Now, processor at once copies all the data item needed for a given piece of code rather than one at a time.

Now it is based on principle of locality

(1) temporal locality

- If we access x, then there is higher probability of accessing x in near future (eg: loops) (∵ copy to cache)

(2) spacial locality

- If we access x, then there is higher prob. of accessing data near to x in near future

eg: Arrays

(If we copy items / subset of data so that if we need a data we need to search only cache (smaller) than 1° or 2° (larger)

floating point numbers not so.

eg: $x = -1.5 \times 10^{38}$

$Y = 1.5 \times 10^{38}$

$z = 1.0$

I] $x + (Y + z) = x + 1.5 \times 10^{38}$

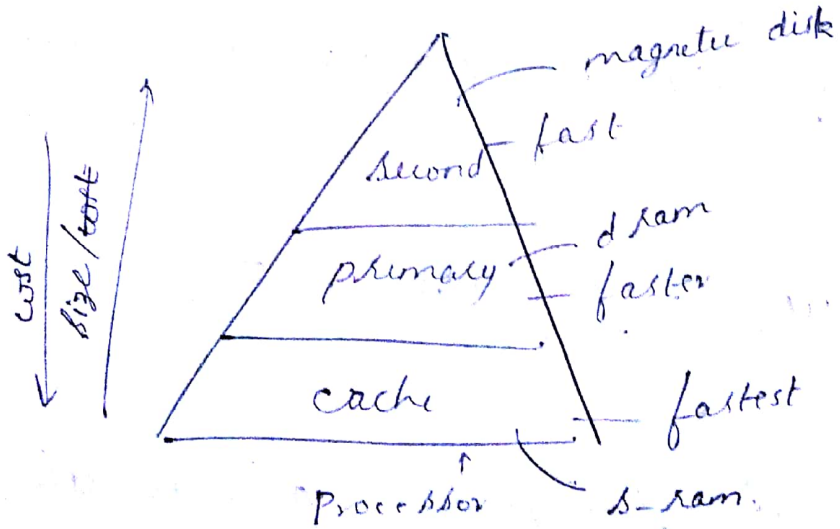
$= 0$

II] $(x + Y) + z = 0 \times 10^{38} + z$

$= 0 + 1$

$= 1$

Memory hierarchy



processor access data from cache

data copied from 2° → 1° → cache → processor

block / line : Minimum unit of info. that is either present or not present in mem.

hit : A hit is occured if a data item is found in a given level else its a miss.

If a hit is occured it copies it to next level closer to processor

else

searches in next level above it

hit ratio : % of memory access satisfied by that level

$$= \frac{\text{no. of hits in a level}}{\text{no. of access in level}} \times 100$$

$$\text{III}^{\text{ry}} \text{ miss ratio} = 100 - \text{hit. ratio}$$

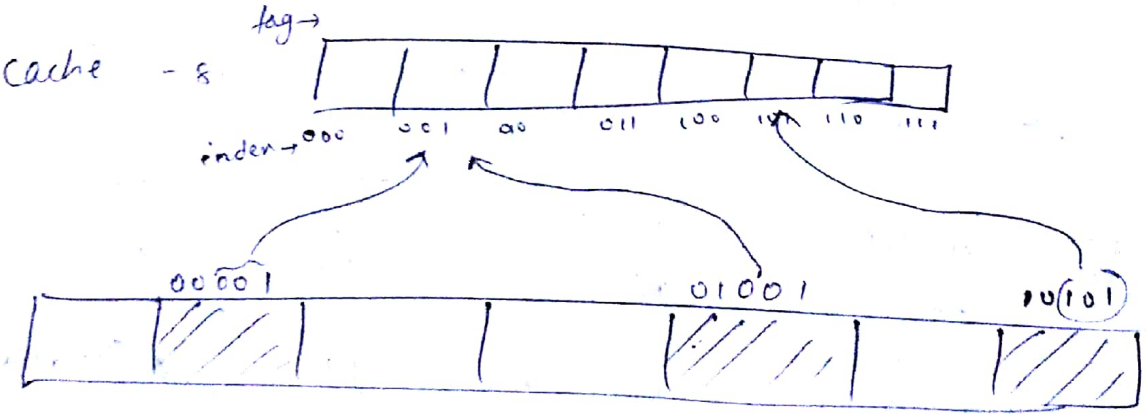
hit - time : time taken to provide a data to CPU including the time to

