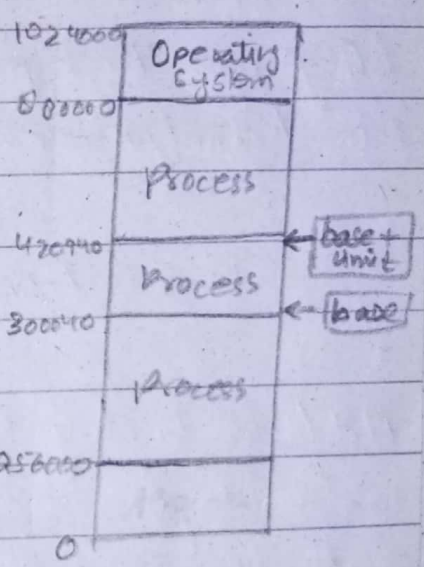
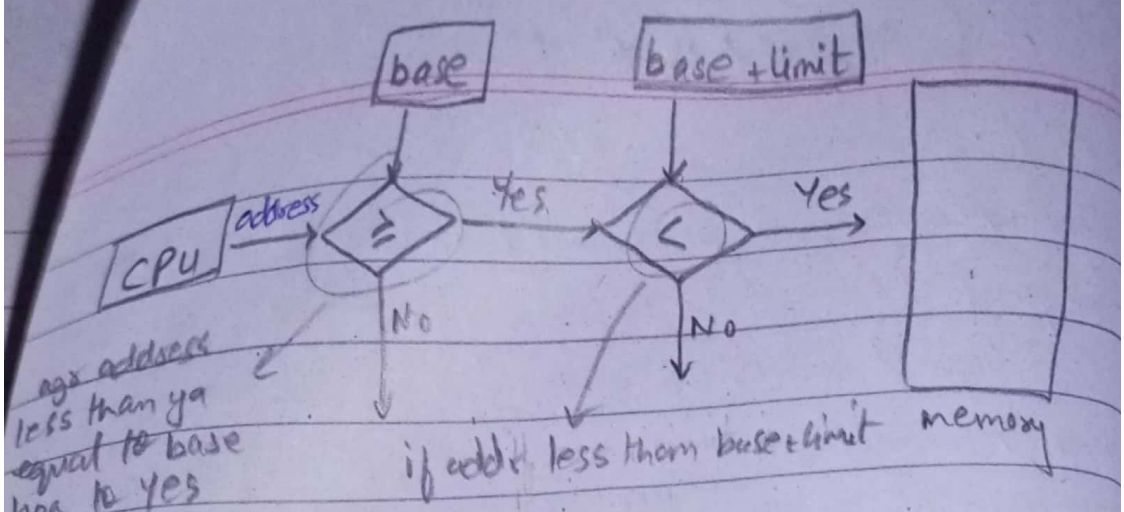
Main Memory
Chapter # 09

→ Agr CPU ko main memory se data load krke apne pass lana hota hai to usmein 1 clock se zyada time lagsakta hai due to transfer of data thru memory bus.
→ Jab tak data nhi ata CPU ko stall (wait) krna parega.
→ Isi fast acess k liye "cache" built hoti hai.
→ We need to ensure that each process acess only its own memory.
→ "Seperate-per process" memory space protects the process from each other. Or is k saath hum ek waqt mein multiple process ko memory mein store kr sakte hai.
→ The protection is provided by two registers.



- Base register: holds the smallest legal physical memory address.
- Limit Register: Specifies the size of the range.

Example:
Base = 300040 & Limit = 120900.
Now the range for memory is (300040 - 420939) Inclusive.



→ Agr koi bhi user program in meinse ko bhi condition fail krta hai to OS will issue a "trap".

→ The base & limit register can only be loaded by OS. They are high priviledge instr., only be executed in Kernel mode.

"ADDRESS BINDING"

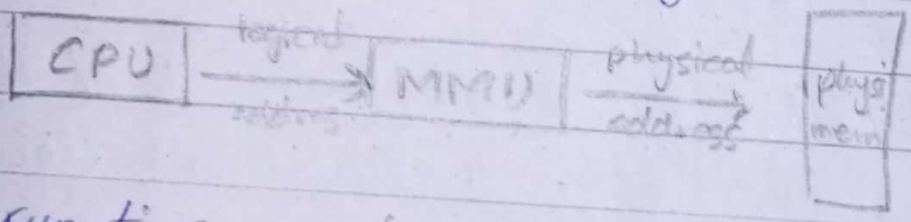
Address binding of instr. & data to memory addresses can happen at three different stages.

1) Compile Time: If you know at compile time where the process will reside, then "absolute code" is generated.
2) Load Time: If it is not known at compile time where the process will reside in memory, then compiler must generate relocatable code. Final binding is delayed until load time. Agr starting address change hoga to hum bs starting mein user code add kr ke req location par aajayenge.

(3) Execution Time : If process can be moved from one mem seg to another during execution, then binding must be delayed until run time.

