→ Concurrent Datastructures

↓

Hashmap → Concurrent — Synchronised

→ How memory is stored for process

→ fragmentation

→ MMU → memory management unit →

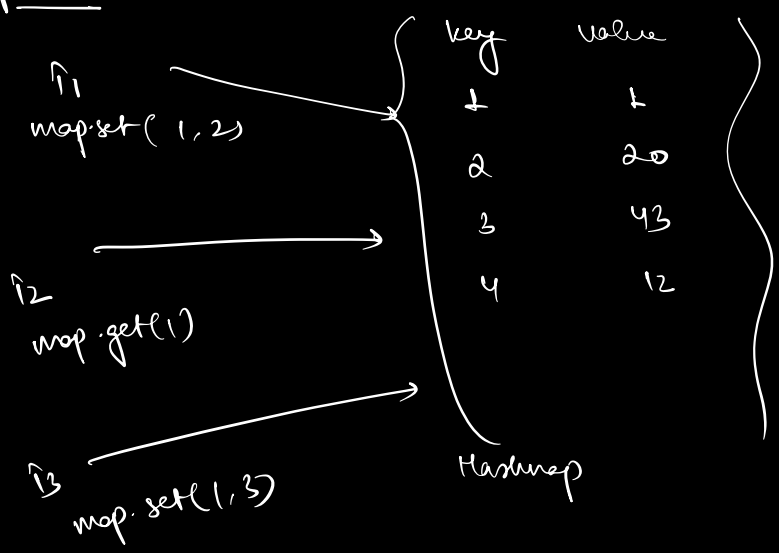
→ Paging

→ Page Replacement algo

