Virtual Memory III

CSE 351 Winter 2021

Instructor:

Mark Wyse

Teaching Assistants:

Kyrie Dowling

Catherine Guevara

Ian Hsiao

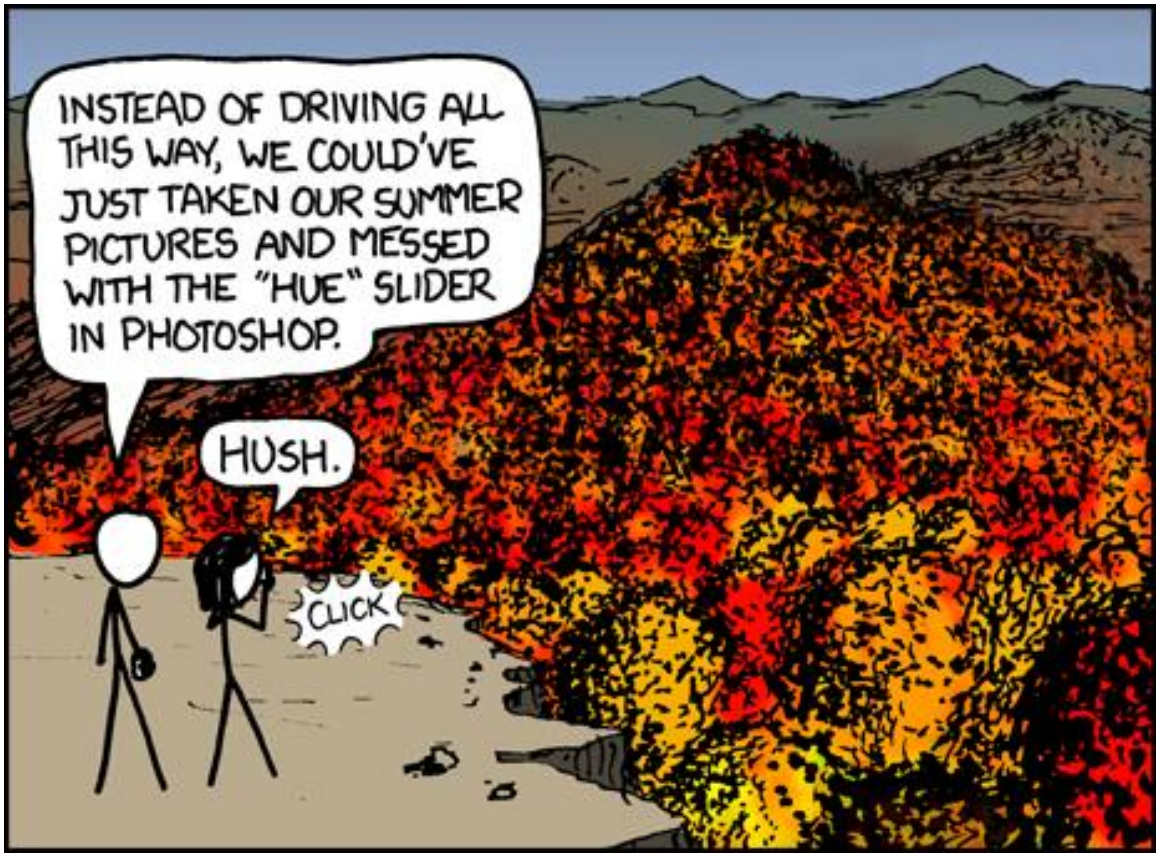
Jim Limprasert

Armin Magness

Allie Pfleger

Cosmo Wang

Ronald Widjaja



https://xkcd.com/648/

Administrivia

- hw18 due Tonight!
- hw 19 due Monday (3/1)
- Study Guide 2 due Monday (3/1)
- hw20 due Friday (3/5)
- Lab 4 due Friday (3/5)

Reading Review

- Terminology:
  - Address translation: page hit, page fault
  - Translation Lookaside Buffer (TLB): TLB Hit, TLB Miss

cache VPN → PPN

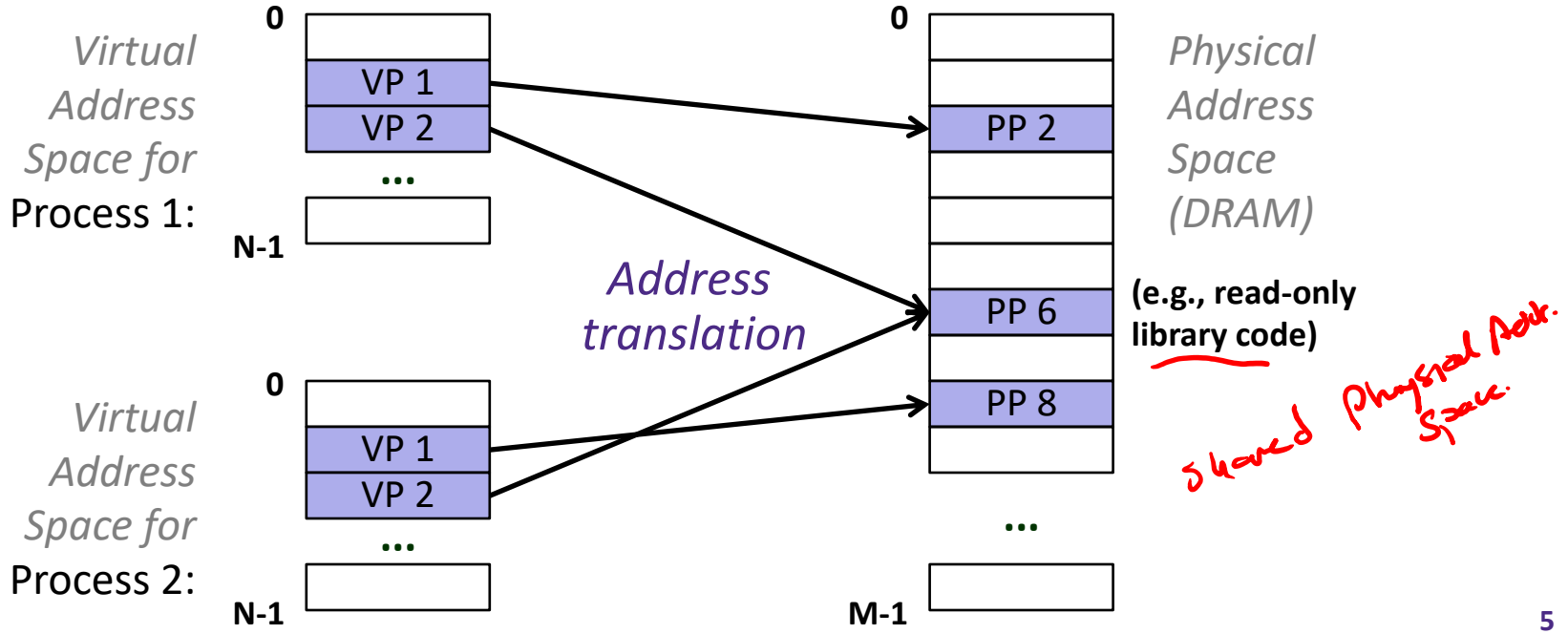
- Questions from the Reading?

Virtual Memory (VM)

- Overview and motivation
- VM as a tool for caching
- Address translation
- **VM as a tool for memory management**
- **VM as a tool for memory protection**

