Virtual Memory II

CSE 351 Summer 2019

Instructor:

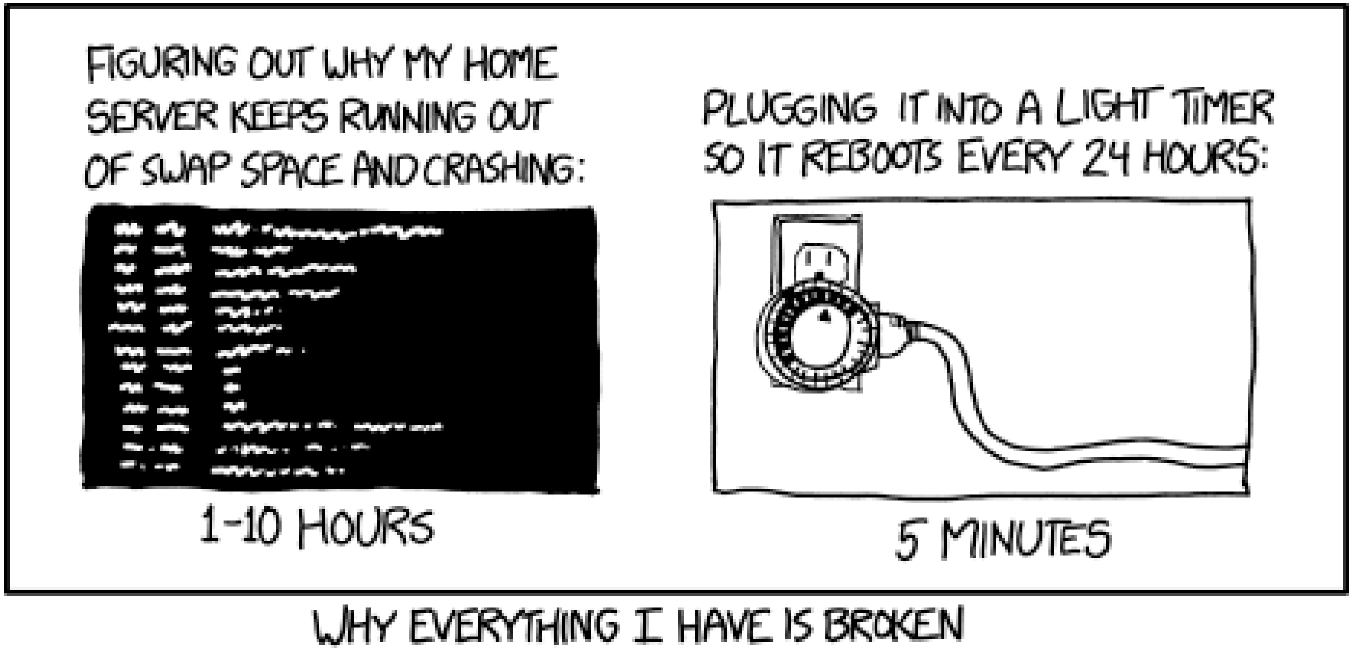
Sam Wolfson

Teaching Assistants:

Rehaan Bhimani

Corbin Modica

Daniel Hsu


FIGURING OUT WHY MY HOME SERVER KEEPS RUNNING OUT OF SWAP SPACE AND CRASHING:
1-10 HOURS
PLUGGING IT INTO A LIGHT TIMER SO IT REBOOTS EVERY 24 HOURS:
5 MINUTES
WHY EVERYTHING I HAVE IS BROKEN

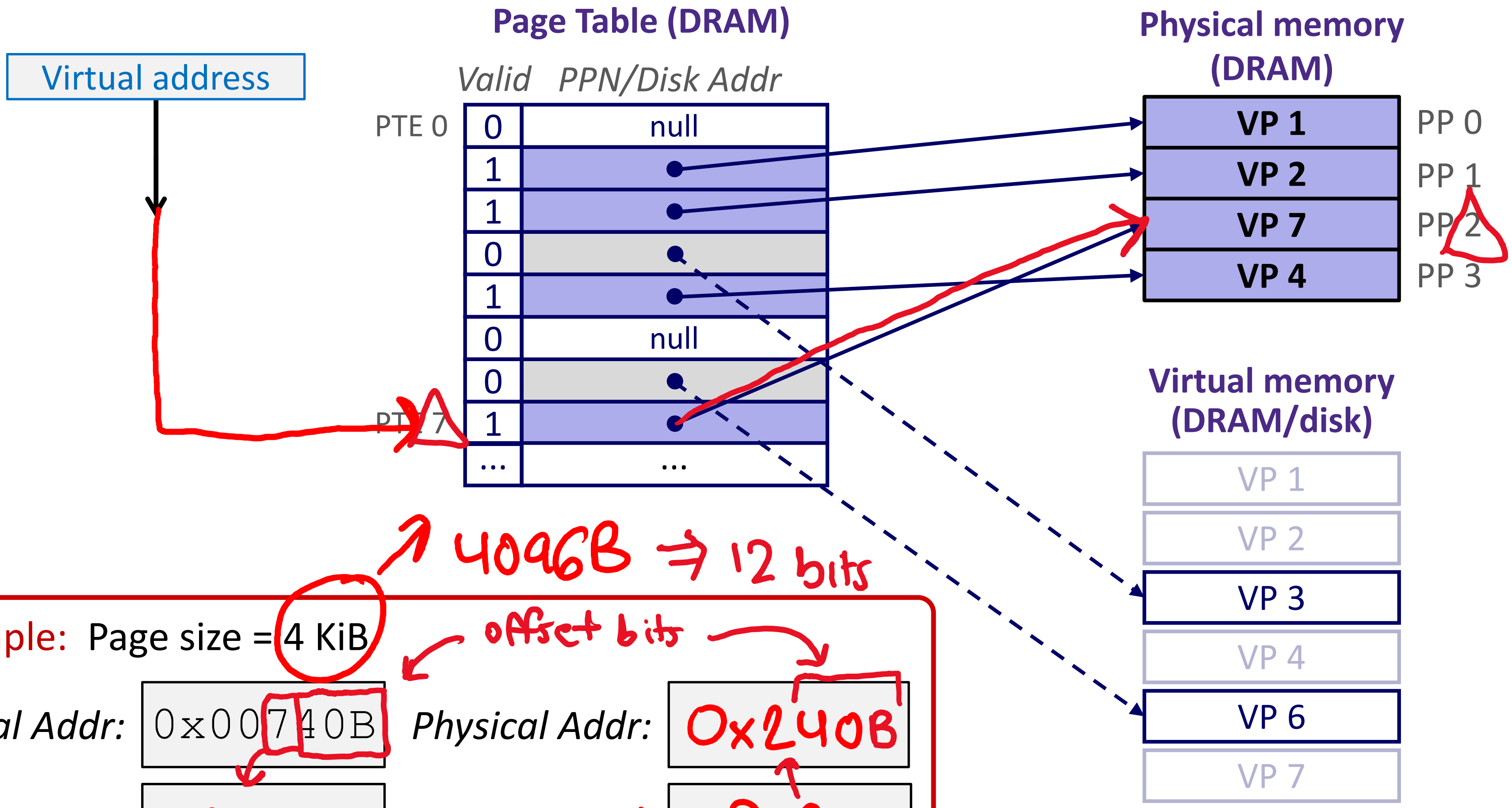
https://xkcd.com/1495/

Administrivia

- Lab 4, due Monday (8/12)
  - Do the Canvas quiz before starting on Part III (blocking)
- HW5 released
- Grades for lab 3 released

Page Hit

- ***Page hit:*** VM reference is in physical memory



4096B ⇒ 12 bits

offset bits

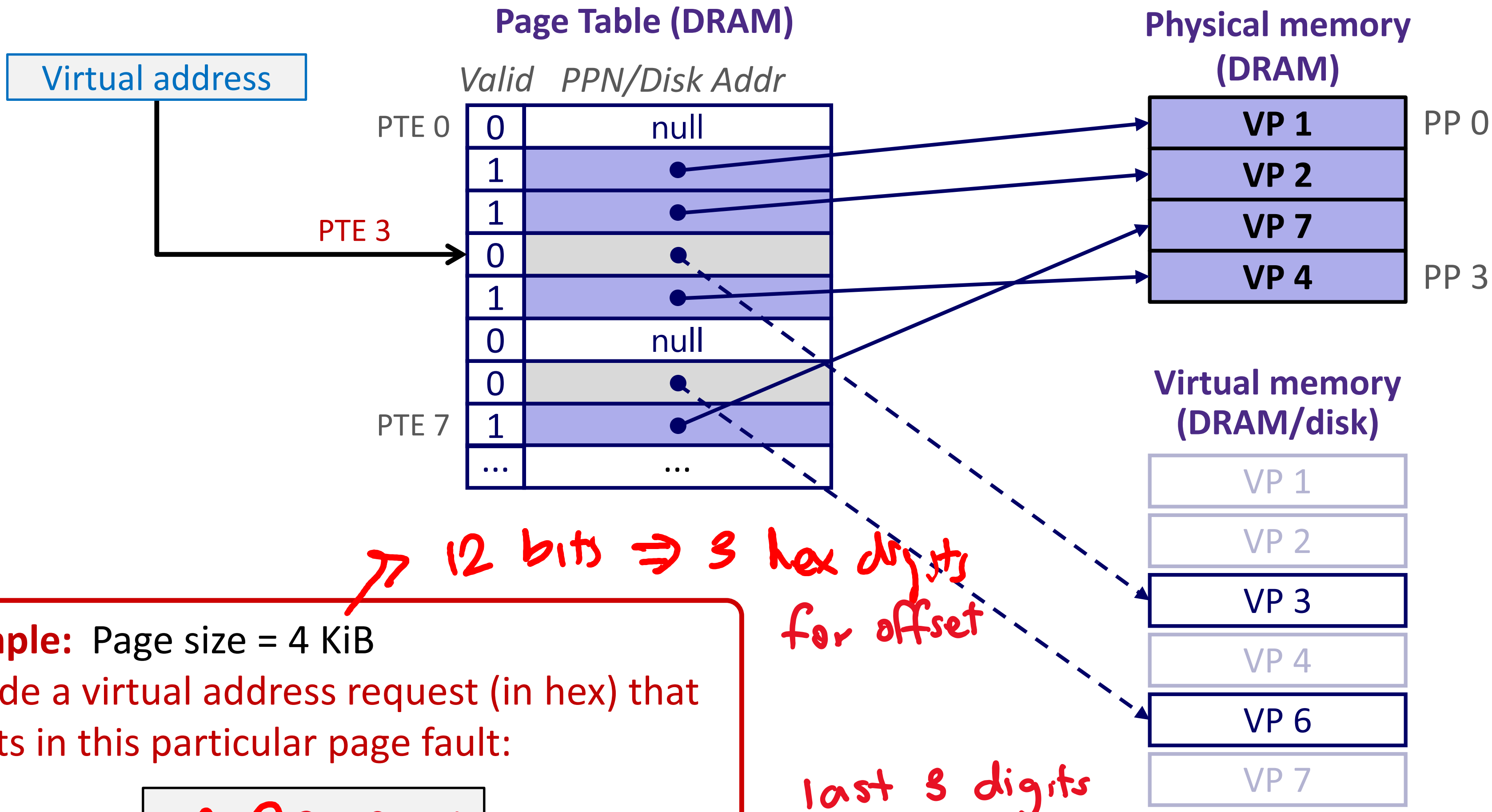
Example: Page size = 4 KiB

Virtual Addr: 0x00740B *Physical Addr:* 0x240B

VPN: 0x7 *PPN:* 0x2

Page Fault

- ***Page fault:*** VM reference is NOT in physical memory



12 bits ⇒ 3 hex digits for offset

Example: Page size = 4 KiB

Provide a virtual address request (in hex) that results in this particular page fault:

Virtual Addr: 0x3000

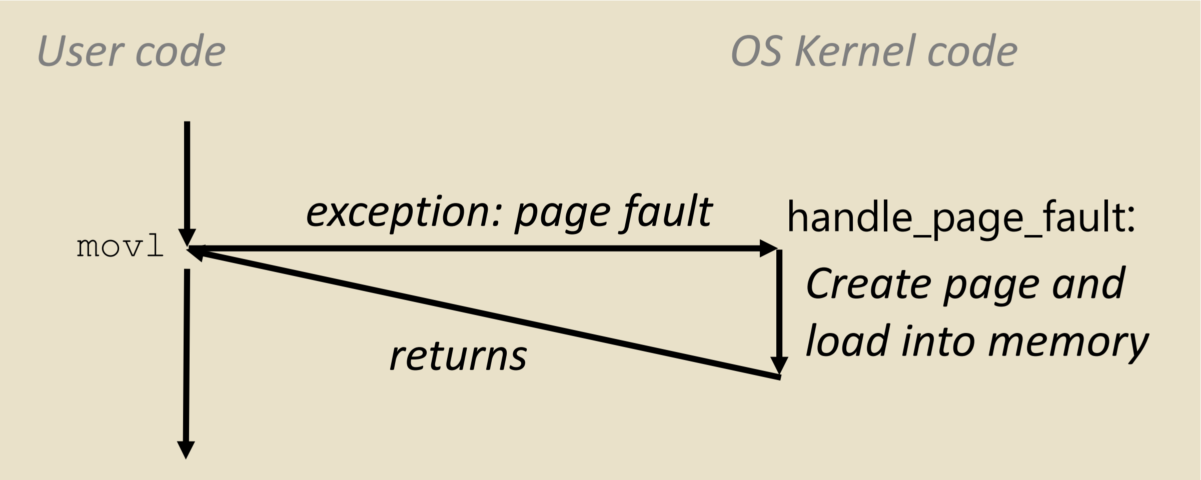
last 3 digits can be anything

Page Fault Exception

```
int a[1000];
int main ()
{
    a[500] = 13;
}
```

- User writes to memory location
- That portion (page) of user's memory is currently on disk