→ Page fault →

→ Deadlocks

=) Hashmaps :-

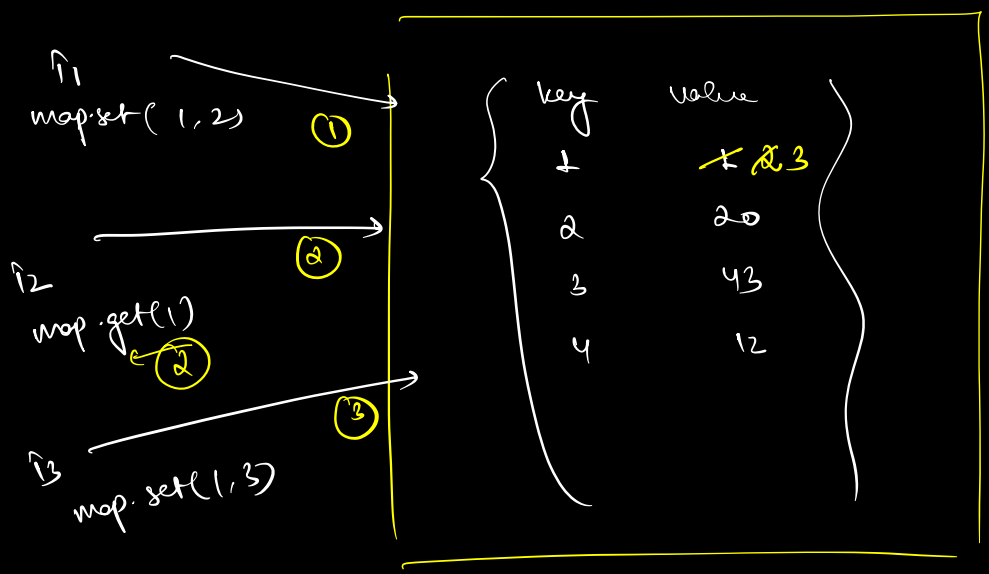


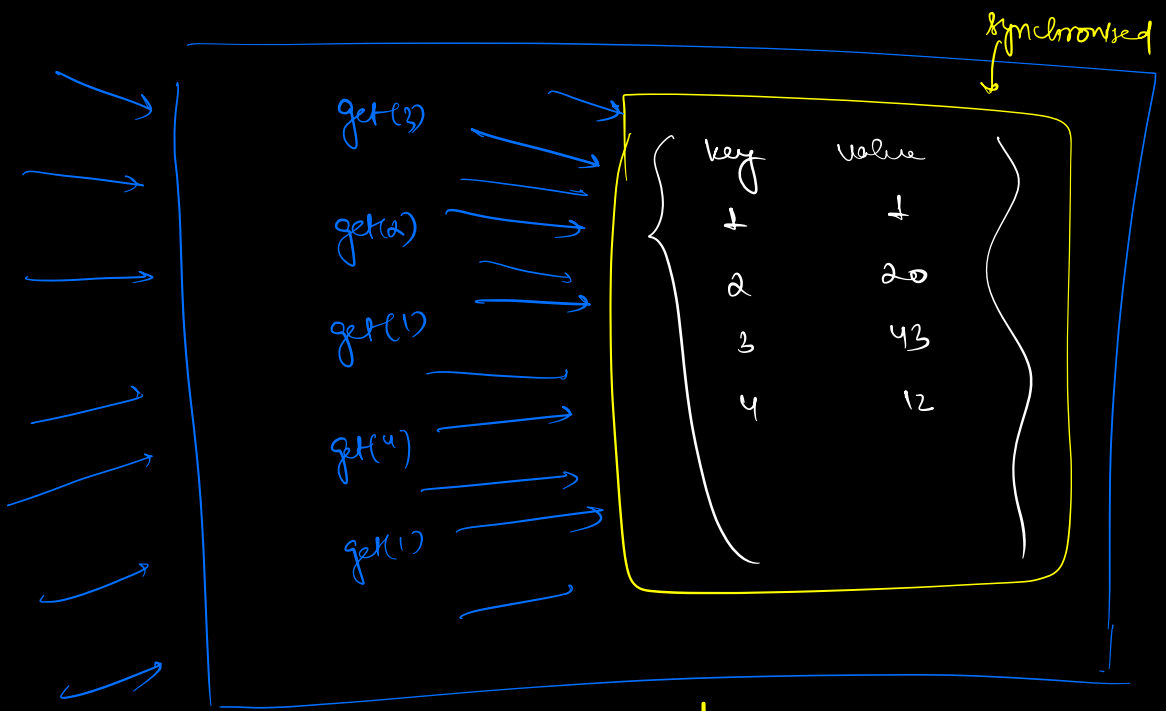
↓

Chances of race condn

: making it synchronised to make it thread safe :-

[Synchronised Hashmap]