check if its a miss/hit

misperality : If its a miss time to copy lower to upper mem + time to deliver data to CPU

Now, in cache limited space is there so how do we copy all the elements needed for a program in there? (duplication or overwrite may occur)

& how the processor can identify the data from cache?



- lower 3 bit as index & upper as tag
- here by using modulo on address %8
- if add data = 00001 we do modulo 8 to find index 001

Now to know if its lower bit of 01001 or 00001 place upper bit in tag associated with that position in cache.

This is called Direct mapped cache

Now to see if its valid data or not we have a valid bit also associated with each cell.

If an address of a data is

$x_n x_{n-1} \ldots\ldots x_0$ (the last x bits) then if size of cache is 2^x, then lower x bits is used to divide the index & rest as tag.

eg:

index (lower bit)	Valid	Tag (higher bit)	Data
000	N	10	M[10000]
001	N		
010	Y	11	M[11010]
011	Y	00	M[00011]
100			
101			
110	Y	10	M[10110]
111	N		

If 10110 is needed by processor, it checks

index 110 (cache = $2^{(3)}$)

first it check 110, since V=N & tag & index not same ∴ its a miss. ∴ copy data & store address in corresponding index

Then we read 11010
tag index → miss ∴ store

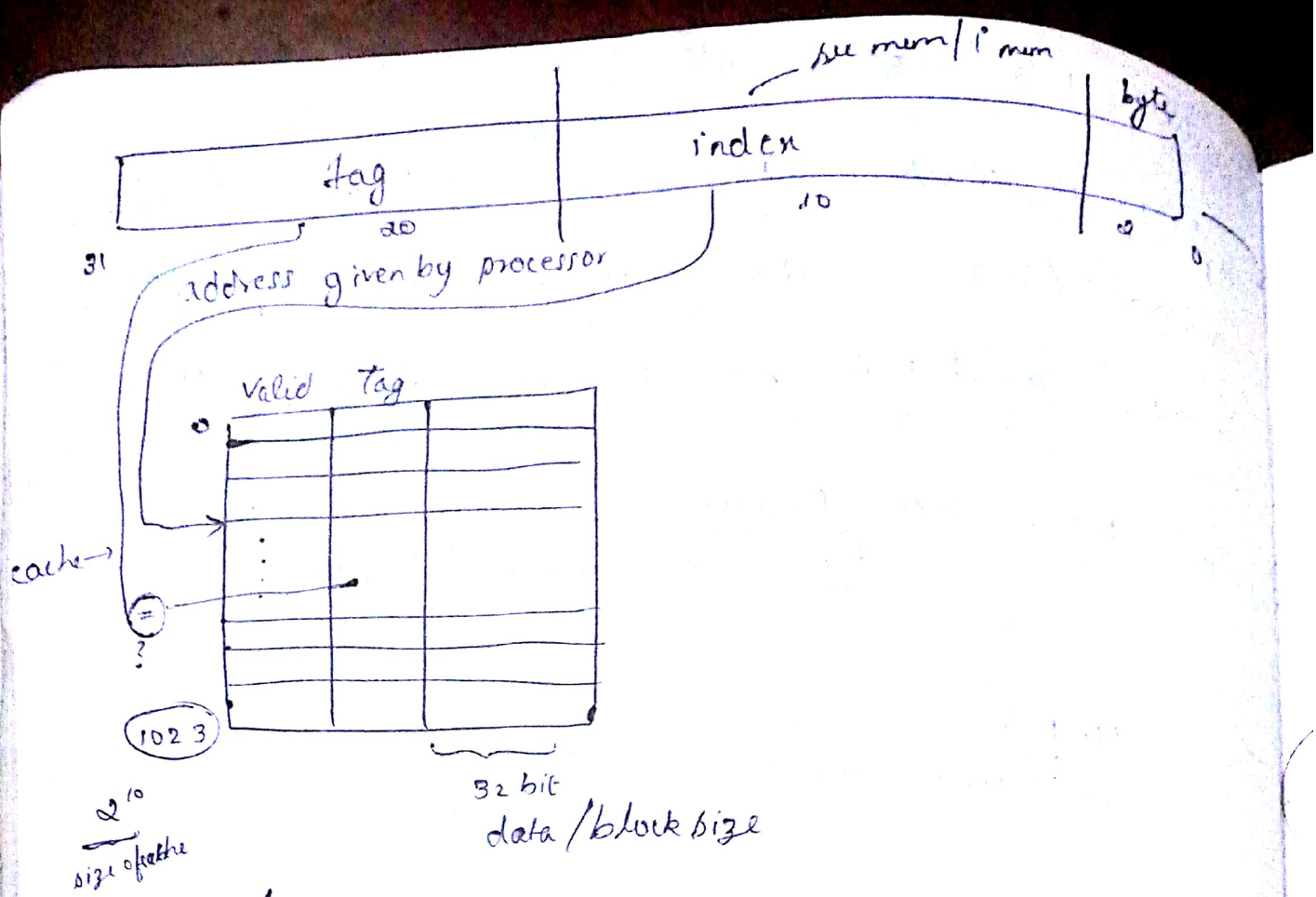
Then 10 000
tag index → miss ∴ store

Then 00011
tag index → miss ∴ store

Then 10 010
tag index → miss [index same but tag not equal]

∴ store update tag

