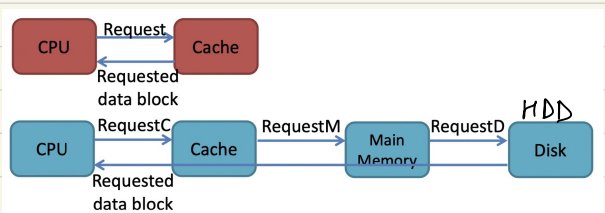
CPU ⟷ Cache ⟷ Main memory ⟷ storage memory

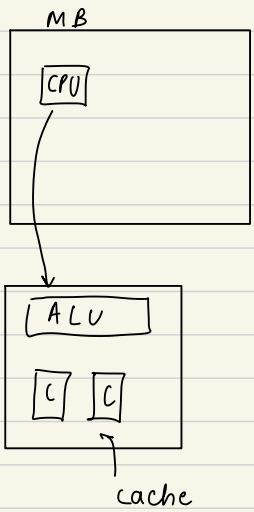
- CPU runs faster than external memory, so ext tends to slow the system down
- a fast memory is needed to buffer b/w CPU and main memory → CACHE
- it speeds up accesses by storing / fetching recently used data closer to the CPU instead of the main memory (slower access)
- best case

worst case



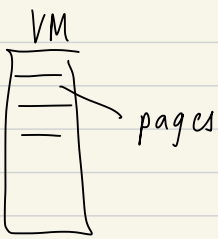
- the required data and a number of its nearby data elements are fetched into cache

tut info

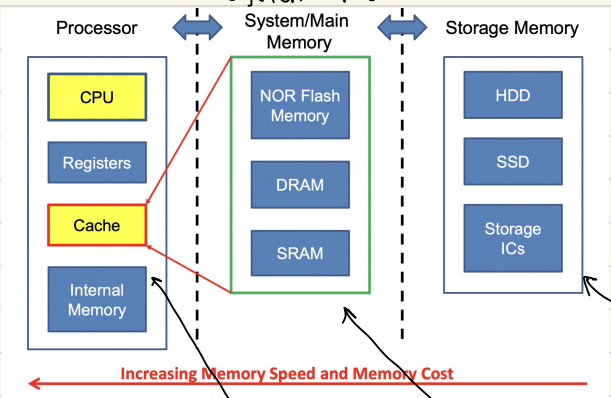


memory is copied in blocks

if loops are maintained in blocks, program can run out of cache itself



system = main



complete code and data is stored here

most recent/ frequently used subsets are stored here.

small parts are transferred here during run-time

example :

cache	system	storage
draw_rect() function	insert object module	PP app.

Principle of locality

↓

once a byte in a program is accessed, it is likely that a nearby data element will soon be needed.

→ locality of space : code/data that is close by is likely to be accessed together

↑ used to justify transferring data b/w main memory and cache in blocks

→ locality of time : recently accessed code is likely to be accessed again

↑ used to decide which item to replace in cache

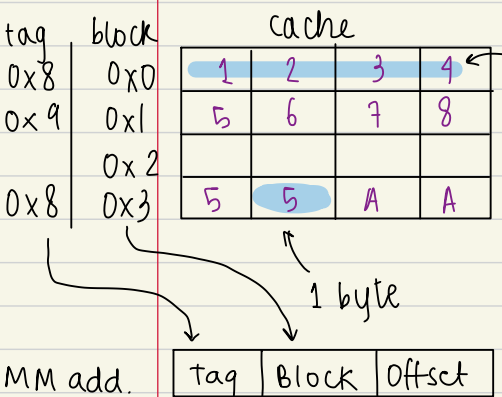
ex. for (i = 0; i < 10; i++) ← temporal / time
y[i] = a[i] * x[i] ← local / space

Cache Mapping Scheme

↓

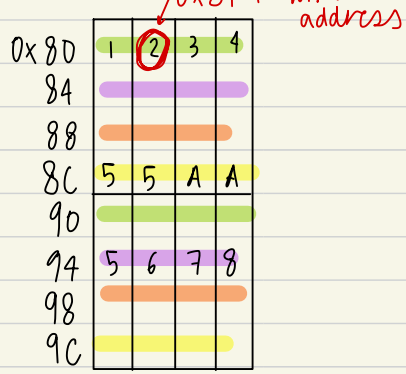
deals with how each main memory block is mapped to the particular cache block.

