

# Virtual Memory II

CSE 351 Summer 2020

## Instructor:

Porter Jones

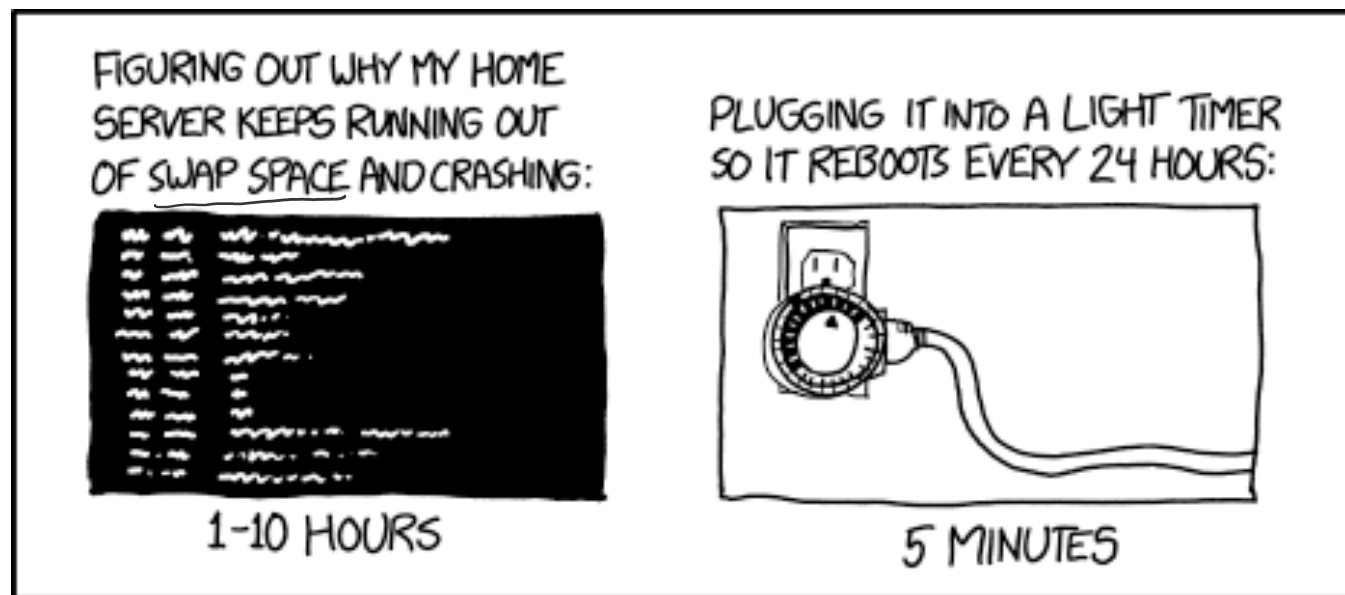
## Teaching Assistants:

Amy Xu

Callum Walker

Sam Wolfson

Tim Mandzyuk



WHY EVERYTHING I HAVE IS BROKEN

<https://xkcd.com/1495/>

# Administrivia

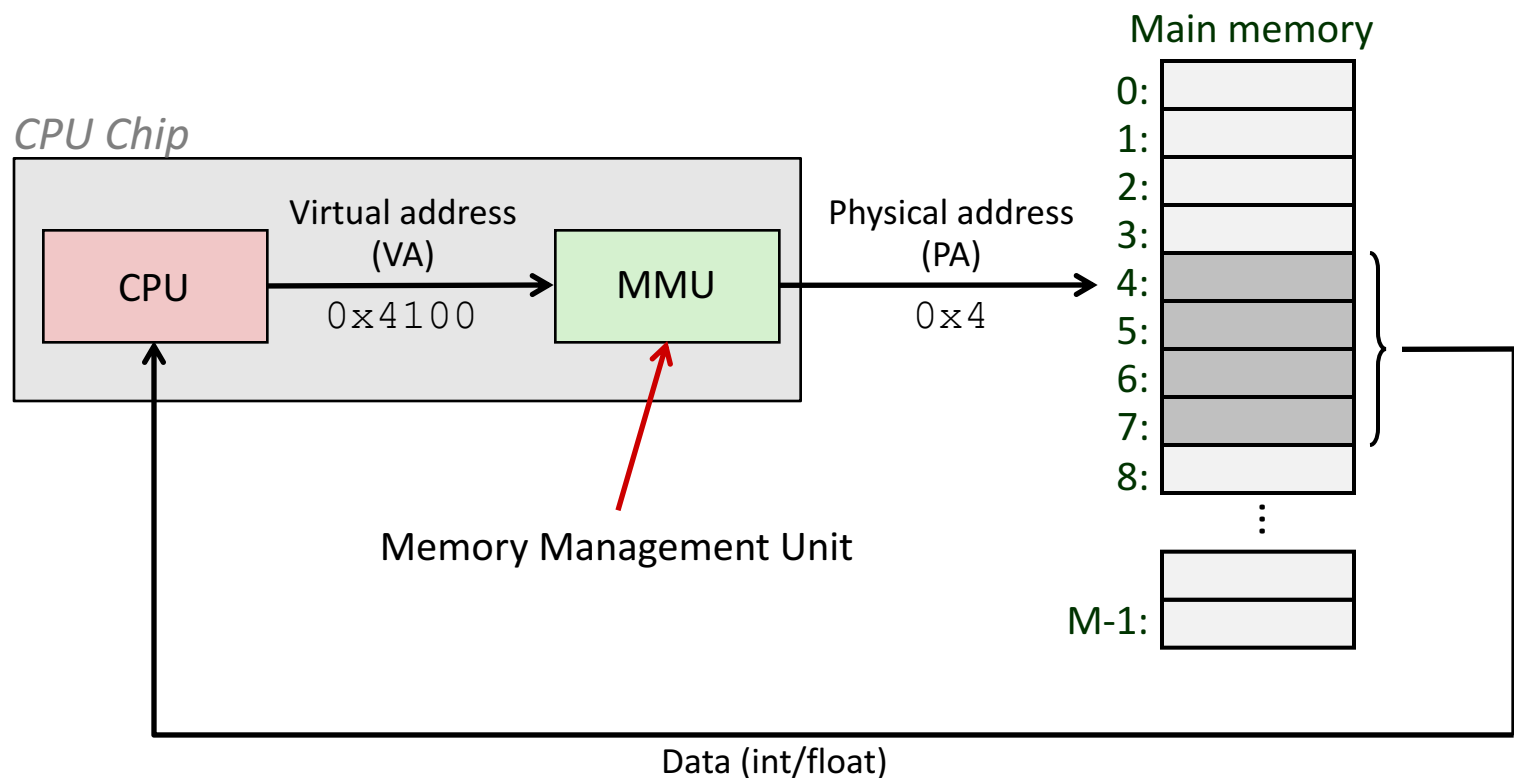
- ❖ Questions doc: <https://tinyurl.com/CSE351-8-10>
  
- ❖ hw19 is optional
  - Can complete it at any point before the quarter ends
  - Practice with virtual memory concepts
  
- ❖ hw20 due Friday (8/14) – 10:30am
  
  
- ❖ Lab 4 due Wednesday (8/12) – 11:59pm
  - All about caches!