∴ if info is changed in sec memory change valid bit of corresponding address as 'N' in cache.



here we need only part of data stored in cache (earlier we accessed all the bits) stored in cache) here if we need 1st byte of data, what to do?

→ processor req all data stored in a block

Here we use 10 bits for index

Now since to access byte by byte.

we need 2 bits to know which portion of word you need to access (words has 4 bytes)

00
10
01
11

$\frac{4}{2^2}$

∴ index − 10 tag − 20 bits.
byte portion − 2

If processor gives a 32 bit address we use its index part to find its location in cache. tag part to check if its the requested data that is stored in that location & valid bit to check validity. If its a hit, then we use the byte part to know which byte portion of the word stored in that cache location is required by processor.

6] 32 bit byte addressed directed mapped cache (→ access byte by byte)

of size 2^n blocks & block size 2^m words.

find size of tag field, total no of bits in directed mapped cache

∴ total = $2^n \cdot 2^m \times 32$.

tag field = $32 - n - 2 - m = 32 - [n + m + 2]$

(n = index to identify a word from 2^n words; m + 2 = to find the byte portion from each word; n + m = index + byte)

→ total = 2^n [Block size + tag size + valid]

(Block size = store data)

$$= 2^n \left[\underbrace{2^m \times 32}_{\text{bits}} + 32 - [n+m+2] + 1 \right]$$

Q] How many total bits r req for dir. mapped cache with 16 kB of data & 4 word block assuming 32 bit address