- Set Associated
- Fully Associate
- Direct Mapped



cache block / line size = 4

usually main memory size is equal to cache block size



cache : N blocks
X : block in main memory
Y : cache block no
Y = X mod N

no of bits of offset : $\log_2$(size of block) → 2 (here)
no of block bits : $\log_2$(size of cache / no of cache lines) → 2 (here)
no of tag bits : MM − 2 − 2 → 4 (here)

↓

8

∴ 0x81 → 1000|00|01

offset : 1
block : 0
tag : 8

Cache mapping

↓

enables us to uniquely identify each and every main memory location

- since CPU will use main memory address as reference to recieve into, we use that only to enable the mapping

ex. MM : 64K bytes → 2^{16}
cache : 256 bytes → 2^8
block : 2^4

offset : 4 ; block : $2^8/2^4 \rightarrow 4$; tag : $16-4-4 = 8$

0x1106 → 0001 0001 | 0000 | 0110

tag = 0x11
block = 0x0
offset = 0x6

- CPU checks if the correct data from MM is in cache by comparing the tag value associated with that block with the tag value derived from MM's data's address

- if tag values match, it's a cache hit
else, it's a cache miss in which case, the corresponding data is transferred into cache

so we know which block to evict

not for direct mapping as that is one-to-one

Cache Memory Replacement Policy

when cache is full, deciding which block gets evicted is based on this policy

- Least recently used (LRU) algo keeps track of the last time a block was accessed and evicts longest unused one
  disad: maintaining access history which slows cache down
- First-in first-out (FIFO): the one thats been there longest
  disad: less efficient than LRU, does not take reusability into account

Cache write policy

- CPU writes in cache → its a Dirty block (a special marking is done)
- dirty blocks must be written into memory (before eviction to maintain data coherency)
- write through: CPU updates cache and main memory simultaneously (increases traffic on system bus)
- write back / copy back: memory is only updated when block is selected for replacement.
  (potential cache coherency issue: copy in cache and mm is not the same.)

write hit: CPU wants to write to MM and that block is in the cache.

write miss handle

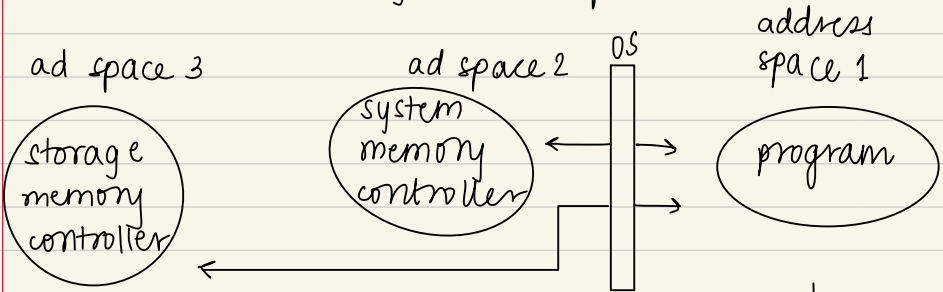
- write allocate : MM block is loaded to cache
  CPU reads cache
  ↑ future read writes are yield a cache hit
  follow one of the dirty blocks procedures
- write no-allocate : CPU will write directly to MM block.

Effective Access Time

- cache & MM are sequentially accessed
  ⇒ access does not overlap
- $EAT = H \times Access_{cache} + (1-H) \times (Access_{cache} + Access_{mm})$
  ↑
  only read operations

virtual memory

memory address space



storage memory addressing (LBA)

- logical block addressing
- stores the entire program code and data.

system memory addressing

- stores a subset of the total program code and data.