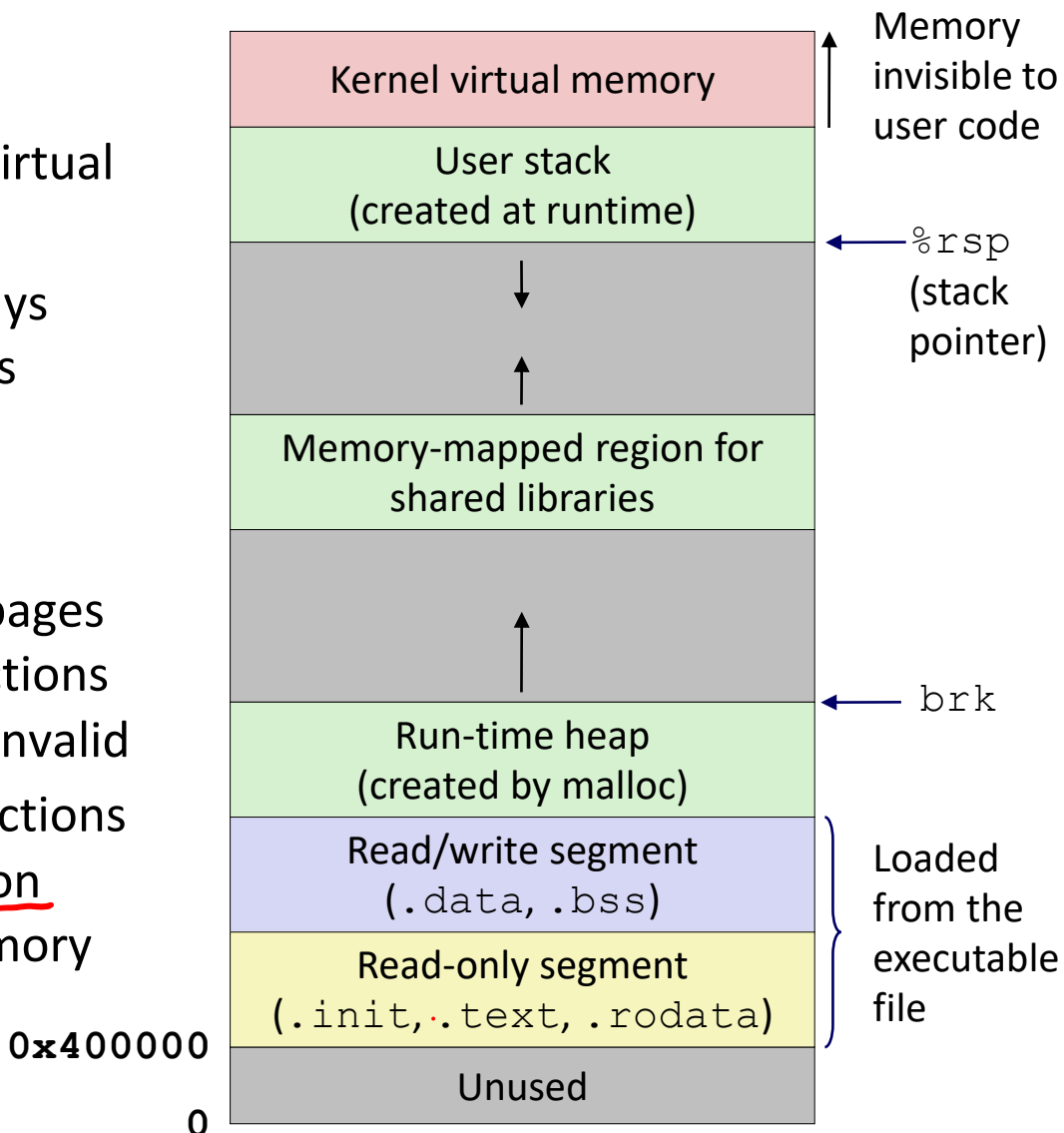
VM for Managing Multiple Processes

- Key abstraction: each process has its own virtual address space
  - It can view memory as *a simple linear array*
- With virtual memory, this simple linear virtual address space need not be contiguous in physical memory
  - Process needs to store data in another VP? Just map it to *any* PP!



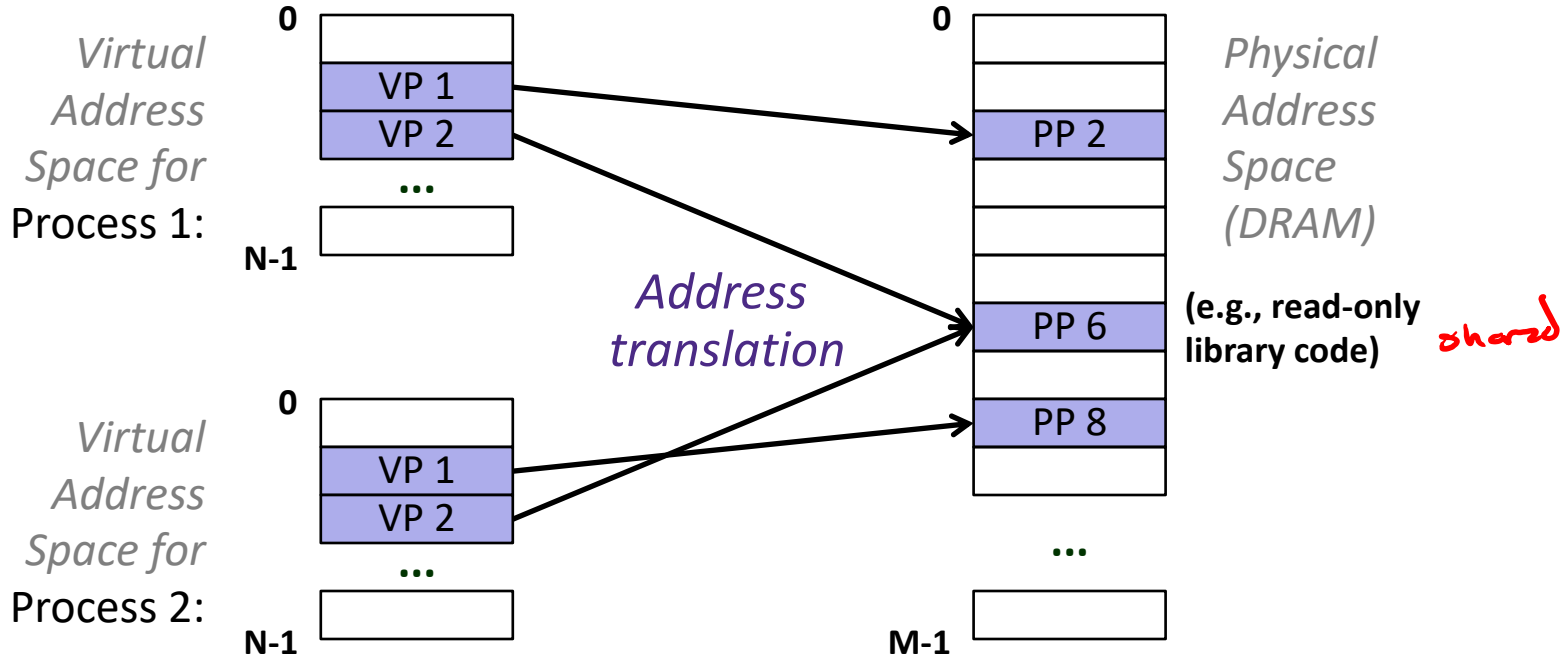
Simplifying Linking and Loading

- Linking
  - Each program has similar virtual address space
  - Code, Data, and Heap always start at the same addresses
- Loading
  - `execve` allocates virtual pages for `.text` and `.data` sections & creates PTEs marked as invalid
  - The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system



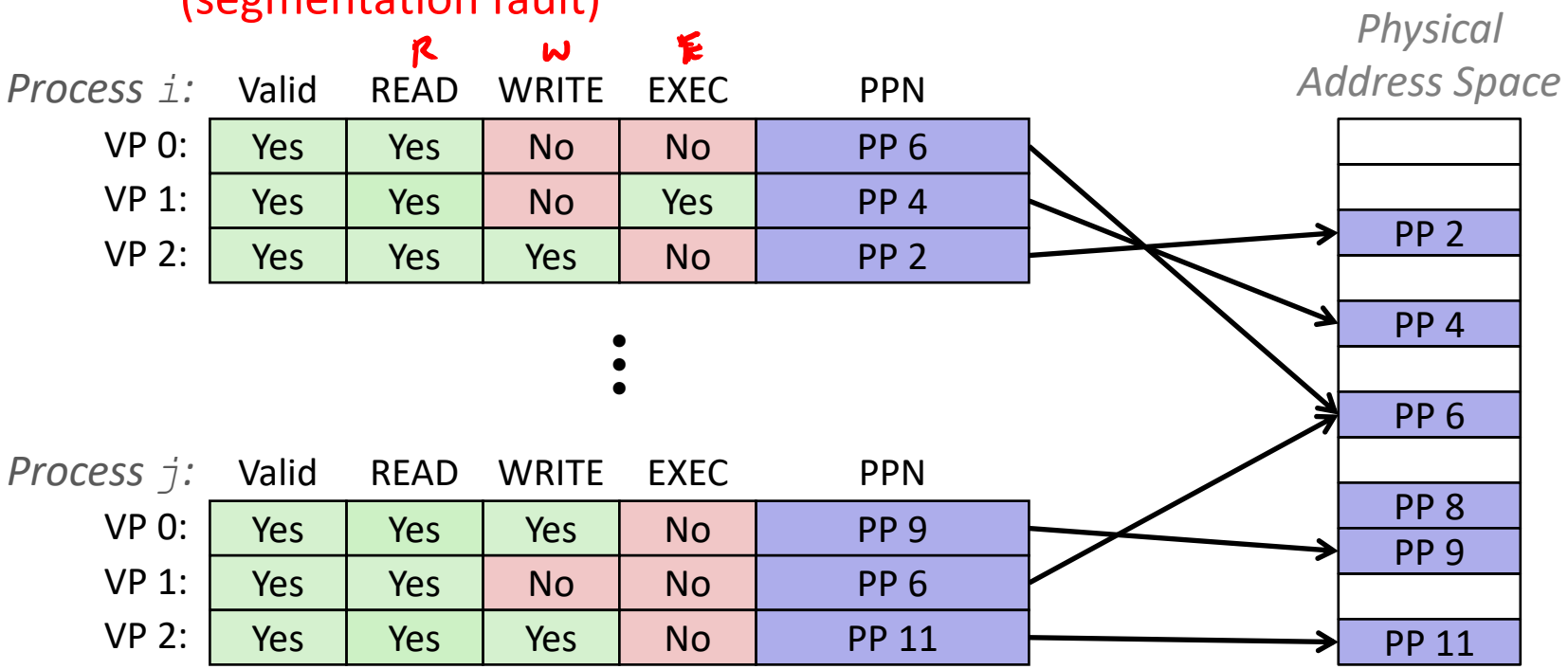
VM for Protection and Sharing

- The mapping of VPs to PPs provides a simple mechanism to *protect* memory and to *share* memory between processes
  - **Sharing:** map virtual pages in separate address spaces to the same physical page (here: PP 6)
  - **Protection:** process can't access physical pages to which none of its virtual pages are mapped (here: Process 2 can't access PP 2)



Memory Protection Within Process

- VM implements read/write/execute permissions
  - Extend page table entries with permission bits
  - MMU checks these permission bits on every memory access
    - If violated, raises exception and OS sends SIGSEGV signal to process (segmentation fault)



