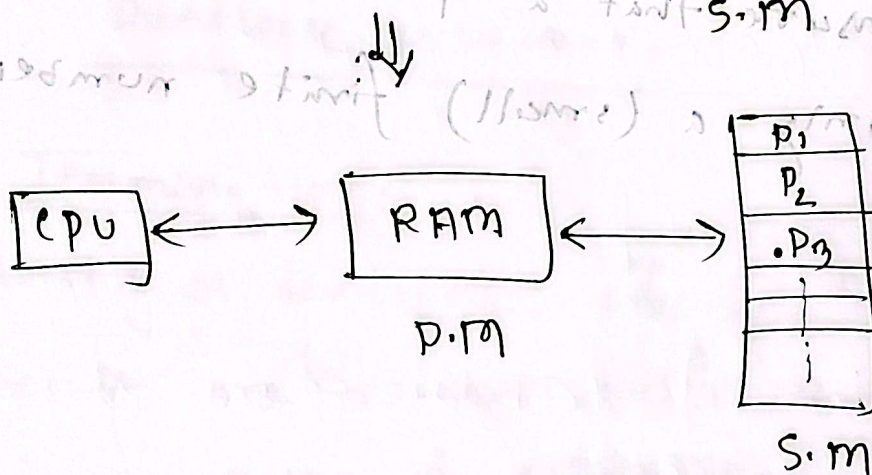
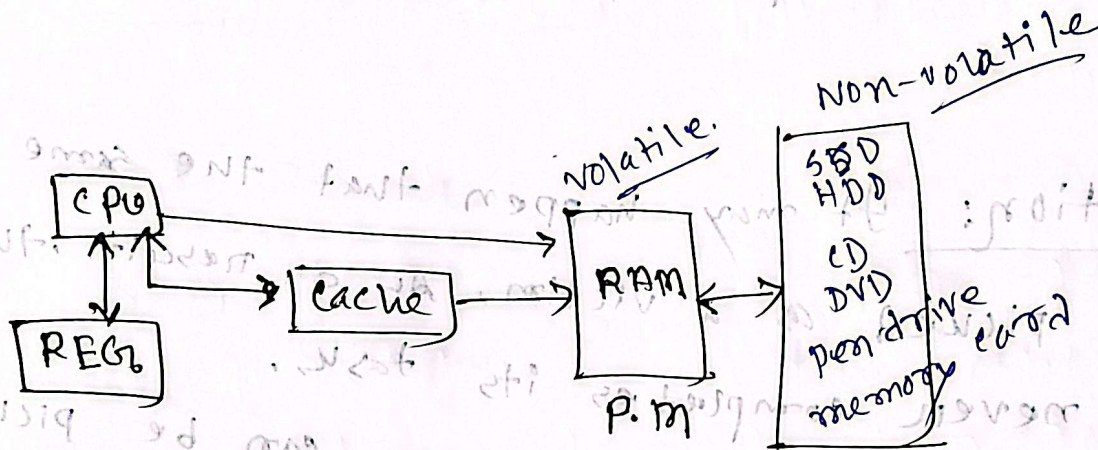
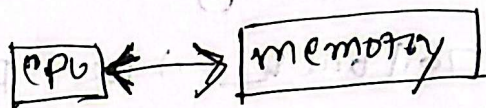


Memory Management

- ① size
- ② Access time
- ③ per unit cost



Memory management function: keeps track of the status of each memory location, either allocated or free. It determines how memory is allocated among competing processes, deciding which gets memory. When memory is allocated it determines which memory location will be assigned. It tracks when memory is freed or unallocated and updates the status.

- ① swap in
- ② swap out

Swapping

→ swapping is a memory management technique in which a process is temporarily moved from the main memory (RAM) to the secondary memory (like a Hard disk or SSD) when the main memory is full, and later brought back to main memory when needed for execution.

Ex: If the RAM is full and a new process needs to be loaded,

→ one of the existing process is swapped out (moved to disk);

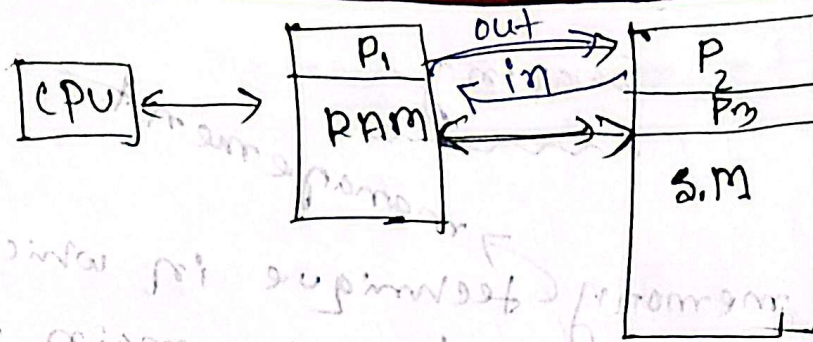
→ the new process is swapped in (brought into RAM)

Purpose:

- To make efficient use of limited RAM
- To allow multiple processes to run concurrently,

Drawback:

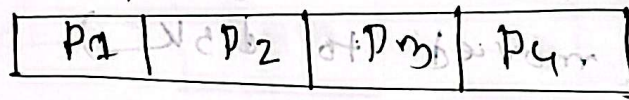
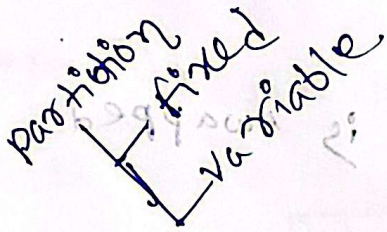
Excessive swapping can use the system to slow down → this is known as thrashing.



Contiguous is a memory allocation method that allocates a single, contiguous section of memory to a process.

→ means that each process is stored in one contiguous block of memory locations

— all its data and instructions are kept together, in a single stretch of physical memory



Advantages:

✓ Access speed is fast (because data are in a contiguous block)

✓ Easy to implement

Disadvantages:

→ leads to external fragmentation (free spaces scattered between blocks)

→ Difficult to allocate large process if free space is scattered.

P₁ → 5MB

OS	
P ₁	10MB
P ₂	20MB
P ₃	30MB

P.M.

~~segmentation~~
 → paging
 → segmentation

Non-contiguous: is a memory allocation method that allocates separate blocks of memory to a process.

→ means that a process is divided into parts, and each part can be stored in different locations of memory, not necessarily together.

Advantage:

P ₁ -1st	} divided.
P ₂ -1st	
P ₁ -2nd	
P ₂ -2nd	

→ No external fragmentation

→ memory is used for more efficiently

→ large processes can fit even if free spaces are scattered!

Dis:

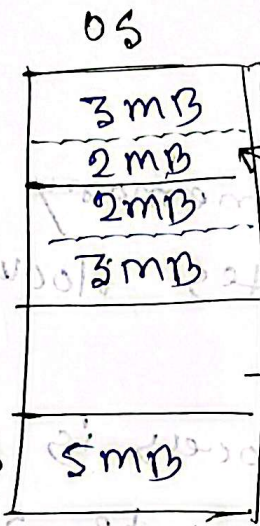
→ access time slower

→ management is more complex (OS must keep track of all pieces)

fixed size partition:

means the main memory (RAM) is divided into a fixed number of equal sized blocks (partition)

→ Each process is allocated exactly one partition so the process must fit within that partition



each block size = 5

$P_1 = 3\text{MB}$

$P_2 = 2\text{MB}$

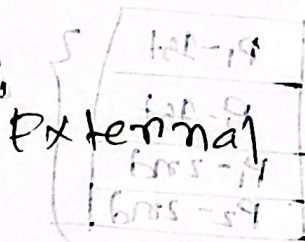
$P_3 = 10\text{MB}$

$P_4 = 5\text{MB}$

Internal Fragmentation

External Fragmentation

$$\text{internal Fragmentation} = (5-3) + (5-2) + (5-5) = 0.5\text{ MB}$$



$$4 \times 5 = 15\text{ MB}$$

Working:

→ OS divides main memory into equal size partition.

