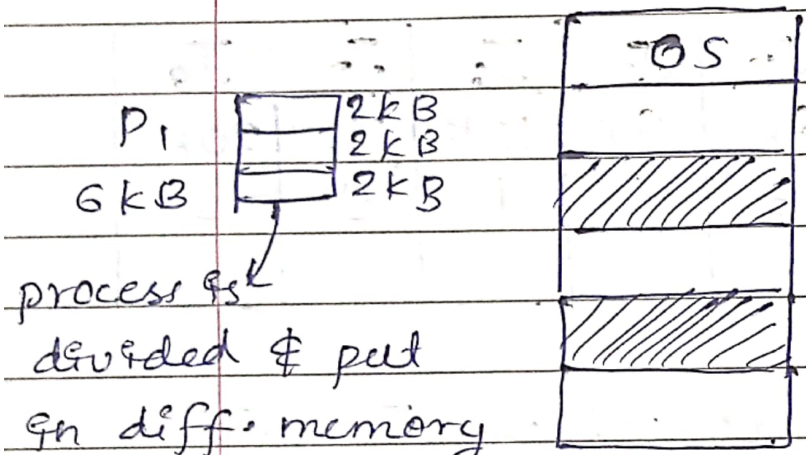
Need of Paging

Non-contiguous memory allocation

A process can be spanned & put up in fragments in diff. memory locations.

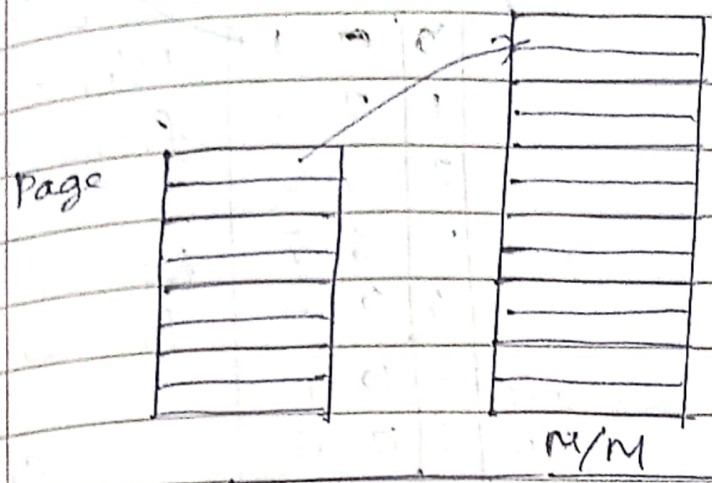


process is divided & put in diff. memory locations

process is divided acc. to the hole size but the hole size changes on runtime so its a a left time consuming operation of process)

So a process is divided into pages before it gets into Main memory

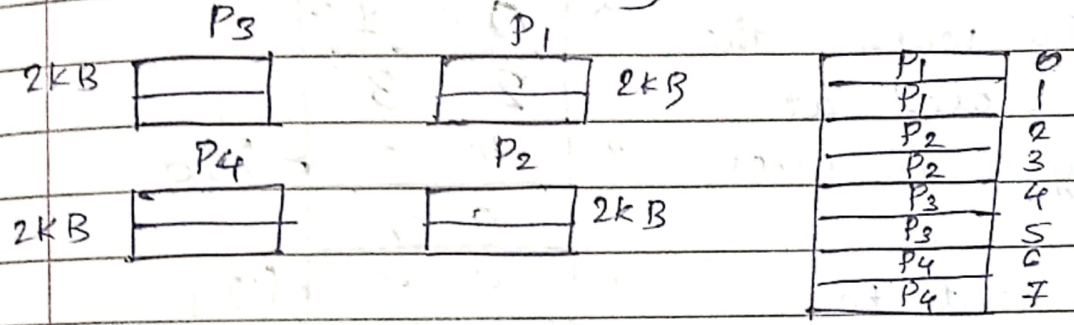
When we do partition of a process in a secondary memory its called paging while when we do partition of a process in RAM its called framing.