" LOGICAL VS PHYSICAL ADDRESS SPACE "

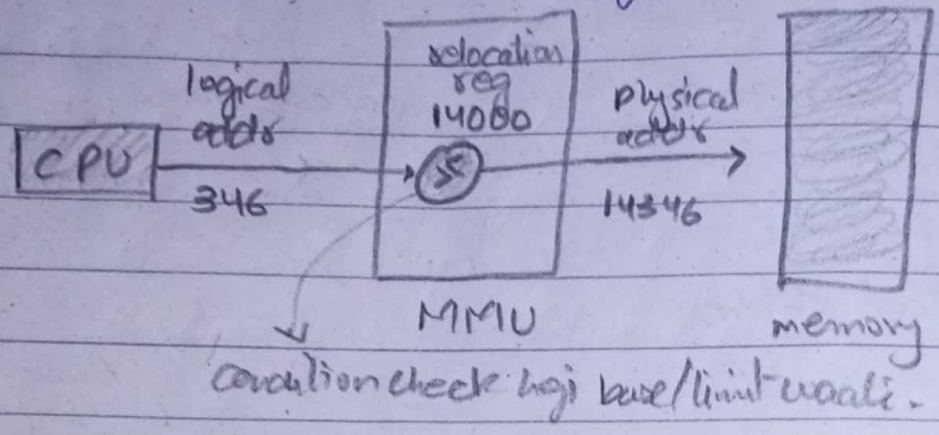
→ An address generated by CPU is known as logical Address.
→ An address seen by memory unit is Physical Addr which is loaded into memory-address register.
→ Logical address also referred as virtual address
→ Logical Address Space is the set of all logical addresses generated by a program.
→ Physical Address Space is the set of all physical addresses generated by a program.



→ The run-time mapping from virtual to phy addr is done by a hardware device memory management unit (MMU)
→ There are many methods for mapping.
→ A simple scheme is in which the address generated by a user program that is a logical address is added to relocation register (base register)

then sent to memory. logical

→ The user program only deals with ~~physical~~ address, it never sees physical adds.



Range for logical (0 to max)

Range for physical (R+0 to R+max), R= base value.

" DYNAMIC LOADING "

→ The entire program does not need to be in memory to execute.

→ Routine (Functions) is not loaded until it is called.

→ Better memory-space & utilization; unused routin never loaded.

→ All routine kept on disk in relocatable load format.

→ Advantage : When large ammount of code are needed to handle infrequently.

"DYNAMIC LINKING"

→ Dynamically linked libraries (DLLs) are system libra that are linked to user programs when programs are run.

→ Static Linking: system libraries & program code combined by the loader into the binary program image.

→ In Dynamic linking, linking is postponed until execution time.

→ Advantage:
  ↳ Saves memory
  ↳ These libraries can be shared among multiple process, so that only one instance of DLL in main memory. Therefore they are also called shared libraries.

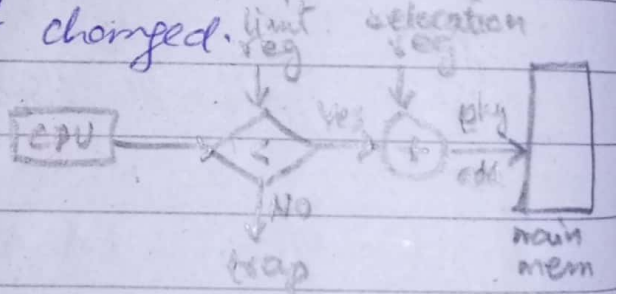
→ Small piece of code, stub, used to locate the appropriate memory-resident library routine.
  ↳ Ensures dynamic loading/linking.

"CONTIGUOUS MEMORY ALLOCATION"

→ Main memory is divided into two parts.
  ↳ OS usually held in ~~low~~ high memory with interrupt ve
  ↳ User process held in high memory.
    ↳ In CMA each process is contained in a single section of memory that is contigous to section containing the next process.

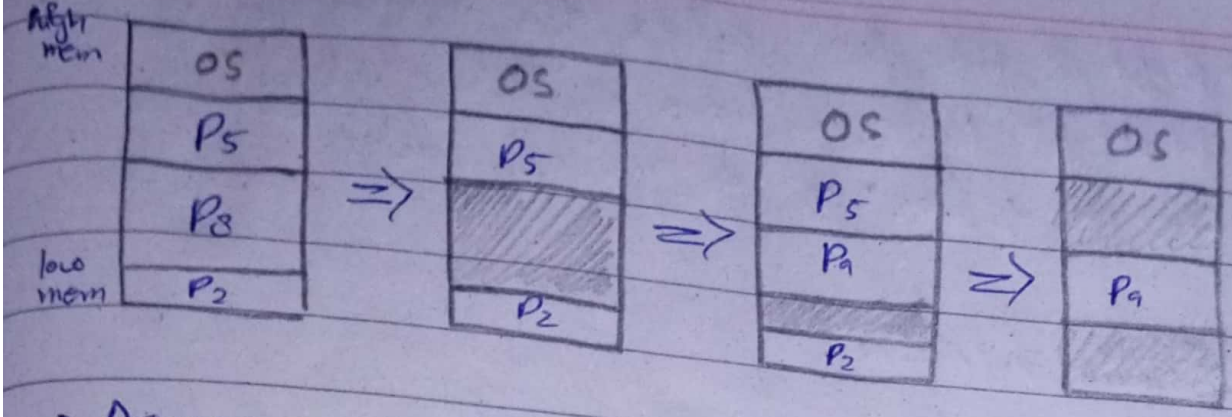
- MEMORY PROTECTION:

→ We can protect memory by using the base/limit registers discussed previously.
→ In every context switch of process the base & limit registers values are also changed.