# 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

# Address Translation

*How do we perform the virtual  
→ physical address translation?*

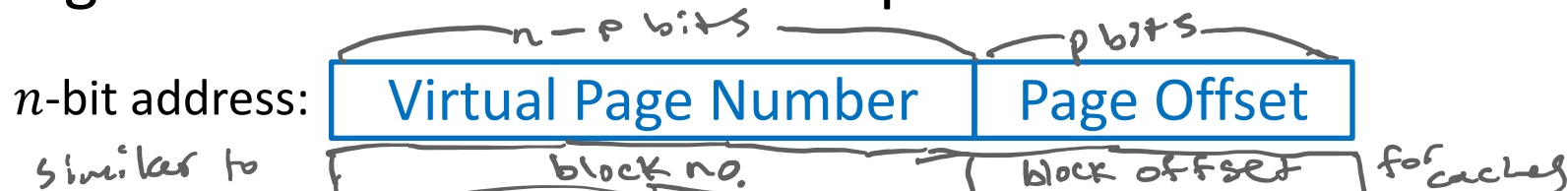


# Address Translation: Page Tables

$2^{n-p}$  total virtual pages

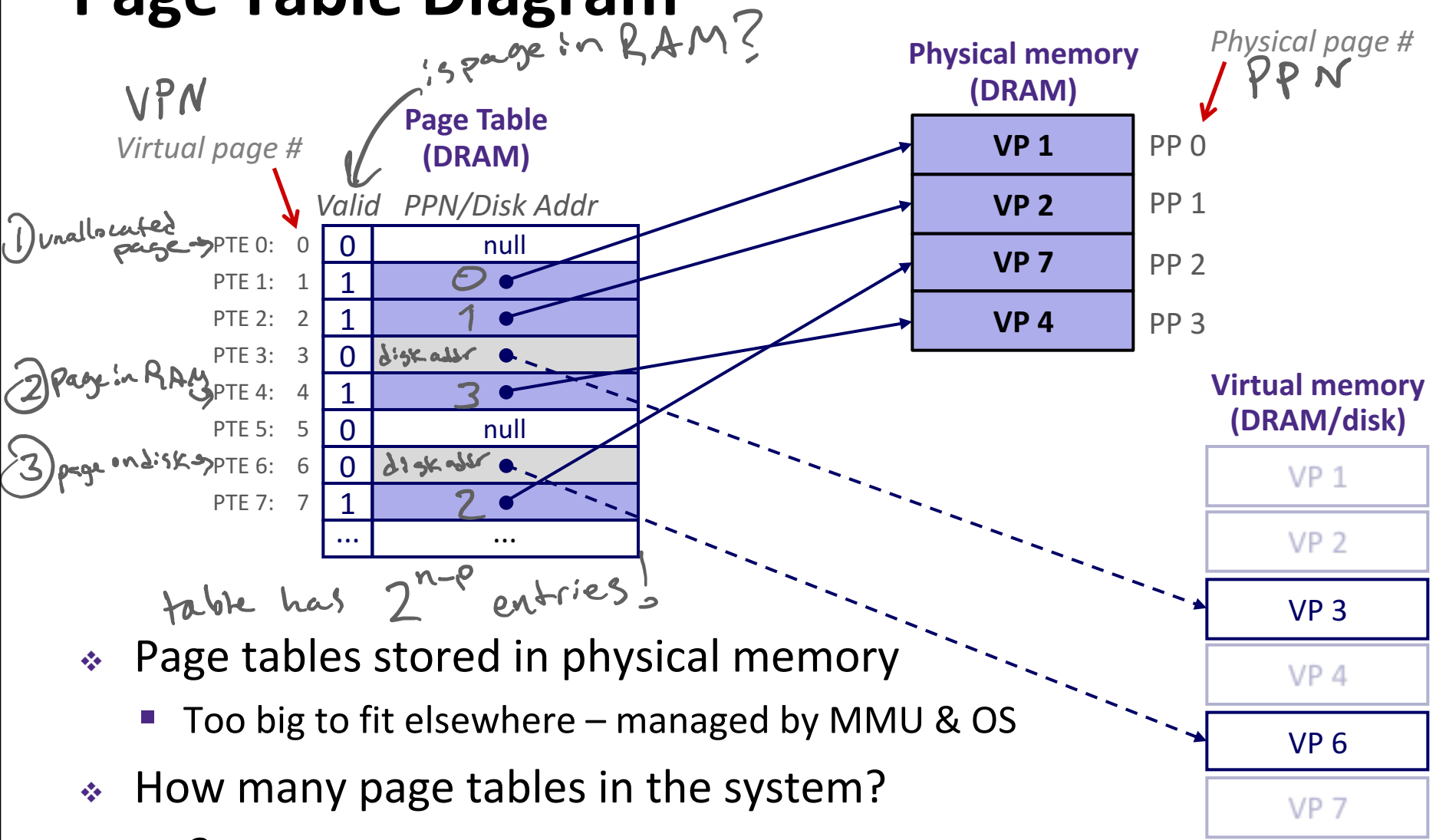
page size  $P$   
 $P = 2^p$   
 $p = \lceil \log_2 P \rceil$