Memory Review Question

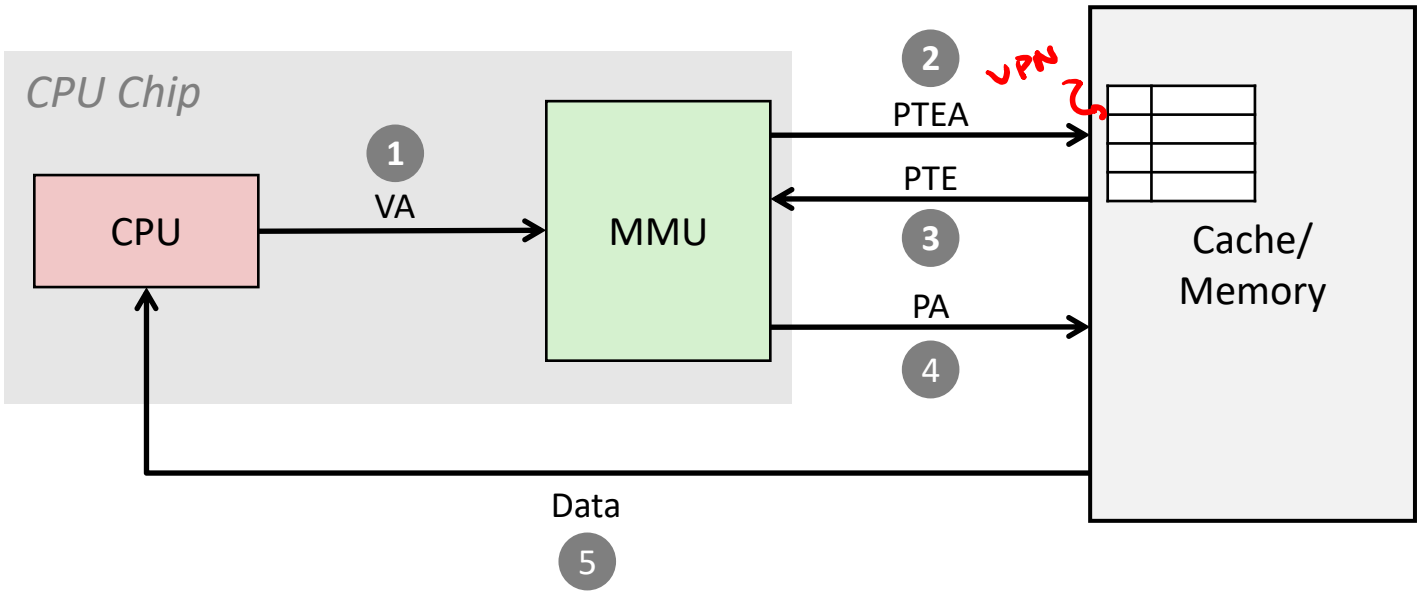
- What should the permission bits be for pages from the following sections of virtual memory?

Section	Read	Write	Execute
Stack	1	1	0
Heap	1	1	0
Static Data	1	1	0
Literals	1	0	0
Instructions	1	0 *	1

Address Translation

- Page Hits and Misses
- Accelerating Translation with the TLB

Address Translation: Page Hit → "page is in physical memory"



1) Processor sends *virtual* address to MMU (*memory management unit*)

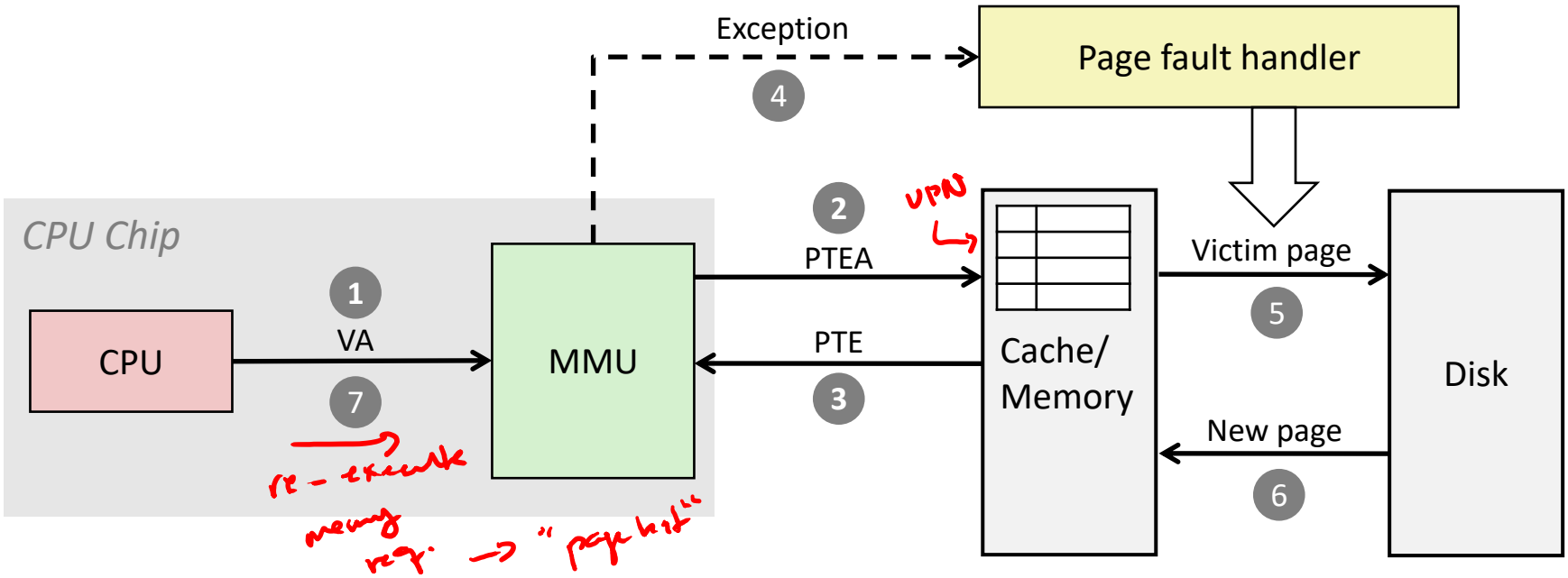
2-3) MMU fetches PTE from page table in cache/memory
(Uses PTBR to find beginning of page table for current process)

4) MMU sends *physical* address to cache/memory requesting data

5) Cache/memory sends data to processor

VA = Virtual Address PTEA = Page Table Entry Address PTE= Page Table Entry
PA = Physical Address Data = Contents of memory stored at VA originally requested by CPU

Address Translation: Page Fault



1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in cache/memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction

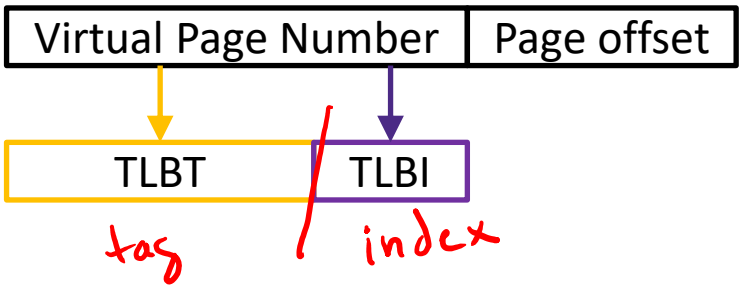
Hmm… Translation Sounds Slow

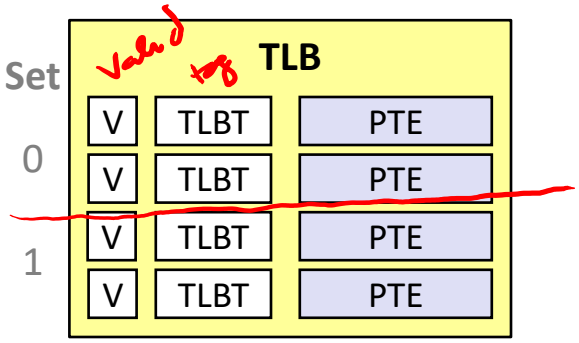
- The MMU accesses memory *twice*: once to get the PTE for translation, and then again for the actual memory request
  - The PTEs *may* be cached in L1 like any other memory word
    - But they may be evicted by other data references
    - And a hit in the L1 cache still requires 1-3 cycles

- *What can we do to make this faster?*
  - **Solution:** add another cache! 

