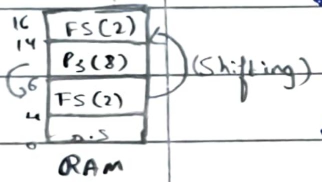
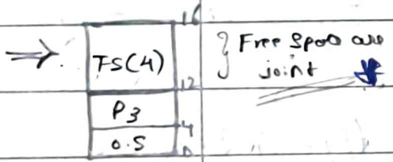
Free Space Management

1). **Defragmentation / Compaction :-**



⇓ Defragmented RAM



* It is Changing 'Relocation Register Base value' from 6 to 4

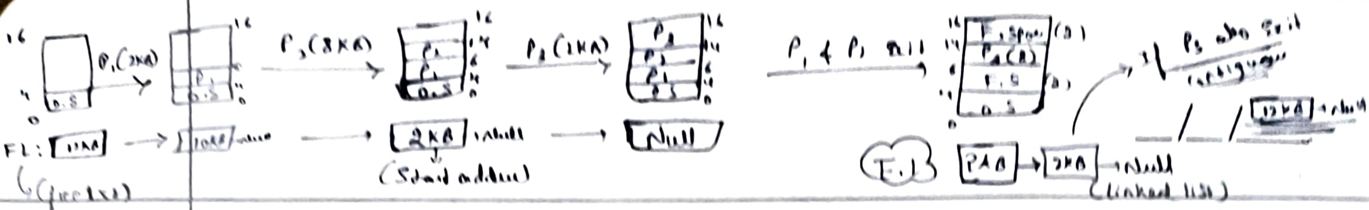
Now, P_5 of 3 KB Can be alloted

- * Dynamic Partitioning Suffers from External fragmentation. (Free Spaces are not Contiguous)
- * Compaction to minimize the Probability of External fragmentation.
- * All the free Partitions are made Contiguous, and all the loaded Partitions are brought together.
- * By applying this technique, we Can Store the bigger Processes in the memory. The free Partitions are merged which Can now be allocated according to the needs of new Processes. It is Called "Defragmentation".
- * the efficiency of the System is ↓ced in the Case of Compaction Since all the free Spaces will be transferred from Several Places to Single Place.

Limitation → overhead

2). How free Space is Stored / Represented in OS?

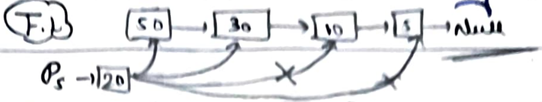
Ans Free holes in the memory are represented by a free List (Linked-List Data Structure).



3) How to Satisfy a request of a 'n' Size from a list of free holes?

Sol→ Various algorithums which are implemented by OS in order to find out the holes in the linked list and allocate them to the Processes:-

(to fit Process in free list)



a) First Fit :

- * allocate the 1st hole that is big enough.
- * Simple and easy to implement.
- * Fast / Less time Complexity. (+ve)

*) Request :<90 KB>

↓ alloted

*) FL:- 50 → 100 → 90 → 200 → 10 → Null

*) New FL: 50 → 10 → 90 → 200 → 10 → Null

b) Next Fit :

- * Enhancement on 1st fit but Starts Search always from allocated hole.
- * Same advantages of 1st fit.

*) Request <90 KB>

*) F.L: 50 → 90|10 → 90 → 200 → 50 → Null

↗ It will start Searching from here for next allotment (P2 → 110 KB)

c) Best Fit :

- * Allocate Smallest hole that is big enough
- * Lesser internal Fragmentation (+ve)
- * May Create many Small holes and Cause major External fragmentation
- * Slow, as required to itterate whole 1st holes list

*) Leaves Least Internal fragmentation

*) Req = 90 <here>

NL : 50 → 100 → 90 → 200 → Null

↓

<Node out>

d) Worst Fit :

- * Allocate the largest hole that is big enough
- * Slow, as required to iterate whole free holes list
- * Leaves larger holes that may accommodate Other Processes.
- * Lesser External fragmentation (+ve)