→ Each process is assigned to one contiguous partition

→ If a process is smaller than partition size, unused space is wasted (called internal Fragmentation)

→ If a process is larger than a partition, it cannot be loaded (~~External~~ Fragmentation)

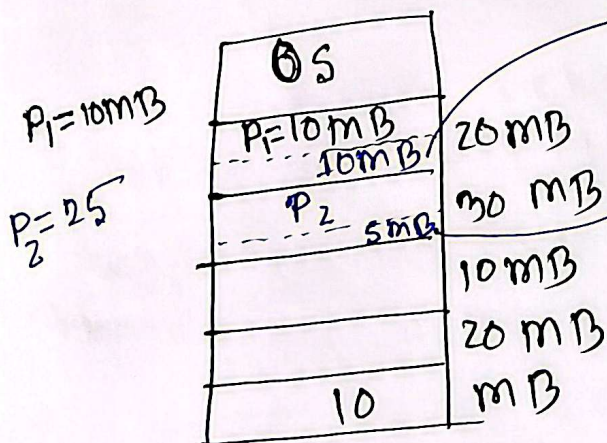
Advant:

- simple to implement
- Easy to manage (fixed boundaries)

Disadvan:

- Internal ~~External~~ Fragmentation
- Process size limitation
- memory utilization is poor.

① Internal ← Fragmentation (Primary memory)
 ② External ← (secondary memory)
 waste space



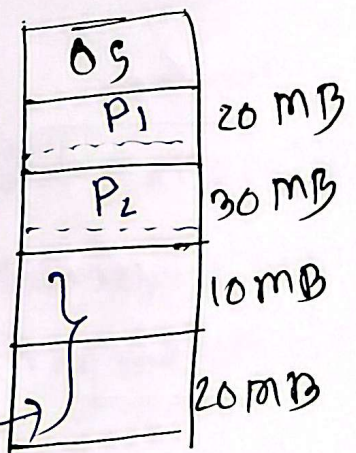
Internal

Internal fragmentation

P₃ = 30MB

wastages

External



Fixed size partitioning algorithm

- ① First Fit
 - ② Next Fit
 - ③ Best Fit
 - ④ Worst Fit
- algorithm

$P_1 = 20$

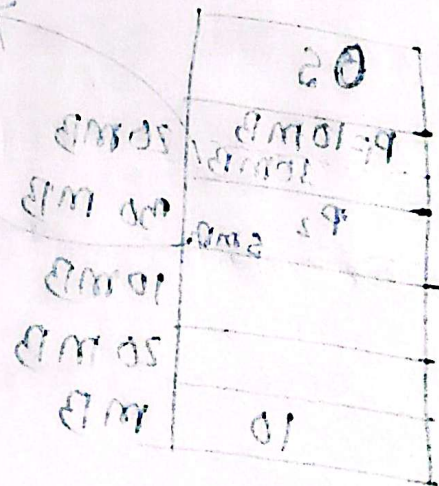
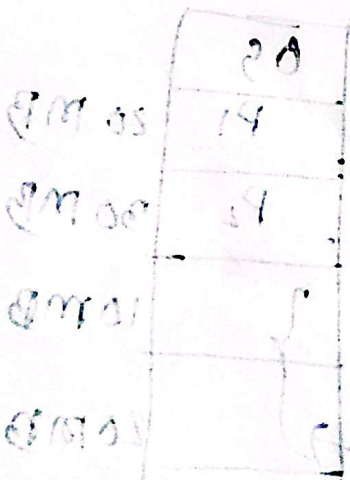
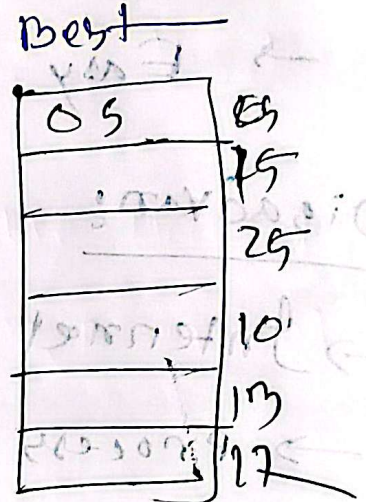
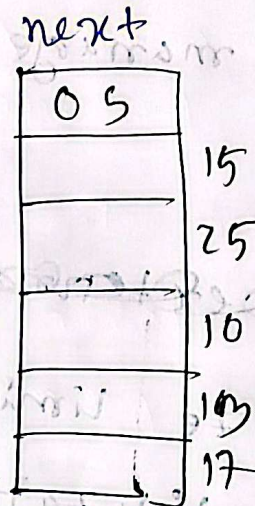
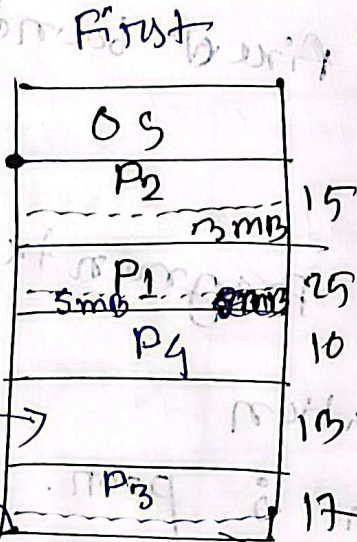
$P_2 = 12$

$P_3 = 15$

$P_4 = 10$

$P_5 = 14$

external
fragmentation



memory Allocation Algorithm :

when a process needs memory, the OS uses one of several allocation strategies to decide where to place that process in main memory.

① First Fit : Allocate the first ^{free} memory block that is large enough to hold the process.

steps:

- Start searching from the beginning of memory
- Allocate the first block that fits.
- stop searching.

Ex:

$P_1 = 20$

$P_2 = 12$

$P_3 = 15$

OS	
P_2	15
P_1	25
	10
P_3	12
2	

→ Fast allocation
→ Causes external Fragmentation

② Next Fit:

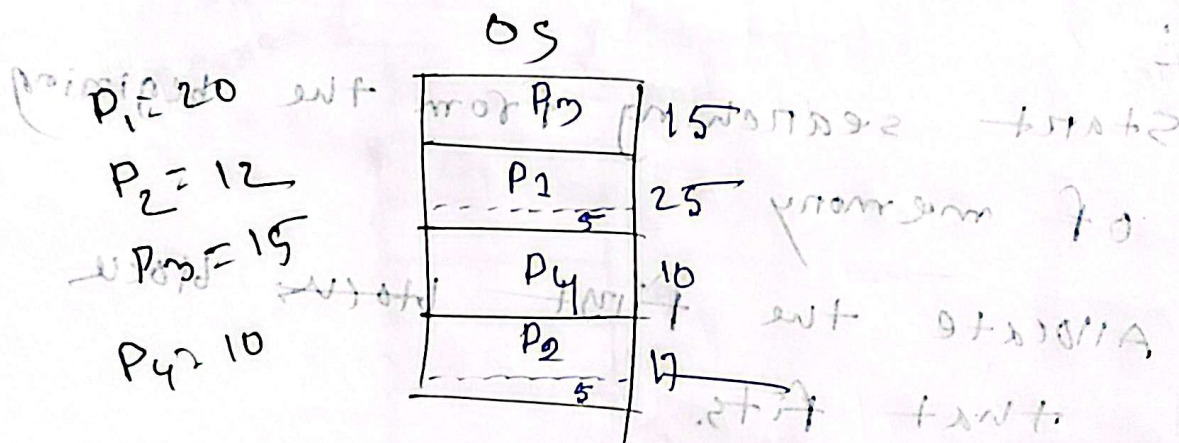
→ similar to first fit, but searching starts from the position where the last allocation ended.