↑ allocated by OS

Virtual Memory addressing

- uses addresses generated by the compiler and allocated by the OS
- each program has its own VMA

- data is stored in different formats throughout
∴ address translation is required.

↑ done by OS

- address space : range of discrete addresses corresponding to some physical or logical entity
- during prog compilation, every piece of data/code is assigned an address by the compiler
- ↑ those addresses are provided to the OS to get corr. info
- compiler generated address $\xrightarrow{OS}$ system memory address

PHYSICAL ≡ MAIN ≡ SYSTEM

a logical entity

Virtual memory management

- each prog has its own VAS (virtual address space)
- virtual address : address used by prog, gen by compiler

- advantages :

→ OS can isolate each VMS and prevent corruption
→ safe and efficient sharing among different programs with shared physical memory
→ allow I/t prog to run in the physical/system/main memory simultaneously even if total size of all is larger than PM size.

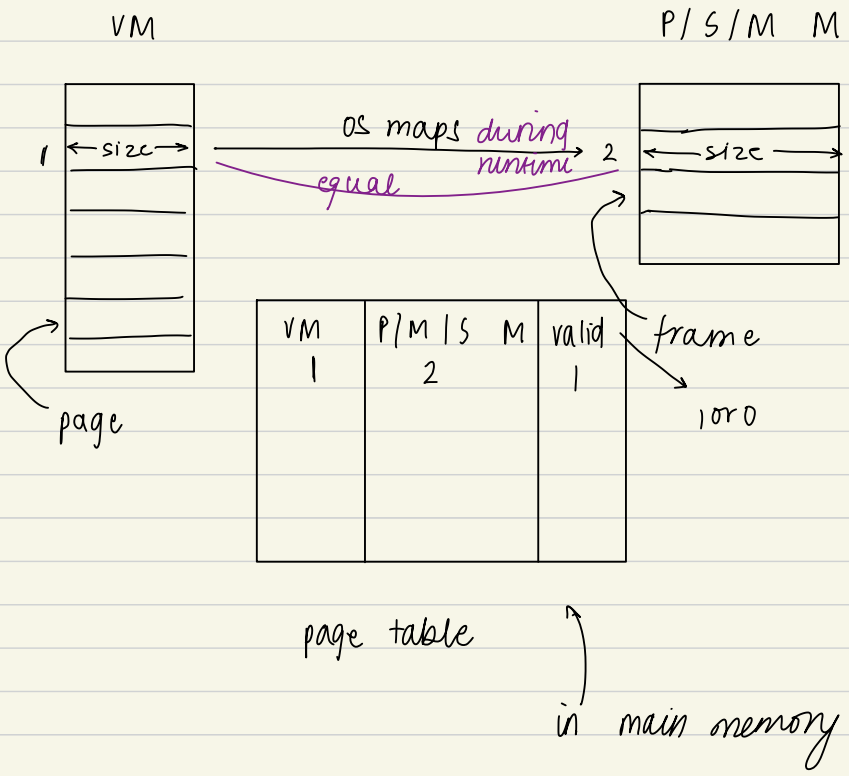
→ VM uses both hw and sw to enable a computer to compensate for physical memory shortages
↓
temp transferring data from RAM to disk storage

→ VM can be contiguous even if that info is not cont. in physical memory
↳ good as software needn't do complicated boundary condition checking and management.

address mapping

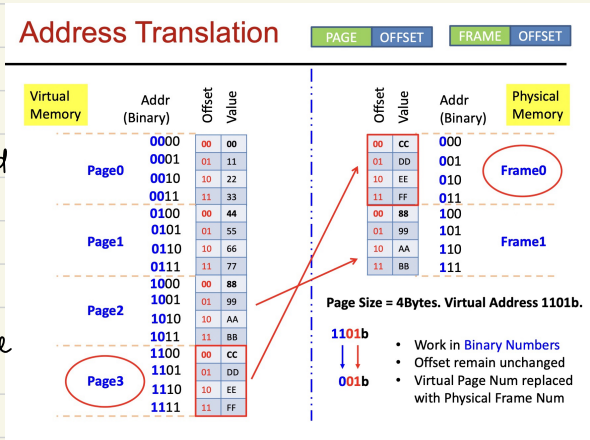
→ paging : mem space partitioned into fixed sized blks
→ segmentation : mem space partitioned into variable sized segments.