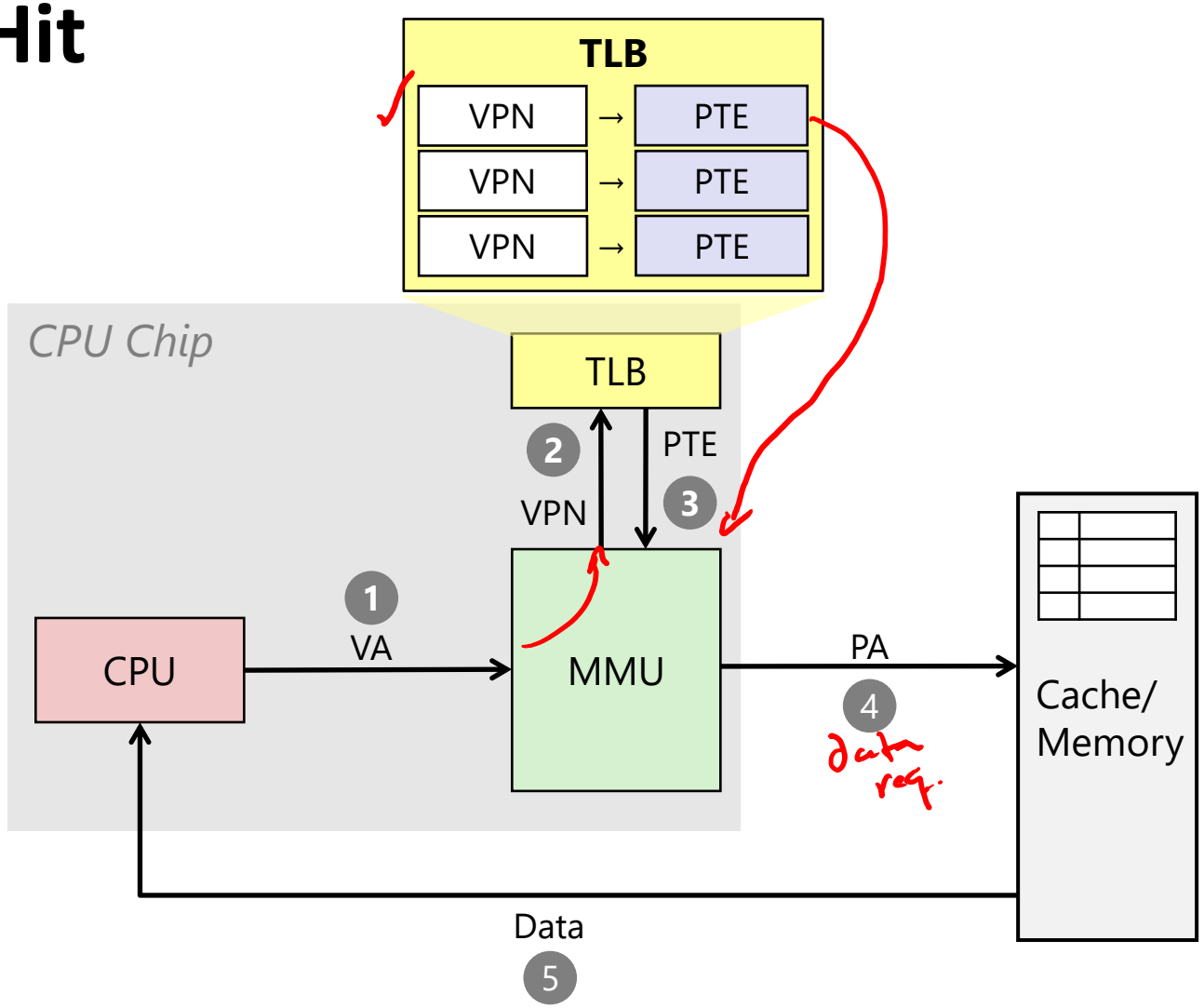
Speeding up Translation with a TLB

- *Translation Lookaside Buffer* (TLB):
  - Small hardware cache in MMU
    - Split VPN into **TLB Tag** and **TLB Index** based on # of sets in TLB
  - Maps virtual page numbers to physical page numbers
  - Stores *page table entries* for a small number of pages
    - Modern Intel processors have 128 or 256 entries in TLB
  - Much faster than a page table lookup in cache/memory



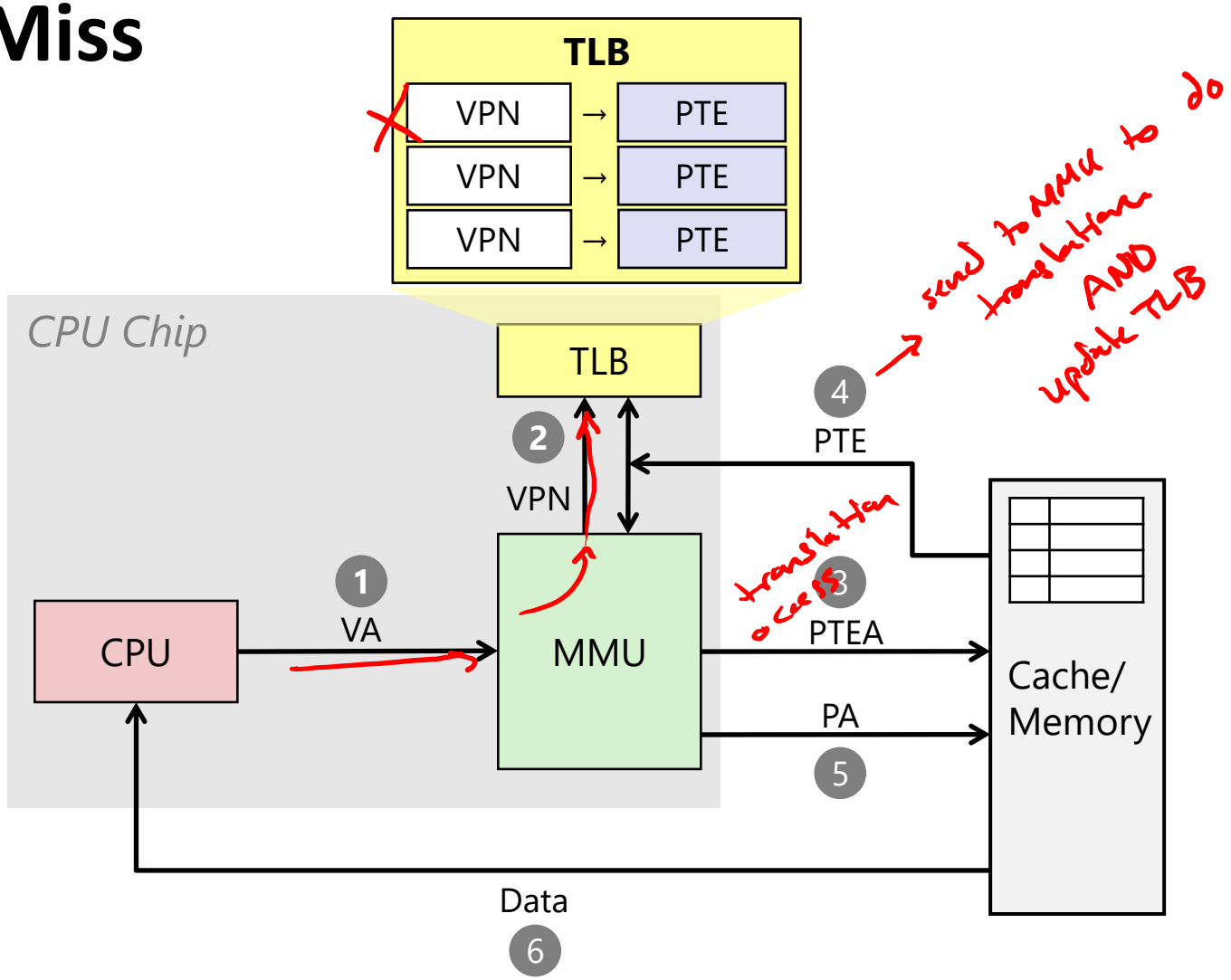


TLB Hit



- A TLB hit eliminates a memory access!