- ❖ CPU-generated address can be split into:



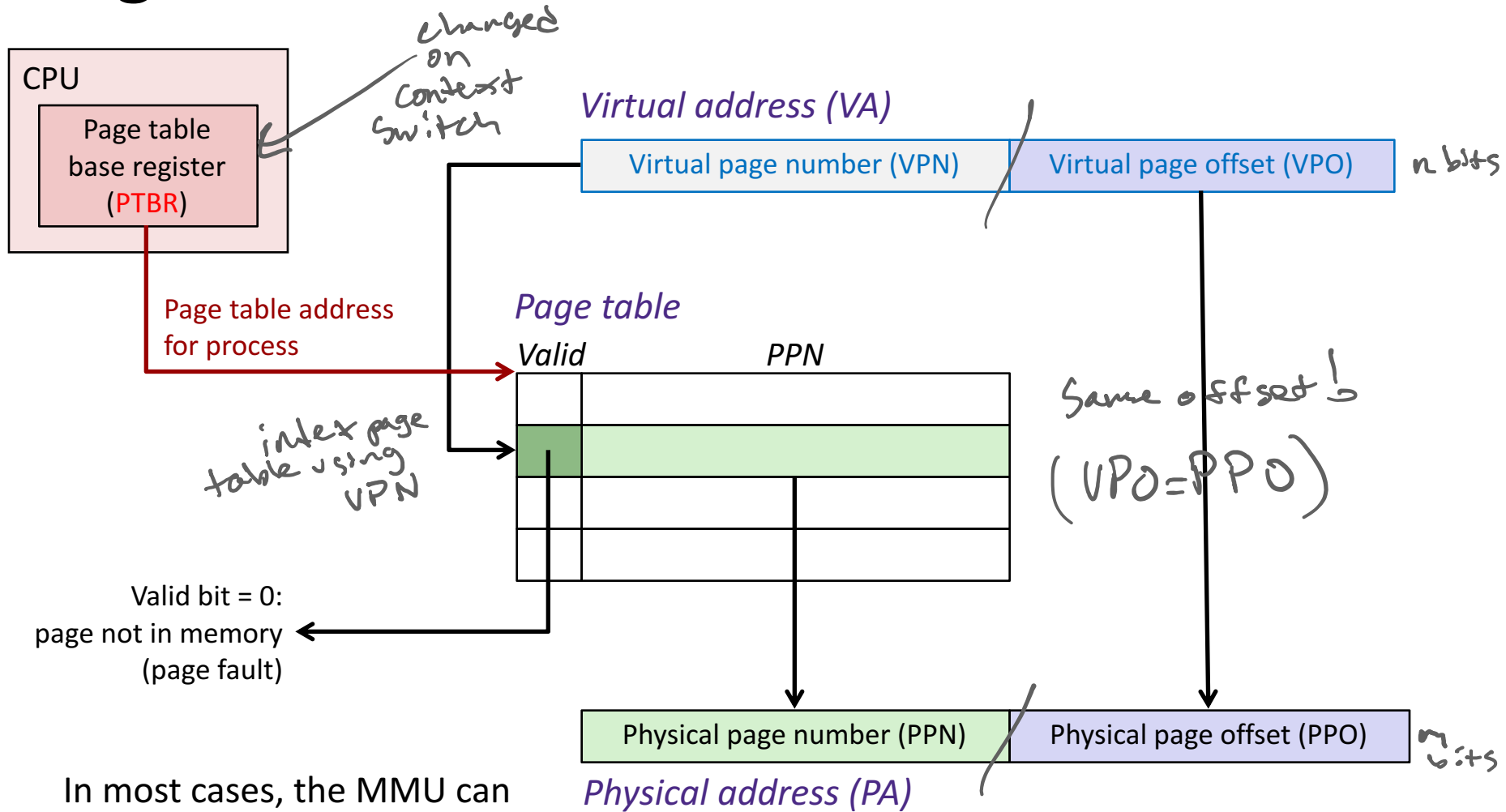
- Request is Virtual Address (**VA**), want Physical Address (**PA**)
  - Note that Physical Offset = Virtual Offset (page-aligned)
- ❖ Use lookup table that we call the **page table (PT)**
  - Replace Virtual Page Number (**VPN**) for Physical Page Number (**PPN**) to generate Physical Address
  - Index PT using VPN: page table entry (**PTE**) stores the PPN plus management bits (e.g. Valid, Dirty, access rights)
  - Has an entry for *every* virtual page