Page size = Frame size

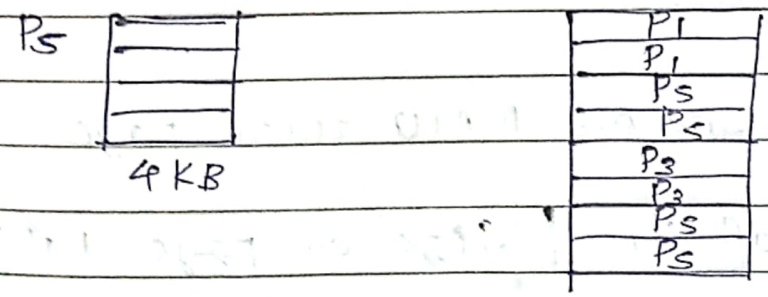
External fragmentation is totally removed

For e-g



Page size = frame size = 1KB

Now P_2 & P_4 complete execution & P_5 of size 4KB comes



Pages P1

0	0	1
1	2	3

(P1)

Bytes

Process size = 4B

Page size = 2B

Nq. of pages/process = $\frac{4B}{2B}$ = 2

frame nq.	Bytes	
0	0 1	M/M size = 16B
1	2 3	frame size = 2B
2	4 5	
3	6 7	Nq. of frames = $\frac{16B}{2B}$
4	8 9	
5	10 11	
6	12 13	= 8 frames
7	14 15	

M/M

Eg.

CPU wants byte nq. 3 of P1

So CPU wants the data stored at 9th byte in M/M

So we need to convert address generated by CPU to addr. of M/M. ~~secondary~~ (Absolute addr.)

0	0	1
1	2	3
2	4	5
3	6	7
4	8	9
5	10	11
6	12	13
7	14	15

← P10

← P11

0 1

→ This task is done by MMU (Memory management unit)

→ For this conversion MMU uses page table.

Page table of P1

0	f2
1	f4

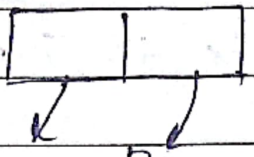
P1

Size of page table = Nq. of pages

Nq. of entries in P.T. = Nq. of pages

CPU generates Logical address

Logical address has 2 parts



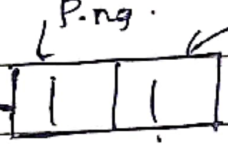
Page offset ~~is~~ the size of a page
Max. no. of bits in page offset ~~is~~ max. no. min no. of bits required to represent page size.

Bits in offset $\Rightarrow$ **Page offset = $\log_2$ (page size)**

Page number = $\log_2$ (No. of entries in P.T.)

$\therefore$ **= $\log_2$ (No. of pages)**

For e.g. in this case CPU generates a logical address



Now we would look at the ~~#~~ entry of ~~pa~~ at pos. 1 of page table

& We'll look at 2nd bit in M/M i.e. bit at pos. 1 in M/M.

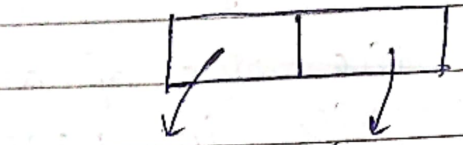
Physical address is the address of the main memory.
P.A. depends on the size of main memory

$\therefore$ Bits in P.A. = $\log_2$ (Size of M/M)

∴ In this case, ~~bits in size~~

bits in P.A. = $\log_2(16) = 4$

∴ P.A.



Frame no. | Frame offset

bits in frame offset = $\log_2$ (frame size)

∴ Here bits in f.O. = $\log_2 2 = 1$

bits in Frame no. = $\log_2$ (No. of frames)

Here

P.A.

3	1
100	1

frame offset = Page offset

Here the P.A. address collectively represents 9 (1001)

In a nutshell

CPU → Logical Address → MMU → Physical Address

$2^{10} = 1K$ $2^{20} = 1M$ $2^{30} = 1G$ $2^{40} = 1T$

PAGE NO.:

Memory is byte addressable

LAS (Logical address space) = 4GB
PAS (Physical —11— —11— →) = 64MB.
Page size = 4KB
No. of pages = ?
No. of frames = ?
No. of entries in Page table = ?
Size of page table = ?

$LA = \log_2 (LAS)$

$LA = \log_2 (2^2 \times 2^{30}) = \log_2 (2^{32})$

LA = 32 bits

LA : | P.N. (20 bits) | P.O. (12 bits) |
32 bits

Page offset = $\log_2$ (page size)

$= \log_2 (2^2 \cdot 2^{10}) = \log_2 (2^{12})$

∴ Page offset = 12 bits

No. of pages = $2^{\text{Page no. (bits)}}$
No. of pages = 2^{20}

Size of P.T. = No. of entries in P.T. × Page table entry size

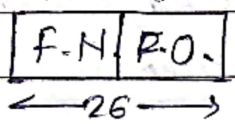
PA = ~~log~~

$PA = \log_2(PA\text{-}S) = \log_2(2^6 \cdot 2^{20})$

$PA = \log_2(2^{26})$

PA = 26 bits

PA =



As we know page size = frame size

∴ frame offset = page offset = 12 bits

∴ No. of frames = $2^{\text{bits in frame no.}}$

No. of frames = 2^{14}

No. of entries in page table = No. of pages in a process

∴ No. of entries in P.T. = 2^{20}

Size of page table = No. of entries in P.T. × Bits in frame no.