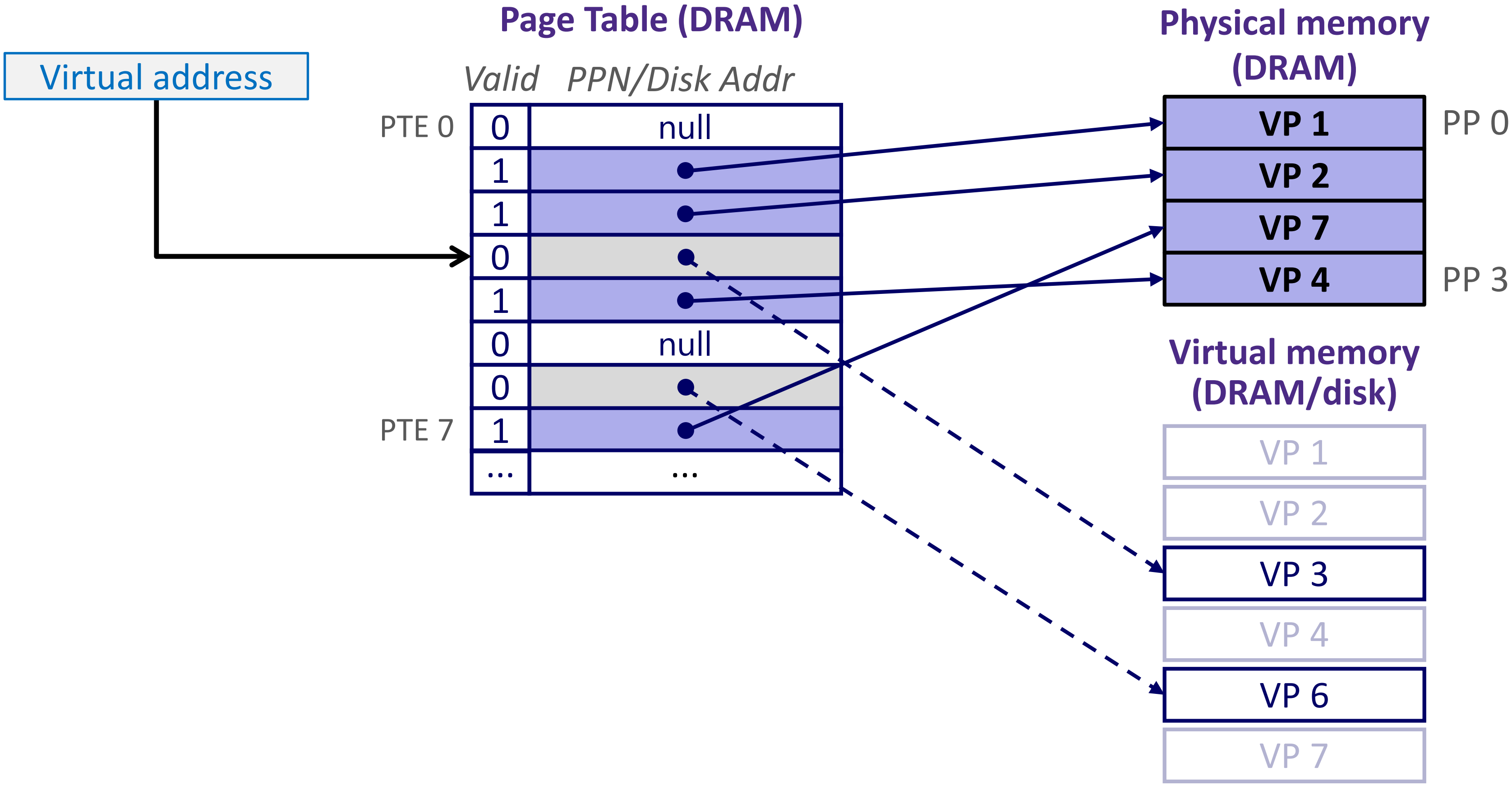
```
80483b7:      c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



- Page fault handler must load page into physical memory
- Returns to faulting instruction: `mov` is executed again!
  - Successful on second try

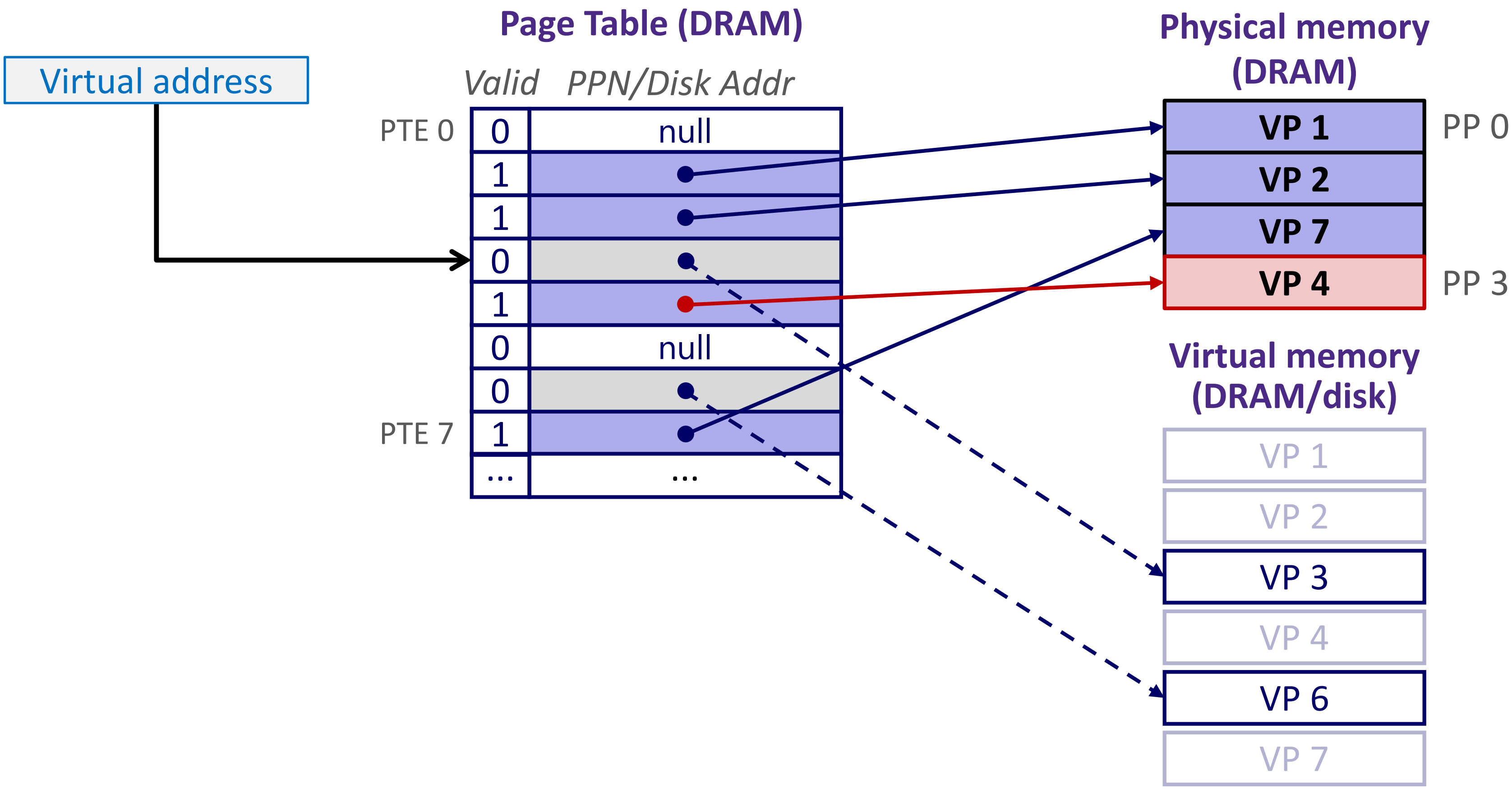
Handling a Page Fault

- Page miss causes page fault (an exception)



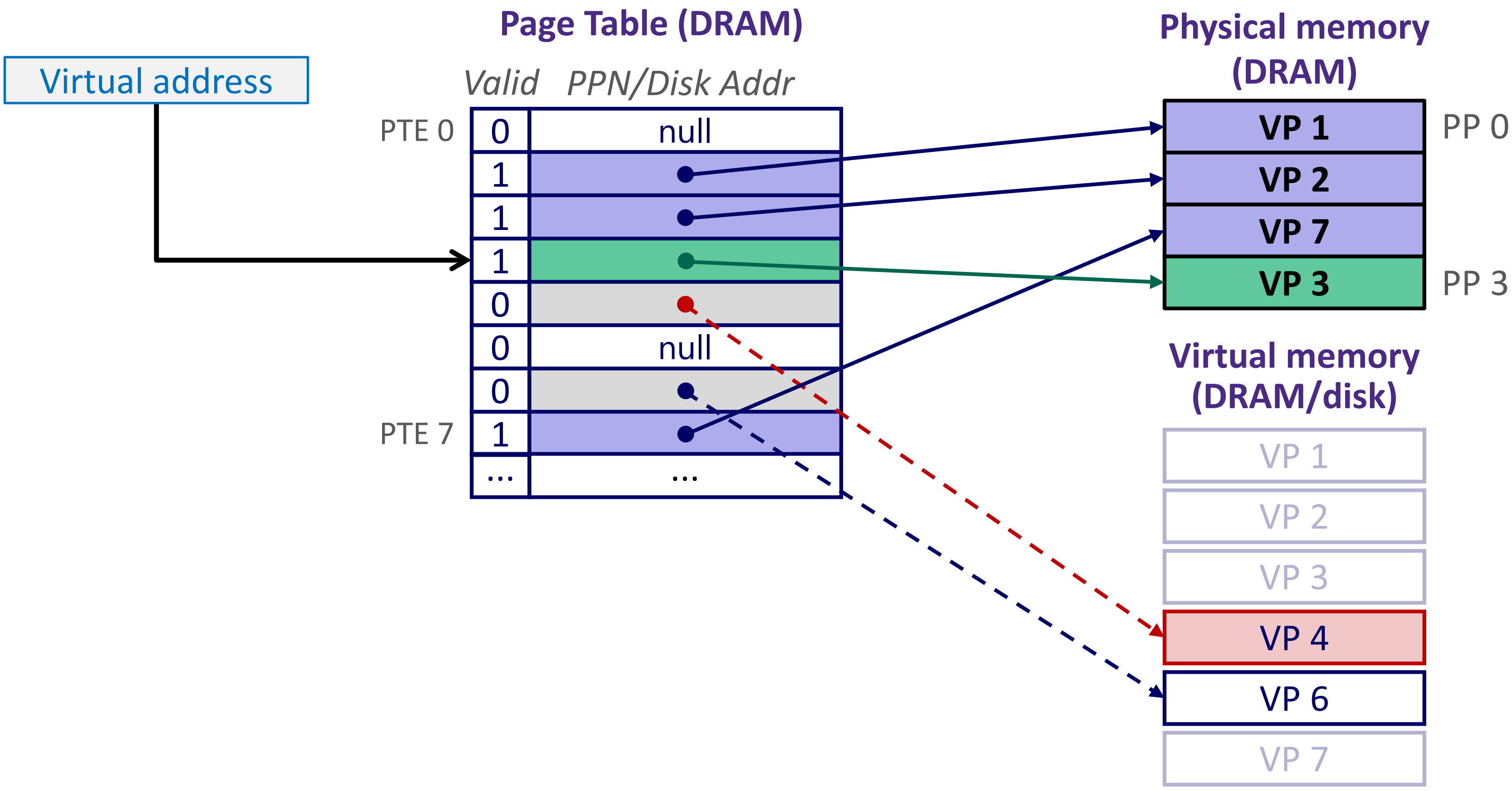
Handling a Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a *victim* to be evicted (here VP 4)



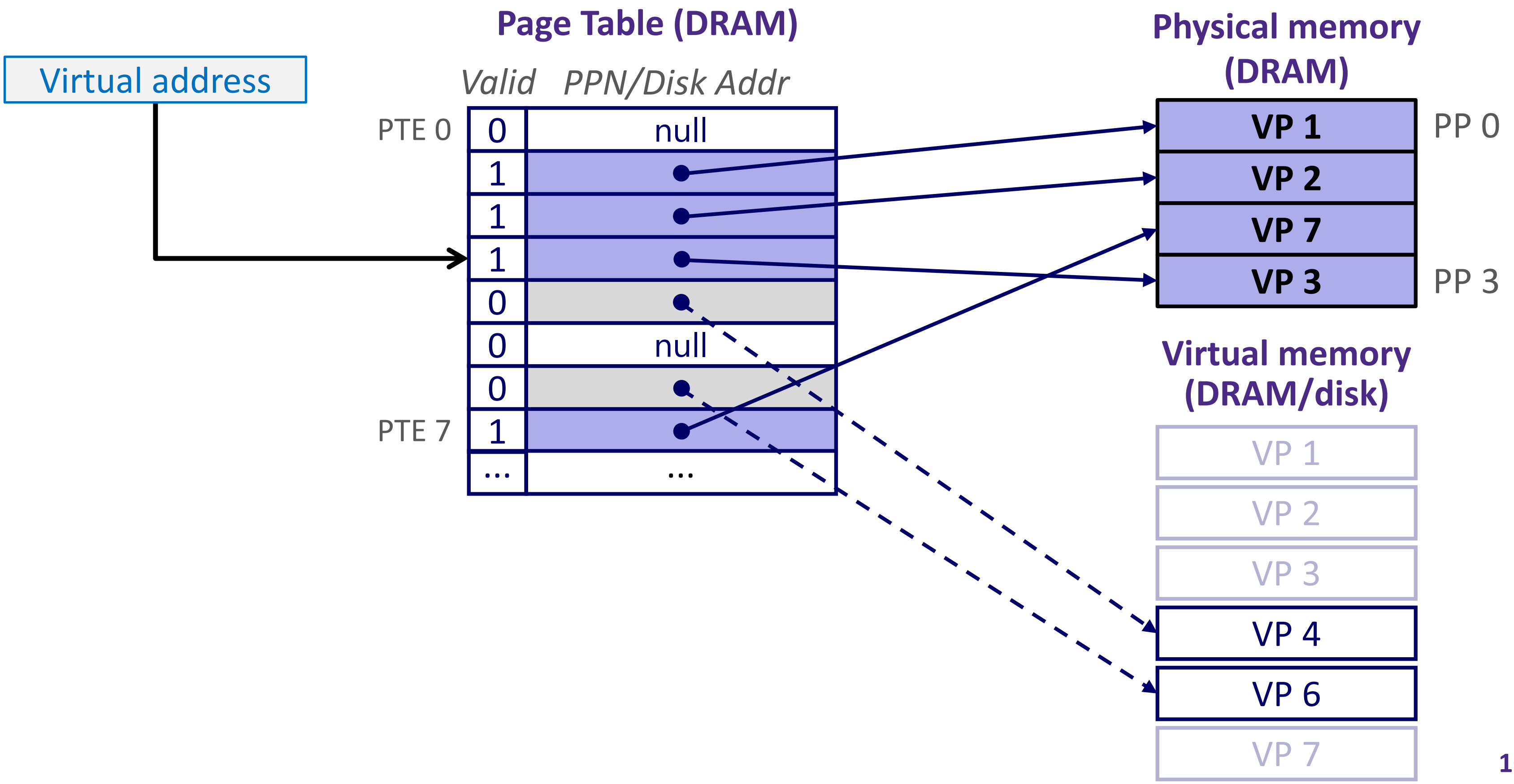
Handling a Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a *victim* to be evicted (here VP 4)



Handling a Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a *victim* to be evicted (here VP 4)
- Offending instruction is restarted: page hit!



Peer Instruction Question

- How many bits wide are the following fields?
  - 16 KiB pages $= 2^{10} \cdot 2^{4} = 2^{14} \Rightarrow$ offset = 14 bits
  - 48-bit virtual addresses
  - 16 GiB physical memory $= 2^{32} \cdot 2^{2} = 2^{34} \Rightarrow$ PA width = 34 bits
  - Vote at: http://pollev.com/wolfson

48 − 14 = 34

34 − 14 = 20

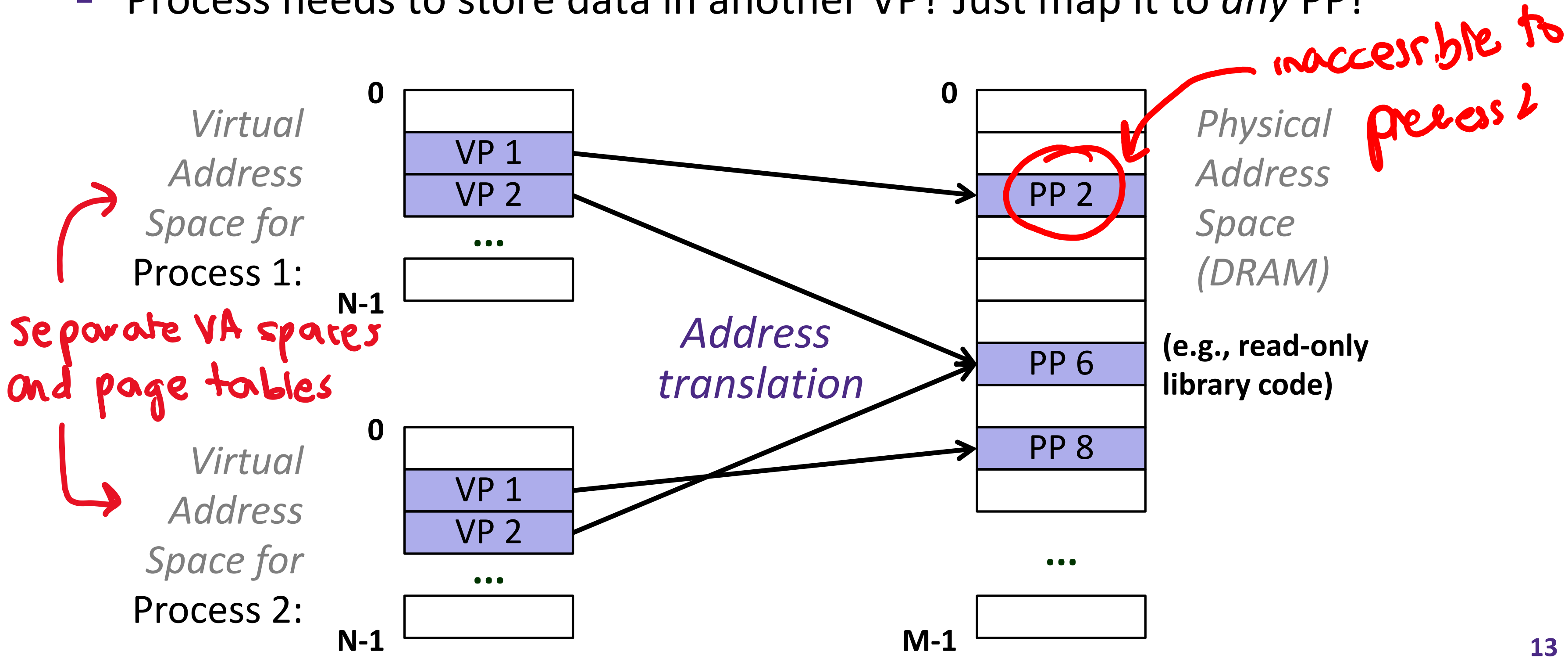
	VPN	PPN
(A)	34	24
(B)	32	18
(C)	30	20
[✓ (D)]	34	20

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**

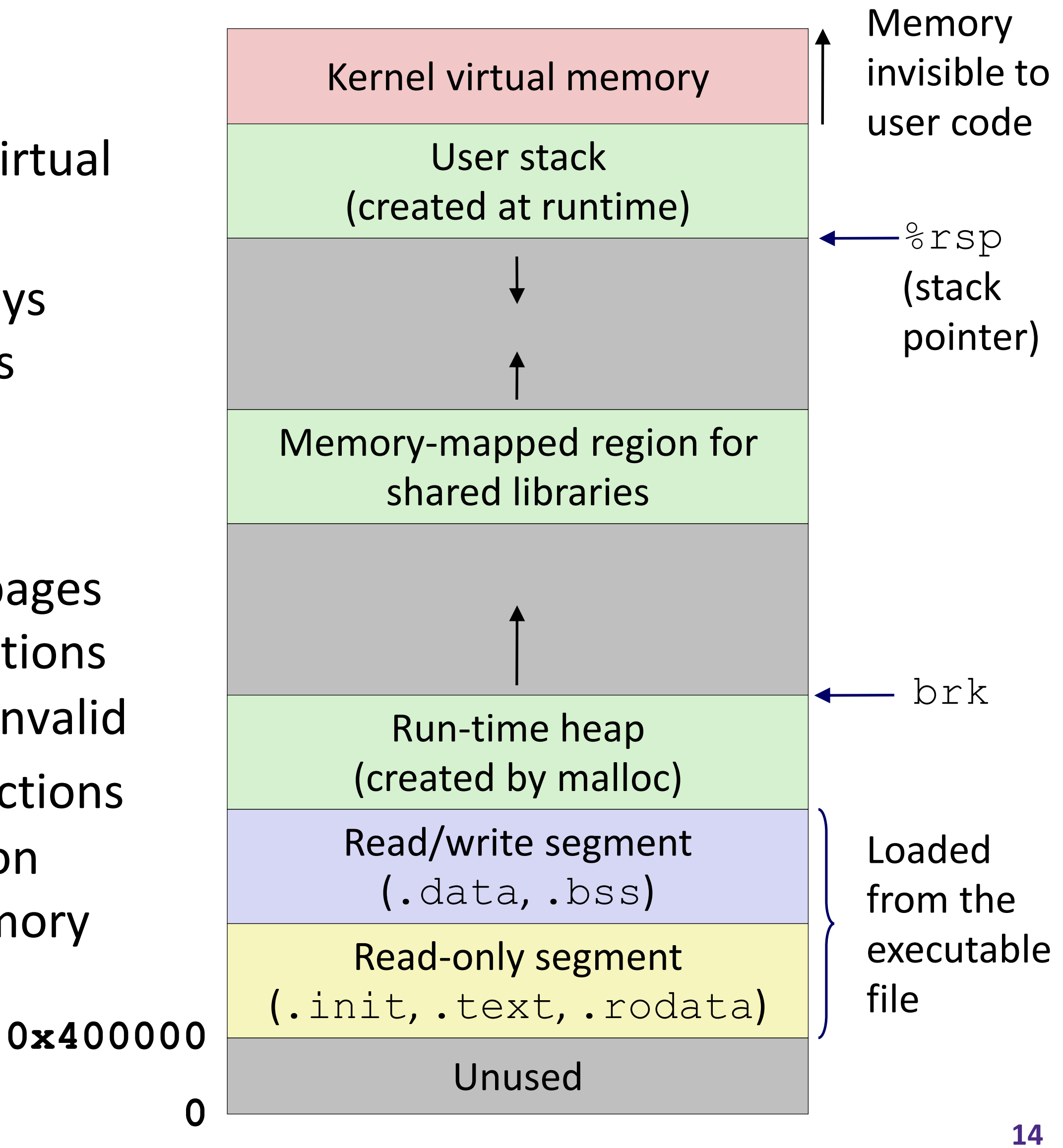
VM for Managing Multiple Processes

- 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!


inaccessible to process 2
Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
separate VA spaces and page tables
Address translation
0
PP 2
PP 6
PP 8
...
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

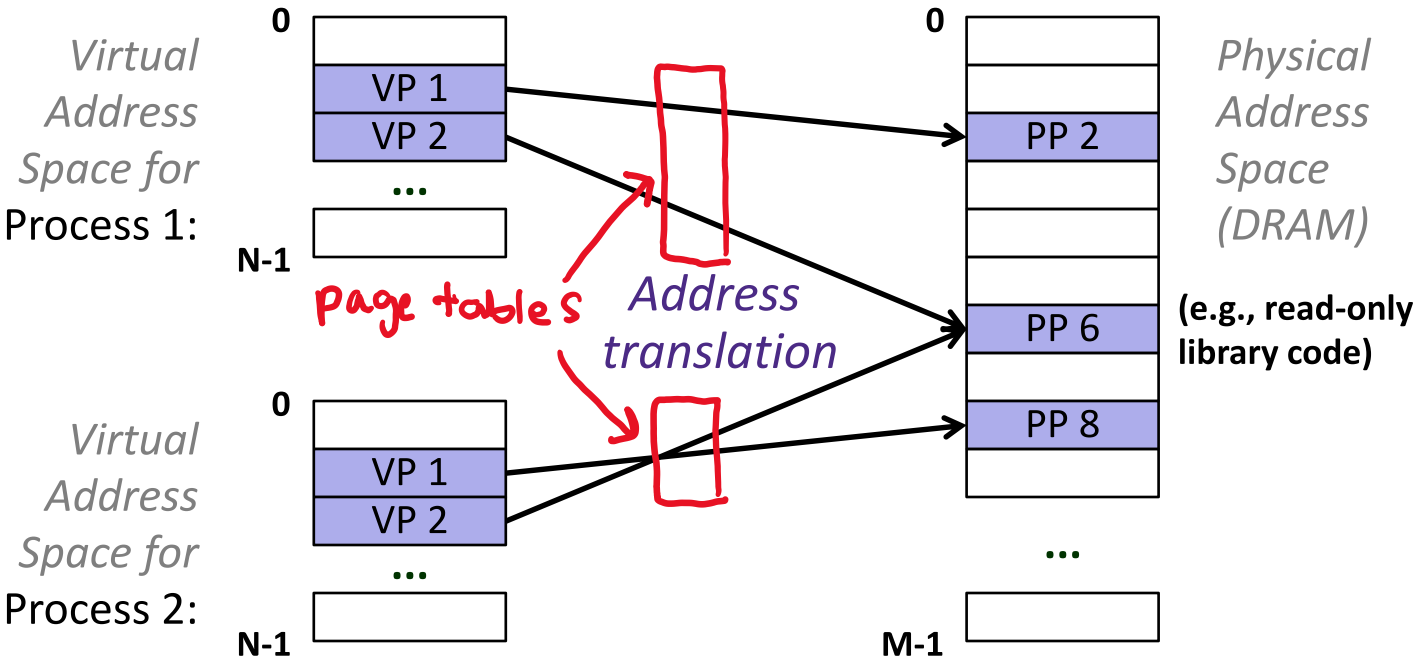
Simplifying Linking and Loading

- 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

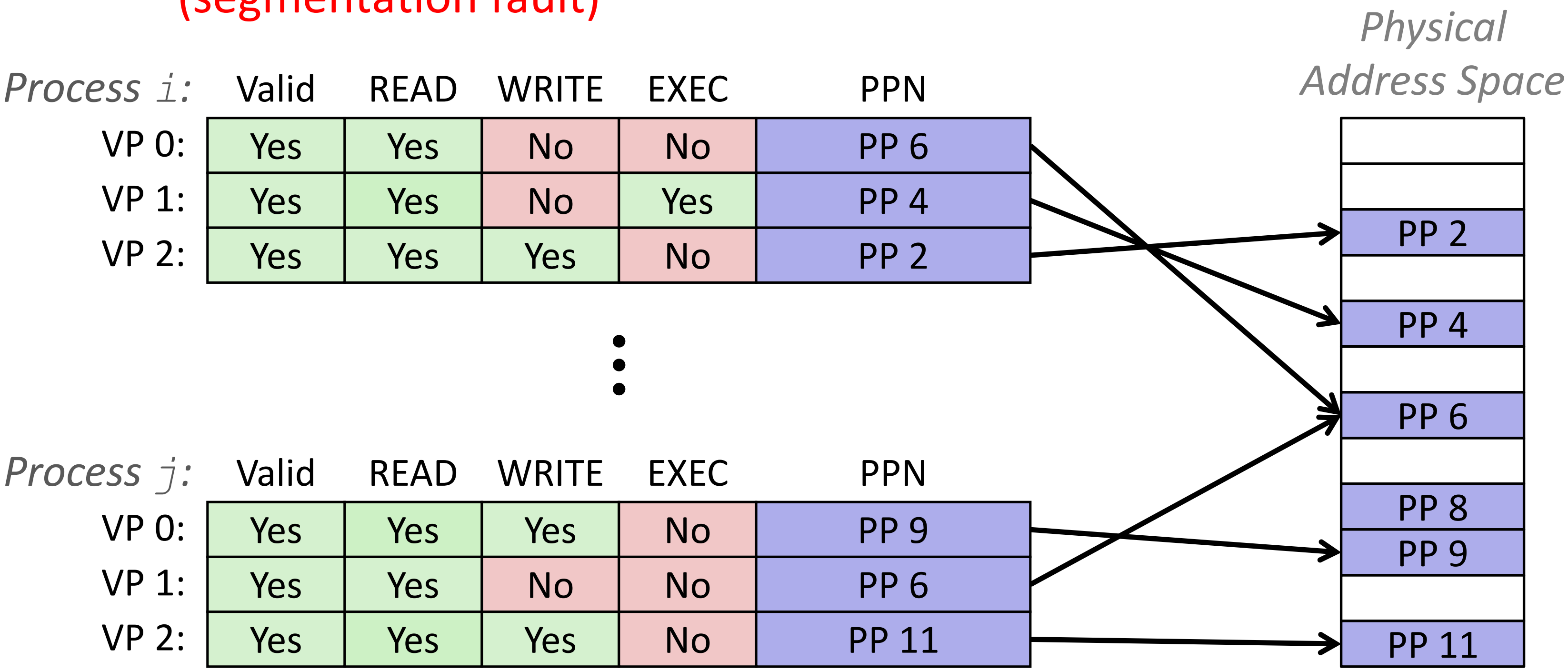
- The mapping of VPs to PPs provides a simple mechanism to *protect* memory and to *share* memory between processes