- MEMORY ALLOCATION:

→ Each process is given variable size according to it's need.
→ In this variable partition the OS keeps a table indicating which part of memory are available and which are occupied.
→ Initially all memory is available & considered one large block, a hole.
  ↳ Holes of various sizes are scattered throughout memory.



→ Degree of multiprogramming limited by no. of partion.
→ When a process arrives, it is allocated memory from a hole large enough to accomodate it.
→ Agr koi process memory free krta hai we usko holes mein convert hojaata hai. Or agr uske adjacent bhi holes ho to we unke sath combine hojayega.
→ Dynamic Storage Allocation Problem:
↳ Satisfying request of "n" size from a list of free holes.
▷ First Fit: Jo phela ~~em~~ hole mile us pas process allocate krde. Searching can start from either at beginning of the set of holes or at the location where prev first-fit search ended.
▷ Best Fit: Allocate the smallest hole that is big enough.
▷ Worst Fit: Allocate the largest hole. More useful than best fit. ? k large hole mein jab process allocate hoga to bachi jo memory bachegi wo hole mein convert hojayegi or use zyada bara hole bnega.

last to fragmentation. 1/3 may be unusable. This property is 50% rule.

→ Best & first fit better ~~for~~ in terms of decreasing
→ Worst fit better in terms of storage utilization
→ First fit ~~better for~~ is generally faster.

FRAGMENTATION:

External Fragmentation:

→ Both first & best fit suffer from external fragmentation.

→ In contiguous memory allocation we can use two methods.

1) Variable Size Partioning
2) Fixed Size Partioning:

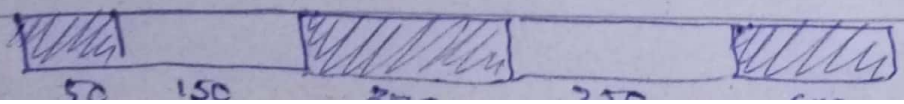
→ left over space can be reu again

Variable Size Partioning: Is mein hum utna hi blocks of memory allocate krte hai jitna required hota hai by a process.

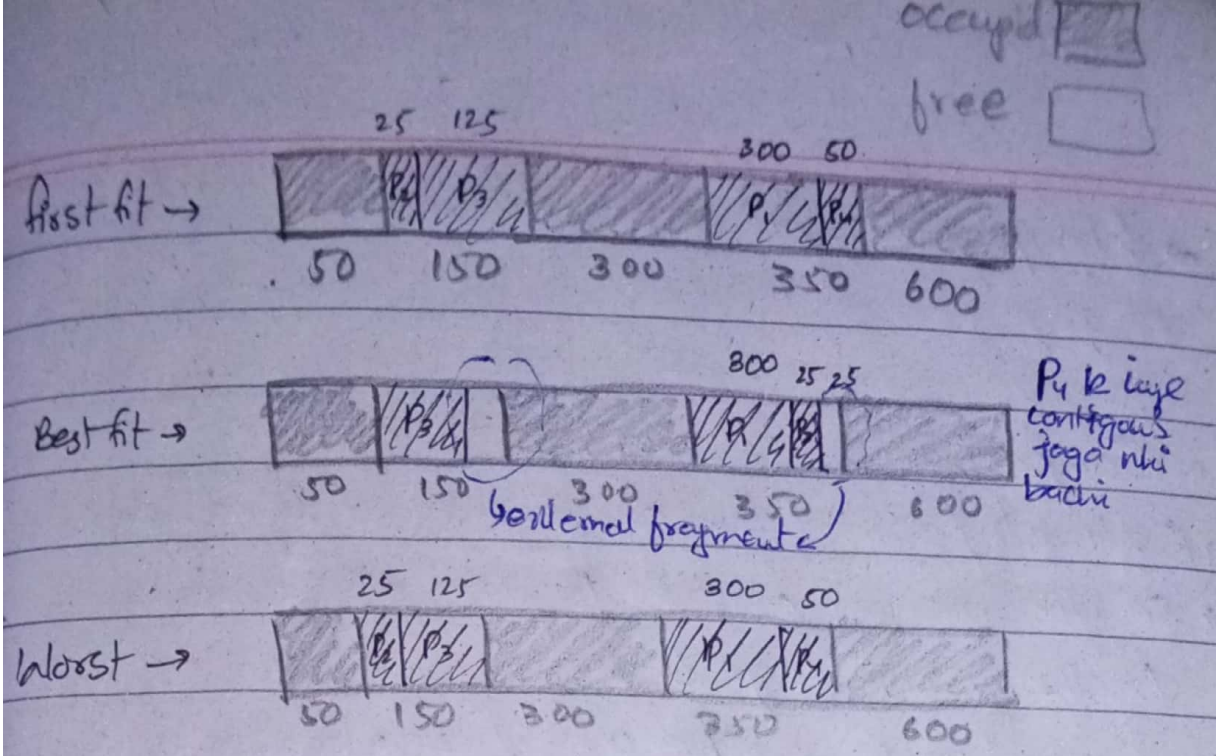
→ Variable size mein we can get "external fragmentation" → "total memory space exist to satisfy a request but it is not contigous".