# Page Table Diagram



- ❖ Page tables stored in physical memory
  - Too big to fit elsewhere – managed by MMU & OS
- ❖ How many page tables in the system?
  - One per process

# Page Table Address Translation



In most cases, the MMU can perform this translation without software assistance

# Polling Question [VM II]

❖ How many bits wide are the following fields?

- 16 KiB pages  
 $2^4 \cdot 2^{10} = 2^{14}$
- 48-bit virtual addresses
- 16 GiB physical memory  
 $2^4 \cdot 2^{30} = 2^{34}$
- Vote at: <http://pollev.com/pbjones>

|     | VPN | PPN |
|-----|-----|-----|
| (A) | 34  | 24  |
| (B) | 32  | 18  |
| (C) | 30  | 20  |
| (D) | 34  | 20  |

$$VA \text{ width} = 48 \text{ bits}$$

$$PA \text{ width} = \log_2(2^{34}) = 34 \text{ bits}$$

$$\text{offset width} = \log_2(2^{14}) = 14 \text{ bits}$$

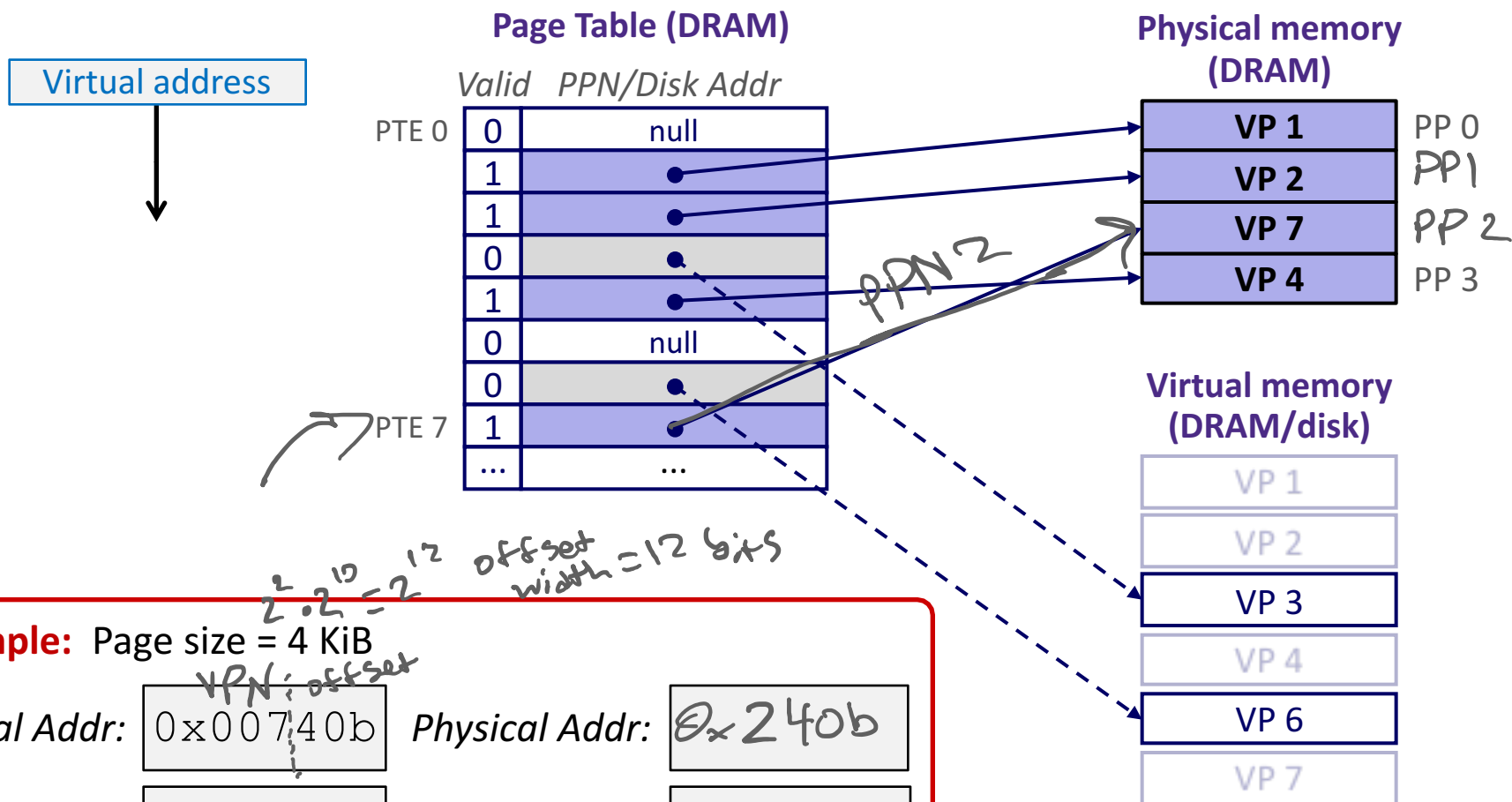
$$VPN \text{ width} = VA \text{ width} - \text{offset width} = 34 \text{ bits}$$