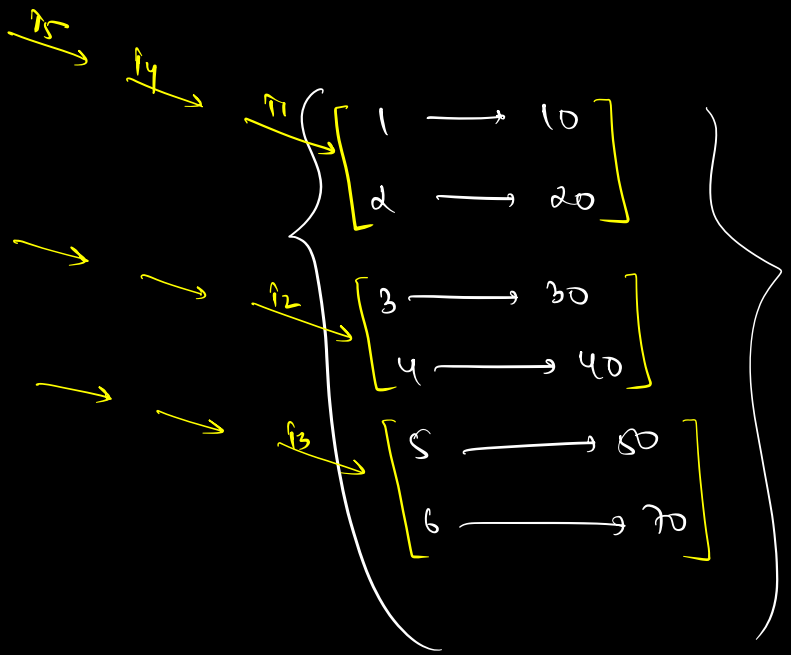




↓

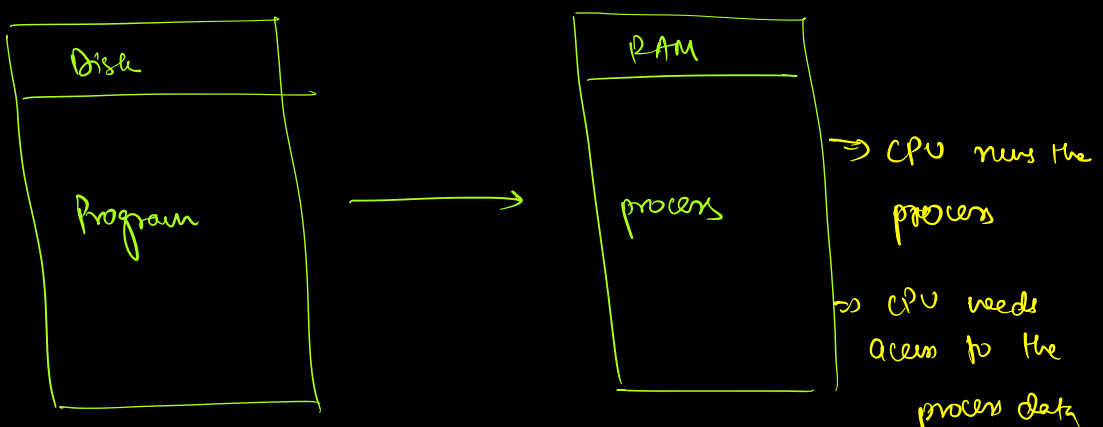
since only 1 thread can execute at a time, this is not fast enough

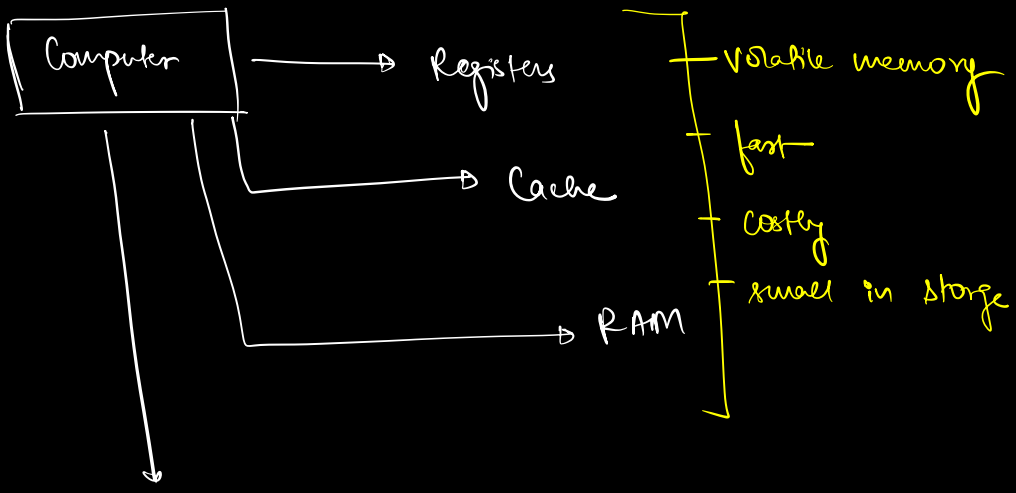
: Concurrent Hashmap: it allows multiple threads to operate on the map but synchronises buckets inside the map.