∴ P.T. contains the frame no. where the particular page is stored

Size of P.T. = $2^{20} \times 14$ bits

1 Word = 1 byte if nothing is mentioned

PAGE NO.:

Q Consider a system which has LA = 7 bits, PA = 6 bits, page size = 8 words then calculate the no. of pages & no. of frames

LA = <4> <3>

P.N.	P.O.

7 bits

P.O. = $\log_2$ (Page size)

P.O. = $\log_2 (2^3)$

= 3

No. of pages = $2^{P.N.} = 2^4$

~~No. of~~

PA = <3> <3>

F.N.	F.O.

6 bits

F.O. = P.O. = 3

No. of frame = $2^{F.N.} = 2^3$

No. of entries in a P.T. = No. of page in a process

= 2^4

Size of P.T. = $2^4 \times 2^7$ B

= $2^4 \times 3$

= 3×2^4 B

Page table entry:

Frame no.	Valid(1)/ Invalid(0)	Protection (RWX)	Reference (0/1)	Caching	Dirty

Mandatory field ← Optional field →

Valid/Invalid = represents whether the page is present there or not

1 → present 0 → absent

Protection:

R = Read W = Write X = Execute

The given process can read (R), write (W) or execute (X) based on these permissions

Reference:

When we swap-in & swap-out pages, then this field tell us whether if we brought the page into M/M at same pt. of time

Caching:

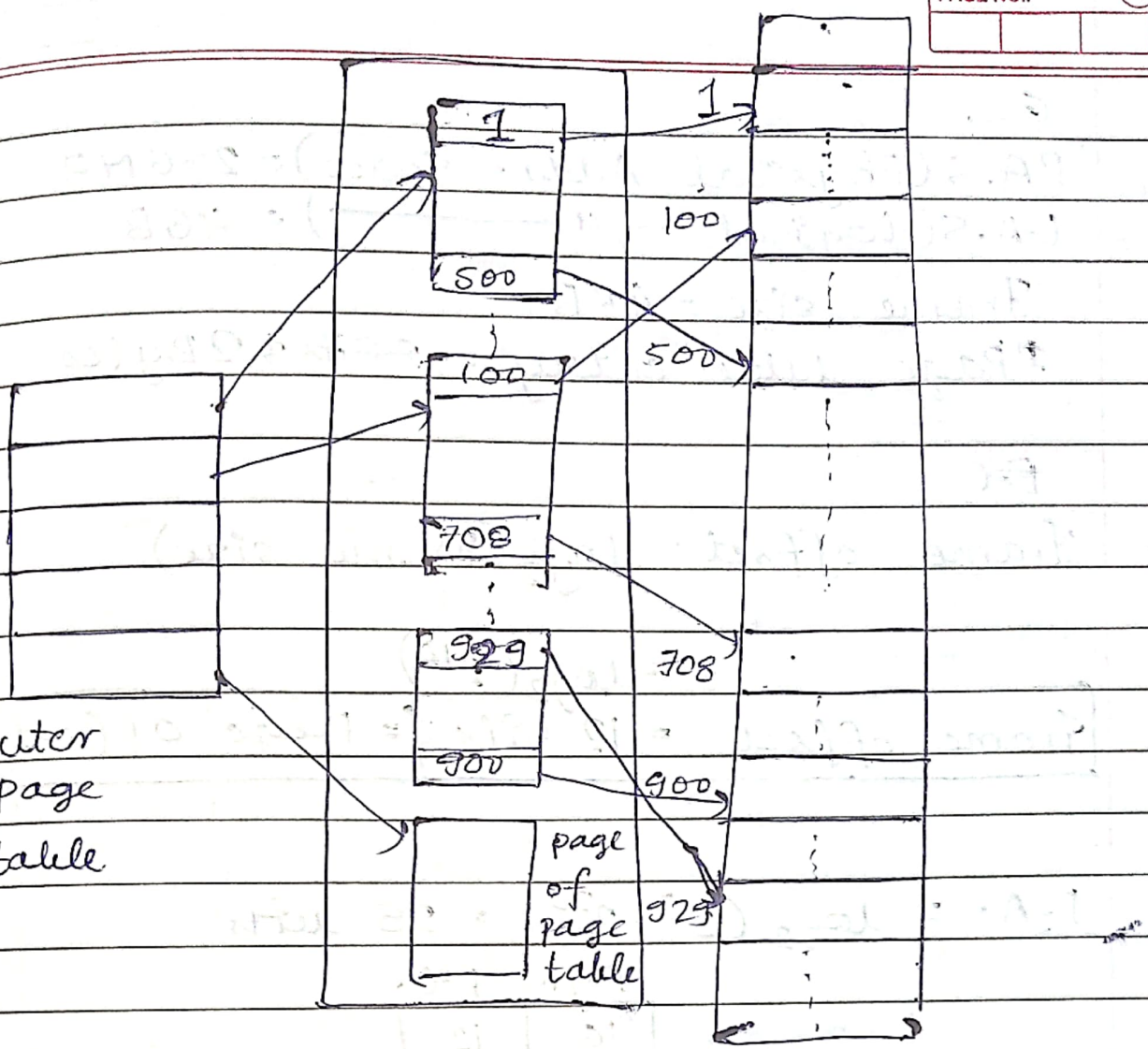
Whenever the user wants fresh data, we should disable caching