  - **Sharing:** map virtual pages in separate address spaces to the same physical page (here: PP 6)
  - **Protection:** process can't access physical pages to which none of its virtual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



Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1
page tables
Address translation
0
PP 2
PP 6
PP 8
...
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)

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)



Process `i`:

	Valid	READ	WRITE	EXEC	PPN
VP 0:	Yes	Yes	No	No	PP 6
VP 1:	Yes	Yes	No	Yes	PP 4
VP 2:	Yes	Yes	Yes	No	PP 2

⋮

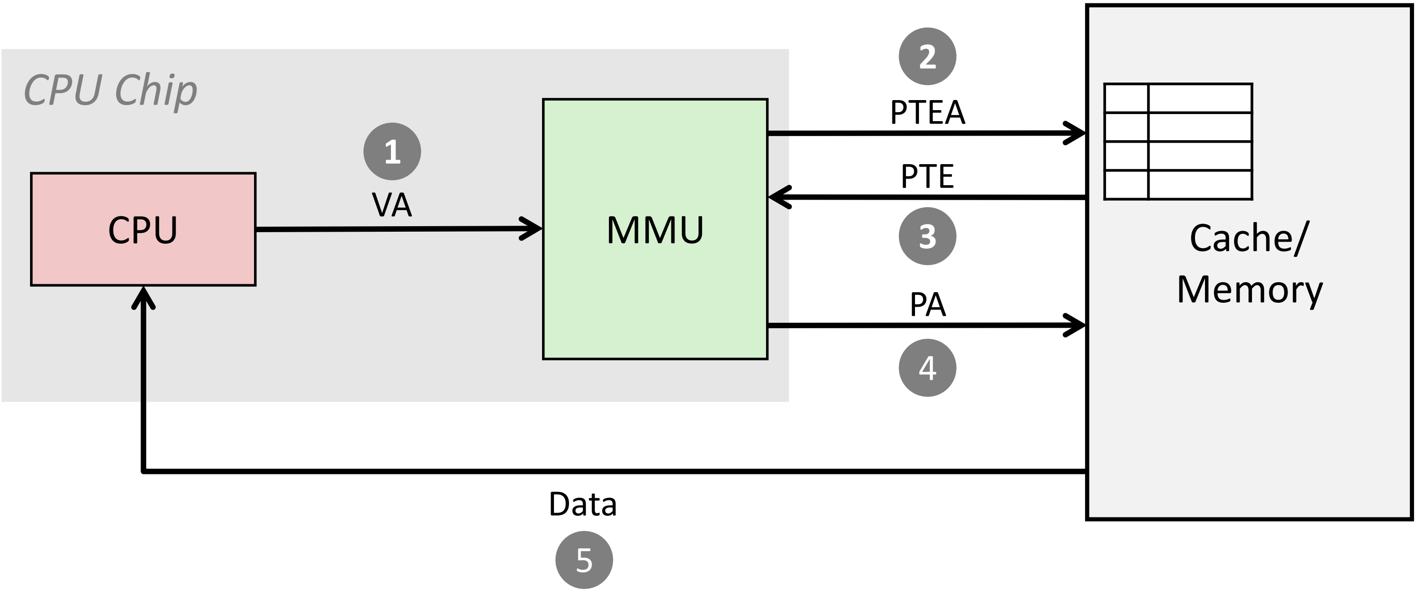
Process `j`:

	Valid	READ	WRITE	EXEC	PPN
VP 0:	Yes	Yes	Yes	No	PP 9
VP 1:	Yes	Yes	No	No	PP 6
VP 2:	Yes	Yes	Yes	No	PP 11

Address Translation: Page Hit

movq (%rdi), %rsi

virtual address



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

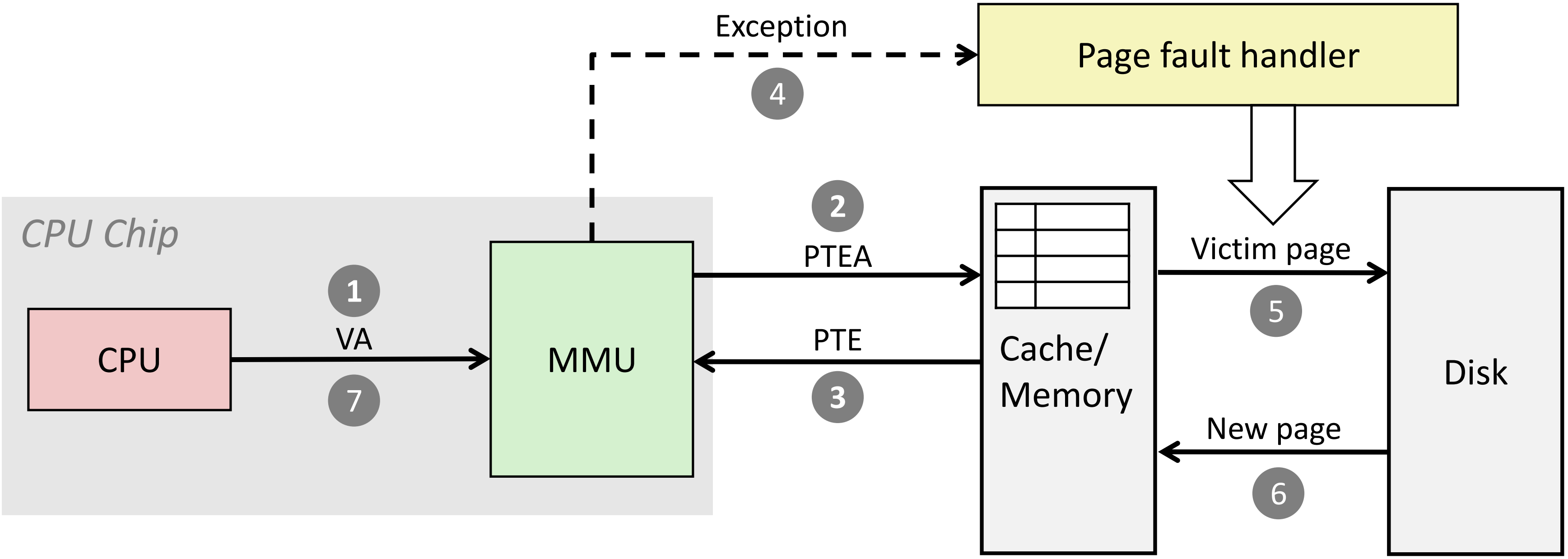
2-3) MMU fetches PTE from page table in cache/memory
(Uses PTBR to find beginning of page table for current process)

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

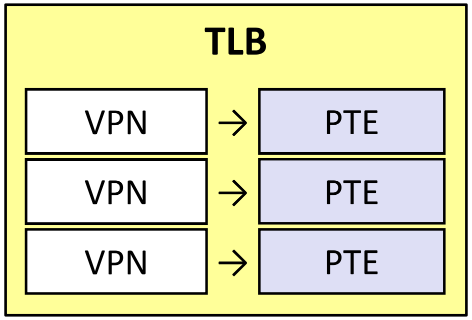
Hmm… Translation Sounds Slow

- 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

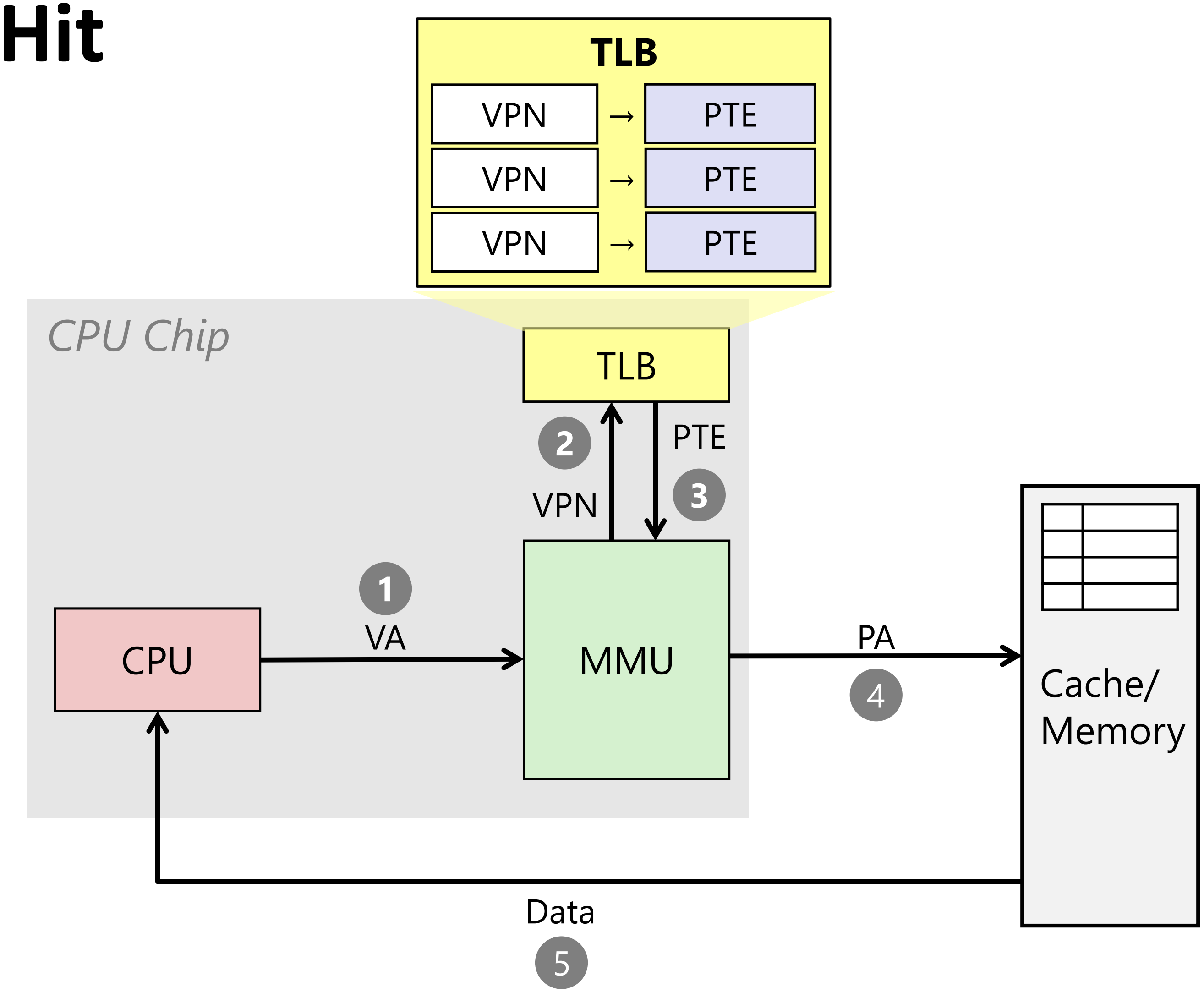
- *What can we do to make this faster?*
  - "Any problem in computer science can be solved by adding another level of **indirection**." *– David Wheeler, inventor of the subroutine*
  - "And all of the new problems *that* creates can be solved by adding another **cache**." *- Sam Wolfson, inventor of this quote*

Speeding up Translation with a TLB

- *Translation Lookaside Buffer* (TLB):
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete *page table entries* for small number of pages
    - Modern Intel processors have 128 or 256 entries in TLB
  - Much faster than a page table lookup in cache/memory

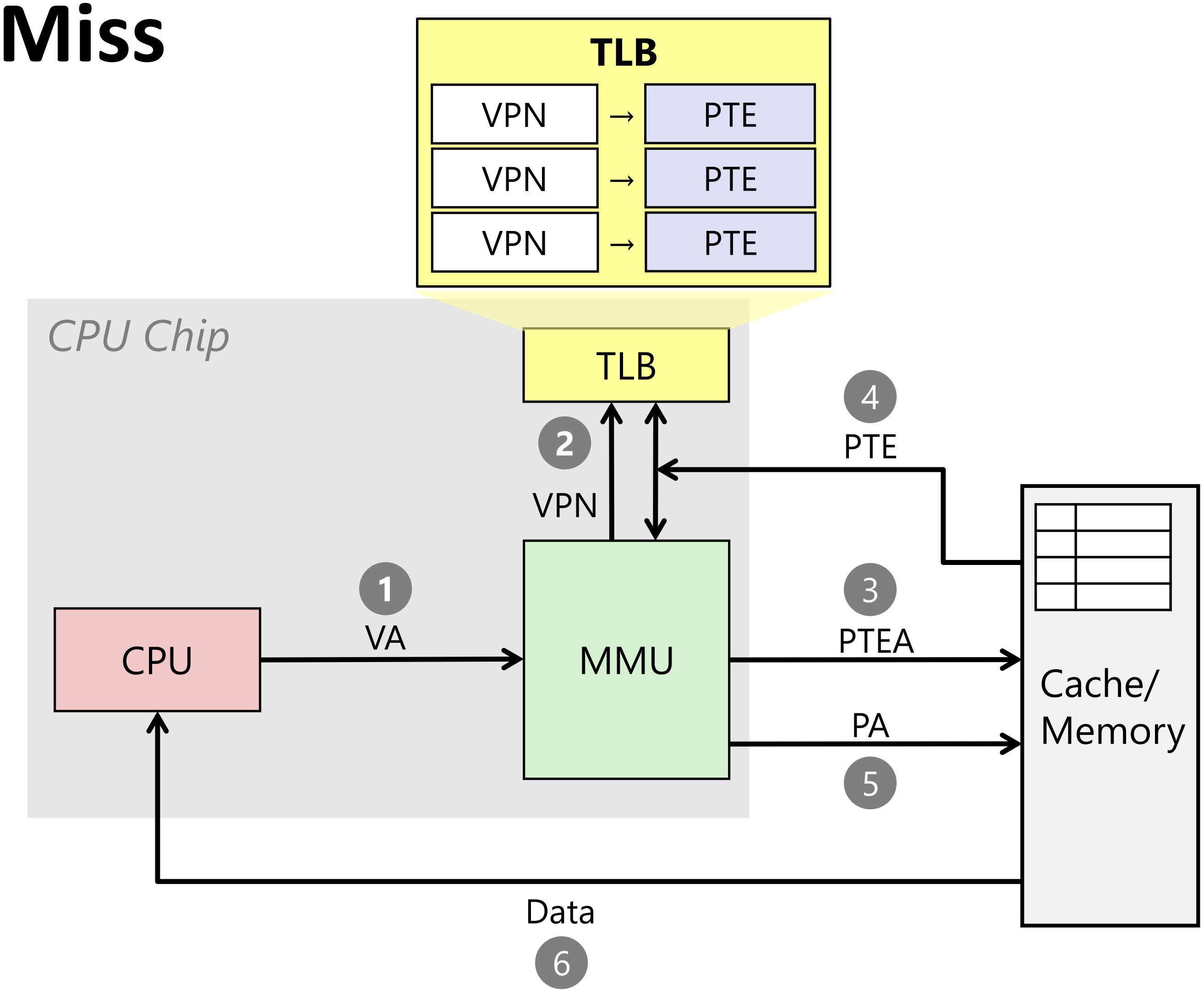


TLB Hit