Steps:

① start searching from where the last allocation happened.

② allocation the last suitable block found.

③ If the end is reached, wrap around to the beginning.



Advant:

① slightly faster than first fit

(No need to always start from the beginning)

② simpler to implement.

→ more fragmentation possible.

→ may skip some free blocks near to start, reducing memory utilization.

③ Best Fit: Allocates the smallest available block that is large enough for the process.

Step:

① check all free blocks

② choose the block that leave the smallest leftover space.

Ex:

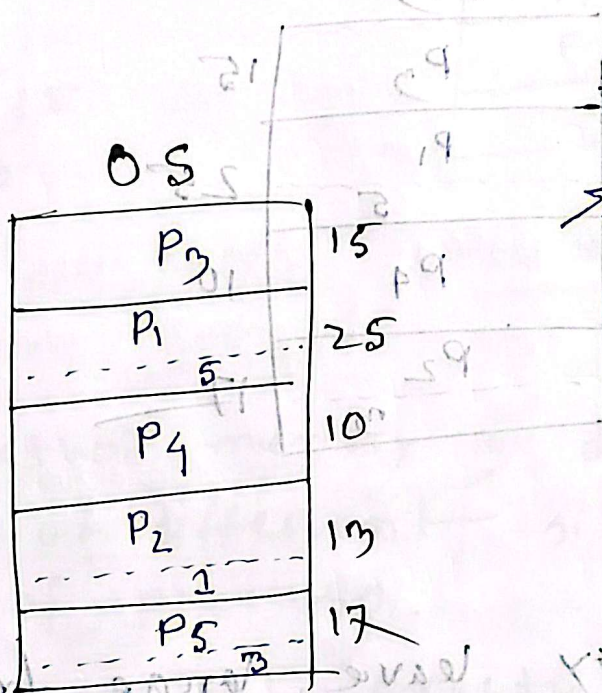
$P_1 = 20$

$P_2 = 12$

$P_3 = 15$

$P_4 = 10$

$P_5 = 14$



internal fragmentation
select no.

advan:

- Reduced wasted space.
- Good for small memory request.
- slower (must search all free blocks)

→ high fragmentation

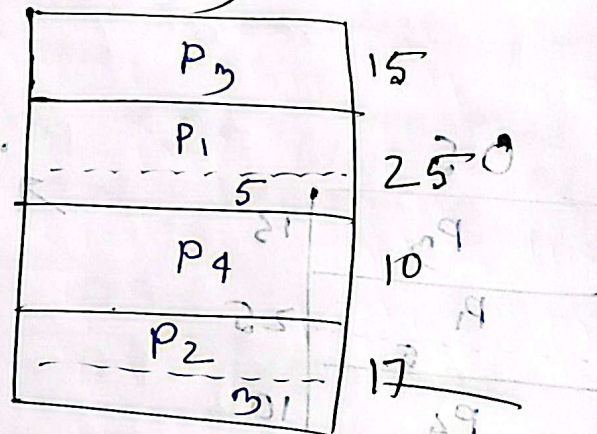
~~danger~~

Worst Fit:

Allocates the largest available block for the process.

Steps:

- ① check all free blocks
- ② choose the largest one



$$P_1 = 20$$

$$P_2 = 12$$

$$P_3 = 15$$

$$P_4 = 10$$

Internal fragmentation
25(62) 10(17)

advant:

→ may leave large free blocks for

future process

→ avoid tiny fragments immediately

→ less waste memory (inefficient use)

→ slower allocation.

Variable Size Partition

$P_1 \rightarrow 5\text{MB}$

P_1

20MB

$P_1 = 10$

$P_2 = 15$

$P_3 = 20$

$P_4 \rightarrow 7\text{MB}$

$P_5 \rightarrow 8\text{MB}$

OS	10
P_1	10
P_2	15
P_3	20
	5

60MB

Execution complete.

OS	
P_4	7
	3
P_2	15
P_3	20
	5

External Fragmentation

↓ solve with

compaction.

→ In this method, memory is divided into partition of different size as per the needs of processes.

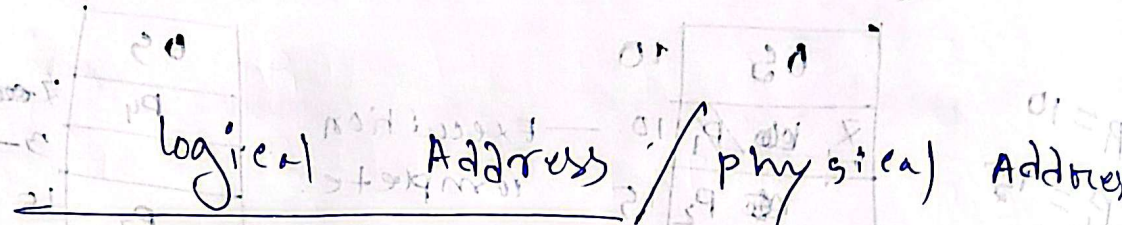
Each process gets exactly as much memory as it requires, no fixed partition size.

Advantage:

- Flexible — memory can be allocated according to process size.
- Better memory utilization than fixed size partitions.

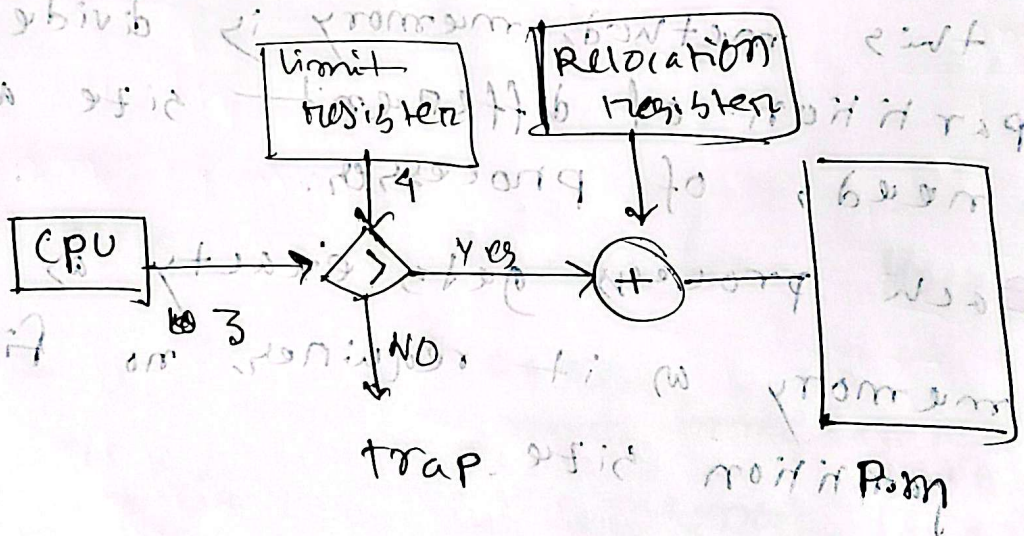
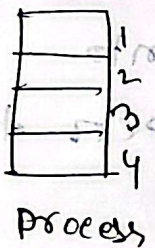
Dis:

- still causes external fragmentation
- requires compaction (to arrange memory & remove gaps)



limit register: store all size of the process.

Relocation register: store the base address of the process.



3.4 → Yes

10.4 → No

Logical address: is a address generated by the CPU during program execution.

→ It is also called virtual address.

→ It exists in the logical address space - not the actual physical memory.

ex: If a program want to access data at address 1000, that's a logical address

not the actual RAM location.

Physical address: is the actual location in the main memory (RAM) where the data or instruction is stored.