→ We have 4 process & we have to allocate memory according to the first, best & worst fit.

$P_1 = 300$, $P_2 = 25$, $P_3 = 125$, $P_4 = 50$



→ current memory sta



→ Worst fit vasible size mein acha kaam karta hai q k agr hum largest block choose karenge to bachne wala block bhi bada hi hoga or we reuse hone ki possibility zyada hai. Agr Best fit use karte hai to remaining block bht small hota hai or we reuse hone ki prob bht kam hojaati hai

Fixed Size Partioning:

→ Is mein hum apni memory ko certain no. of memory blocks mein divide krdete hai. Or har block ka size same ya different ho.sakta hai depends on machine

→ Lekn agr ek process ko 50KB chahiye or memory mein 100KB mojood hai to use 100KB assign hoga or baqi 50KB unused rhega. This is internal fragmentation

"Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used."

→ Yaha par Bestfit aata hota hai.

• Compaction:

→ External fragmentation can be reduced by compaction.
→ Shuffle memory contents to place all free memory together in one large block.
→ Compaction is possible only if relocation is dyn and is done at execution time.

"PAGING" → no external fragmentation but have internal fragmentation.

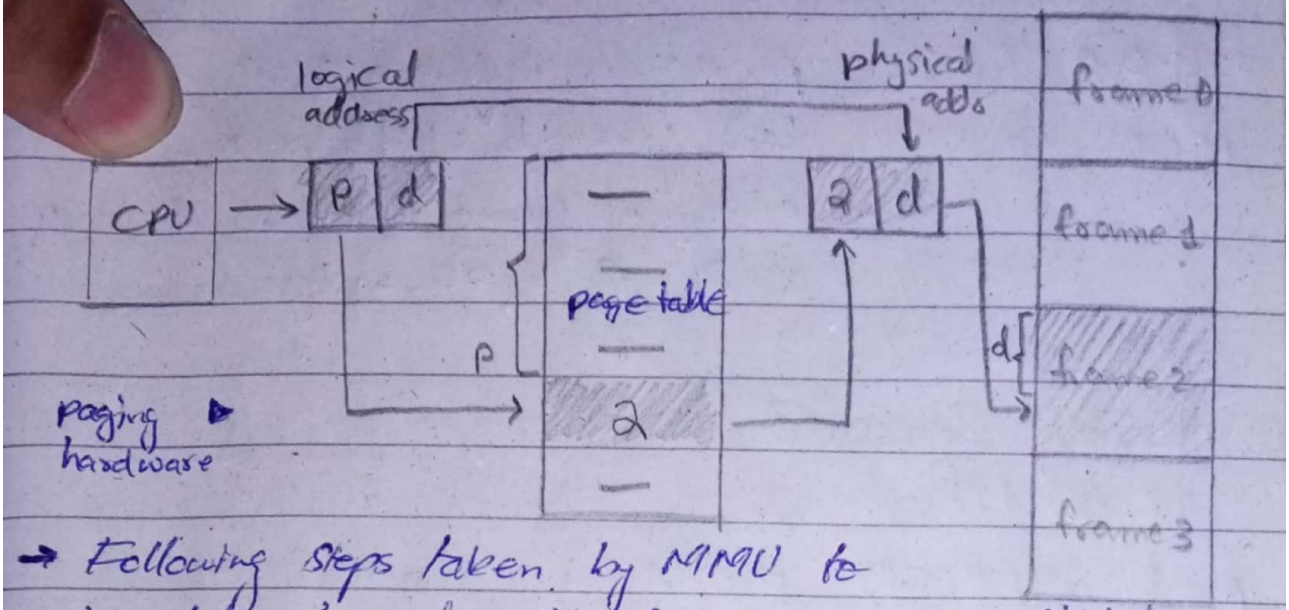
→ It is non-contigous memory allocation.
→ The physical memory is partioned into fixed size of equal blocks called frames. or main memory
→ The process memory is also divided into fixed size equal block called pages.
→ The size of page and frame are equal.
→ Jab bhi kisi process ko main memory mein load karna hoga to uske pages ko load karenge frames me
→ Every address generated by CPU is divided into two parts: a page number (p) & a page offset (d).

p	d

page no. page offset → particular frame mein kis jagah jaana hai yeh offset hai.