- 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

movq (%rsi), %rax

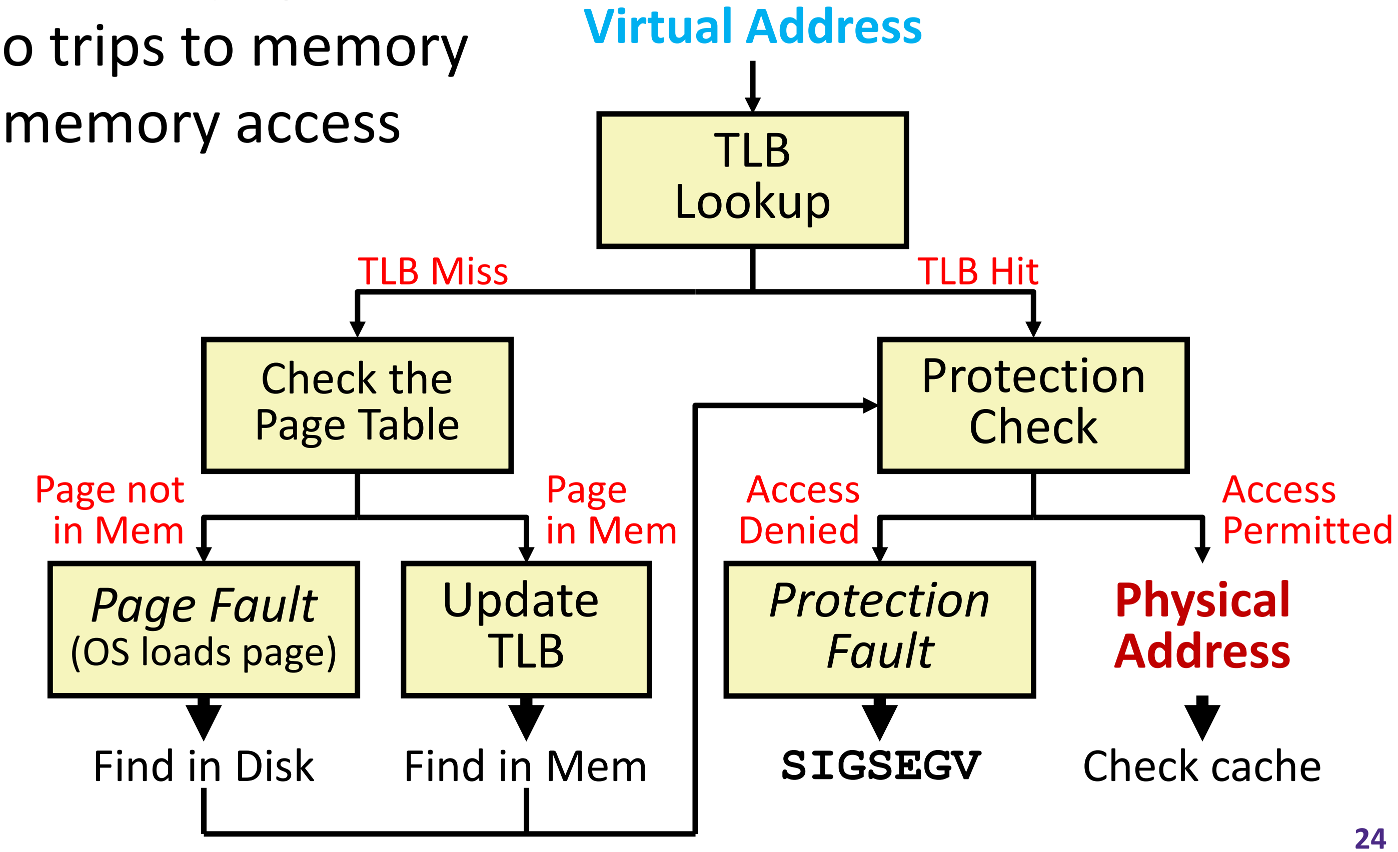
1) Check TLB
   - Input: VPN, Output: PPN
   - *TLB Hit:* Fetch translation, return PPN
   - *TLB Miss:* Check page table (in memory)
     - *Page Table Hit:* Load page table entry into TLB
     - *Page Fault:* Fetch page from disk to memory, update corresponding page table entry, then load entry into TLB

virtual address
↓
physical address

2) Check cache
   - 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

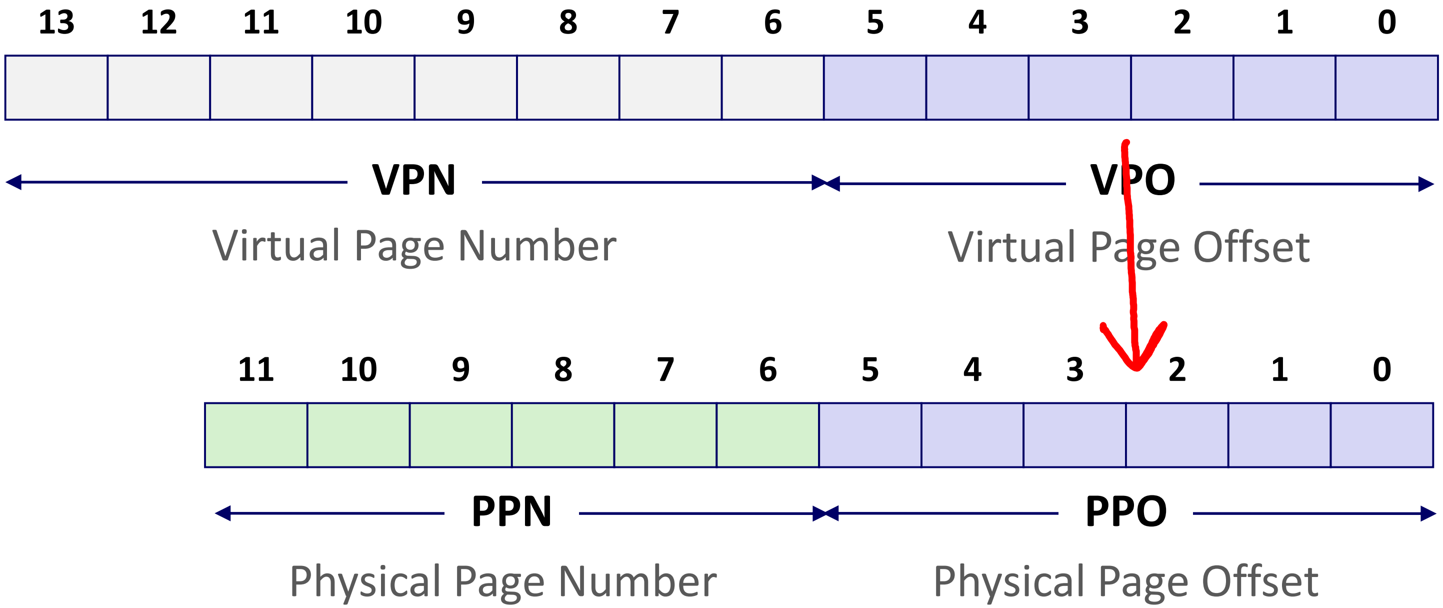
Address Translation

- VM is complicated, but also elegant and effective
  - Level of indirection to provide isolated memory & caching
  - TLB as a cache of page tables avoids two trips to memory for every memory access



Simple Memory System Example (small)

- Addressing
  - 14-bit virtual addresses $n=14$
  - 12-bit physical address $m=12$
  - Page size = 64 bytes $\log_2 64 = 6 \Rightarrow 6$ offset bits



Simple Memory System: Page Table

- Only showing first 16 entries (out of 2^{n-p} ______) $= 2^{14-6} = 2^{8}$
  - **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

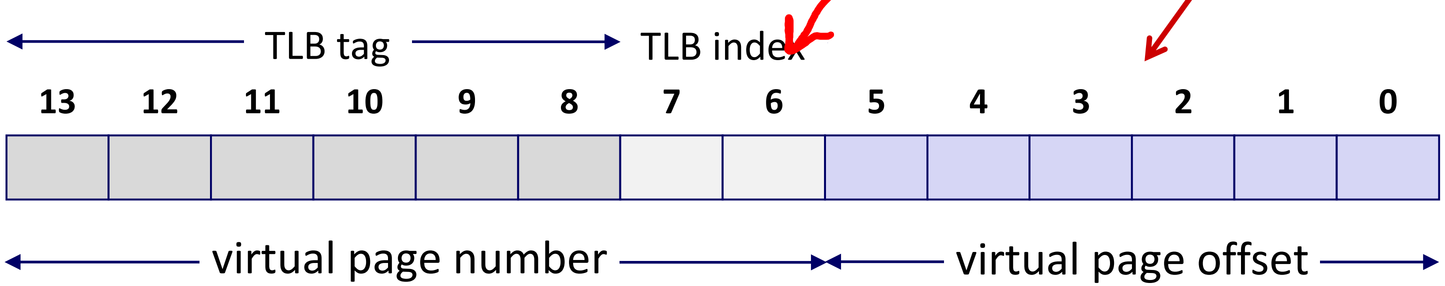
VPN	PPN	Valid
8	13	1
9	17	1
A	09	1
B	–	0
C	–	0
D	2D	1
E	–	0
F	0D	1

Simple Memory System: TLB

- 16 entries total
- 4-way set associative

16/4 = 4 sets

Why does the TLB ignore the page offset?

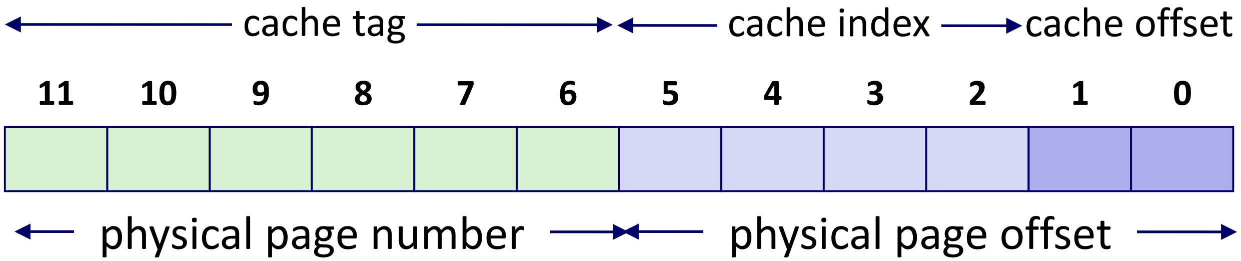


Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Simple Memory System: Cache

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

- Direct-mapped with K = 4 B, C/K = 16
- Addressed using *physical addresses*



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	–	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

Page table (partial):

VPN	PPN	V	VPN	PPN	V
0	28	1	8	13	1
1	–	0	9	17	1
2	33	1	A	09	1
3	02	1	B	–	0
4	–	0	C	–	0
5	16	1	D	2D	1
6	–	0	E	–	0
7	–	0	F	0D	1

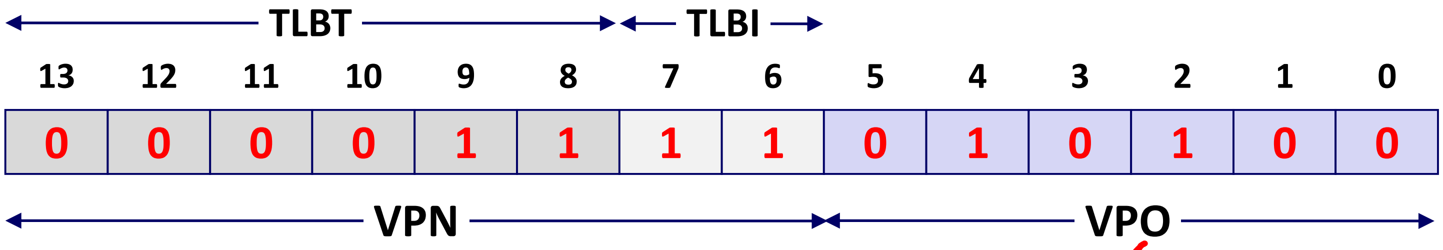
Cache:

Index	Tag	V	B0	B1	B2	B3	Index	Tag	V	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

Memory Request Example #1

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

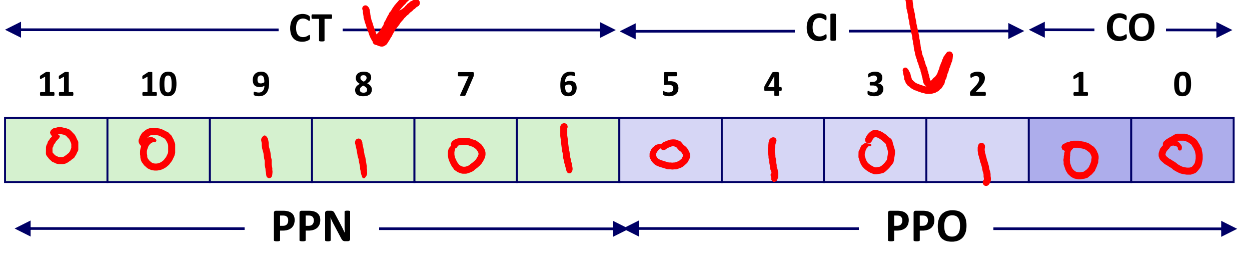
- Virtual Address: `0x03D4`



TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0
VPN								VPO					

VPN 0xF TLBT 0x3 TLBI 0x3 TLB Hit? ✓ Page Fault? ✗ PPN 0xD

- Physical Address:



CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0
PPN						PPO					

CT 0xD CI 0x5 CO 0x0 Cache Hit? ✓ Data (byte) 0x36

Memory Request Example #2

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

- Virtual Address: `0x038F`

TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	0	1	1	1	1
VPN								VPO					

VPN 0xE TLBT 0x3 TLBI 0x2 TLB Hit? X Page Fault? ✓ PPN ¯_(ツ)_/¯

- Physical Address: unknown since no PA until OS swaps

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
PPN						PPO					

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

- Virtual Address: `0x0020`

TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0
VPN								VPO					

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

- Physical Address:

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
PPN						PPO					

CT 0x28 CI 0x8 CO 0x2 Cache Hit? X Data (byte) ¯_(ツ)_/¯

Memory Request Example #4

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

- Virtual Address: `0x036B`

TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1	0	1	0	1	1
VPN								VPO					

VPN 0xD TLBT 0x3 TLBI 0x1 TLB Hit? ✓ Page Fault? ✗ PPN 0x2D

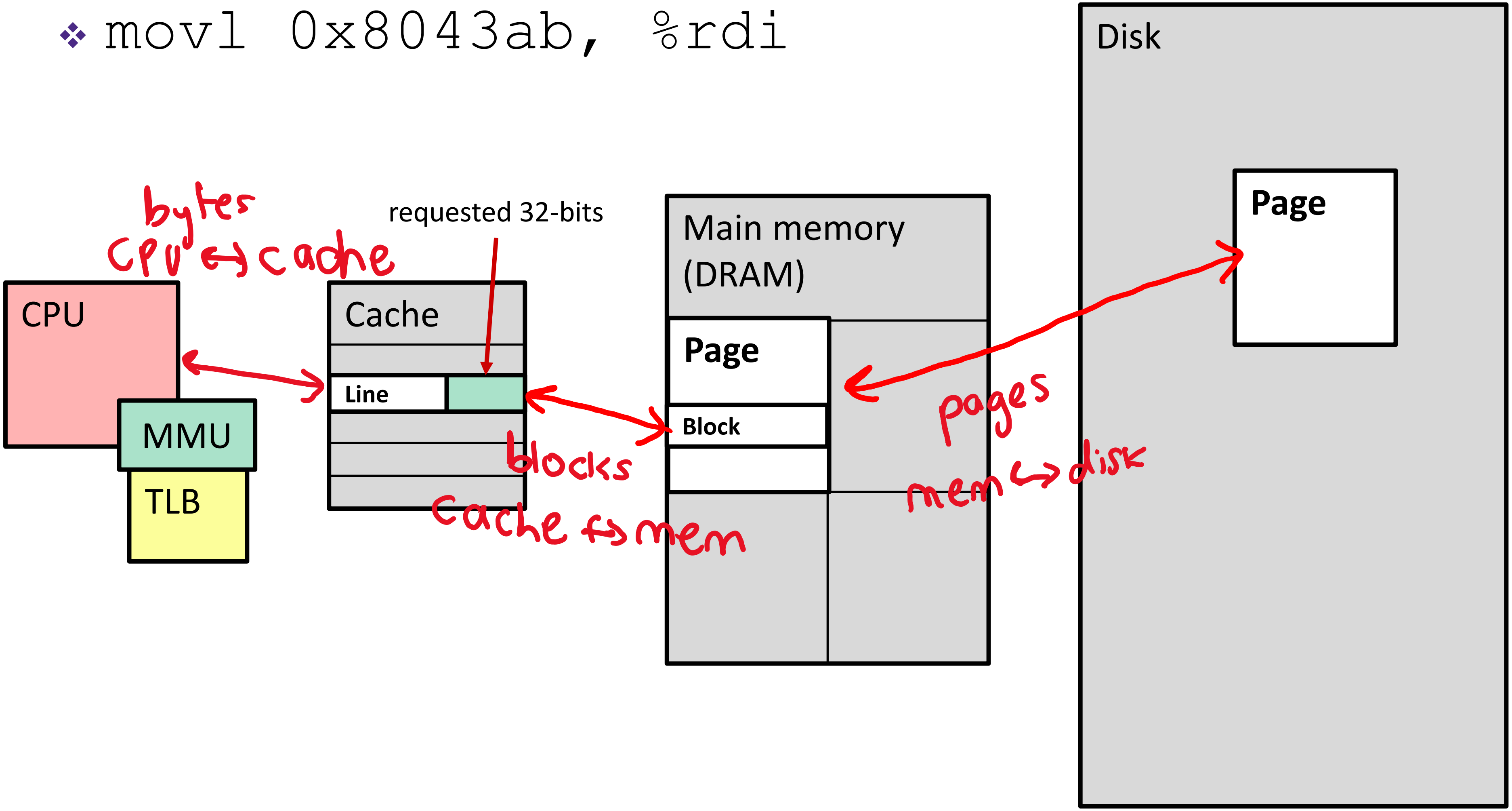
- Physical Address:

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	1	0	1	1
PPN						PPO					

CT 0x2D CI 0xA CO 0x3 Cache Hit? ✓ Data (byte) 0x3B

Memory Overview

- `movl 0x8043ab, %rdi`



Page Table Reality

This is extra (non-testable) material

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

2^{13} B

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