*) for 200 it will be alloted

↳ 90|110

200

__/__/__

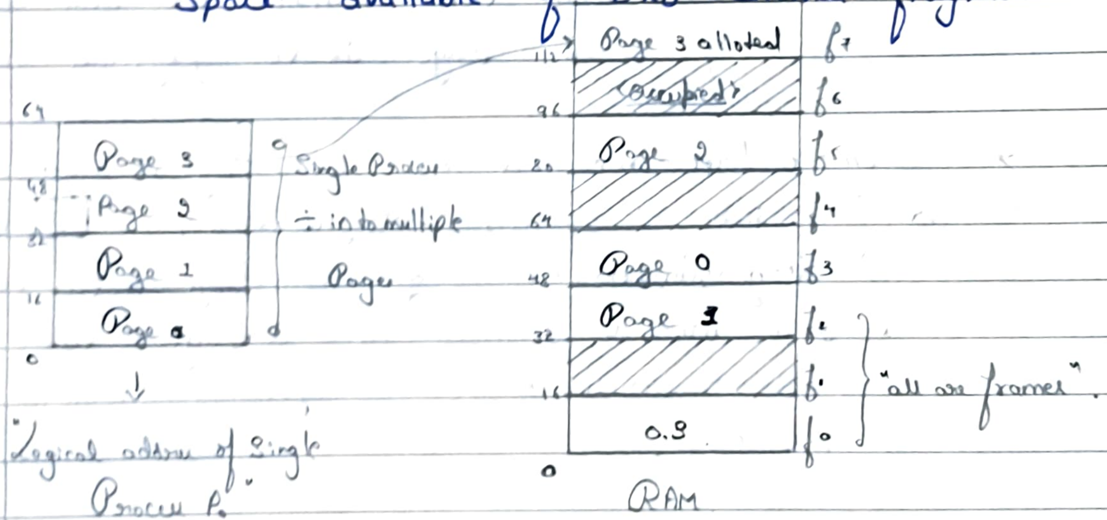
"Paging / Non-Contiguous Memory Allocation".

→ The main disadvantage of Dynamic Partioning :-

- ★ Is External Fragmentation.
- ★ Can be removed by Compaction, but with overhead
- ★ we need more dynamic / flexible / optimal mechanism, to load Process in the Partition.

→ Idea Behind Paging :-

- ★ If we have only 2 Small non-contiguous free holes in the memory, say 1KB each.
- ★ If OS wants to allocate RAM to a Process of 2 KB, in Contiguous allocation, it is not possible, as we must have Contiguous memory Space available of 2 KB (External fragmentation)



→ Paging :-

- ★ Paging is a memory-management Scheme that Permits the physical address Space of a Process to be 'non-contiguous'.

_/_/_

★ It avoids External Fragmentation and need of Compaction.

★ Idea is to divide physical memory into fixed-Sized blocks Called 'Frames', along with divide logical memory into blocks of Same Size Called 'Pages'.
(## Page Size = frame Size)

⊙ Page Size: It is usually determined by the Processor architecture. Traditionally, Pages in a System had uniform Size, Such as 4,096 bytes. However, Processor designs often allow 2 (or) more, Sometimes Simultaneous, Page Size due to its benefits.

→ It is Stored in 'memory' ⊙ Each Process has its 'Pagetable'

⊙ Page Table:

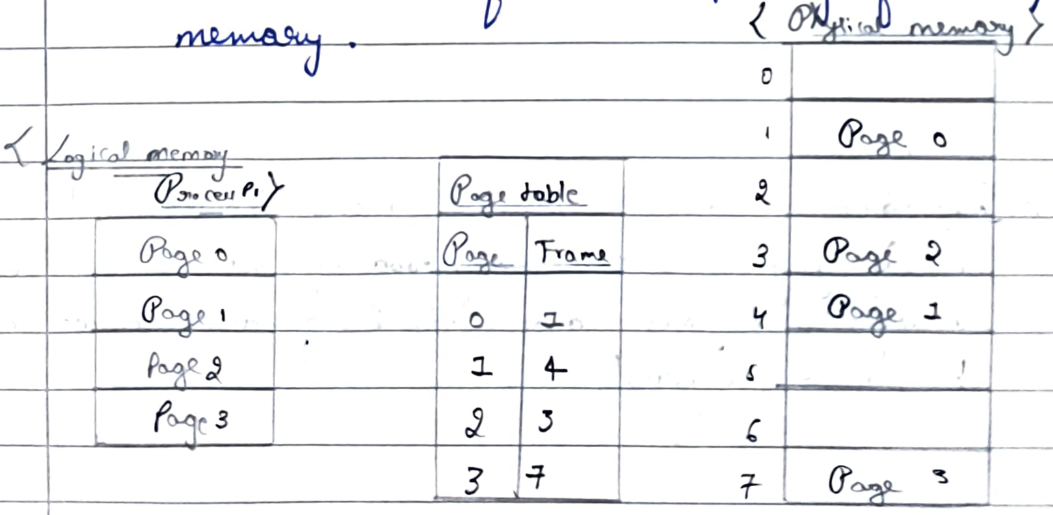
★ A data Structure Stores which Page is mapped to which frame.