Disadvantage: We need to remember the effect of every page frame in our page table in which our pages are stored. There is also internal fragmentation. Memory access time is doubled this frame use nhi.

→ Has process ka ek page table hota hai. Os us page table mein ek i^{th} page kis (frame no.) k sath mapped hai wo store hota hai. (base address of that frame)

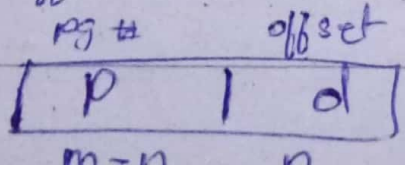


→ Following steps taken by MMU to translate logical adds to phys adds.

1. Extract page no. 'p' & use it as an index into page table
2. Extract corresponding frame no. 'f' from page table
3. Replace page no. 'p' in logical address with frame no. f.

→ The size of a page is power of 2.

→ If the size of logical address space is 2^m and a page size is 2^n bytes, then high order "m−n bytes" of logical address designate page no. & lower n bit page offset.



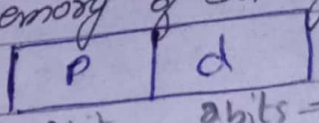
If address is of n bit then size of main memory is 2^n x size of each frame
to be added → 2^m x (2)

2^n

2^4 = 8 bits for logical address space.

→ Logical address: n=2, (m=4). Using a page size of 4 bytes and a physical memory of 32 bytes 18 pages.

See figure 9.10



(a) Logical Address: 0

| 00 | 00 |

Pg# 0, offset: 0

page#0 is in frame 5.
Each frame is of 4 bytes so base address of frame 5 will be. 5x4=20
& 20 + 0 = (20) → physical address.
offset

(b) logical address: 3

| 00 | 11 |

Pg# 0 offset: 3

page#0 is in frame 5.
so (5x4+3) = 23 physical addr.

(c) logical address: 4

| 01 | 00 |

Pg# 1 offset: 0

page#01 is in frame 6
(6x4+0) = 24

(d) logical address: 13

| 1 1 | 0 1 |

Pg# 3 offset: 1

page#03 is in frame 2
(2x4+1) = 9

Memory = n x (size of location)
64KB = n x 1B (suppose)
n = 64KB/1B ⇒ n = 64K

64K = 64,000
= $2^6 \times 2^{10}$
= 2^{16} → 16 bits required to represent a memory of 64 KB

→ jitna page size kam hoga utna internal fragmentation kam hoga but no of pages badh jayega or page table

32B = n x 4
n = 8

→ Example: A page have size 2048 bytes and a process have size 72,766 bytes. It will need pages + 1086 byte to accomadte mean 36 mes. 4n 36 mein se 35 mein internal fragmentation nhi hoga lekn last wale mein kuk 1086 bytes hai or ek frame 2048 bytes ka hai to internal fragmentation of 962 bytes.

→ Agr koi esa process hai jise n pages + 1 byte chahiye hoga to usko n+1 frames allocate honge or ek frame ka almost internal fragmentation hoga.

→ Frame Table: OS keeps tracks of all details of physical memory like - allocated frames, free frames, total frames & so on in a data structure called frame table.

Q: Why paging increases context switch time?
Because when a process is switched out, its entire page table must be saved & restored. This involves updating the page table base register, which can be time consuming especially if process has a large page table

page-table base register (PTBR).

→ Har ek process ka alag page table bnta hai. Os
Ki base value ek register mein store hoti hai
→ The page table is kept in main memory, and a pag
table base register (PTBR) points to the page table
→ Jb agr context switch krna hoa to sirf is register ki val
change hoti poora PT load unload nhi krna pdega

▷ Translation Look Aside Buffer:

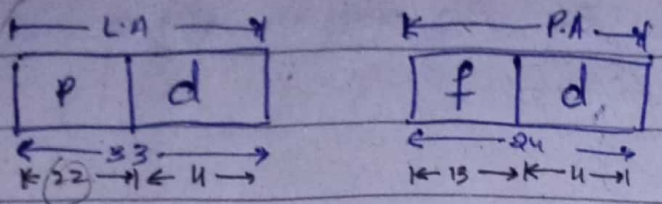
Why needed?

↳ To get any data from the memory first we need
to traverse the page table from its base value t
the required page, then that ~~page~~ frame no. w
combined with page offset to get desired place i
memory. Here ~~two~~ memory acess are need to
acess data, Hence memory acess is slowed 2
which is intolerable in most circumtances.