    2^{33} B
  - How many page table entries is that?

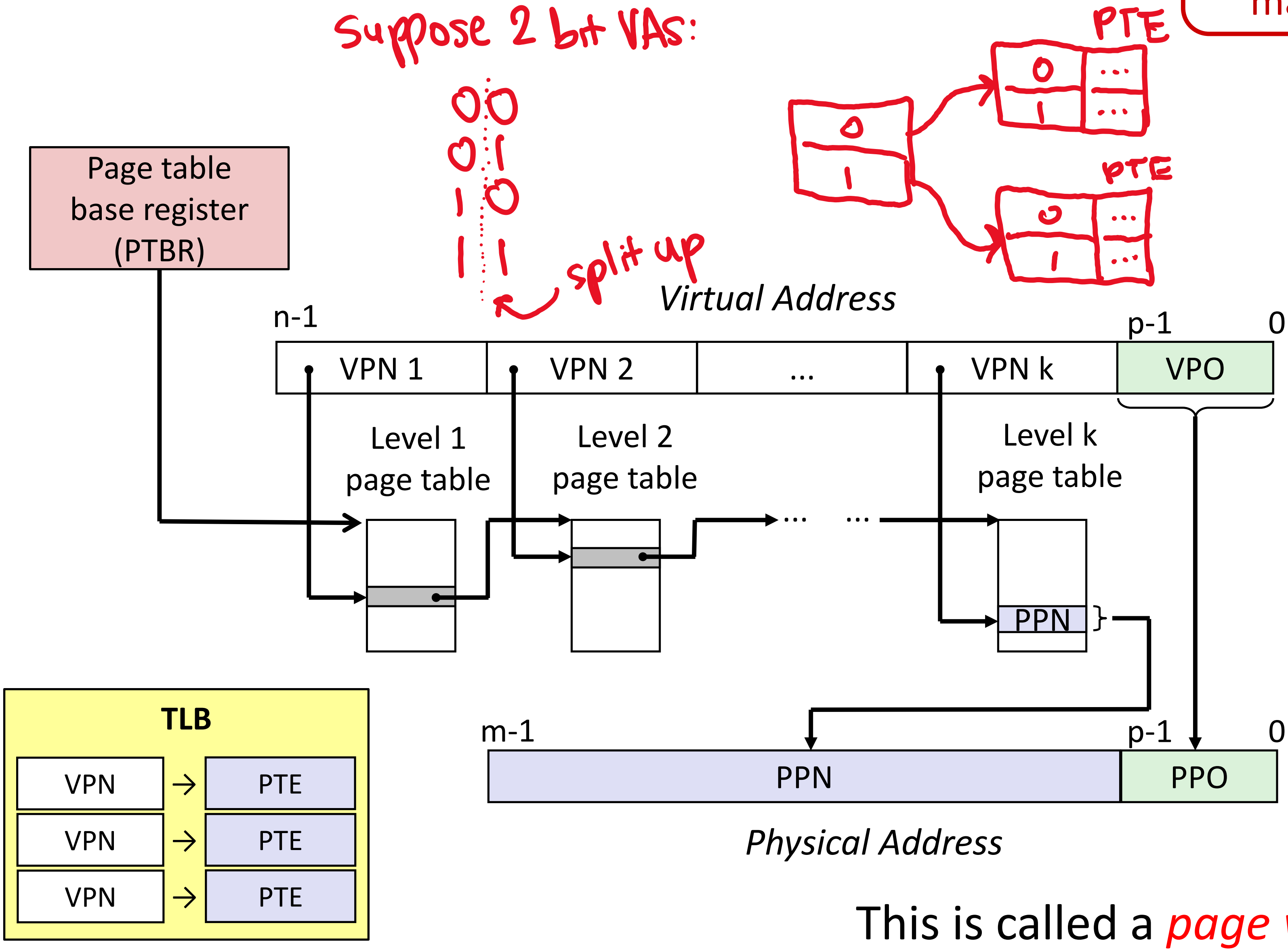
    $2^{64}/2^{13} = 2^{51}$
  - About how long is each PTE?

    $2^{33}/2^{13} = 2^{20} \Rightarrow$ 20 bit PPN + 3 mgmt. bits ≈ 3B
  - **Moral:** Cannot use this naïve implementation of the virtual → physical page mapping – it's *way* too big

total space: 3B · 2^{51} entries = $2^{52} + 2^{51}$ B
per process!! ö

This is extra (non-testable) material

A Solution: Multi-level Page Tables



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

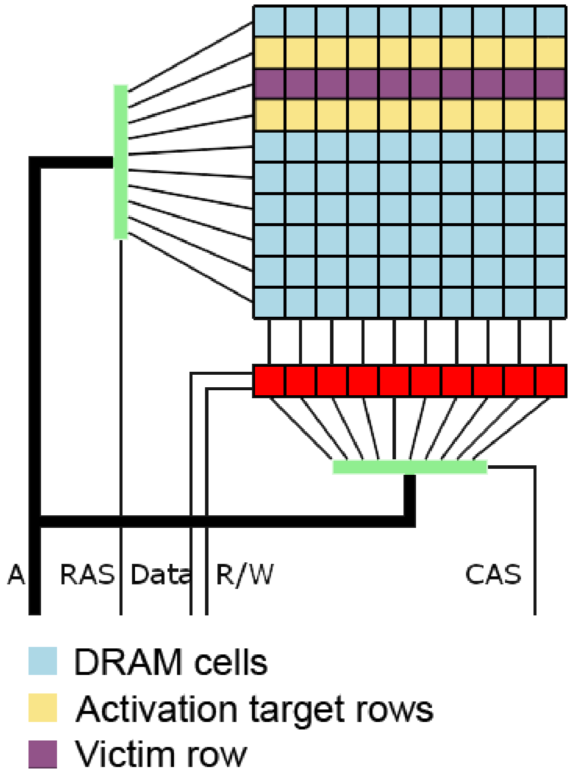
BONUS SLIDES

For Fun: DRAMMER Security Attack

- Why are we talking about this?
  - **Recent(ish):** Announced in October 2016; Google released Android patch on November 8, 2016
  - **Relevant:** Uses your system's memory setup to gain elevated privileges
    - Ties together some of what we've learned about virtual memory and processes
  - **Interesting:** It's a software attack that uses *only hardware vulnerabilities* and requires *no user permissions*

Underlying Vulnerability: Row Hammer

- Dynamic RAM (DRAM) has gotten denser over time
  - DRAM cells physically closer and use smaller charges
  - More susceptible to "*disturbance errors*" (interference)
- DRAM capacitors need to be "refreshed" periodically (~64 ms)
  - Lose data when loss of power
  - Capacitors accessed in rows
- Rapid accesses to one row can flip bits in an adjacent row!
  - ~ 100K to 1M times



By Dsimic (modified), CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=38868341

Row Hammer Exploit

- Force constant memory access
  - Read then flush the cache
  - `clflush` – flush cache line
    - Invalidates cache line containing the specified address
    - Not available in all machines or environments
  - Want addresses `X` and `Y` to fall in activation target row(s)
    - Good to understand how *banks* of DRAM cells are laid out

