.

2. Slab Allocation :

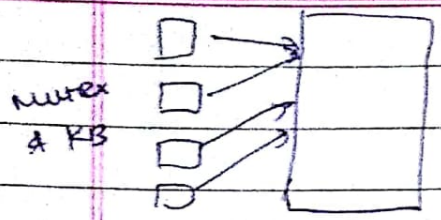
→ It will have cache for diff. d.s.



Teacher's Signature

* If multiusers ⇒ more semaphores used → size of cache ↑↑

→ there are many caches in system
  ↳ L1, L2
  ↳ TLB
  ↳ Below one



Use : in Cache, everything will be of equal sizes → ∄ slabs

Slabs : section of contiguous physical pages
(2-3 pages together → 1 slab)

→ Depending on size of slab & size of objects, I will have fixed no. of slots in cache

→ whenever a new obj. come, it'll be given space (of same size)

→ slabs can be :
  ↳ full
  ↳ empty
  ↳ partial

→ when new obj. comes : If partial & empty both available : put in partial one.

→ size of page is same. Only it is divided into diff. parts.

* It is possible that build of this cache is not same as the cache that we use learnt earlier → avoids memory wastage

→ In Buddy system : have to do lot of breaking down & coalescing, then also ; fragmentation
(4 20 KB size ⇒ it will allocate 32 KB → 12 KB wasted)

→ Here, I can use 20 KB size slot
⇒ minimized fragmentation

Teacher's Signature

FILE SYSTEMS

DATE: / /
PAGE NO.:

→ File : A coll^n of related information

→ File could be : data, program
- data → numeric, text, x-numeric

→ In any case → seq. of bits

→ Can be free structure

→ smallest unit of secondary storage from user p.o.v.

→ File System : How to store files & how to retrieve them.

→ Many attributes are associated with any file :
- ↳ Permission
- ↳ Name
- ↳ Type
- ↳ Size
- ↳ Location
- ↳ Time
- ↳ Date
- ↳ User
- ↳ Identifier

→ In file system, ∃ a table (in directory) where name of file is given an identifier. After that, internally, file will be ~~reff~~ referred through that no. → will point to all attributes of that file
(has ⇓ all info. about file)

→ What can you do with a file ? (Abstract data type)

1.) Create : have to find space in file system for this file, update the file directory

2.) Read / Write : have to locate the file in file system, then check file ptr & update it (pehle kahaan tk tha, ab kaha jaayega)

* It will keep ptr per process for a file.

Teacher's Signature