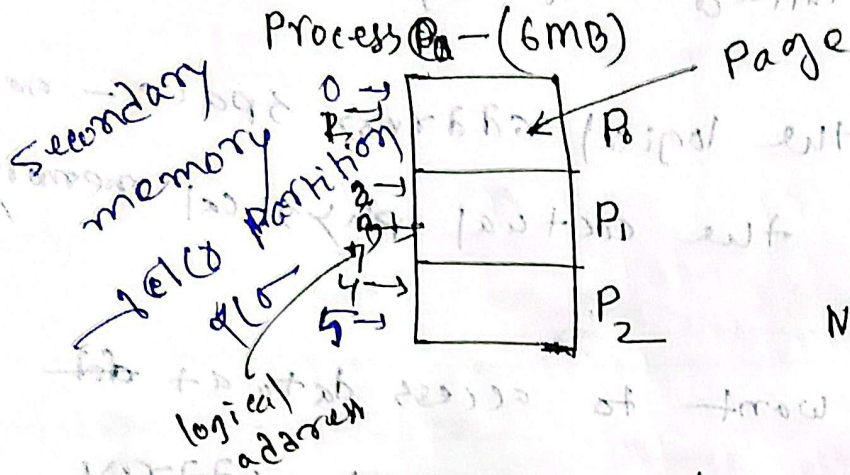
→ It exist in the physical address space.

Relocation
(base register)

$$\text{physical address} = \text{Base address} + \text{logical address (offset)}$$

Process	limit reg	Relocation reg	logical address	
P ₀	380	700	210	→ $210 < 380 = 210 + 700 = 910$ (physical address)
A	425	1100	500	→ $500 > 425$ (invalid)
P ₂	115	200	80	→ $80 < 115 = 80 + 200 = 280$ (phys)

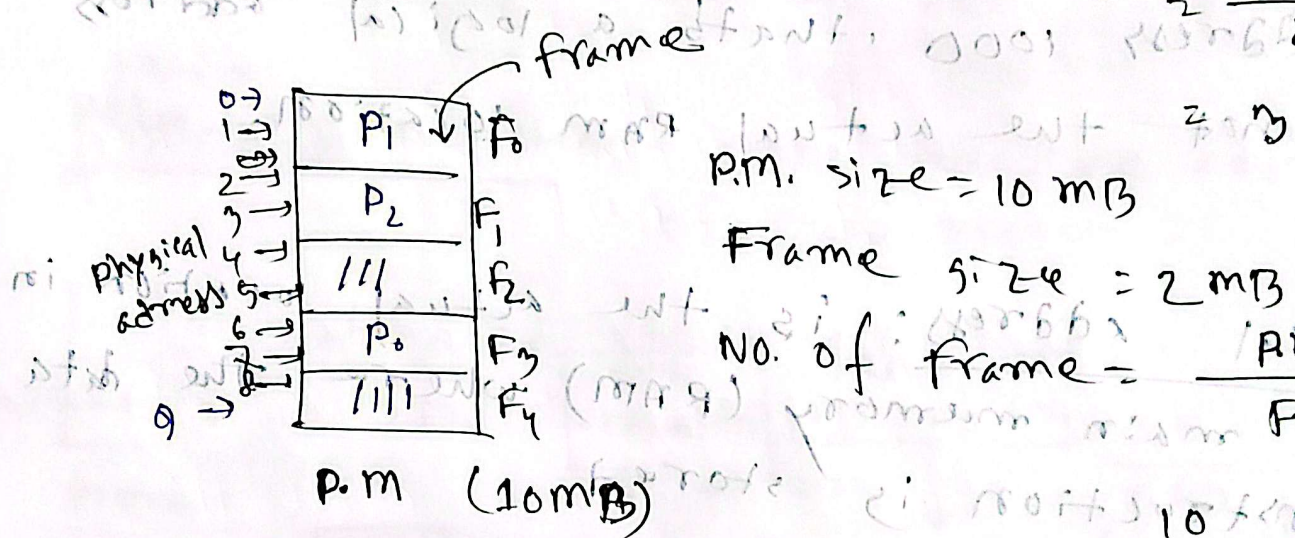
Paging (Non-continuous)



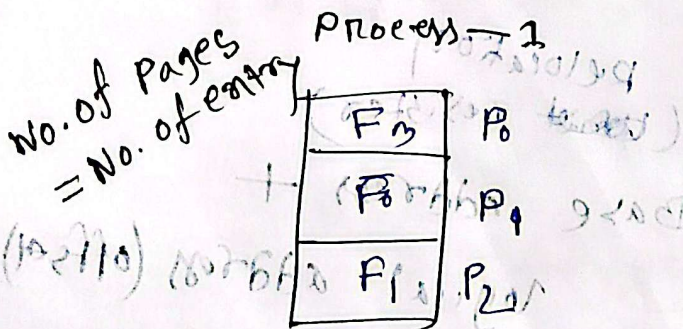
Process - 6MB

Page size - 2MB

$$\text{No. of Page} = \frac{\text{Process size}}{\text{Page size}} = \frac{6}{2} = 3$$



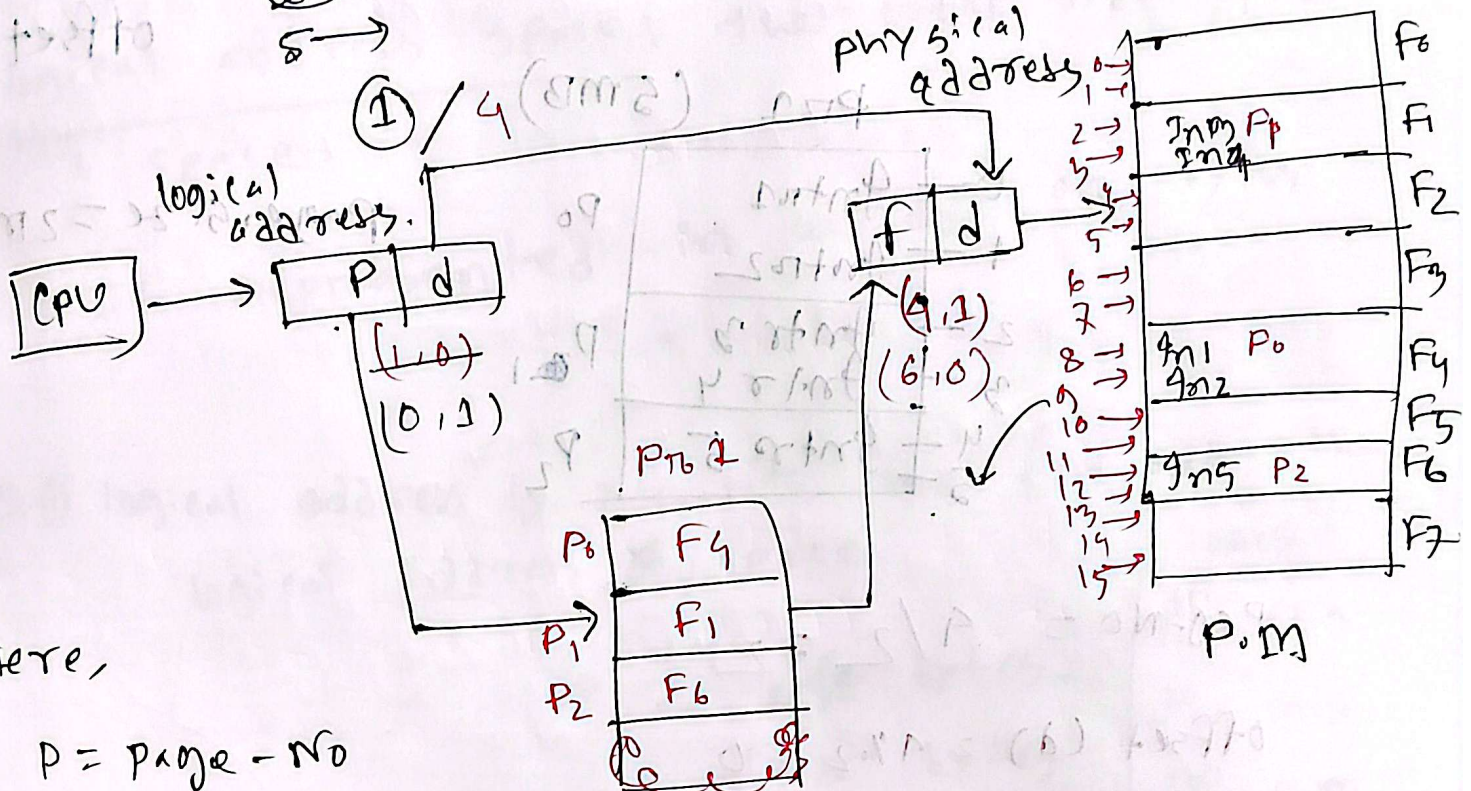
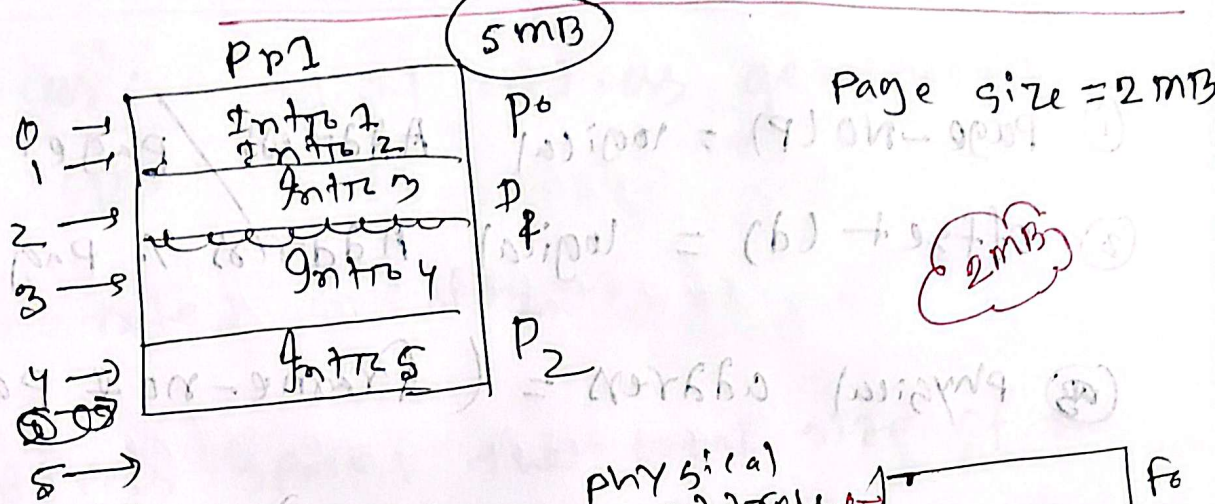
$$\text{No. of frame} = \frac{\text{P.M. size}}{\text{Frame size}} = \frac{10}{2} = 5$$