TLB Miss

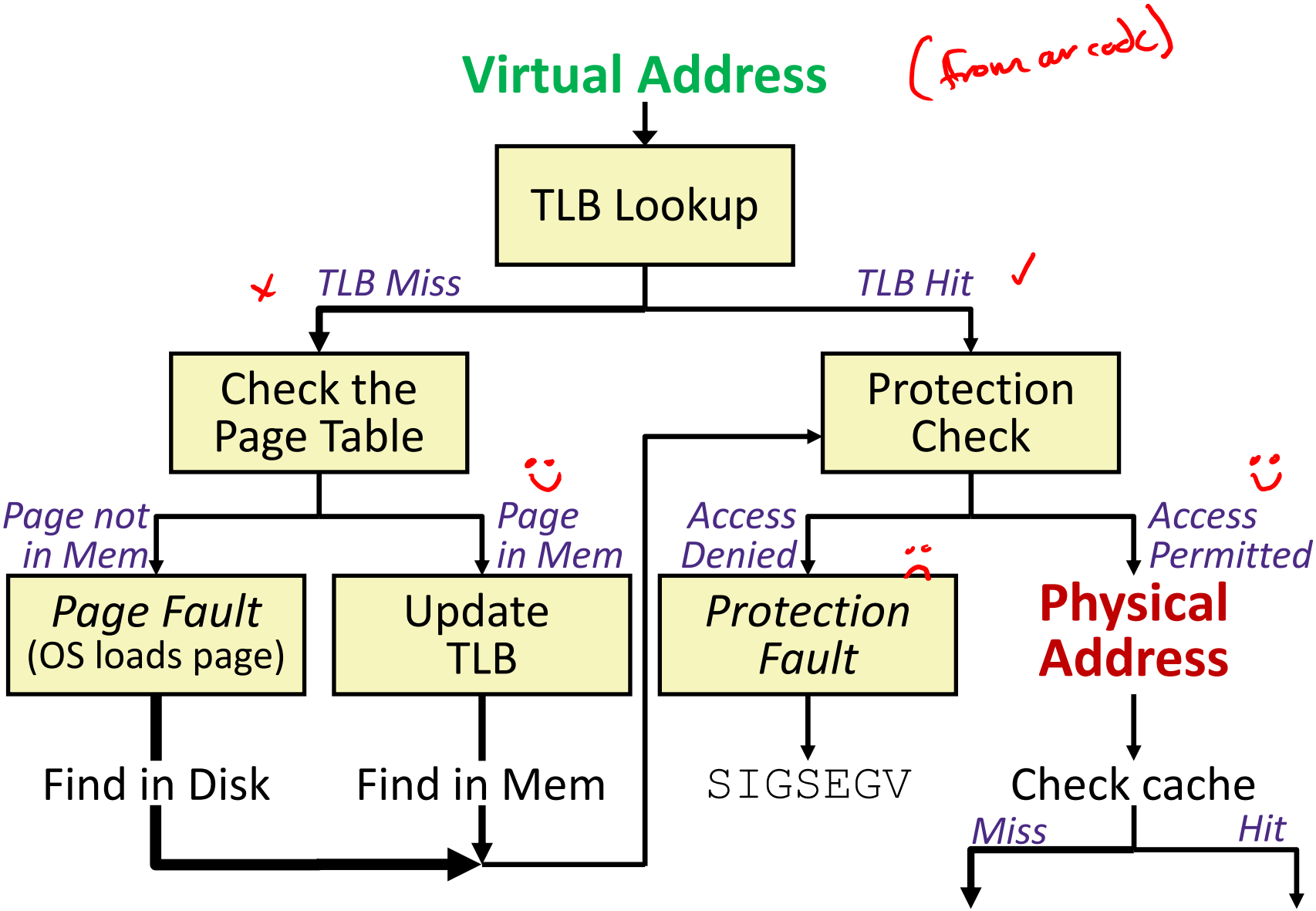


- A TLB miss incurs an additional memory access (the PTE)
  - Fortunately, TLB misses are rare

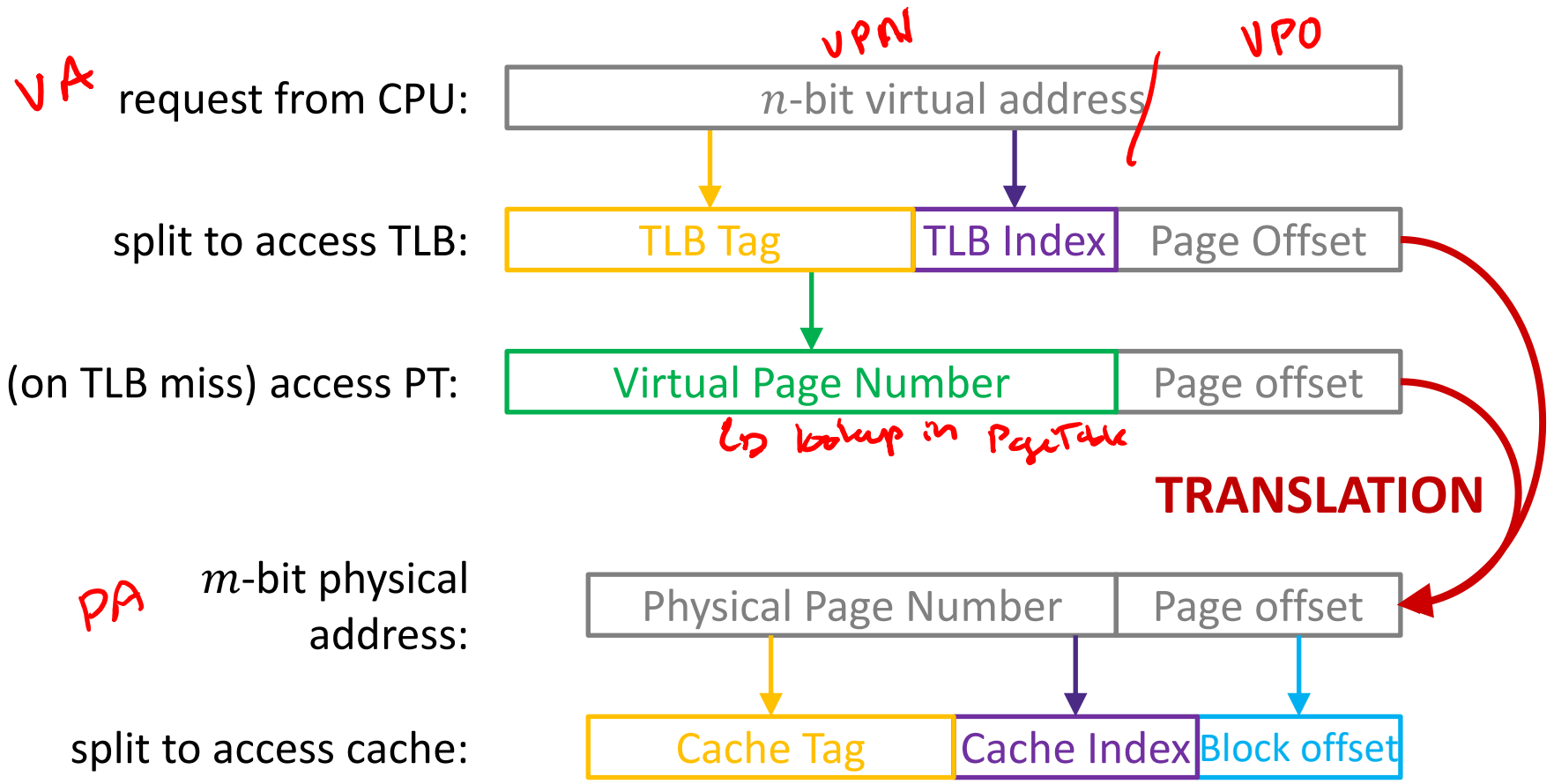
Fetching Data on a Memory Read

1) Check TLB VA->PA
   - Input: VPN, Output: PPN
   - *TLB Hit:* Fetch translation, return PPN
   - *TLB Miss:* Check page table (in memory)
     - *Page Table Hit:* Load page table entry into TLB
     - *Page Fault:* Fetch page from disk to memory, update corresponding page table entry, then load entry into TLB
2) Check cache fetch data
   - Input: physical address, Output: data
   - *Cache Hit:* Return data value to processor
   - *Cache Miss:* Fetch data value from memory, store it in cache, return it to processor

Address Translation



Address Manipulation



Context Switching Revisited

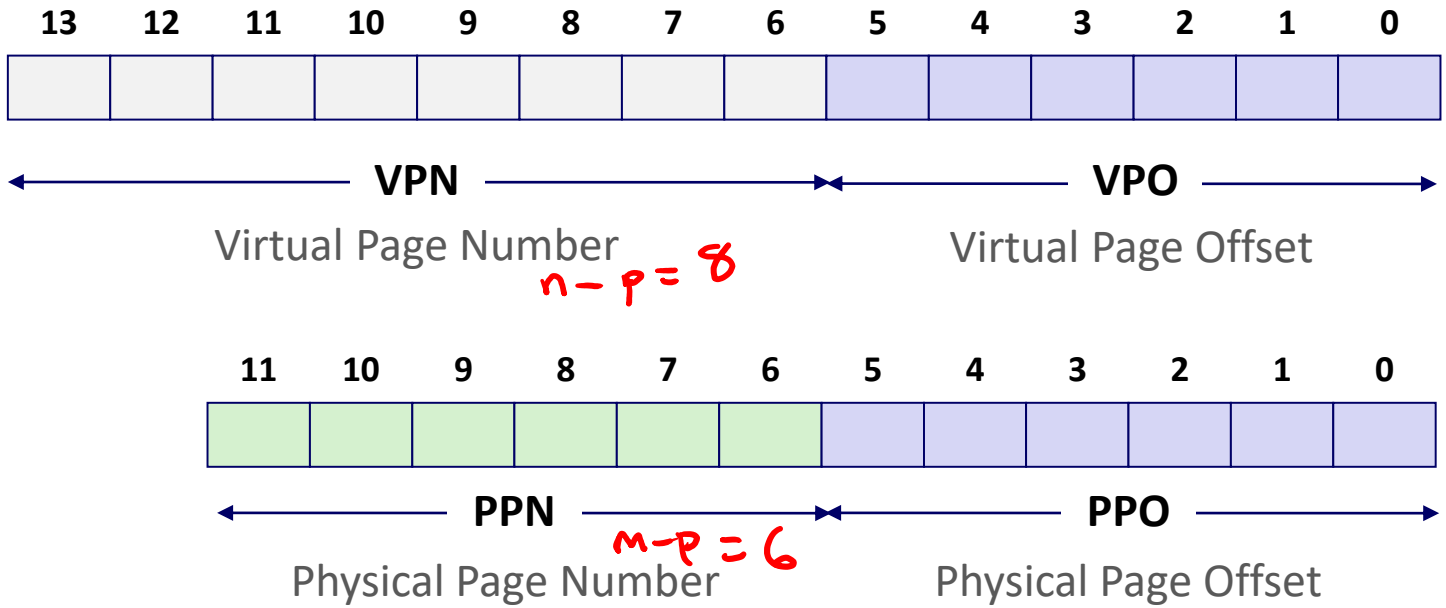
- What needs to happen when the CPU switches processes?
  - Registers:
    - Save state of old process, load state of new process
    - ✓ Including the Page Table Base Register (PTBR)
  - Memory:
    - Nothing to do! Pages for processes already exist in memory/disk and protected from each other
  - TLB:
    - *Invalidate* all entries in TLB – mapping is for old process' VAs
  - Cache:
    - Can leave alone because storing based on PAs – good for shared data