Solution:

↳ To stop the acess of main memory two times we use
special, small, fast lookup hardware cache called
translation look aside buffer (TLB), a high speed m
- Consist of two parts a Key (page #) & a value (frame
- TLB contains ~~only~~ only few page table entries.
- When LA is generated by CPU, the MMU first checks
page no. is present in the TLB. Agr mil jata hai to
it can directly access memory, this is called TLB hit

each page no. can be represent by 22 bits
so total no. of regs = $2^{22} = 2^2 \times 2^{20} = 4M$



LA = 33b
PA = 24b
PS = 2KB

size of each page / frame = 1B

$2^{35} = 2^3 \times 2^{30}$
$= 8 \times 1G$
$= 8G \times 1B$
size of each location

Secondary Memory
↳ 8GB

$2^{24} = 2^4 \times 2^{20}$
$= 16 \times M \times 1B$
$= 16MB$
size of each loc

main memory

size of whole page / size of each page = no. of pages $\Rightarrow \frac{2KB}{1B} = 2K = 2000 = 2^{11}$

2×2^{10}

so, d = 11

- If the page no. is not in TLB (known as TLB miss) to paging ka procedure follow hota hai k uske page ko memory mein load krke page table mein safe krdenge. Lekin uske baad, the page & frame no. is also saved in TLB so it can be accessed quickly in next references.
- Agr TLB full hai to kisi page ko replace krdenge according to some certain policies.
- Some entries in TLB are wired down, meaning they cannot be removed.
- Some TLBs store address-space identifiers (ASIDs). In each TLB entry, which uniquely identifies each process and used to provide address space protection for that process. (Agr ASIDs nhi honge to har process k context switch mein TLB needs to flush the data of prev process)

- The percentage of times that the page no. of interest is in TLB is called hit ratio.

eg: TLB use karne se average memory time = 2 x memory access time

Calculating effective memory-access time:

→ If it takes 10 ns to access memory
→ Then mapped memory access takes 10 nanoseconds when page# is in TLB.
→ Agar TLB mein page nhi milta to 20ns or agar mil jata hai to 10ns.
→ So if we have hit ratio of 80%. then

effective-access time = 0.80 x 10 + 0.20 x 20
= 12 ns

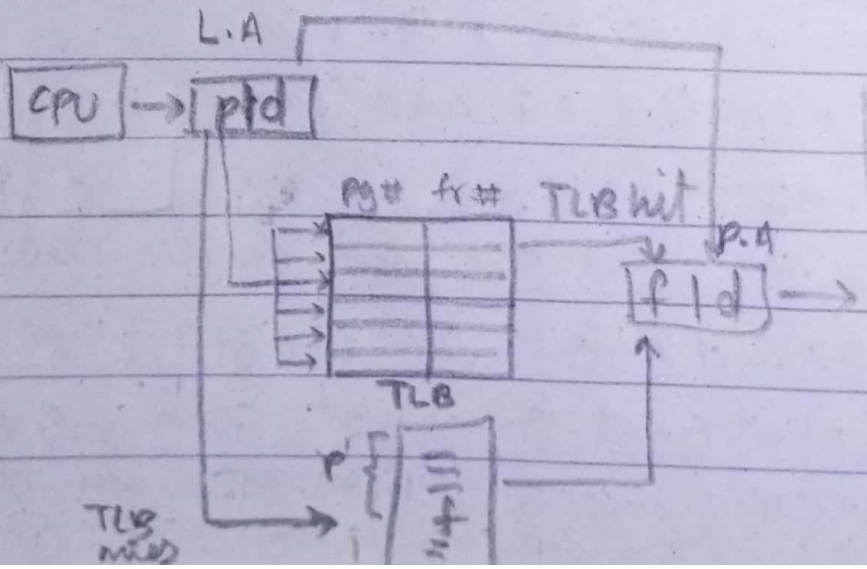
(TLB hit mein 10ns, TLB miss mean 20ns)

↳ in this we suffer 20% slowdown

→ If hit ratio is 99%.

effective-access time = 0.99 x 10 + 0.01 x 20
= 10.1 ns

↳ only 1% slowdown.



MM = 400μs, TLB = 50μs, h = 90%.

TLB hit mein usko TLB time lega q/k humein direct to kona hi parega ka page hai ya nhi.

effective-access time = $\frac{h}{100}[TLB + MM] + \frac{(1-h)}{100}[TLB + MM \times 2]$

= 0.9 [50 + 400] + 0.1 [50 + 400 × 2]

= 490μs

- Disadvantages of TLB:
  - → It is a hardware so each process cannot access it simultaneously.
  - → If there is a context switch the data of prev process is flushed out.
  - → Or jab doosra process ayega os usmein jese hi TLB full hone wala hoga to wapis CS hojayeg plus os jab doobara ayega to use wapis se TLB ko fill krna parega.

Solution: We can add multiple TLBs to accomodate multiple process but it is costly.

↳ Memory Protection in Paging:

→ It is done with the help of protection bits associated with each frame

→ Bits are maintained in Page Table, so during the search of a frame no. in Page table, protection bits are also checked.

Example: Agr koi frame ~~mein~~ sirf Read-Only hai mein we can not write so we can ensure this by using protection bit.