Page table

Process	Page	Frame
P ₀	0	F ₀
P ₁	1	F ₁
P ₂	2	F ₂

Paging hardware Architecture



Here,

P = page - No

d = offset

f = frame - No

$$P = \frac{1}{2} = 0 (P)$$

$$d = 1 \times 2 = 2$$

$$P.A = (4 \times 2) + 1 = 9$$

or

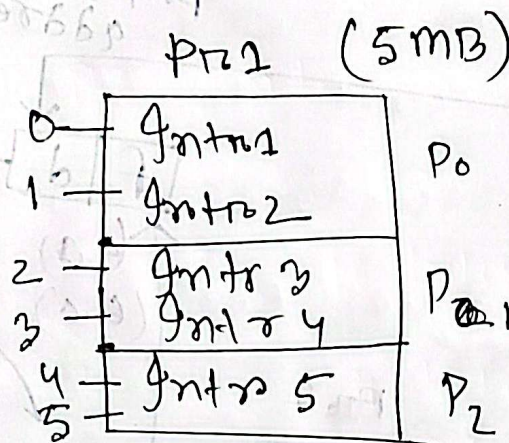
P.A = Base address of frame + offset

$$8 + 1 = 9$$

logical address to physical Address

- ① Page-NO (P) = logical Address / Page Size
- ② Offset (d) = logical Address % Page Size

③ Physical address = (frame-no × page size) + offset



Page size = 2MB

PAG-NO = $4 / 2 = 2$

Offset (d) = $4 \% 2 = 0$

(Pr1) location P₂ @ 0 location Intro 5 any

P.A = $(6 \times 2) + 0 = 12$

P.M 12 physical address @ Intro 5 any

logical address to logical address space

logical address :- logical address generated by the CPU.

→ It generated in bits.

logical address space :- the total size of the space.

→ It represented in words or bytes.

Ex: ① logical address is 24 bits, now find out the logical address space.

$$\begin{aligned} & \rightarrow 2^{24} \text{ bytes} \\ & = 2^4 \times 2^{20} \text{ bytes} \\ & = 16 \text{ MB} \end{aligned}$$

bytes

$$\begin{aligned} 2^{10} &= 1 \text{ KB} \\ 2^{20} &= 1 \text{ MB} \\ 2^{30} &= 1 \text{ GB} \\ 2^{40} &= 1 \text{ TB} \end{aligned}$$

② logical address space 8GB,

$$\begin{aligned} \text{logical address} &= 2^3 \times 10^{30} \text{ bytes} \\ &= 2^{30} \text{ bytes} \end{aligned}$$

2³⁰ bytes = 8GB

$$= 30 \text{ bits}$$

(2³⁰ to 30 bits)

Physical address to physical address space.

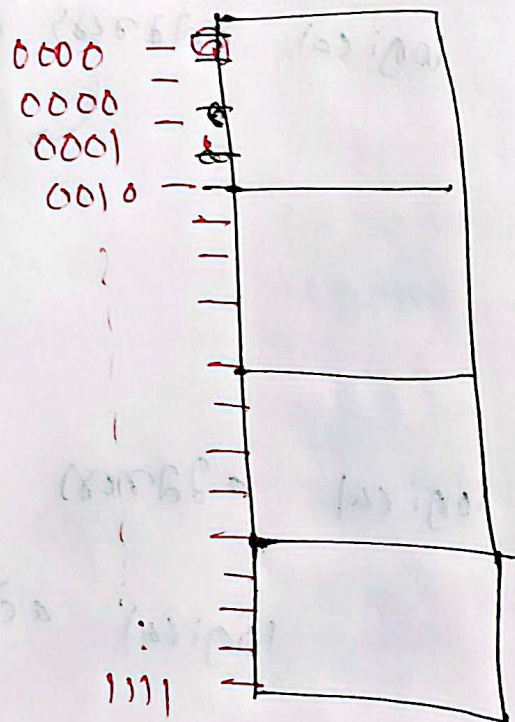
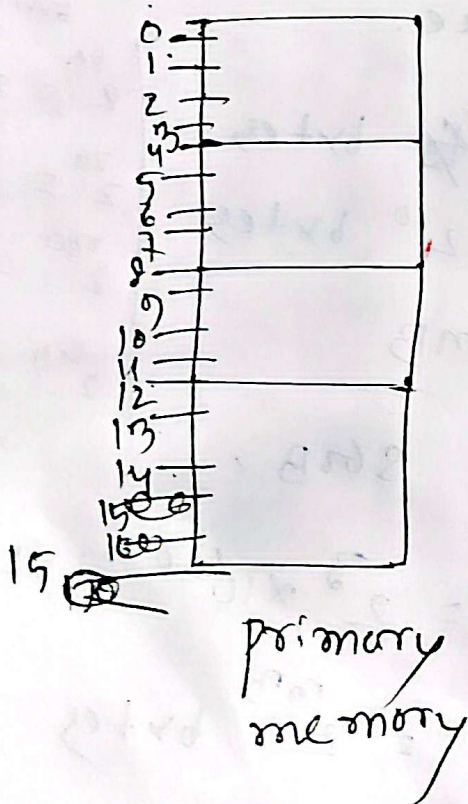
P.A: The address of the primary memory
→ represented by bits.