Summary of Address Translation Symbols

- Basic Parameters
  - $\mathrm{N} = 2^n$ Number of addresses in virtual address space
  - $\mathrm{M} = 2^m$ Number of addresses in physical address space
  - $\mathrm{P} = 2^p$ Page size (bytes)
- Components of the virtual address (VA)
  - **VPO** Virtual page offset
  - **VPN** Virtual page number
  - **TLBI** TLB index
  - **TLBT** TLB tag
- Components of the physical address (PA)
  - **PPO** Physical page offset (same as VPO)
  - **PPN** Physical page number

Simple Memory System Example (small)

- Addressing
  - 14-bit virtual addresses $n = 14 \rightarrow 16$ K.B
  - 12-bit physical address $m = 12 \rightarrow 4$ K.B
  - Page size = 64 bytes $p = 6$



Simple Memory System: Page Table

$2^{n-p} = 2^8$

- Only showing first 16 entries (out of 256)
  - **Note:** showing 2 hex digits for PPN even though only 6 bits
  - **Note:** other management bits not shown, but part of PTE

VPN	PPN	Valid
0	28	1
1	–	0
2	33	1
3	02	1
4	–	0
5	16	1
6	–	0
7	–	0

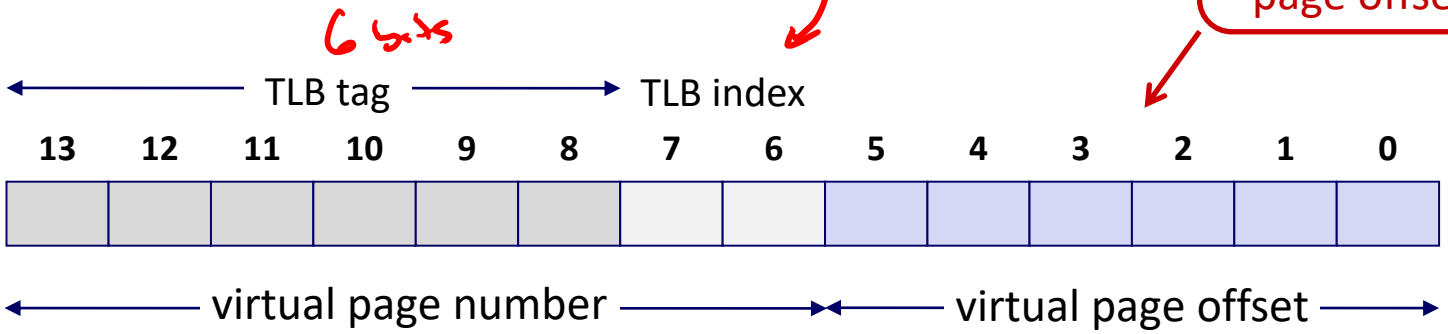
VPN	PPN	Valid
8	13	1
9	17	1
A	09	1
B	–	0
C	–	0
D	2D	1
E	–	0
F	0D	1

Simple Memory System: TLB

- 16 entries total
- 4-way set associative

16/4 = 4 sets
TLBI = 2 bits

Why does the TLB ignore the page offset?



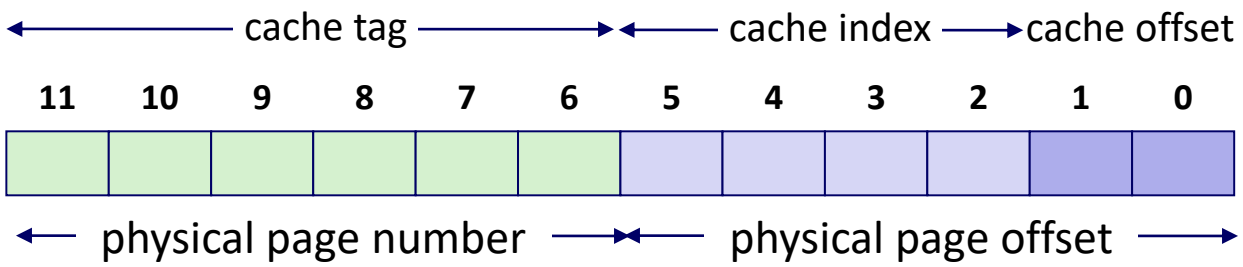
Set	Way 0 Tag	PPN	Valid	Way 1 Tag	PPN	Valid	Way 2 Tag	PPN	Valid	Way 3 Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Simple Memory System: Cache

Note: It is just coincidence that the PPN is the same width as the cache Tag

E=1

- Direct-mapped with K = 4 B, C/K = 16 4 Sets
- Physically addressed m= 12 bits



Index	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Index	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Current State of Memory System

TLB:

Set	Tag	PPN	V	Tag	PPN	V	Tag	PPN	V	Tag	PPN	V
0	03	–	0	09	0D	1	00	28	1	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Page table (partial):

VPN	PPN	V
0	28	1
1	–	0
2	33	1
3	02	1
4	–	0
5	16	1
6	–	0
7	–	0

VPN	PPN	V
8	13	1
9	17	1
A	09	1
B	–	0
C	–	0
D	2D	1
E	–	0
F	0D	1

Cache:

Index	Tag	V	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

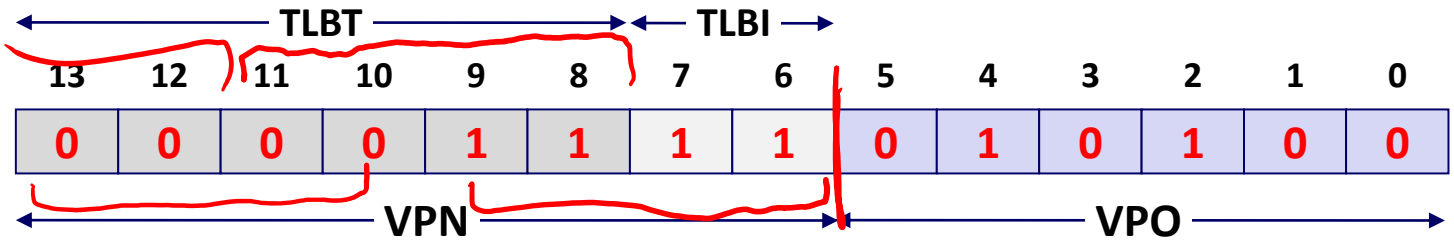
Index	Tag	V	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Memory Request Example #1

TLB Hit, Cache Hit

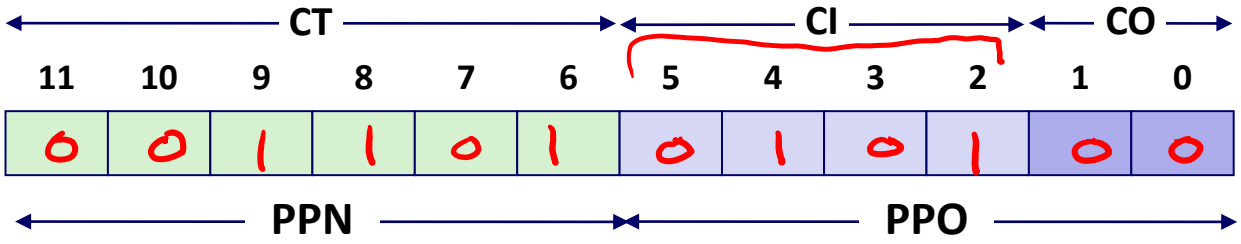
Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: 0x03D4



VPN 0x0F TLBT 0x03 TLBI 3 TLB Hit? Y Page Fault? N PPN 0x0D

- Physical Address:



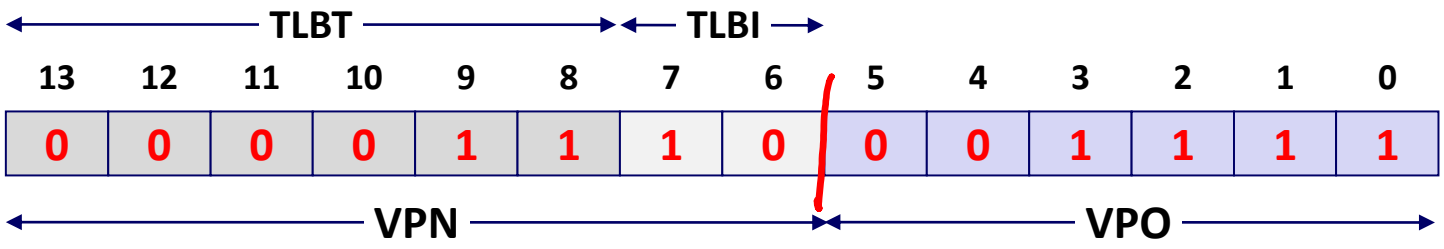
CT 0x0D CI 5 CO 0 Cache Hit? Y Data (byte) 0x36

Memory Request Example #2

Note: It is just coincidence that the PPN is the same width as the cache Tag

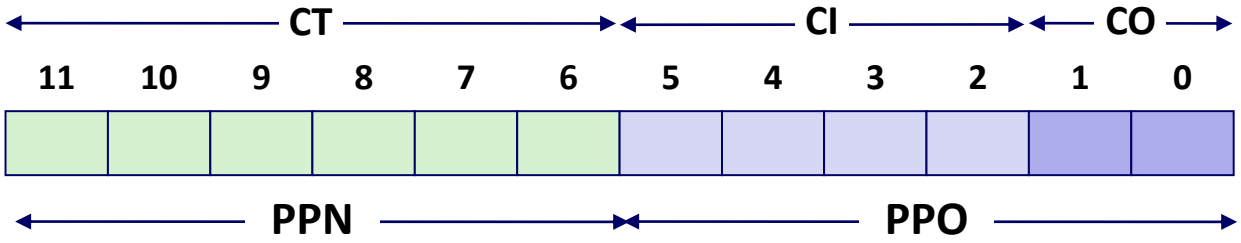
TLB Miss, Page Fault!