A. tot. block size = 16 kB

= size of 1 block × total no. of block lines

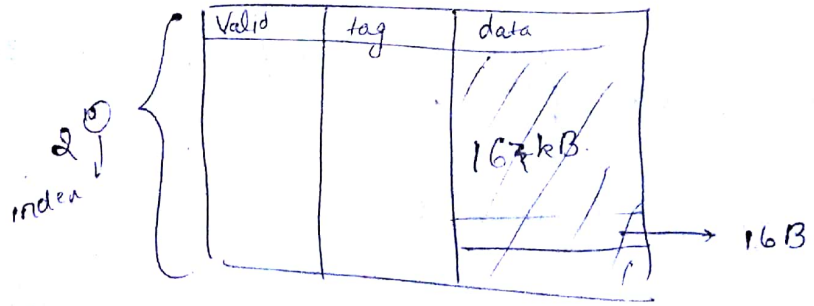
4 × 4 byte

$\Rightarrow$ total no. of block lines = $\frac{16\,kB}{16\,byte}$

$K = 1024 = 2^{10}$

= ~~1 kB~~

$= 2^{10}$



$\therefore$ tag = $32 - [10 + 2 + 2]$ (2 for word, 2 for byte)

$= 32 - [14] = 18$

$\therefore$ 1 line = $1 + 18 + 4 \times 32 = 19 + 128$

$= 147$

total = $2^{10} \times 147$ bits = 147 kBit

~~tag + valid~~ = ~~18+2 bits~~

~~total~~

Fully associative mapping

If the cache has n blocks, data add can be mapped to any one the block.

Use comparator to check tag bit of each block to that of address of data needed

$\Rightarrow$ (mm block no) % (No. of block in cache)

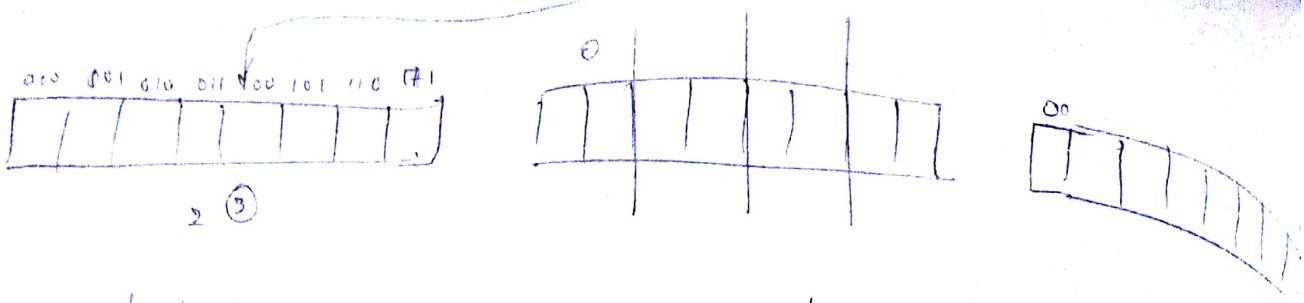
Set associative

n-way set associative $\Rightarrow$ a set has n block

first we need to identify the set & then use tag to find block within that set.

$\Rightarrow \left(\begin{matrix}\text{mm}\\ \text{block}\\ \text{no}\end{matrix}\right) \% \left(\begin{matrix}\text{no. of sets}\\ \text{in cache}\end{matrix}\right)$

eg: mem add = 12 = 1100



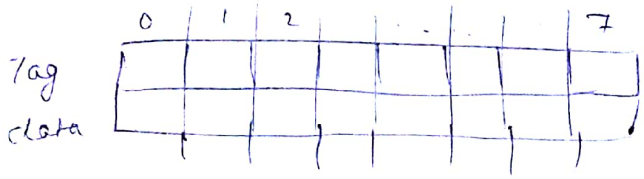
direct map

2-way set associative

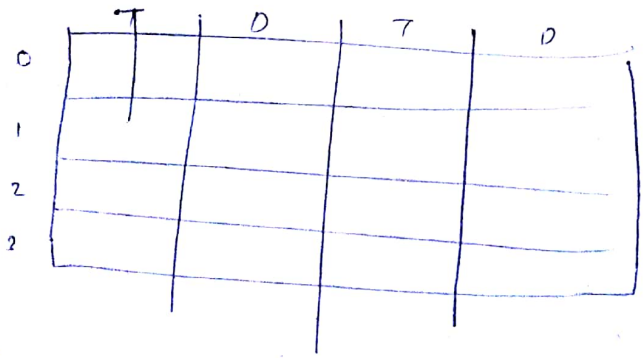
no. of set = 8/2 = 4

∴ 12 % 4 = 0

1-way set ass. with 8 block.