	Frame #	Valid bit
0	2	V
1	4	V
2	6	V
3	7	V
4	10	I
5	8	I

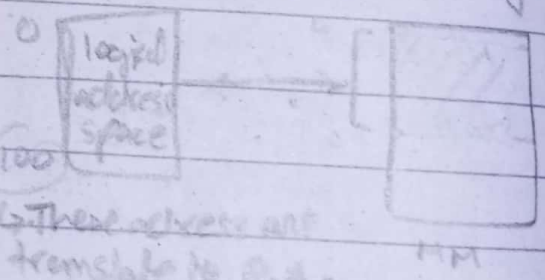
↳ Valid- Invalid Bits:

Valid-invalid bits can also be associate with page table

bit → Valid → Page is in the Process Logical Address space of address that the process can use to access memory.

→ Invalid → Page is not in the P.L.A.

→ Any violation results in a trap to kernel.



Problem:

For eg humare pas 14-bit address space hai i.e (0 – 16383 $(2^{14}-1)$) lekin humare process ko sirf (0–10468) allocate kiya gaya hai means ek process will have 6 pages $(10468/2KB) \approx 6$ is the size of a page ($2KB = 2 \cdot 2^{10} = 2^{11}$).

Lekin agr hum 6 pages ki total deklein to 12,287 bnega jabke humein 10,468 tak ka access hai ~~so ham~~ jo logical fragmentation se bachega wo illegal hoga lekin page table par

valid bit se hi show hoga because that page is in our process memory space.

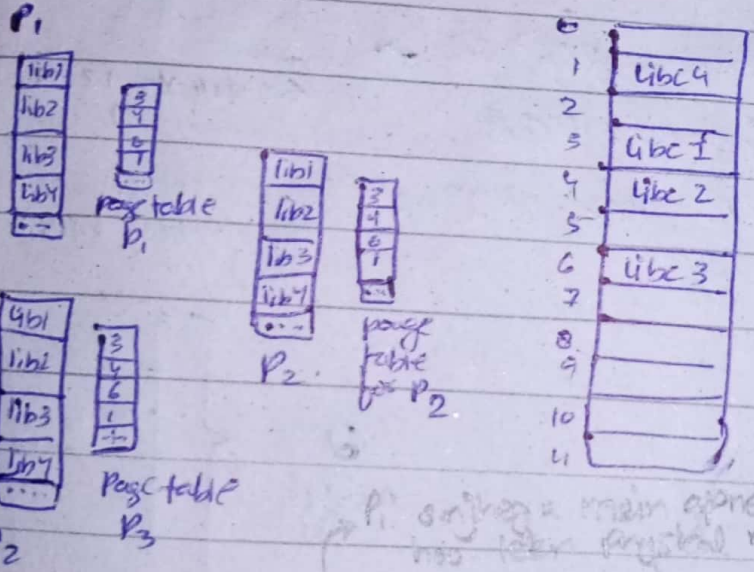
* Key

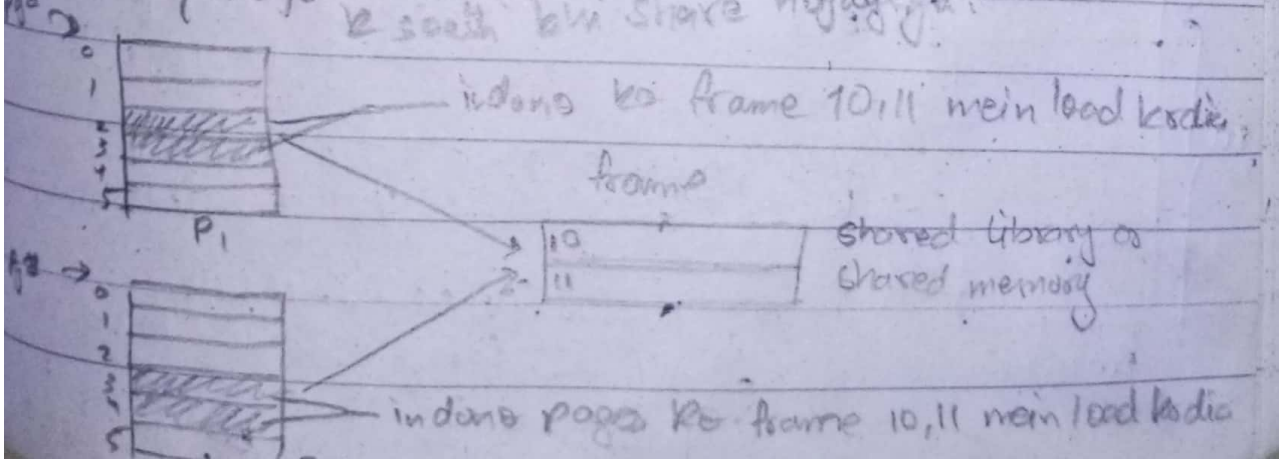
SHARED PAGES:

→ One copy of readonly (reentrant) code shared among process (i.e text editors, compilers, window system)

→ Similar to multiple threads sharing same process space

→ Also useful for interpo— IPC if sharing of read-write pages is allowed.





"Structure Of Page Table"

① Hierarchial / Multilevel Paging :

→ Agr hamare page-table ka size hamare frame k size se bara hogaya to humein apne page table ko further pages mein divide krna hoga.