- Virtual Address: 0x038F



VPN 0x0E TLBT 0x03 TLBI 2 TLB Hit? N Page Fault? Y PPN n/a

- Physical Address:



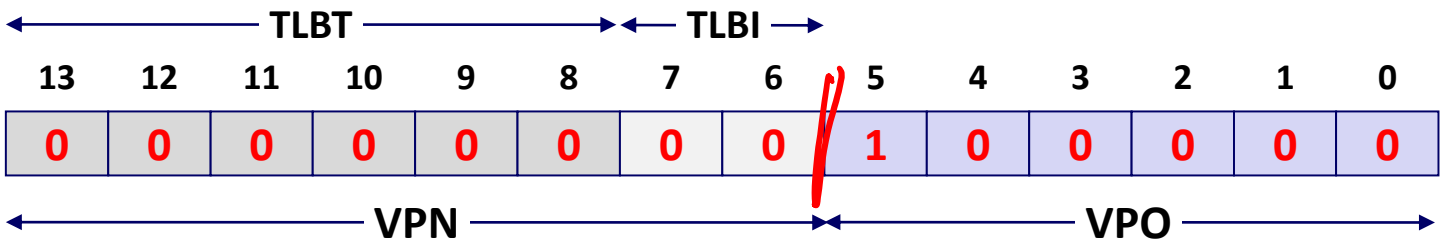
CT _____ CI _____ CO _____ Cache Hit? ___ Data (byte) _____

Memory Request Example #3

Note: It is just coincidence that the PPN is the same width as the cache Tag

TLB Miss, Page Table Hit, Cache Miss

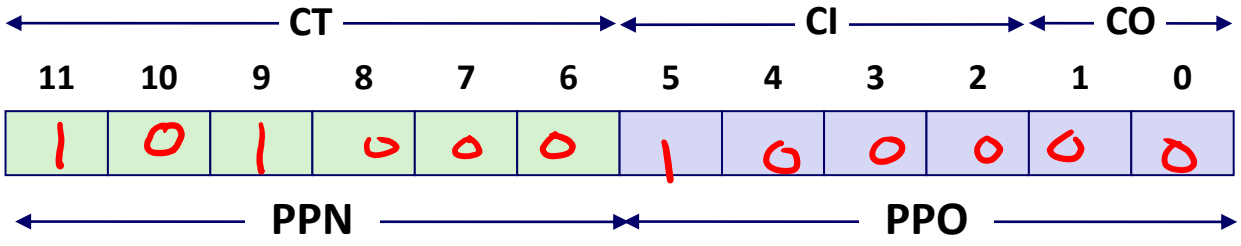
- Virtual Address: 0x0020



VPN 0x00 TLBT 0x00 TLBI 0x0 TLB Hit? N Page Fault? N PPN 0x28

PTE was V

- Physical Address:



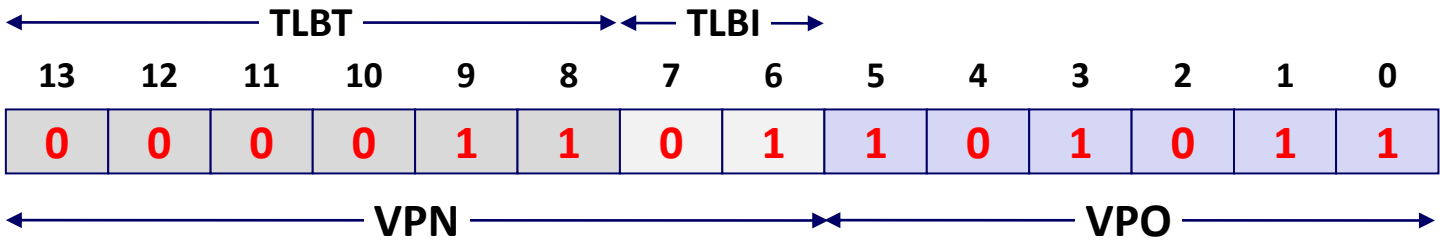
CT 0x28 CI 8 CO 0 Cache Hit? N Data (byte) n/a

Memory Request Example #4

TLB Hit, Cache Hit

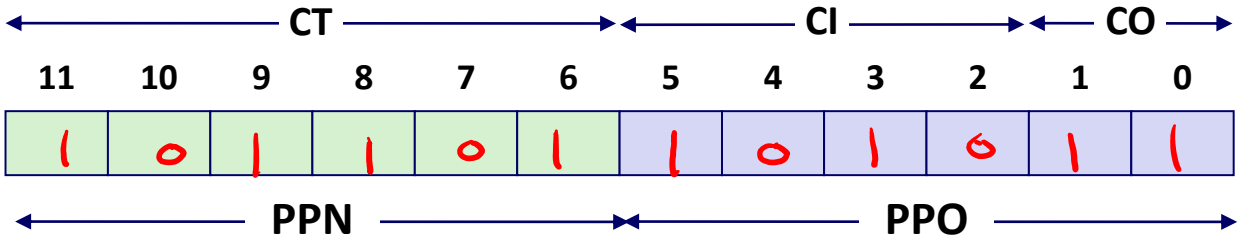
Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: 0x036B



VPN 0x0D TLBT 0x03 TLBI 1 TLB Hit? Y Page Fault? N PPN 0x2D

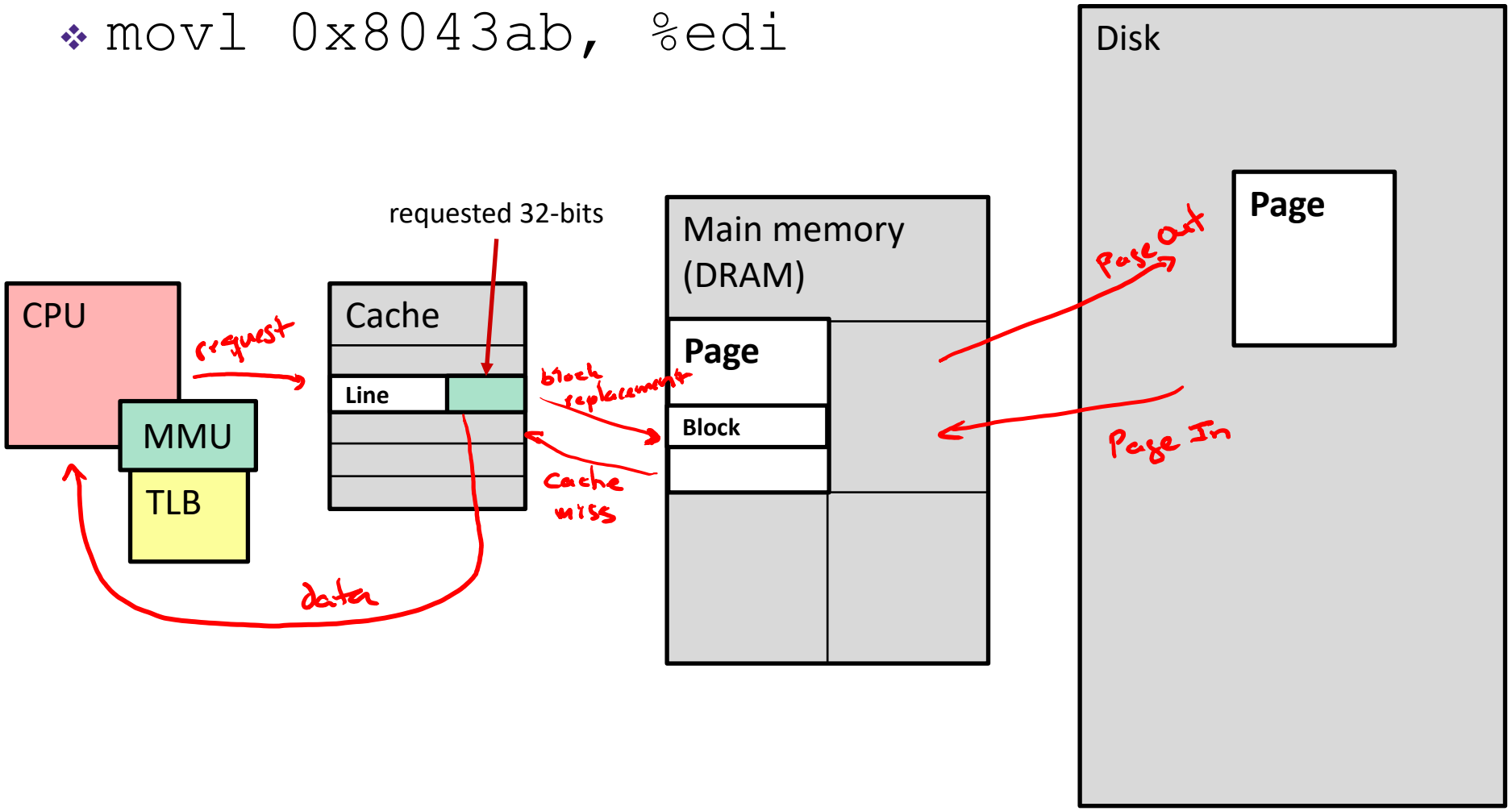
- Physical Address:



CT 0x2D CI 0xA CO 3 Cache Hit? Y Data (byte) 0x3B

Memory Overview

- `movl 0x8043ab, %edi`



Page Table Reality

This is extra (non-testable) material

- Just one issue… the numbers don't work out for the story so far!