★ The Page table Contains the base address of each Page in the physical memory.

★ Every address generated by CPU (logical address) is divided into 2 Parts: ⊙ Logical address Space :- $\boxed{P \mid d}$

i) Page Number (P) and ii) Page offset (d).

★ P is used as a index into a Page table to get base address of the Corresponding frame in physical memory.



{Logical memory (Process P1)}

Page 0
Page 1
Page 2
Page 3

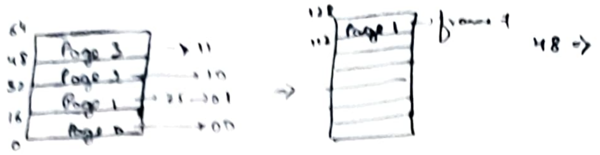
Page table

Page	Frame
0	1
1	4
2	3
3	7

{Physical memory}

0	
1	Page 0
2	
3	Page 2
4	Page 1
5	
6	
7	Page 3

{Paging model of logical & physical memory}



__/__/____

* Logical Address of P₀ → 64 bytes ∴ $2^6 = 64$ ∴ 6 bits are required

Eg 6 bits => 25 => 011001
(P) <Page No> ⓟ
offset (d) ⓓ
offset base address (9)
(16 + 9 => 25)
(Page No base) (offset)

⊙ In Physical address; 121 => 111001 ∴ 112 + 9 => 121
{it will only vary} ← (frame 7) d (it will be same)

*) Logical Address → 01 | 1001

*) Physical Address $\boxed{\text{Address}}$ 111 | 1001 Same

(to allocate this Page table is used)

⊙ Page table is stored in main memory at the time of Process Creation and its base address is stored in PCB.

⊙ * Page table base register (PTBR) is Present in the System that Points to the Current Page table. Changing Page tables requires only this register, at the time of Context-Switching.

Q) How Paging avoids External Fragmentation?

Ans) Non-Contiguous allocation of the Pages of the Process is allowed in the random free frames of Physical Memory.

Q) Why Paging is Slow and how do we Make it fast?

Ans, There are too many memory references to access the desired location in physical memory.

↓

*) We Can't reach a physical address directly; 1st after knowing Logical address, we Should go with Page table to known offset and later we can reach reach desired location on physical address (RAM)

↳ To overcome this we use 'TLB'.

★ Internal Fragmentation in Paging :-

* Program → 15 KB : Then at Page
* Page Size → 2 KB : (can't be [illegible])