→ Page table bhi main mem mein store hote hai ~~to unka~~ or main memory apke frames mein divided hoti hai. to apka page table frame k size se bara nhi hosakta

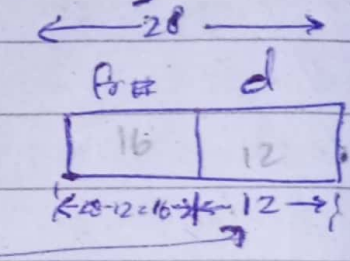
Example:

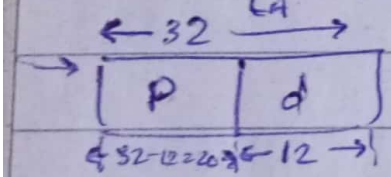
Physical Address Space = 256 MB → 28

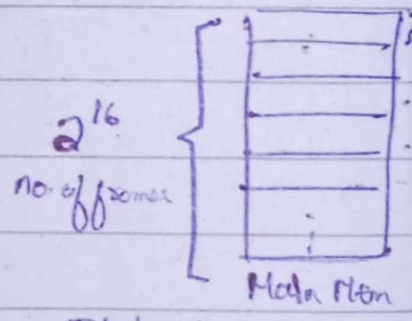
Logical Address Space = 4GB

Frame size = 4KB ($4KB = 2^2 \cdot 2^{10} = 2^{12}$ (means ek frame mein 12 bit address hi store hosakta ha).

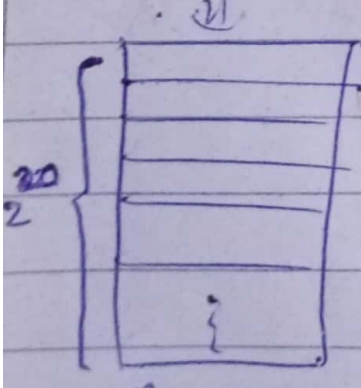
Page Table Entry = 2B



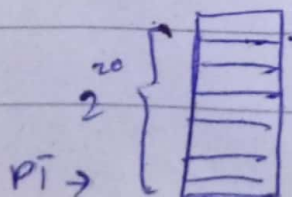




Total 2^{16} = 65536 frames hai aur har ek ka size 4KB hai



⇒ Ab is process ka page table bnega to usmein bhi 2^{20} no. of entries ki jaga reserve hai

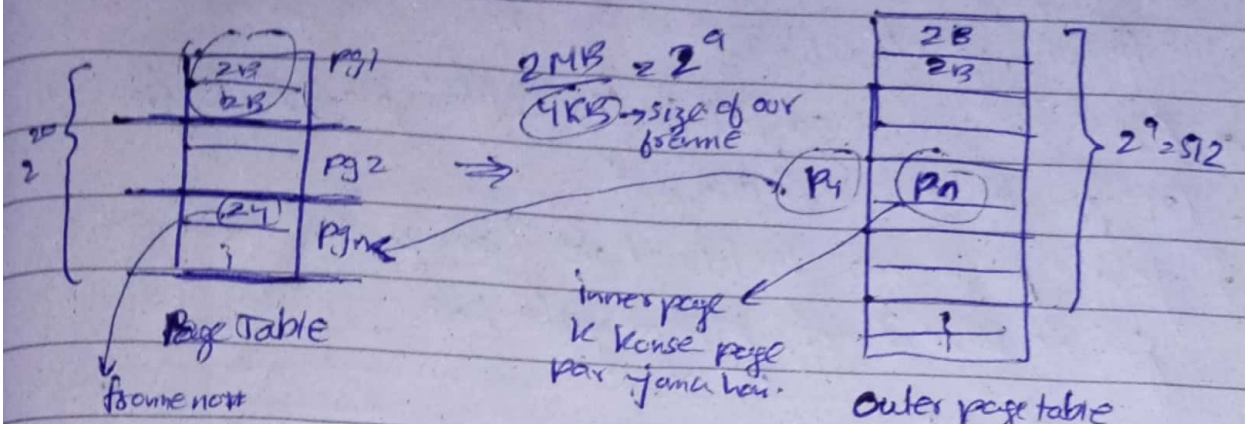


32B (Given) = 16 bits

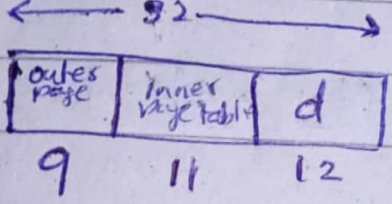
↳ page table mein to sirf frame aata hai to uske liye 2B hi kafi hai aur upar humne

Size of page table = $2^{20} \times 2B = 2MB$

→ Ab page table ka size 2MB or ek frame ka size 4KB so we need to divide PT.



Size of OPT = $2^9 \times 2$
= 1KB < 4KB ✓



→ Phele outer page mein jayenge usha hamare inner page ki value hogi. Phir us inner page par jayenge. Us inner page par frame no. hoga or phir frame no. se required offset par move karenge.

This scheme is called forward-mapped page table.