- The problem is the page table for each process:
  - Suppose 64-bit VAs, 8 KiB pages, 8 GiB physical memory
  - How many page table entries is that?

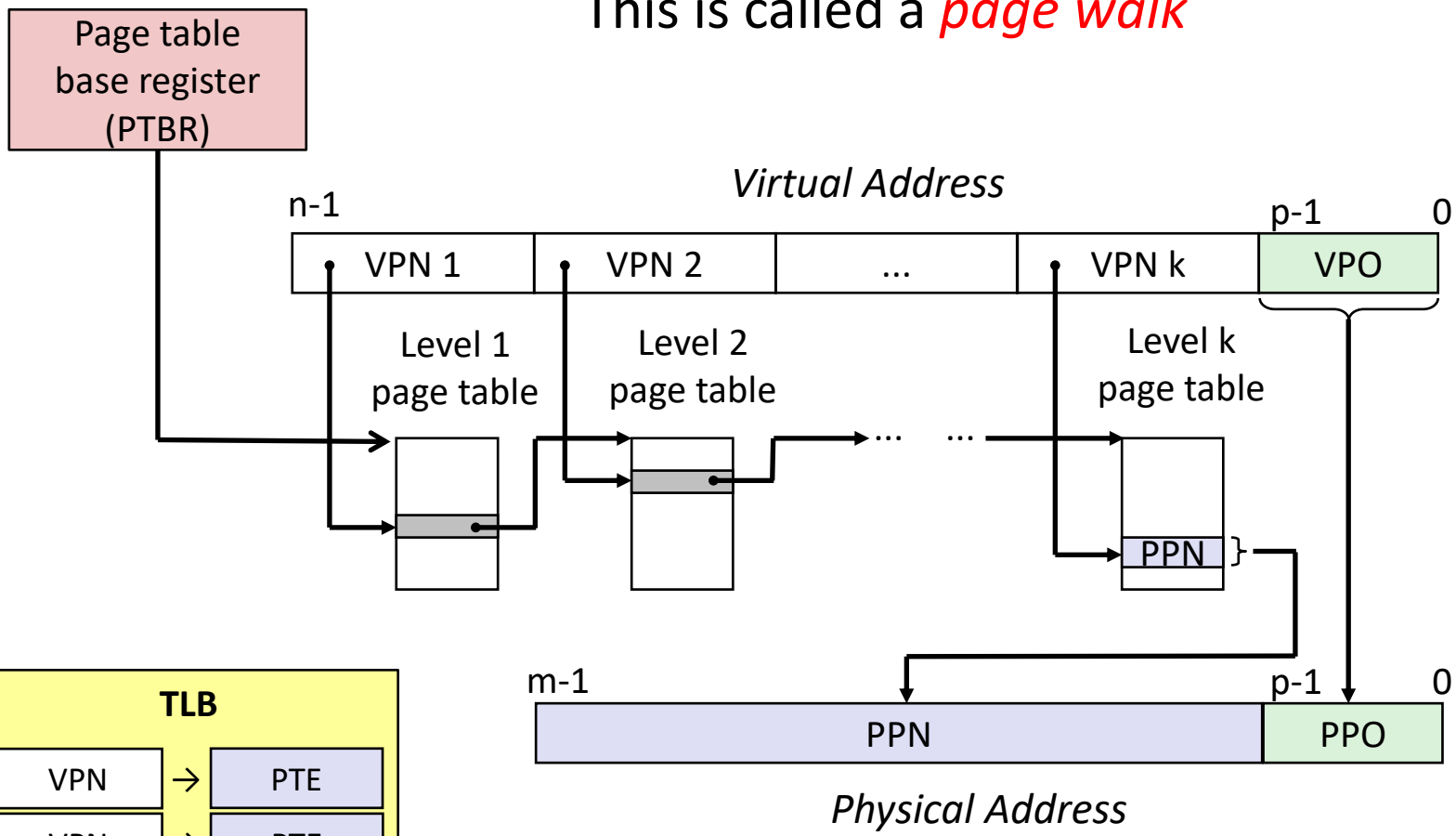
  - About how long is each PTE?

Moral: Cannot use this naïve implementation of the virtual→physical page mapping – it's *way* too big

A Solution: Multi-level Page Tables

This is extra (non-testable) material

This is called a *page walk*



Multi-level Page Tables

This is extra (non-testable) material

- A tree of depth k where each node at depth i has up to 2^j children if part i of the VPN has j bits
- Hardware for multi-level page tables inherently more complicated
  - But it's a necessary complexity – 1-level does not fit
- Why it works: Most subtrees are not used at all, so they are never created and definitely aren't in physical memory
  - Parts created can be evicted from cache/memory when not being used
  - Each node can have a size of ~1-100KB
- But now for a k-level page table, a TLB miss requires $k + 1$ cache/memory accesses
  - Fine so long as TLB misses are rare – motivates larger TLBs

Practice VM Question

- Our system has the following properties
  - 1 MiB of physical address space m = 20 bits
  - 4 GiB of virtual address space n = 32 bits
  - 32 KiB page size p = 15 bits
  - 4-entry fully associative TLB with LRU replacement

a) Fill in the following blanks:

_____ Entries in a page table

2^{17}

$2^{n-p} = 2^{32-15}$

_____ Minimum bit-width of PTBR (PA width)

20

_____ TLBT bits

17

VPN bits = n-p = 17

TLB is fully assoc → Ø TLBI bits

_____ Max # of valid entries in a page table

2^5

(# of physical pages) = $2^{m-p} = 2^{20-15}$

Practice VM Question

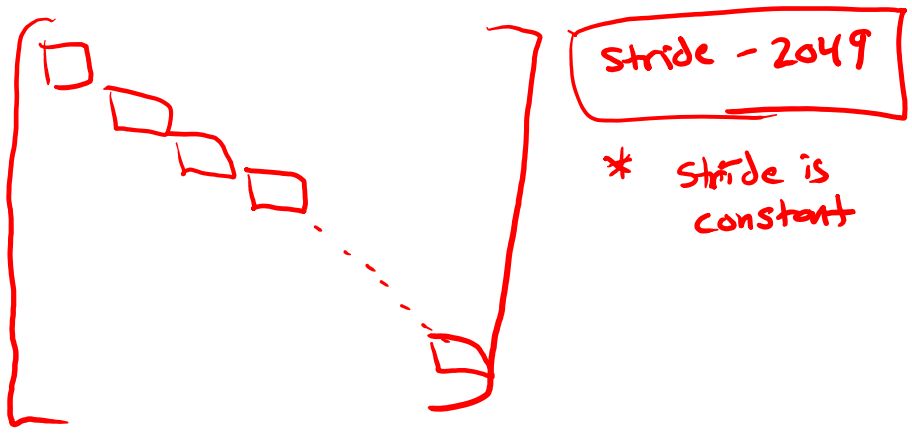
- One process uses a page-aligned *square* matrix `mat[]` of 32-bit integers in the code shown below:

```
#define MAT_SIZE = 2048
for(int i = 0; i < MAT_SIZE; i++)
  mat[i*(MAT_SIZE+1)] = i;
```

2048 ints in mat[]

b) What is the largest stride (in bytes) between successive memory accesses (in the VA space)?

i	idx of mat
0	0
1	2049
2	2 * 2049



Practice VM Question

- One process uses a page-aligned *square* matrix `mat[]` of 32-bit integers in the code shown below:

```
#define MAT_SIZE = 2048
for(int i = 0; i < MAT_SIZE; i++)
  mat[i*(MAT_SIZE+1)] = i;
```

c) Assuming all of `mat[]` starts on disk, what are the following hit rates for the execution of the for-loop?

3/4 = 75% TLB Hit Rate

Page Size = 32 KiB = 2^{15} B

MAT_SIZE = 2^{11} ints = 2^{13} B

→ each page holds 4 rows of mat[]

so, for each page: M-H-H-H in TLB

0% Page Table Hit Rate

- PT only accessed on TLB Miss
- since mat[] is on disk, first access to each page results in Page Fault
  - i.e., never hit in PT
  - when page is loaded, translation loaded to TLB, and TLB hit doesn't access PT

Virtual Memory Summary

- Programmer's view of virtual memory
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes

- System view of virtual memory
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and sharing
  - Simplifies protection by providing permissions checking

Memory System Summary

- Memory Caches (L1/L2/L3)
  - Purely a speed-up technique
  - Behavior invisible to application programmer and (mostly) OS
  - Implemented totally in hardware
- Virtual Memory
  - Supports many OS-related functions
    - Process creation, task switching, protection
  - Operating System (software)
    - Allocates/shares physical memory among processes
    - Maintains high-level tables tracking memory type, source, sharing
    - Handles exceptions, fills in hardware-defined mapping tables
  - Hardware
    - Translates virtual addresses via mapping tables, enforcing permissions
    - Accelerates mapping via translation cache (TLB)

Quick Review

- What do Page Tables map?

  VPN to PPN or disk address
- Where are Page Tables located?

  physical memory
- How many Page Tables are there?

  One per process
- Can your program tell if a page fault has occurred?

  No, but it has to wait a long time
- What is thrashing?

  constantly paging in and out
- True / False: Virtual Addresses that are contiguous will always be contiguous in physical memory

  - an array that crosses a page boundary
  - Virtual Pages not necessarily mapped to contiguous physical pages!
- TLB stands for Translation Lookaside Buffer and stores Page Table Entries