: Memory Management

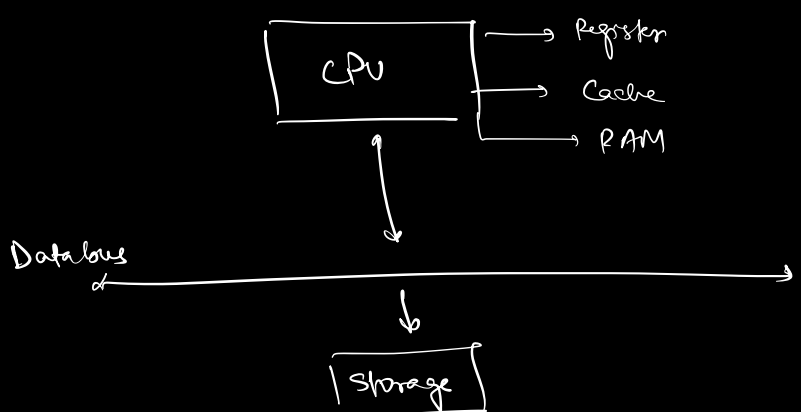
→ How exactly data is stored in a process ?





Disk storage → HDD | SSD → Persistent
Slow
Cheap
Large in storage

: CPU can't directly talk to storage devices
(because of speed differences)



=> Some programs / apps can be huge in size

There can be scenarios where complete process may not be able to be stored in RAM.

RAM => 4 GB.

ex => i) Games

ii) Too many tabs on Chrome

iii) Video editing

iv) IDEs. (Android Studio)

v) Highly data intensive algo.

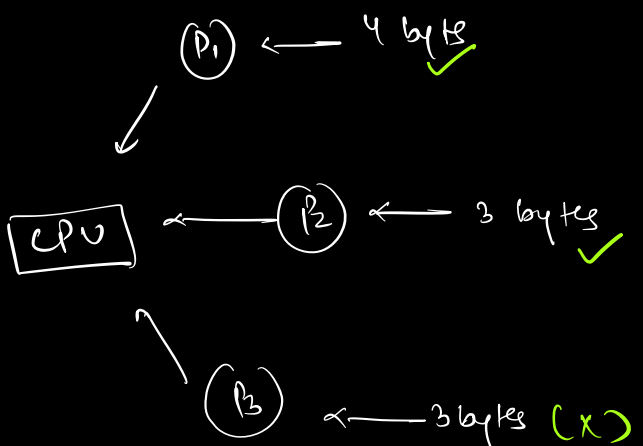
→ How does it work?

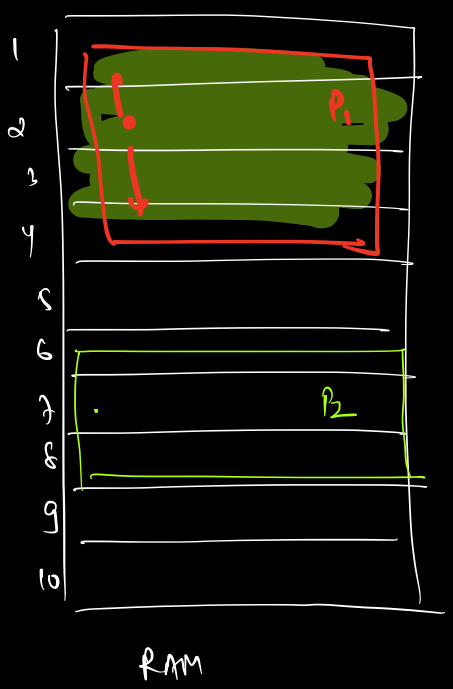
→ How process is stored in memory:-

* CPU allocates memory for process in RAM, then starts executing.