2-way – 8 block ⇒ 4 set



There r 3 cache each with 4 1 word block

1 is fully assouative

2 is 2 way ass

3 is directed

find no of misses for each cache for

given seq of block address

0, 8, 0, 6, 8

In 2 way set, if a set [x | y] has 2 data & another data is mapped to it, by least recent principle, the least recently used is replaced (eg if x is placed after y y is replaced)

Address block	hit/miss	Contents of cache block 0	1	2	3
0	M	m[0]			
8	M	m[8]	update		
0	M	m[0]			
6	M	~~M[7]~~		m[6]	
8	M	m[8]	update		

in dicid $\underbrace{0 \% 4 = 0}_{\text{no of block}}$; for direct mapping

$3 \% 4 = 0$

$6 \% 4 = 2$

⇒ for 2 ways - set associative

add	hit/miss	set 0	set 0	set 1	set 1
0	M	M[0]			
8	M		M[8]		
0	H	M[0]			
6	M		M[6]		
8	M	M[8]			

→ replace least recently used
∴ M[0]
∴ M[6]

no of set = 4/2 = 2
(no of block / no of set)

0 % 2 = 0 ← set
8 % 2 = 0
6 % 2 = 0

→ fully associative in a set
place in any block of set 0

direct mapped about set

⇒ for fully associative

add	H/M	0	1	2	3
0	M	M[0]			
8	M		M[8]		
0	H	M[0]			
6	M			M[6]	
8	H		M[8]		

We have to store in such a way to minimise the misses ∴ if they have free block store it there

★ 8 block & 16 block
no-of-set = 4 8

Handling Cache misses

If an instruction is checked, if its a miss in cache, we check the address of that instruction (= PC - 4) ϕ
already incr by 4 we need prev instr add

chick in 1° mem.

Handling Cache writes

After changing value of say variable n, if we change only the value of n in cache (copied from its original loc in 1° mem during fetching) the cache will have modified value ϕ 1° mem will have old value causing an inconsistancy.

To avoid this, we can use

(i) Write through:

" processor writes to cache ϕ at the same time it writes to memory.
takes more time
This causes more delay to processor

(ii) Buffered write

processor writes to buffer instead of writing to i° mem (reducing the time to write to i° mem) & mem writes/ updates itself from buffer
by processor as buff is faster

Now if buffer is full, it creates a sta

we can avoid this by:

(i) increase buffer size

(ii) keep multiple queues/ buffers

(iii) Write Back/copy back

Update i° mem only if a replacem occurs to corresponding add. in cache

or ~~write~~ copy back ~~add~~ data in cache when power is offed

---

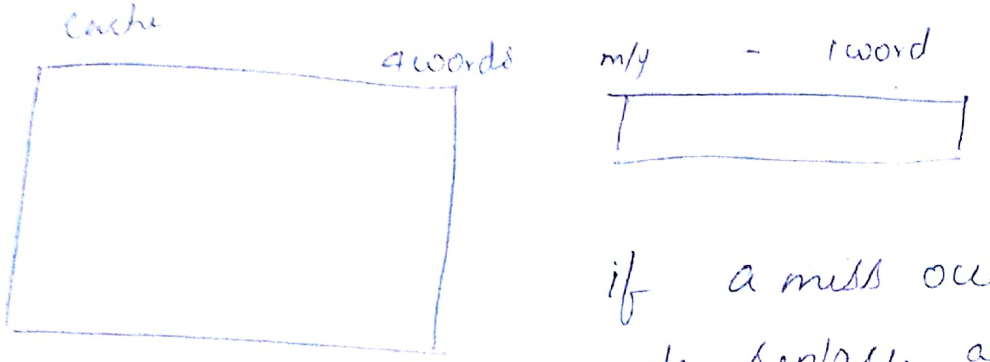
DESIGNING M/Y SYSTEMS

Q. Assume 1 mem bus clk cycle is needed to send address. 15 mem bus clk cycles for each d-ram access initiated.

i° mem

1 mem clk cycle to send a word of data

If we have a cache block of 4 words & 1 word ~~byte~~ wide bank of d-rams Calculate miss penalty in terms of mem. bus clk cycle?



if a miss occur, we have to replace all the 4 words in cache (not that 1 word alone)

If miss happens

- send add to lower mem → 1
- → access 4 words in 1 mem → 15 × 4
- → copy back data → 1 × 4 (word, data bus)

= 65

if we can access the 4 words seq. if we provide only the starting address

If we have to provide add. of each of 4 word in cache, then

$4 \times 1 + 15 \times 4 + 1 \times 4 = 68$

Now, if we ↓ cache size, miss penality ↓ but miss rate ↑.

but if we ↑ size of 1° mem (instead of ↓ cache size), then

miss penality = 1 + 15 + 4 × 1 = 20

- 1: send one add
- 15: d-ram size is 4 word ∴ only 1 access
- 4 × 1: bus = 1 byte

" it ↓

Now if we ↑ bus size,

miss penality = 1 + 15 + 1 = 17

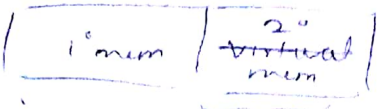
- 1: send one add
- 15: only 1 access
- 1: only 1 access to return data

- ↑ cache size ⟶ ↓ miss rate
- ↓ " ⟶ ↓ miss penality
- ↑ bus ⟶ ↓ miss penality
- ↑ 1° mem size ⟶ ↓ miss penality

Virtual memory

If RAM is having less size miss rate can be higher along with miss penalty

Here we attach a part of 2° mem (virtual) to 1° mem so that processor has a larger view. This is called virtual mem



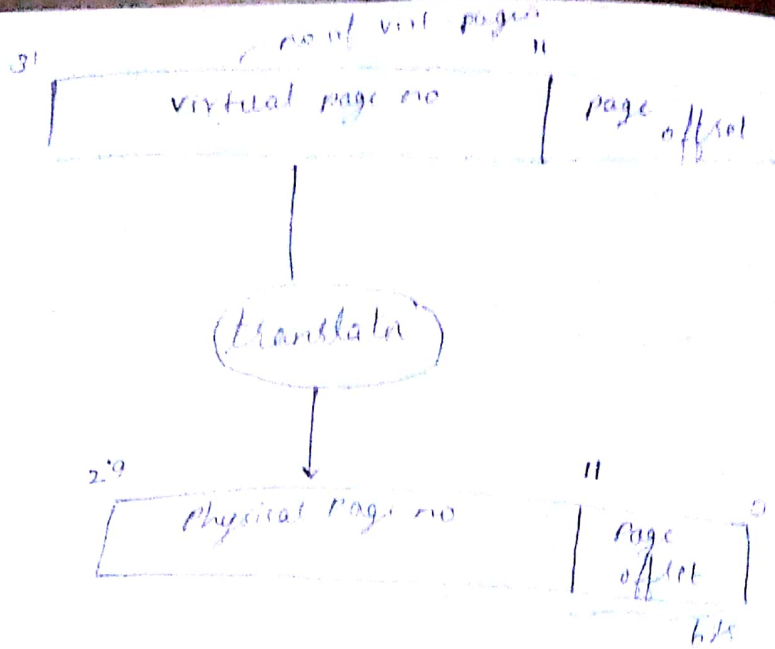
Also, virtual mem allows sharing of prgms (virtual mem)

page: block is called a page

Pagefault: miss is called pagefault

virtual add: add corresponding to virtual space

Address mapping / translation: translation of virtual add to a physical address (1° mem add) using s/w & h/w