```
hammertime:
   mov  (X), %eax
   mov  (Y), %ebx
   clflush (X)
   clflush (Y)
   jmp hammertime
```

- The row hammer effect was discovered in 2014
  - Only works on certain types of DRAM (2010 onwards)
  - These techniques target x86 machines

Consequences of Row Hammer

- Row hammering process can affect another process via memory
  - Circumvents virtual memory protection scheme
  - Memory needs to be in an adjacent row of DRAM

- Worse: privilege escalation
  - Page tables live in memory!
  - Hope to change PPN to access other parts of memory, or change permission bits
  - **Goal:** gain read/write access to a page containing a page table, hence granting process read/write access to *all of physical memory*

Effectiveness?

- Doesn’t seem so bad – random bit flip in a row of physical memory
  - Vulnerability affected by system setup and physical condition of memory cells

- **Improvements:**
  - Double-sided row hammering increases speed & chance
  - Do system identification first (e.g. Lab 4)
    - Use timing to infer memory row layout & find “bad” rows
    - Allocate a huge chunk of memory and try many addresses, looking for a reliable/repeatable bit flip
  - Fill up memory with page tables first
    - `fork` extra processes; hope to elevate privileges in any page table

What’s DRAMMER?

- No one previously made a huge fuss
  - **Prevention:** error-correcting codes, target row refresh, higher DRAM refresh rates
  - Often relied on special memory management features
  - Often crashed system instead of gaining control

- Research group found a *deterministic* way to induce row hammer exploit in a non-x86 system (ARM)
  - Relies on predictable reuse patterns of standard physical memory allocators
  - Universiteit Amsterdam, Graz University of Technology, and University of California, Santa Barbara

DRAMMER Demo Video

- It’s a shell, so not that sexy-looking, but still interesting
  - Apologies that the text is so small on the video

How did we get here?

- Computing industry demands more and faster storage with lower power consumption
- Ability of user to circumvent the caching system
  - `clflush` is an unprivileged instruction in x86
  - Other commands exist that skip the cache
- Availability of virtual to physical address mapping
  - **Example:** `/proc/self/pagemap` on Linux (not human-readable)

- Google patch for Android (Nov. 8, 2016)
  - Patched the ION memory allocator

More reading for those interested

- DRAMMER paper: https://vvdveen.com/publications/drammer.pdf
- Google Project Zero: https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html
- First row hammer paper: https://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf
- Wikipedia: https://en.wikipedia.org/wiki/Row_hammer