* CONTIGUOUS MEMORY ALLOCATION :-

CPU allocates a part of the free RAM to complete for complete execution





PROS

=> faster

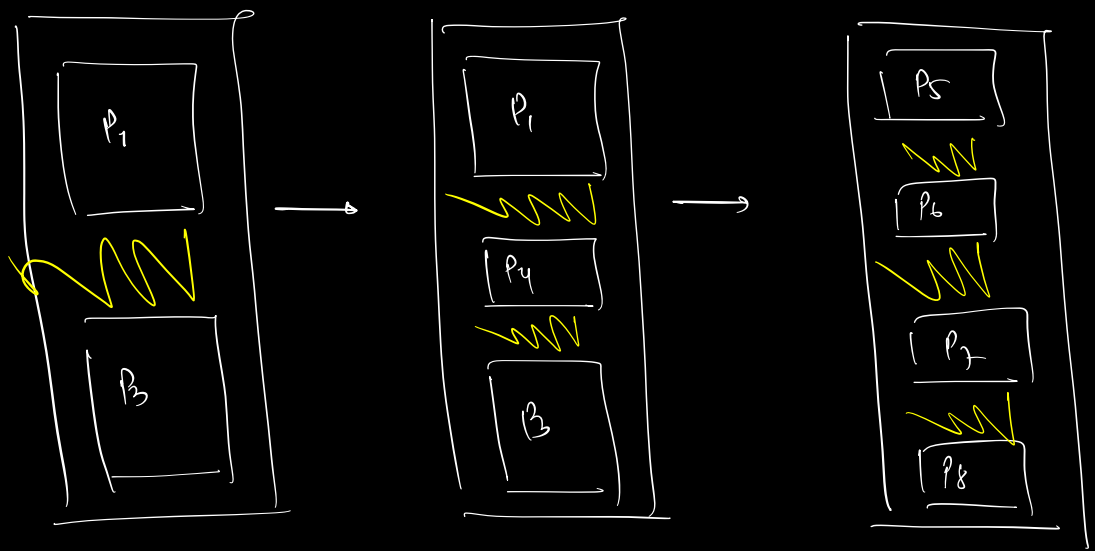
=> complete data of the process is always in memory

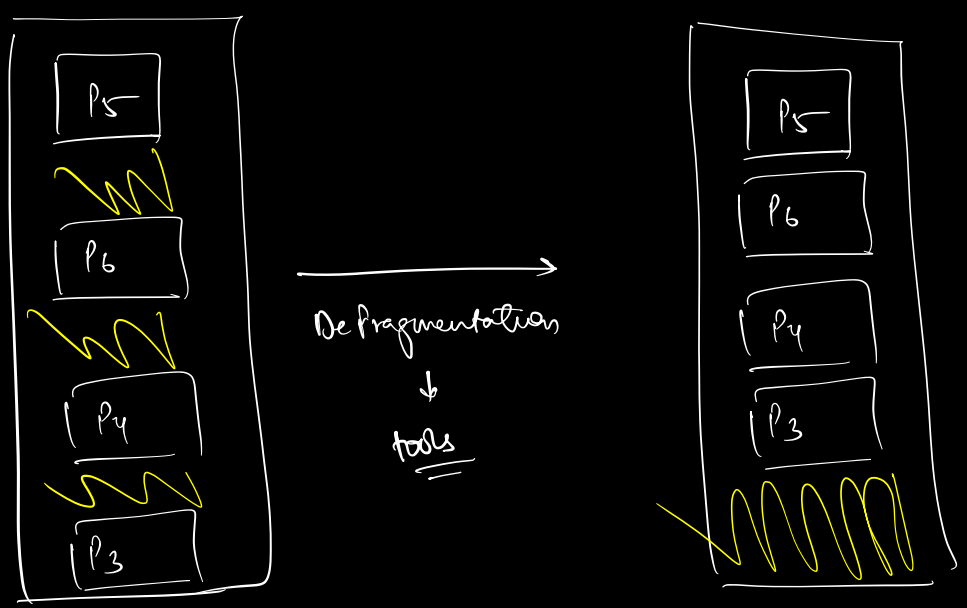
CONS:

=> wastage of memory

+ fragmentation
  - (pieces, parts)
  - → memory divided into smaller chunks
  - → leads to wastage of memory

When a CPU is accessing some data from RAM, it also reads nearby data, and stores it in cache, assuming it might be required later on.





* If a process needs 6 GB RAM, but we have 4 GB, how do we actually execute it ?

Paging → reality.