$$PPN \text{ width} = PA \text{ width} - \text{offset width} = 20 \text{ bits}$$



# Page Hit

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



**Example:** Page size = 4 KiB

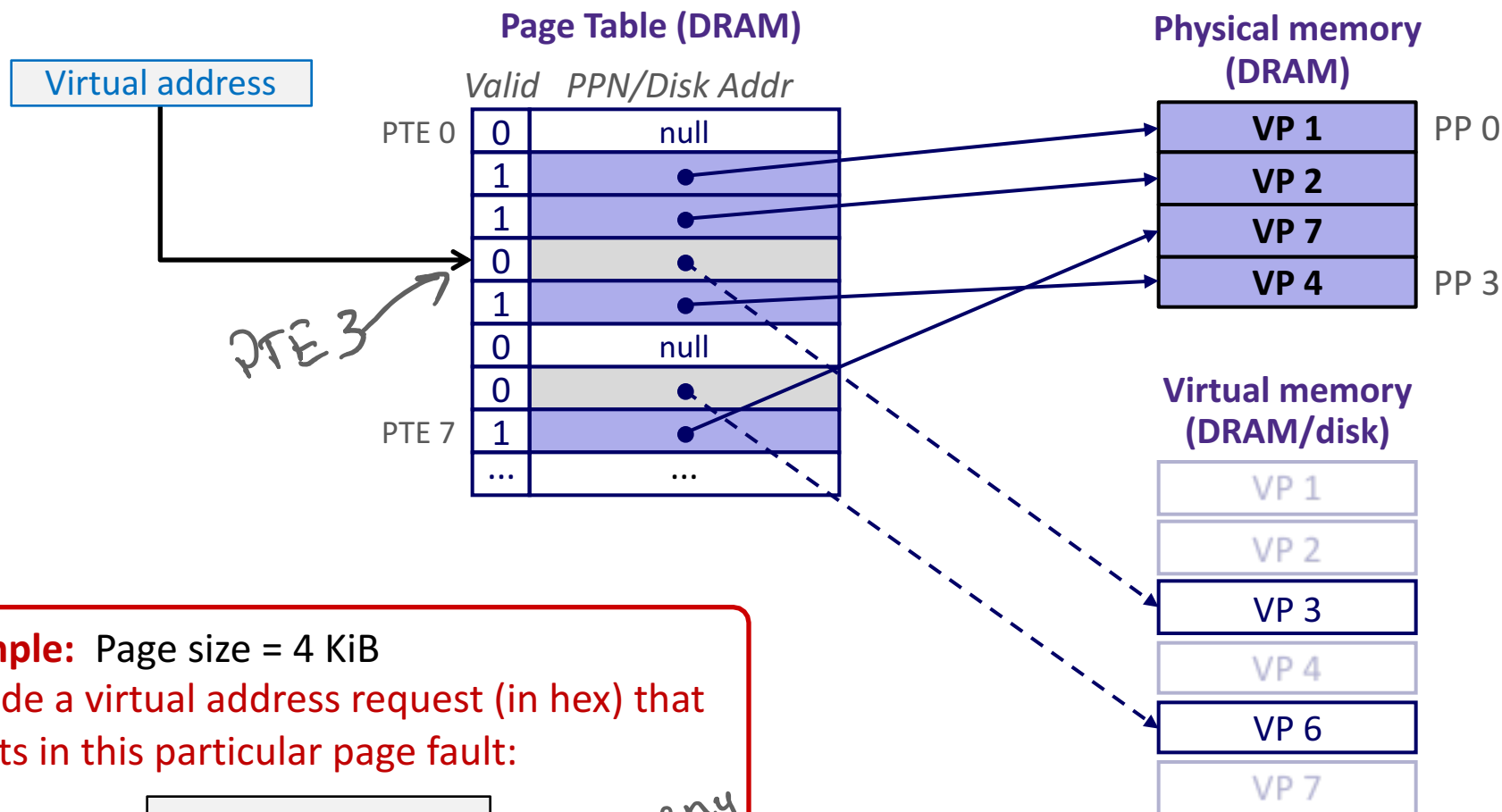
$2^2 \cdot 2^{10} = 2^{12}$  offset width = 12 bits