If the user wants redundant data then enable caching

Dirty:

Tells us whether the page is modified or not.

1 → Modified 0 → Not Modified



page table

Need of multi-level paging:

If the size of P.T. (page table) is bigger than the size of the frame
So we need to divide the page table into smaller pages

E.g.

Q1 P.A.S (Physical Addr. Space) = 256MB

L.A.S (Logical " ") = 4GB

Frame size = 4KB

Page table entry = 2 Bytes

Frame offset = $\log_2$(Frame size)

= $\log_2(2^{12})$

Frame offset = 12 bits = Page offset

P.A. = $\log_2(2^8 \cdot 2^{20})$ = 28 bits

F.N.	P.O.
16	12

← 28 →

L.A. = $\log_2$(LAS) = $\log_2(2^{32})$ = 32 bits

P.N.	P.O.
20	12

← 32 →

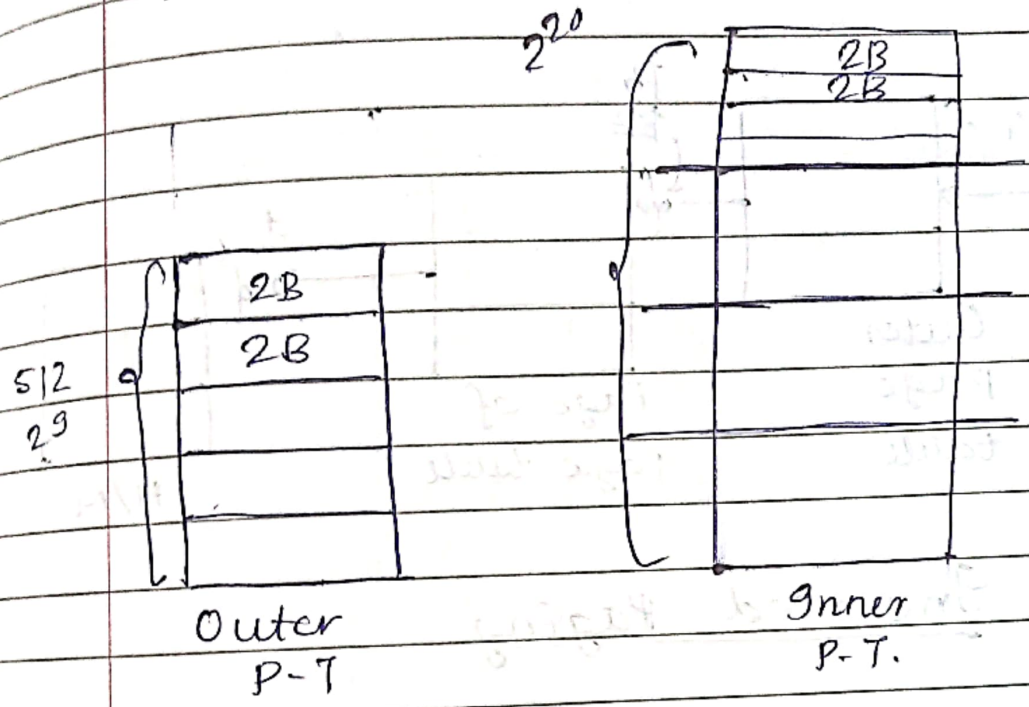
No. of entries in P.T. = $2^{P.N.} = 2^{20}$