RAM → 4 GB
Req → 6 GB
} extra 2 GB stored in a disk

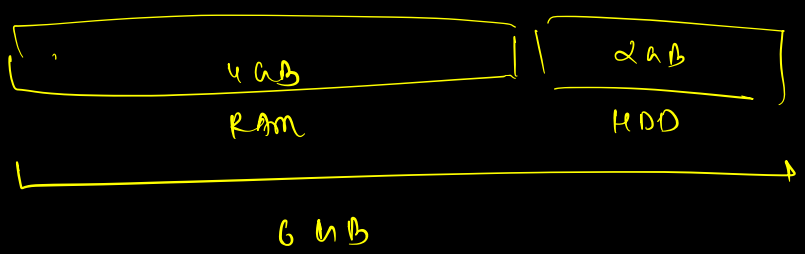
↓

HDD | SSD | flash drives

principle :-

→ store whatever is possible in RAM

→ everything else to be stored in to disk



⇒ whenever CPU needs to access something, from the disk, it is first brought into memory, then accessed by CPU.

↓

LAG

↓

who does all this work?

MMU (memory management unit)

⇒ Deep dive on Paging :-

There are 2 types of addresses :-

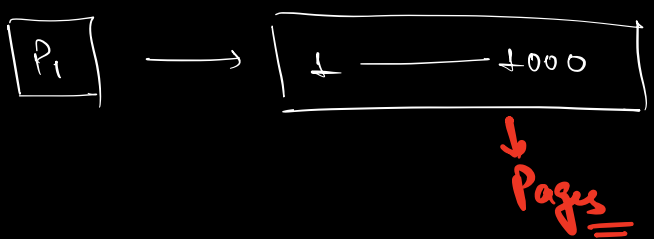
i) Logical address → conceptual address

ii) Physical address → real address.

⇒ Apps will only know ⇒ "logical addresses"

⇒ As soon as a process starts, MMU allocates a huge no. of conceptual address (logical addr to it.

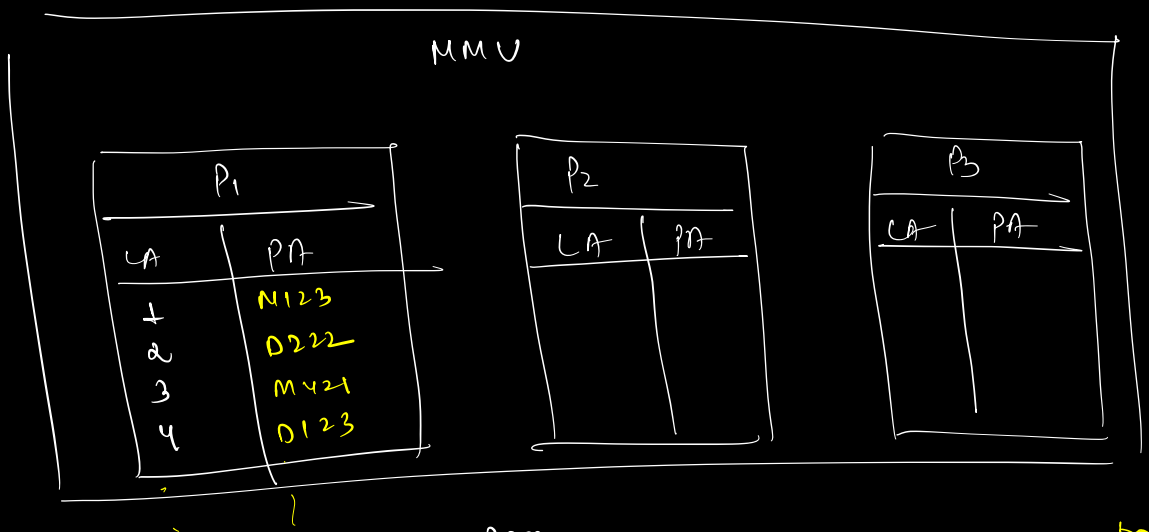
[Page Table]