Virtual Addr:  $0x00740b$       Physical Addr:  $0x240b$

VPN:  $0x007$       PPN:  $0x02$

# Page Fault

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



**Example:** Page size = 4 KiB  
 Provide a virtual address request (in hex) that results in this particular page fault:

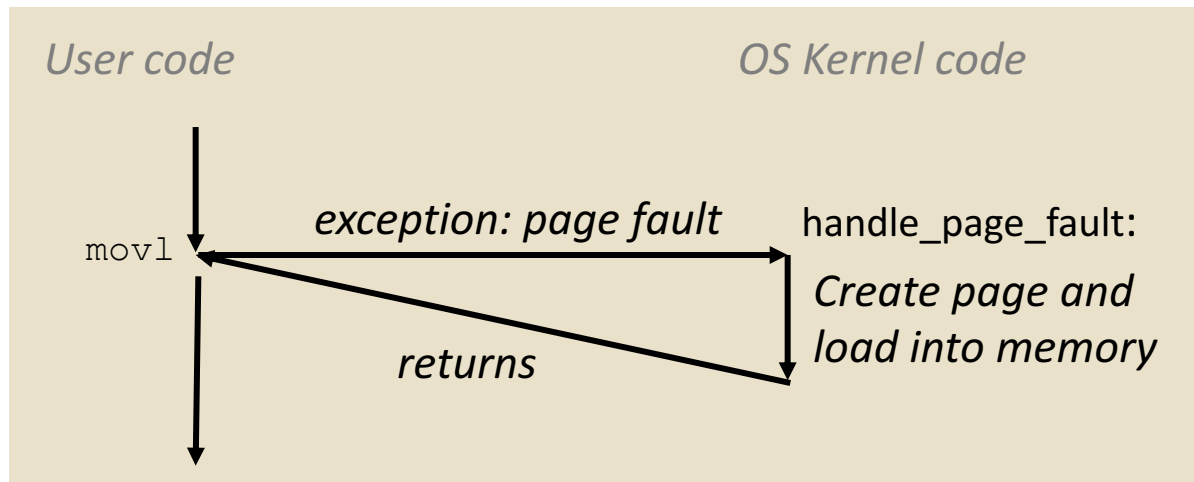
Virtual Addr: 0x003\_ \_ \_ ← any offset