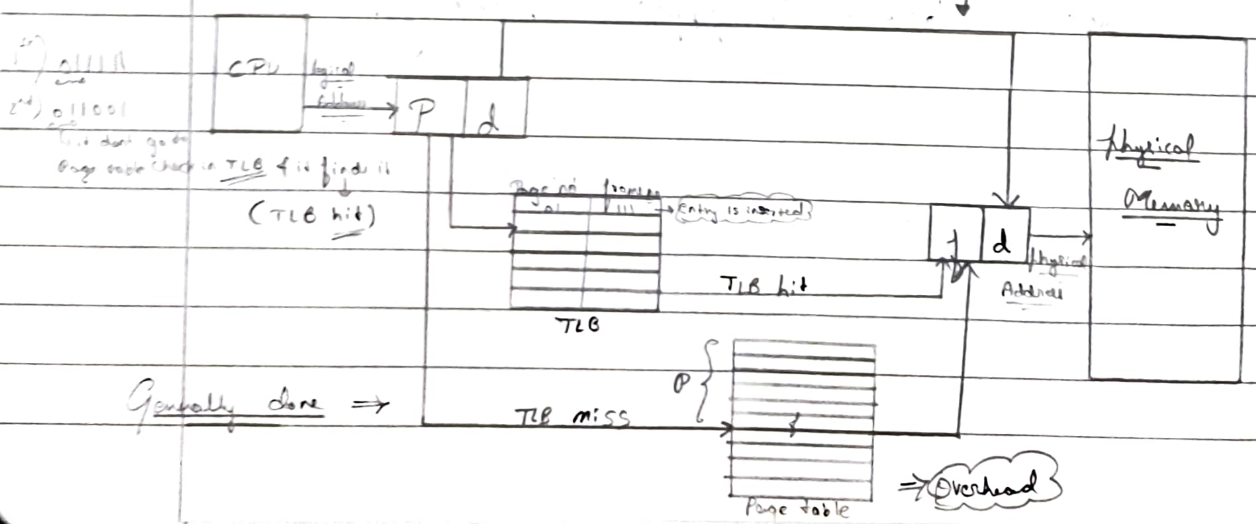
x thus Cause internal fragmentation of 1 KB

___/___/___

⇒ TLB {Translation Look-aside buffer (TLB)}.

- ★ A hardware Support to Speed up Paging Process.
- ★ It's hardware Cache, high Speed memory.
- ★ TBL has key and Value.
- ★ Page Table is Stored in main memory & because of this when the memory reference is made, the translation is Slow.
- ★ when we are retrieving physical address using Page table, after getting frame address Corresponding to the Page Number, we put an entry of that into TLB. So that next time, we can get the values from TLB directly without referencing actual Page table. Hence, make Paging Process faster.
- ★ 'TLB hit' if TLB Contains the mapping for the requested logical address.
- ★ Address Space Identifier (ASID's) is Stored in each entry of TLB. ASID uniquely identifies each Process & is used to Provide address Space Protection & allow to TLB to Contain entries for Several different Processes.
- ★ when TLB attempts to resolve Virtual Page numbers, it ensures that the ASID for the Currently Executing Process matches ASID associated with virtual Page. If it doesn't match, the attempt is treated as TLB miss.

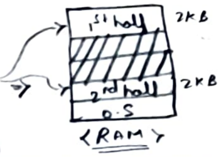
Paging Hardware with TLB



(Paging) → (Problem)

* Pages are ÷ into fixed Size bits.

Eg. fun()
{ Code →
}
(4KB)



∴ It Causes Overhead of time delay
(we use Segmentation to __/__/___ Overcome this)

★ TLB accesstime of Cost >> Main memory accesstime of Cost.,

★ whenever there is a Context Switch we need to flush the previous TLB So as to maintain Security.

I 1st method ★ TLB is reset after Context Switching (But it is Cost effective).

II 2nd method ★> So we use unique Identifier that will identify unique Processes.

	ASID	Pg. no	Frame no
P₁ →	0	10	100
P₁ →	1	10	101

" if P₁ of 10 enter " → it will be missed due to mismatch in ASID

★> TLB Can have multiple entries of multiple Process with "ASID"

" Segmentation | Non-Contiguous Memory Allocation " (See up)

★ An important aspect of memory management that become Unavoidable with Paging is Seperation of user's view of memory from the actual physical memory.

⊙ Segmentation is memory management technique that Supports the user view of memory.

★ A logical address Space is a Collection of Segments, these Segments are based on 'user View' of logical memory.

⊙ Each Segment has Segment number and offset, defining a Segment. <Segment-number, offset> {s, d}

★ Process is ÷ into 'variable Segments based on user view'.

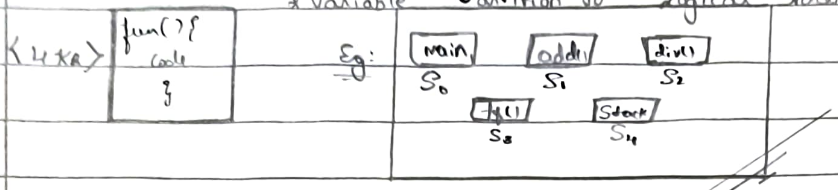
★ Paging is Closer to OS rather than user. It divides all the Processes into the form of Pages although a Process Can have Some relative Parts of funⁿ which need to be loaded in the Same Page.