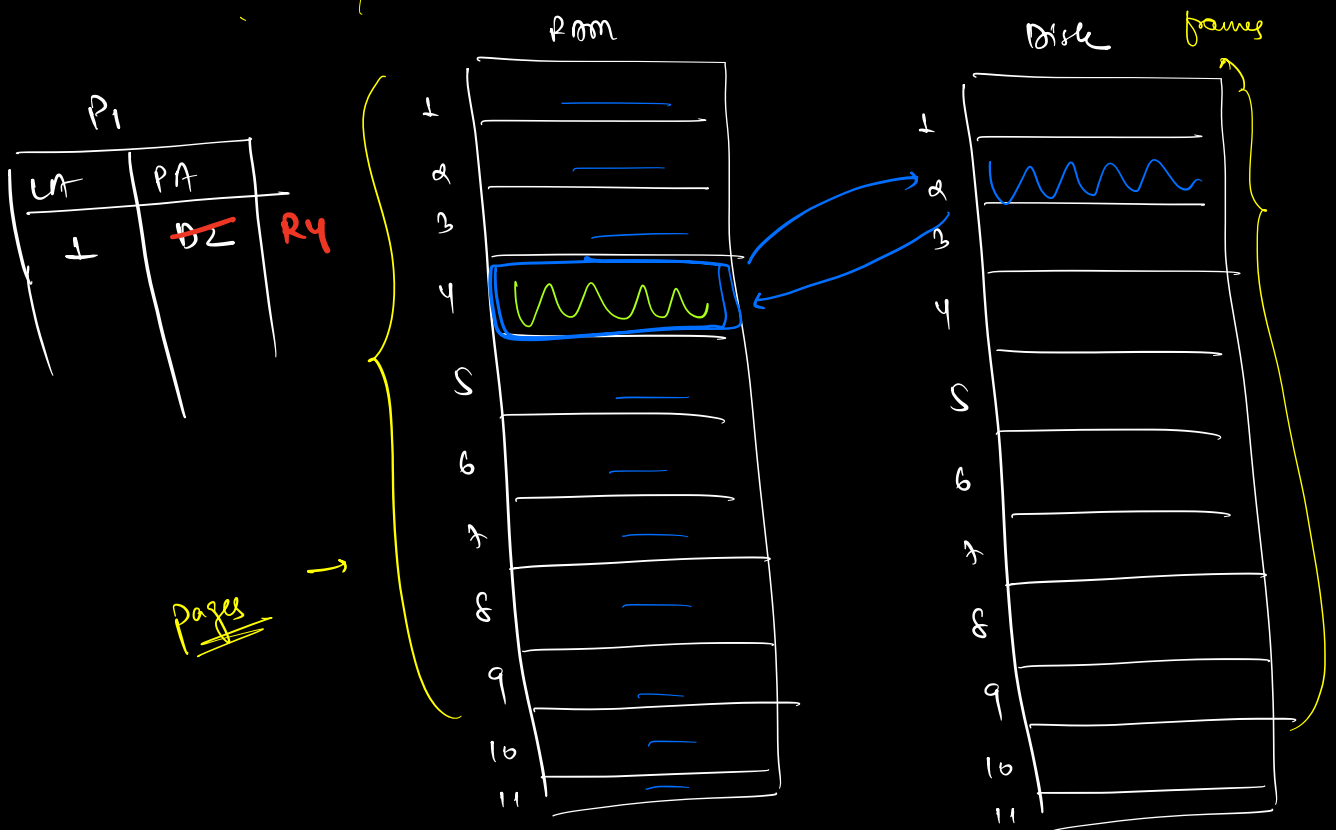


⇒ MMU internally maintains a table to map the logical address to real physical address.