P.A.S: The total size of the primary memory
→ represented in bytes or words.

Ex:

P.M. size = 16 bytes

Frame $n = 4$ bytes.



$$2^n = 16$$

$$n = 4 \text{ (no. of bits)}$$

4 bits,

$$\begin{aligned} \text{P.A.S} &= 2^4 \text{ bytes} \\ &= 16 \text{ bytes} \end{aligned}$$

$$\checkmark 16 \text{ MB} = 2^4 \times 10^{20} \text{ bytes} = 2^{24} \text{ bytes} = 2^4 \text{ (bits)}$$

Ex: P.A is 32 bits, P.A.S = 2^{32} bytes
 $= 2^2 \times 2^{30}$ bytes
 $= 4 \text{ GB}$

Ans.

Ex: P.A.S is 16 GB, P.A = 16 GB
 $= 2^4 \times 10^{30}$ bytes
 $= 16 \times 2^{34}$ bytes
 $= 34 \text{ bytes}$

calculate no. of page & no. of Frame

Q. Here a process size is 8MB & physical address space is 16MB. If the page size is 4KB then calculate total no. of page and total number of frames.

Soln:

$$\text{Process size} = 8 \text{ MB}$$

$$\text{Physical address space} = 16 \text{ MB}$$

$$\text{Page size} = 4 \text{ KB}$$

$$\text{No. of Page} = \frac{\text{process size}}{\text{page size}}$$

8 MB

4 KB

$$= \frac{2^{10} \times 8 \text{ MB}}{4 \times 2^{10} \text{ MB}} = \frac{8 \times 2^{20} \text{ bytes}}{4 \times 2^{10} \text{ bytes}}$$

$$= \frac{2^{10}}{2^{12}} = 2^{11} \text{ Ans}$$

Ans:

$$\text{No. of frame} = \frac{\text{P. A. S}}{\text{frame size}}$$

frame size = page size

P. A. S

Page size

16 MB

4 KB

$$2^4 \times 2^{20} \text{ bytes}$$

$$2^2 \times 2^{10} \text{ bytes}$$

$$2^4$$

$$2^{12}$$

$$2^{12}$$

Ans:

Q1) logical space address is 2^3 bytes. and
~~P.A.S~~ physical memory size 16MB.
 If the frame size 4KB. Then calculate
 total no. of page and total no. of
 frame.

Solⁿ:

$$\text{No. of page} = \frac{\text{L.A.S}}{\text{Page size}}$$

$$= \frac{2^3 \text{ bytes}}{4 \times 2^{10} \text{ bytes}}$$

$$= \frac{11}{2}$$

Ans:

Given,

$$\text{L.A} = 2^3 \text{ bits}$$

$$\text{L.A.S} = 2^3 \text{ bytes}$$

$$\text{P.M.S} = 16 \text{ MB}$$

$$\text{Page size} = \text{frame size} = 4 \text{ KB}$$

$$\text{No. of frame} = \frac{\text{P.M.S}}{\text{frame size}}$$

$$= \frac{16 \text{ MB}}{4 \text{ KB}}$$

$$= \frac{2^4 \times 2^{20} \text{ bytes}}{2^2 \times 2^{10} \text{ bytes}}$$

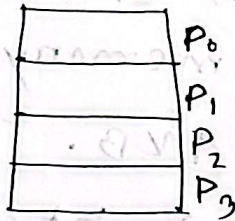
$$= \frac{2^2 \times 2^{10} \text{ bytes}}{2^2 \times 2^{10} \text{ bytes}}$$

$$= \frac{12}{2}$$

Ans:

3

P_{n-1} (8 bytes)



page size = 2 bytes
represent

P_2	F_0	— 000
P_1	F_1	— 001
P_0	F_2	— 010
P_1	F_3	— 011
	F_4	— 100

p.m (10 bytes)

$$\text{no. of frame} = \frac{10}{2} = 5$$

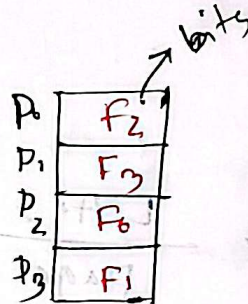
$$= 2^n = 2^3$$

$$L.A.S = 8MB$$

$$P.A.S = 36 \text{ bits}$$

$$P.S = 20 \text{ bits} \oplus 1MB$$

Page table size = ?



Page table.

Formula:

$$\text{no. of entries} = \frac{\text{page table size}}{\text{entry size}}$$

$$\Rightarrow \text{page table size} = \text{no. of entries} \times \text{entry size}$$

5 bit Frame for p.m
(6 represent no. of
total no. of
frame bits = 3

$$\text{no. of frame} = \frac{P.A.S}{F.S} = \frac{36}{20} = 1.8$$

$$= 2^3 \times 16$$

$$= 2^3 \times 2$$

$$= 2^4 \text{ bytes}$$

$$\text{no. of Page} = \frac{L.A.S}{P.S} = \frac{8MB}{1MB} = 8$$

$$= \frac{2^3}{2^20} = 2^3 \text{ (no. of entries)}$$

(Translation lookaside Buffer)

Paging theory

Paging is a memory management technique in which both ~~main~~ ^{physical} memory and ~~process~~ ^{logical memory} are divided into fixed size blocks.

- ~~process~~ ^{logical memory} blocks are called pages.
- ~~memory~~ ^{physical} blocks are called frames.

The main goal of paging is to eliminate external fragmentation and to allow non contiguous memory allocation.

when a process is loaded into memory:

→ It is divided into small pages.

→ The main memory is divided into small frames (same size as pages)

→ Each page is placed into any free frame in memory.

Page Table:

Each process has a page table that keep tracks of which page is stored in which frame.

0	F ₅
1	F ₉
2	F ₂
...	...
n	F ₆

that means page 1 is stored in:

Frame F₉, page n is stored in
Frame F₆ etc.

Advantages:

- No external fragmentation
- Easy to manage memory
- Each page can be loaded separately

→ supports virtual memory

Disadvantages:

- Internal fragmentation (unused space inside a page)
- Extra memory needed for page table
- Extra time for address translation

TLB

Translation Lookaside Buffer.

- TLB is a special small cache memory used in paging systems to speedup virtual address to physical address translation.
- Page table look up in main memory are slow
- TLB stores recently page table entries so the CPU can access them much faster.

Why TLB Need

In a paging system:

1. CPU generates a virtual address.
2. It must be translated into a physical address using the page table.
3. Page table resides in main memory, so lookup takes extra time.

TLB solve this problem by caching most recently accessed page table entries.

Working:

1. CPU generates a virtual address.
2. Check the TLB to see if the page number exists.

TLB Hit $\rightarrow$ physical frame number found.
No need to access page table.

That's why faster.

TLB miss $\rightarrow$ physical frame number not found. Then go to page table.
Then update TLB.

TLB Types:

① Fully associative TLB: $\underline{\hspace{1cm}}$

- $\rightarrow$ any page no. can go in any TLB entry
- $\rightarrow$ most flexible, faster for small TLB

② Set Associative TLB

- $\rightarrow$ TLB divided into sets, each page maps to a set

$\rightarrow$ balance between speed and complexity

③ Direct mapped TLB:—

→ Each page number to exactly one TLB entry.

→ Simple but can cause collision.

TLB location:

→ Usually inside CPU as part of MMU (memory management unit)

→ It's a hardware cache — very fast compared to main memory.

Advans:

① ~~can~~ speedup virtual → physical address

② Reduces page table access time.