__/__/___

★ Segmentation.

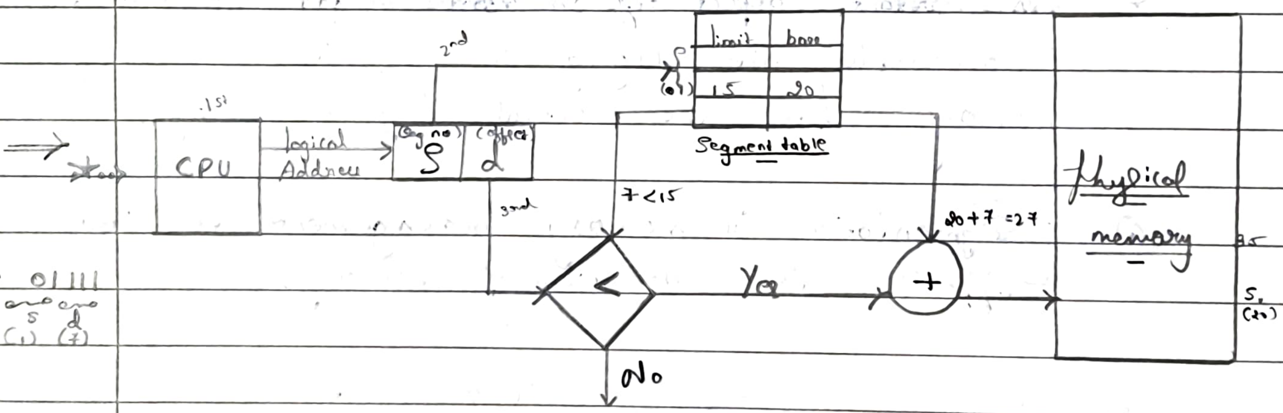
* diff. Segments * Varying Sizes * users view Priority

* Variable Partition to Logical Address Space



★ O.S doesn't Care about user's view of the Process. It may divide the Same funⁿ into different Pages and those Pages may (or) may not be loaded at the Same time into the memory. It ↓ces the efficiency of System.

★ It is better to have Segmentation which divides the Process into Segments. Each Segments contains the Same type of funⁿ Such as the main function Can be included in one Segment and the library functions Can be included in other Segment.



Trap : addressing Error

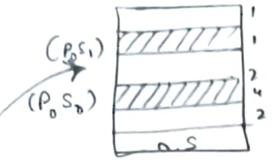
⊙ Advantages :-

1) No internal fragmentation

2) Size of Segment table < Size of Page table

3) 1 Segment has a Contiguous allocation, hence efficient working within Segment.

4) It results in more efficient System because the Compiler keeps the Same type of funⁿ in one Segment.



< P1 → S0 → 1 KB >
S1 → 3 KB (cannot be alloted)

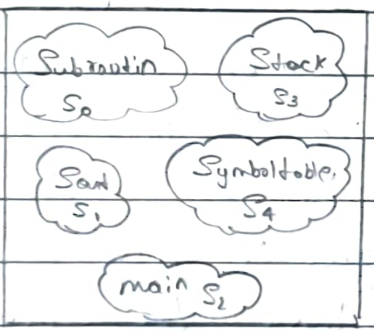
① **Disadvantages :-**

a> External fragmentation

⇒ The different Size of Segment is not good that the 'time of Swapping'.

*) Modern System architecture Provides both Segmentation and Paging implemented in some hybrid approach.

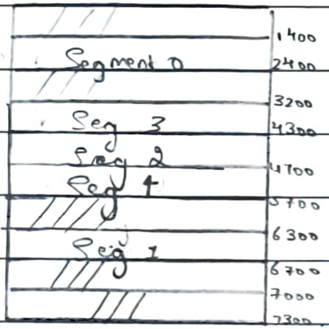
Address Space



Segment table

Limit	Base
1000	1400
400	6300
400	4300
1100	3200
1000	4700

Physical memory



Eg. S_2 : 4300 + 53 = 4353
< Logical Address >