RAM or disk

table ⇒ Process Page table





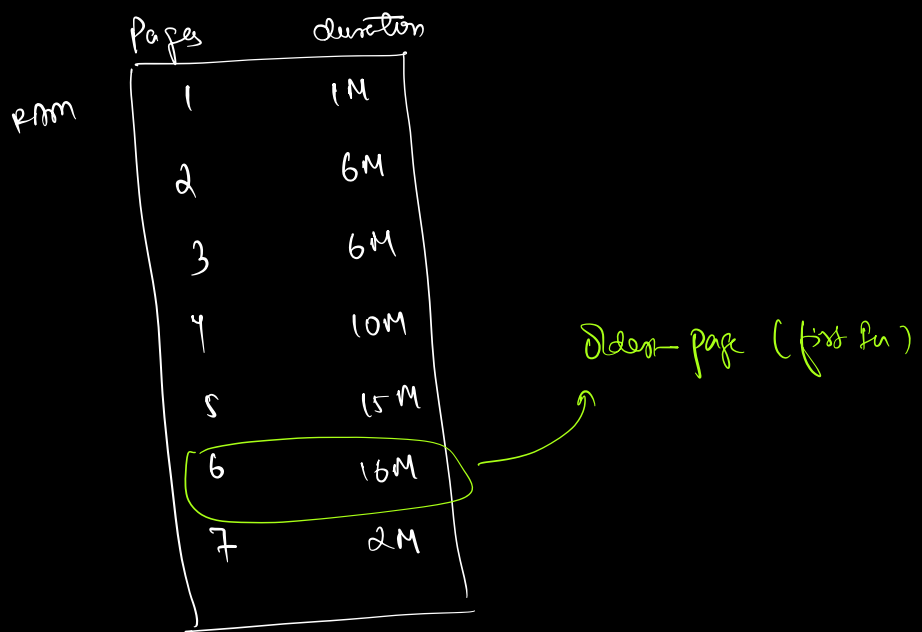
size of 1 page == size of 1 frame

=> [Page Replacement]

↓

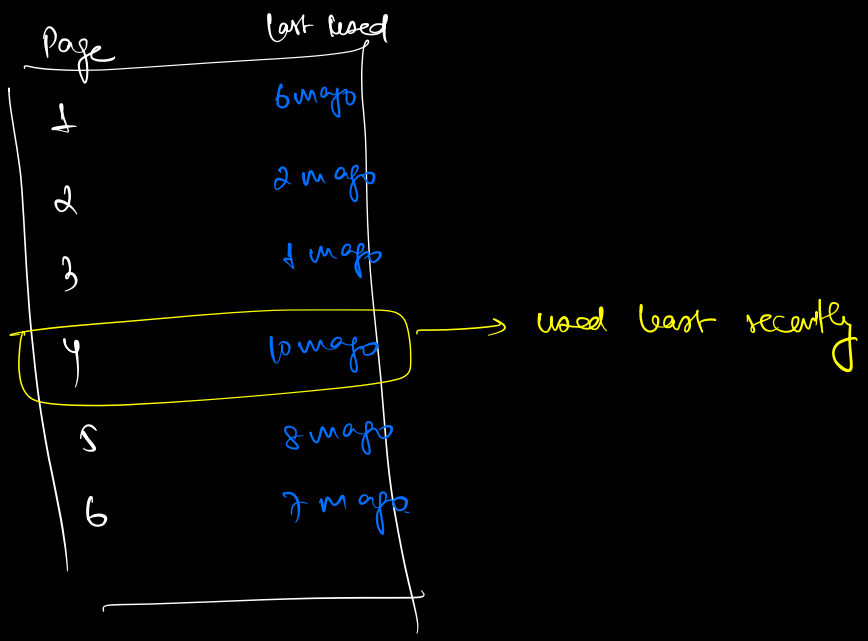
Page replacement algo → decides which page to remove from RAM, when a new page from page disk has to be brought in RAM.

=> 1. FIFO → First In First Out



2) LRU → Least recently used

replace the page which was least recently used



(iii) LIFO → last in first out

⇒ PAGINGi—

PRO

* no memory limit
* no memory wastage.

CONS

* complex to design
* slower
* appn doesn't know where exactly the data is present.

→ How process access the data:

i) CPU needs the data to execute

ii) CPU will get the logical address.

iii) CPU goes to MMU with logical addr.

iv) MMU maintains (logical → physical)

↓

page process table

MMU gets physical address

present in RAM

↓

return to CPU

present in disk

↓

i) run a page replacement algo

ii) replace the page and update the page process table

iii) return to CPU.

Page fault :- when CPU asks MMU for data but data is present in disk of instead of RAM.

→ Higher no. of page fault

→ Lag, slowness.