PAGING



address translation

→ VM : 16 bytes → 4 bit Ad
→ PM : 8 bytes → 3 bit Ad
→ page size : 4 bytes
↙
2 bits



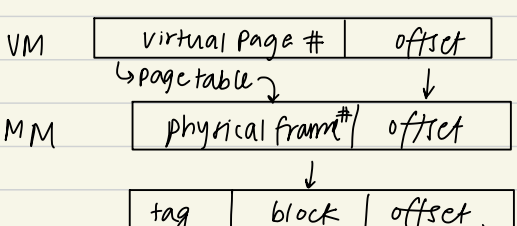
- offset remains the same
- page no is replaced w/ frame no.

relative position of data in the frame doesn't change

found from page table

- if valid bit is 0 ⇒ page is not there, must be loaded from storage to physical memory

- cache & virtual memory



① VM → MM add.
② CPU → cache hit/miss
③ hit → CPU takes info from cache
miss → cache manager transfers info from system/storage mem.

Translation lookaside buffer (TLB)

↓

SRAM → page table cache ← contains subset of the table

(or some fast memory)

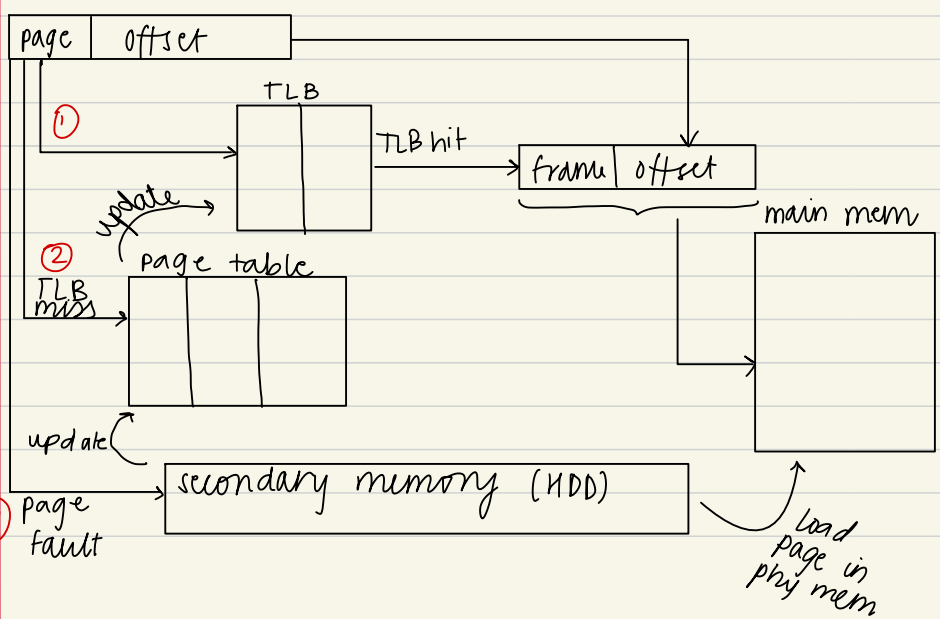
↑ for quick access ∵ page table exists in MM.

in the processor

page #	frame #

this info is "address translation information"

all entries are valid



8

0010 | 00 | 00

2, 2

EAT : $0.95 \times 20 + 0.05 \times 120$

$19 + 6 = 25$

64K : 16

32 → 5

size : 16 - 5 = 11 → 2K

off : 10

00 | 11 0000 0000

00 → 3

1111 0000 0000

F00

K : 10

64 : 6

256 : 8

0000 1010 0000 0011

offset

0010 03

t 2	b 4		w 2
01	0100	4	01
10	0101	5	01
10	1000	8	11
10	1100	12	11
11	0101	5	01
01	0100	4	01