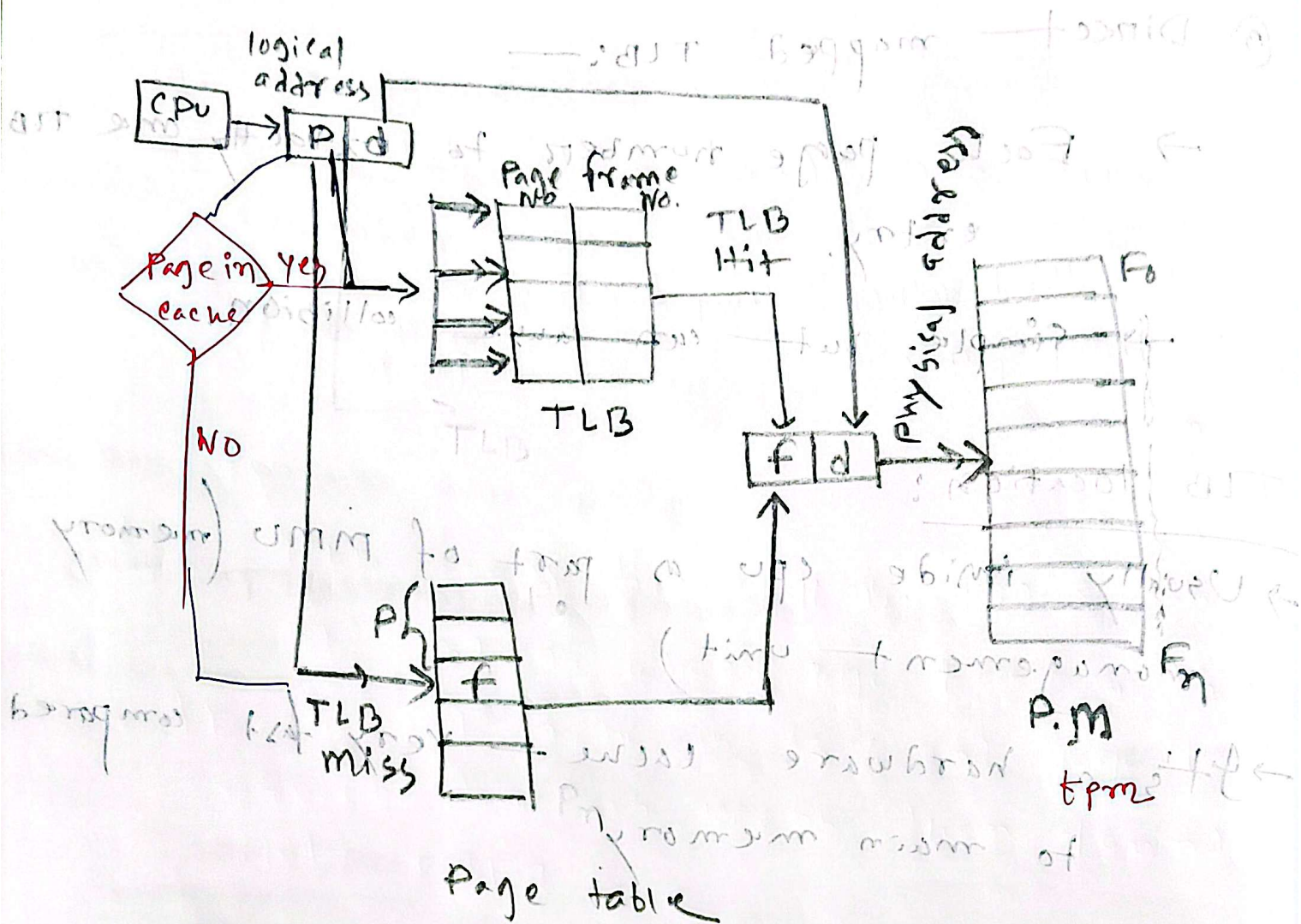
③ Improves overall CPU performance.

Dis:

① Limited size — cannot store all page

② If TLB miss occurs → extra delay

③ Hardware ~~cost~~ cost.



$$E.A.T = \text{hit} (T_{TLB} + t_{pm}) + \text{miss} (T_{TLB} + t_{pm} + t_{pm})$$

$$= \text{hit} (T_{TLB} + t_{pm}) + \text{miss} (T_{TLB} + 2t_{pm})$$

$$= h(T_{TLB} + t_{pm}) + \text{miss} (T_{TLB} + 2t_{pm})$$

(1-h) TLB Hits

$$TLB \text{ Hit rate} = \frac{TLB \text{ Hits}}{\text{Total memory access}}$$

Note:

① If TLB hit ratio is increased then EAT will be decreased.

② If TLB hit ratio is decreased then EAT will be increases.

Ex: TLB access time 25 ns, primary memory access time is 150 ns. If hit ratio is ~~150 ns~~ 70%. then find out EAT?

we know,
$$EAT = h(T_{TLB} + t_{pm}) + miss(T_{TLB} + 2t_{pm})$$

$$= (0.7)(25 + 150) + (0.3)(25 + 2 \times 150)$$

$$= (0.7)(175) + (0.3)(325)$$

$$\therefore EAT = 220 \text{ ns}$$

Ans:

without TLB total access time

$$= 2 \times t_{pm}$$

$\therefore 220 < 300$
that's why TLB effective $= 2 \times 150 = 300 \text{ ns}$

TLB Page fault

$$EAT = \text{hit} \left(T_{TLB} + t_{pm} \right) + \text{miss} \left(T_{TLB} + t_{pm} + \text{Page fault \& service time} + \text{Page fault not found} \times t_{pm} \right)$$

Ex: TLB access time 40 ns, P.M. access time 137 ns where page fault is 45%.

If the hit ratio is 65% and page fault service time is 1000 ns then find out EAT .

$$EAT = \text{hit} \times (T_{TLB} + t_{pm}) + (1 - \text{hit}) \times (T_{TLB} + t_{pm} + \text{Page fault \& service time} + \text{Page fault not found} \times t_{pm})$$

$$= 0.65 \times (40 + 137) + (1 - 0.65) \times (40 + 137 + 1000 + 137 \times 0.45)$$

$$= 0.65 \times 177 + 0.35 \times 1261$$

$$= 116.05 \text{ ns}$$

TLB access time = 40 ns

Page fault service time = 1000 ns

Page fault not found = 137 ns

~~multi-level~~ / ~~Paging~~

Segmentation

Segmentation

① Memory Management

where logical memory is divided into variable sized segments.

② memory block size — variable size (each segment can be different)

③ can occur

④ NO occur

⑤ Segments represent program parts (code, data, stack)

⑥ Address translation: segment no. + offset

⑦ Table used: segment table $\rightarrow$ base + limit

Paging

① Memory Management where logical memory is divided into fixed-size pages.

② Fixed size (each page and frame can be same size)

③ external Fragmentation is avoidance

④ can occur (last part of page may be unused)

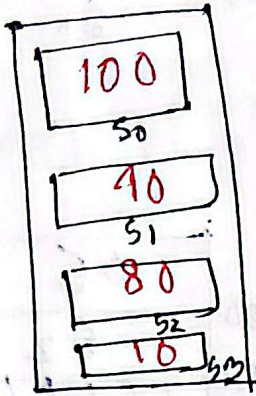
⑤ page are fixed size block of the program

⑥ page no + offset

⑦ Page table $\rightarrow$ Page $\rightarrow$ frame mapping.

segmentation

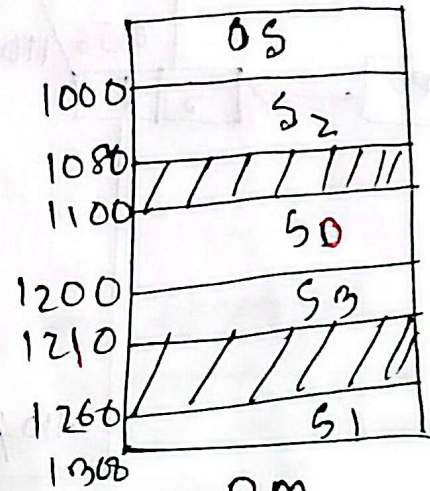
Process



Base Address
↓
Limit

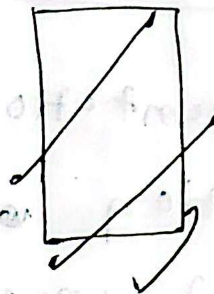
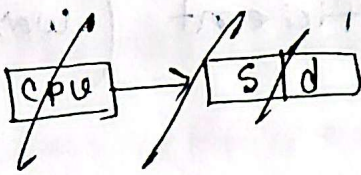
S0	1100	100
S1	1260	40
S2	1000	80
S3	1200	10

segment table



P.M.