Size of P.T. = $2^{20} \times 2$ = 2 MB

Here we can see size of P.T. = 2 MB & size of frame of M/M = 4KB, as 2 MB won't come in 4KB we divide the pages further

Divide the page table by 4KB

No. of pages after division = $\frac{2MB}{4KB} = \frac{2^{21}}{2^{12}} = 2^9$



Size = $2^9 \times 2$

$= 2^{10}$

No. of entries in each page = $\frac{\text{Page size}}{\text{size of page table entry}}$

$= \frac{4KB}{2B} = \frac{2^{21} \cdot 2^{10}}{2}$

$= 2^{11} = 2 \cdot 2^{10} B$

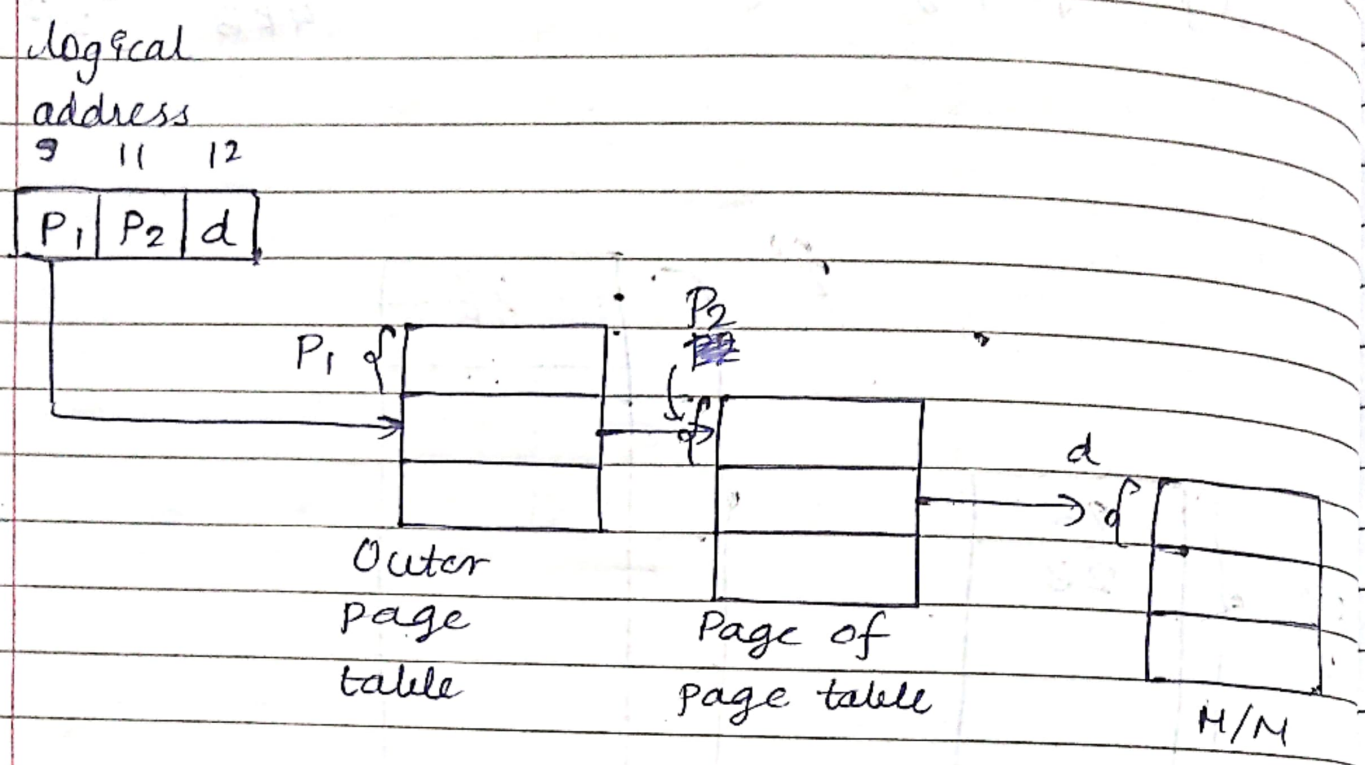
$= 2KB$ 2^{11} entries