Page size 2^{12} = 4kB

No of pages in physical/1° mem = 2^{18}

No. of pages in virtual mem = (2^{20}) - large

block

Now, we need to know whether a page is present in 1° mem / 2° mem of virtual mem

∴ we need a page table with each all having :

i) valid bit V = 0 ⇒ page fault ∴ page in 2°

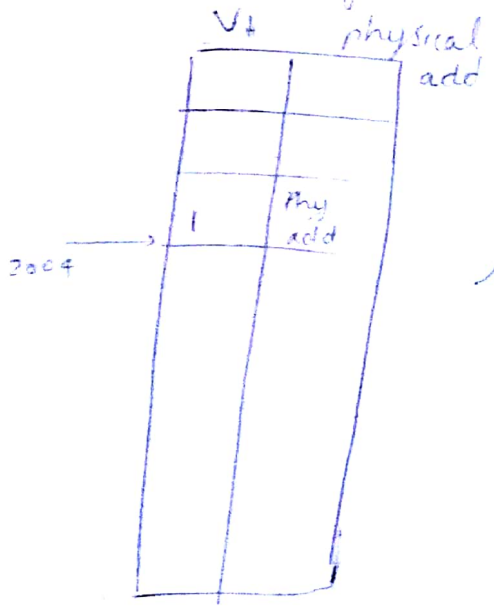
V = 1 ⇒ page in 1° mem

Once we know the virtual memaddr, we use it as an index to search the

virtual page no

page table if v = 1, the ~~add~~ phy. add corresponding to it shows page is in 1° mm

if v = 0, ⇒ page in 2° mm. then we use phy add adjacent to that index to search 2° mm & update page table by moving data to 1° mm from 2° mm



index = virtual page no

∴ no of index in page table

$= 2^{20}$

each prgm has its own page table & page table reg. store the starting add of page table.

⇒ Another type of mapping is called segmentation

⇒ Now, when we have to replace some value in 1° mem when a page fault occur in page table, we use copy back mechanism. (value in 2° mem moved to add in 1° mem & the value at add n is moved to a space called swap space in 2° mem.

⇒ & we replace the least recently used data from 1° mem incase of a page fault. To know the least recently used block, we keep an extra bit called reference bit R

⇒ When a data is used we set R=1 but after a time stamp (eg 6ms) its value is cleared saying that it is not a recently used data.

In case of cache, we need to copy back incase of replacement of if data in cache was updated keep dirty bit = 1 if cache was updated (wently & then we have to copy back data in cache ≠ data in corresponding i° mem

on before replacing the value in cache in case of miss (if dirty bit = 0, no copy back just replace)

III^rly in case of virtual mem space in page table dirty bit is set as 1 if data in that table is changed if its 1, the we have to copy back the data in that all to corresponding byte in in that all to corresponding byte in 2° mem before replacing /swaping

Rmk

Page table resides in physical mem.

Translation look aside buffer (TLB)

→ page table in cache

for 1 mem.

V	D	R	Phy add
1	0	1	
1	0	0	
0	0	0	
0	1	1	
1	0	0	

We keep cache to get faster access to page table hence we keep only the entries of the recently used index in cache

∴ TLB

V	D	R	Phy addr
V	D	R	

→ recently used index in page table of 1°

Cache TLB can store may be max of 1 or 2 entries. If cache be miss occur in this case we replace the recently used ~~If the 2 index is passed to~~

TLB miss → entry miss not that it's a page fault. that may be available in page table. Access page table modify if its page fault & then replace the least recently used entry in TLB

{ii) 1st TLB if miss goto page table update TLB after modifying page table in case of page fault }