$$(1200 - 1100) = 100$$



logical address to physical address

segment no	Base Address	limit
0	1500	300
1	500	250
2	1400	70
3	1200	150
4	150	50

logical address	$d < \text{limit}$	Base address + offset	physical address
0: 280	$280 < 300$	$1500 + 280 =$	1780
1: 252	$252 < 250$	$500 + 252 = 752$ error	752 error
2: 180	$180 < 250$	$500 + 180$	680
3: 65	$65 < 70$	$1400 + 65$	1465
3: 160	$160 < 150$		
4: 40	$40 < 50$	$150 + 40$	190

Segmentation disadvantage

- External Fragmentation occurs.
- memory utilization is inefficient (wastage of unused space)

(1) External Fragmentation :-

- Although segmentation avoids internal fragmentation, it suffers from external fragmentation because free memory is scattered in different places.
- As segments vary in size, it becomes difficult to find a contiguous block large enough to hold a new segment.

(2) Complex memory management :-

- Keeping track of variable sized segments and managing their allocation/deallocation is more complex than paging.
- The segment table is more complicated than a simple page table.

(3) ✶ Difficult in swapping :—

since segments are of different sizes, swapping them in and out of memory is harder and less efficient compared to fixed-size pages.

(4) Address translation slower :—

logical to physical address translation requires two memory access (one to the segment table one to the actual address), making it slower than simple paging.

(5) Overhead of segment table :—

- Each process needs a segment table that stores the base and limit of every segment.
- This table consumes extra memory and must be accessed for every memory reference (unless cached with hardware support).

Paging with segmentation

~~What~~ does segmentation: segmentation provide a program into logical parts, such as:

→ code segment

→ Data "

→ stack "

→ Heap "

Each segment has different size based on the program's logical structure.

✓ Segmentation = logical division of memory

~~What~~ does paging does:

Paging divides memory into fixed-size blocks.

→ logical memory → pages

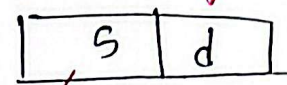
→ physical " → frames

✓ Paging is fixed-size blocks to avoid external fragmentation.

working:

step-01: segmentation

A logical address is given as



offset
inside
the
segment

segment
No.

OS check the segment table and finds the base address of the page table for that segment.

step-02: Paging

Each segment has its own page table.

Offset d is divided into

• page no (P)

• page offset (O)

physical address = frame no. + offset (O)

Why use:

segmentation advantages:

- logical organization
- Different protection for each segment (code = read-only)

Paging advan:

- No external fragmentation
- Easy memory allocation

combine Benefits:

- logical separation & efficient physical memory use
- Better protection
- large program handled easily
- No external fragmentation

Real-life:

A book (program)

→ chapters = segments

→ Each chapter broken into equal-sized pages = paging

Advantage:

→ logical program structure retained

→ Efficient physical memory use

→ No external fragmentation

→ Strong protection.

→ Good for large program

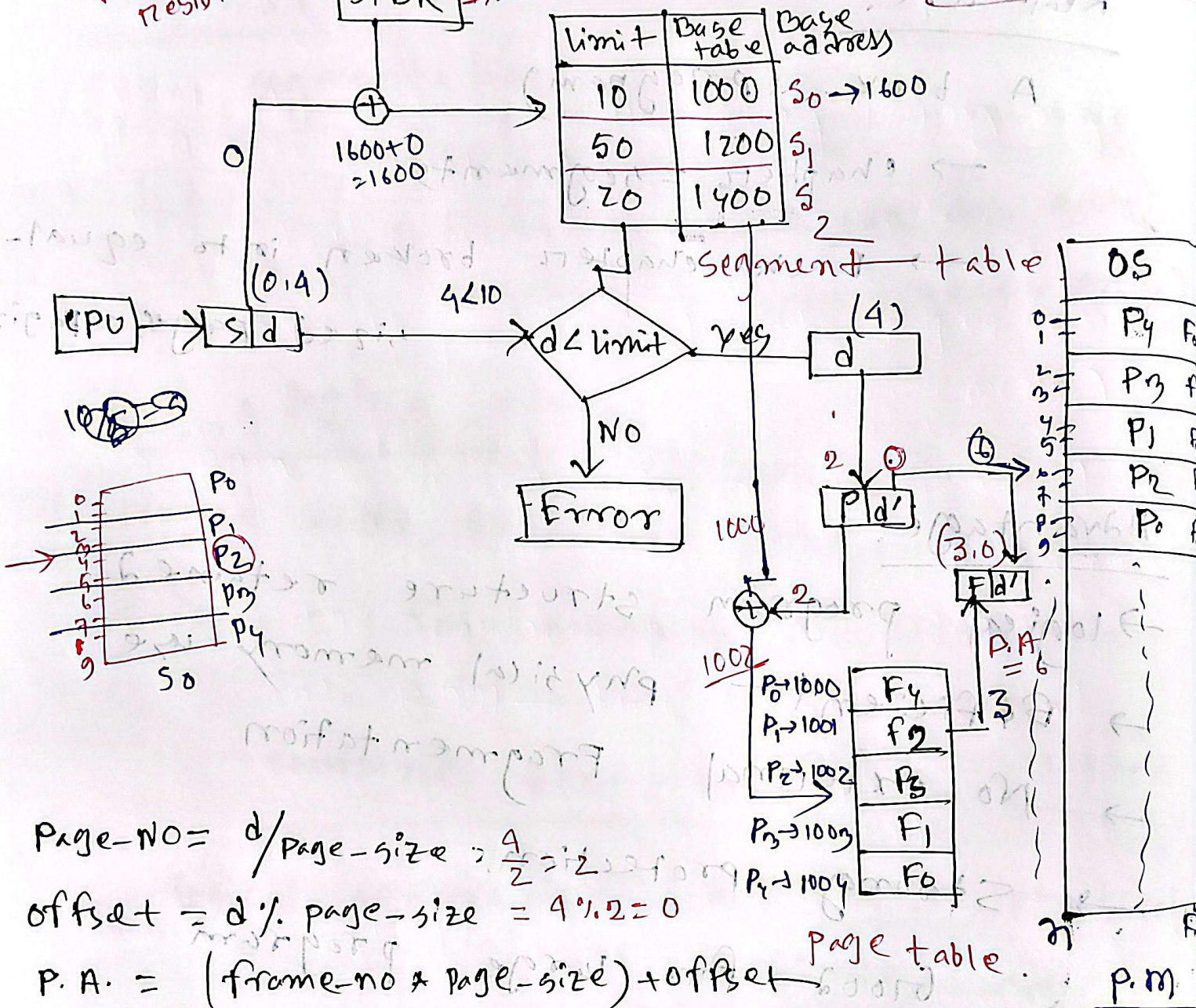
Dis:

→ address translation becomes complex

→ more memory needed for multiple page table.

segment base register → STBR → 1600

Page-size = 2



$$\text{Page-NO} = d / \text{Page-size} = \frac{4}{2} = 2$$

$$\text{offset} = d \% \text{Page-size} = 4 \% 2 = 0$$

$$\text{P.A.} = (\text{frame-no} \times \text{Page-size}) + \text{offset}$$

$$= (2 \times 2) + 0$$

$$= 4$$