No. of entries in each page = 2^{11}

L.A. =

P.N.	P.O.
20	12
9	11

Address translation scheme



Inverted Paging

For this let's understand normal paging

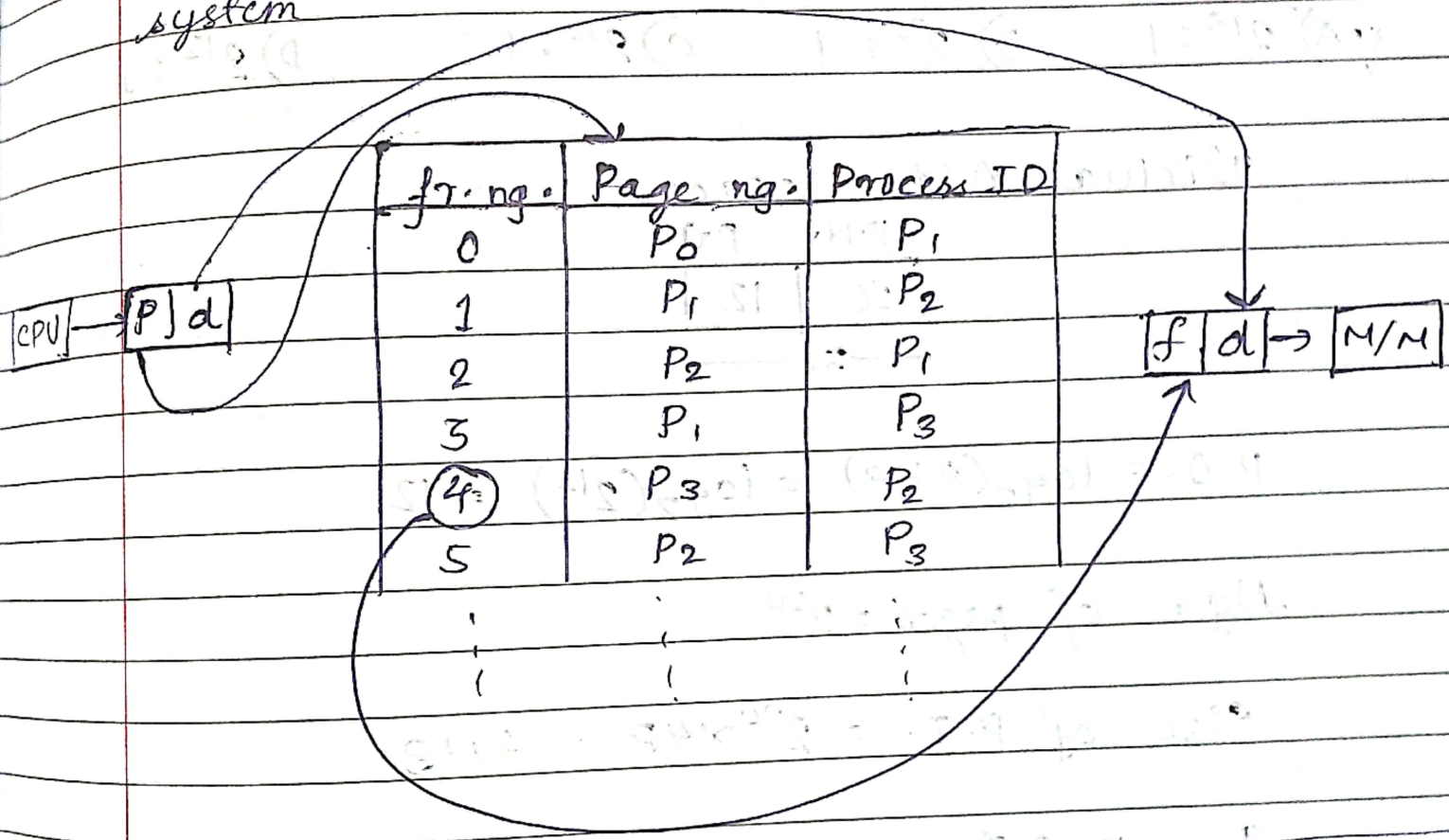
Normal paging

→ Each process has its own page table
→ P.T. would be stored in M/M
→ For e.g. if we have 10 processes then each one of them has their own P.T.
→ If every process has 1 or 2 pages of its in M/M
→ We need to use 10 frames to store the P.T. in the M/M
→ As we know M/M is limited so if we use some frames only for storing P.T. it would be very ineffecient.

Inverted paging

→ The concept says that OS would maintain only 1 P.T. for entire system

→ The OS maintains one global P.T. for entire system



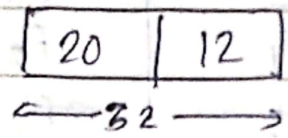
No. of entries in Inverted page table (I.P.T.) = No. of frames in M/M

Searching time is very high as we need to search linearly for each page

Q Consider a virtual address space of 32 bits & page size of 4KB. System is having a RAM of 128KB. Then what will be the ratio of P.T. & Inverted P.T. size if each entry in both is of size 4B?

✓(A) $2^{15}:1$ B) $2^{20}:1$ C) $2^{10}:1$ D) $2^{12}:1$

Virtual Addr. space:



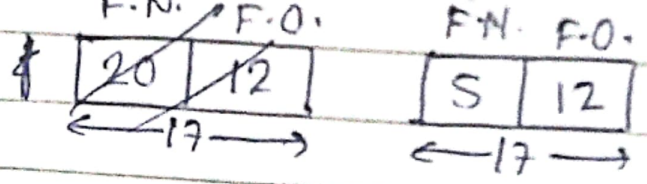
P.O. = $\log_2(4KB) = \log_2(2^{12}) = 12$

Ng. of pages = 2^{20}

Size of P.T. = $2^{20} \times 4B = 4MB$

In I.P.T.,
Ng. of entries = Ng. of frames in M/M
in P.T.

Physical Addr. space:



Ng. of frames = 2^S

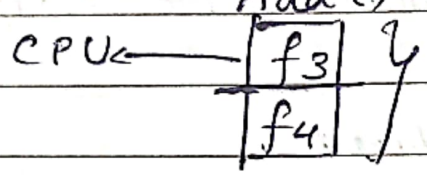
I.P.T size = $4 \times 2^S B$

$$\frac{\text{P.T. Size}}{\text{I.P.T. Size}} = \frac{4 \times 2^5}{4 \times 2^{20}} = \frac{1}{2^{15}} \qquad \frac{4 \times 2^{20-15}}{4 \times 2^8} = \frac{2^{15}}{1}$$

Paging vs Segmentation:

Suppose we have a add() function which performe addition.

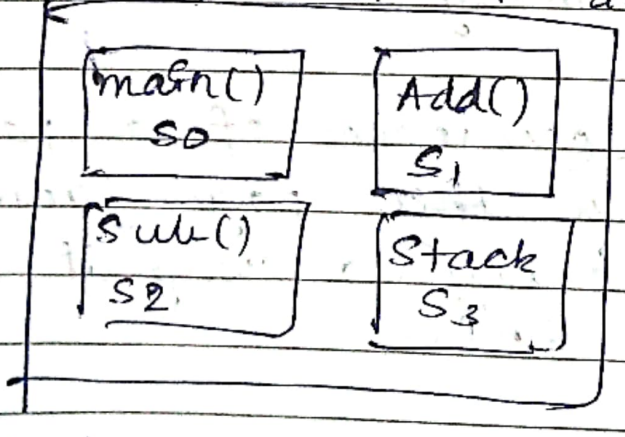
So we divide the program in fixed partition (pages) on the concept of paging



f_3 doesnt contain the entire code, after f_3 executes then the CPU needs to quickly fetch f_4 otherwise add^n won't get executed properly

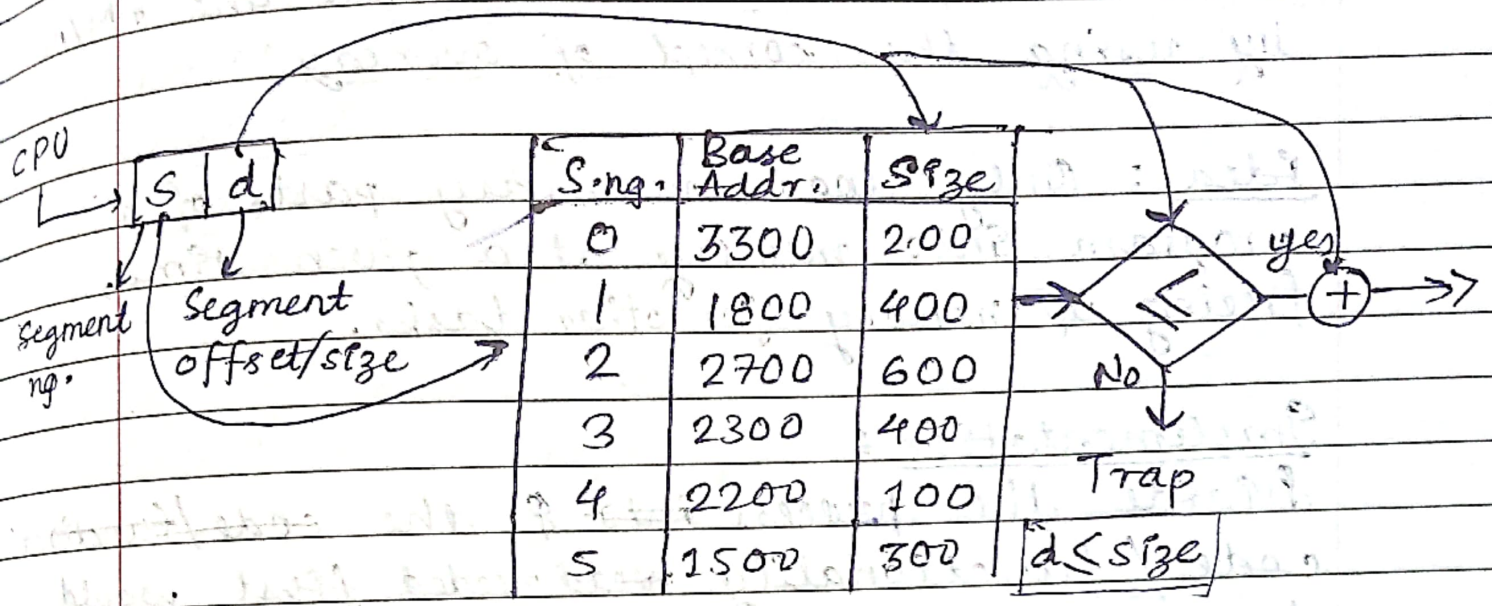
Here if after execution of f_3 we don't get f_4 then the problem of page fault arises

Segmentation: It divides the code into unequal segments for e.g. consider the above scenario. So we divide the code into segments & store it in M/M.



CPU would generate a logical addr. which we need to convert into Physical addr.

We use segment table to convert logical addr. to physical addr.



Advant

For. e-g.

L-A. | 001 | 200 |

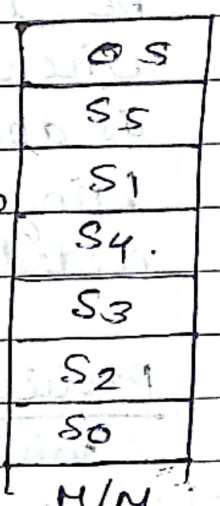
So we want segment 3 first.

Then we check whether the offset is less than size in Segment table

So we read the data from 2300 to 2500 & give it to the CPU.

→ If offset size is greater than the size in S.T. we end up in a trap.

→ Here we group the related data into same segments



Overlay: If the size of my process is more than the size of memory, then I can make that process accomodate in the M/M by using the concept of overlay

Idea: Only load necessary parts of a program into memory at a given time, freeing up memory for other tasks.

Implementation:
Divide the process & the ~~code/function~~ code / functionality ~~need~~ needed first would be loaded into M/M.
When the work of that functionality is done then we would bring out & bring in another partition which is needed & the process would continue.

Problem: No driver in the OS ~~to decide~~ which supports the overlay
→ Here the user needs to decide on its own how to divide that process / program so that whenever we bring the partition, we should be able to execute its functionality properly & the subsequent process should continue.
→ Also the code in one partition shouldn't depend on the other

The concept of overlay is only used in embedded systems ∵ the functions therein are fixed.

But in PC's processes are of diff. sizes so we can't leave it over to the user.

Q Consider a 2 pass assembler pass 1: 90KB, pass 2: 90KB
Symbol table: 30KB
Common routine: 20KB

At a time only one pass is in use, what is min. partition size reqd if overlay driver is of 10KB size

If we load both the ~~passes~~ passes, then mem. reqd = 90+90+30+20+10 = 230KB

For pass 1 = 90+30+20+10 = 140KB

For pass 2 = 90+30+20+10 = 150KB

So, we need to choose max of pass 1 & pass 2, ∵ the larger one would definitely accomodate both the passes

∴ Ans: 150KB