# Reminder: Page Fault Exception

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

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

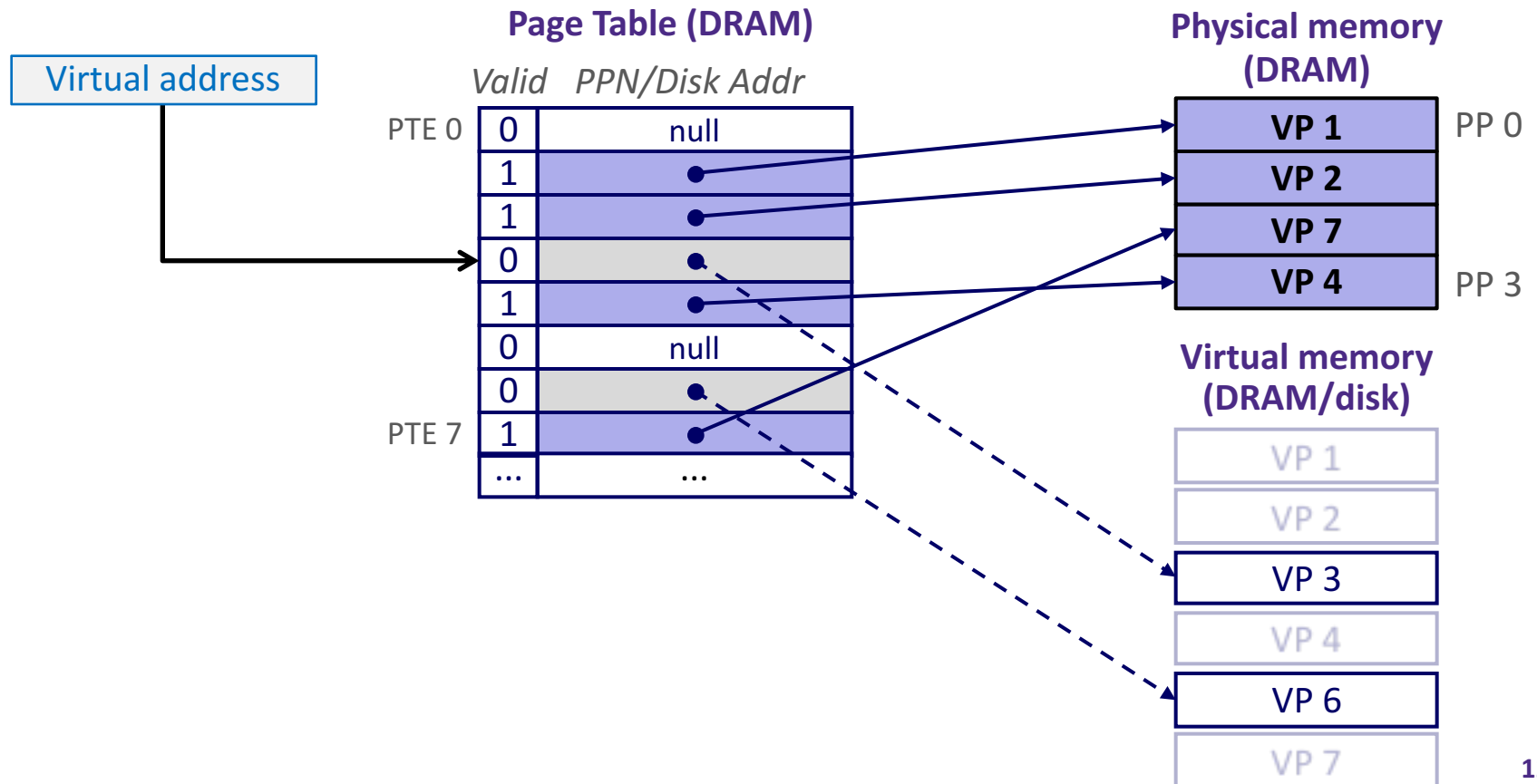
```
80483b7:      c7 05 10 9d 04 08 0d  movl    $0xd,0x8049d10
```



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

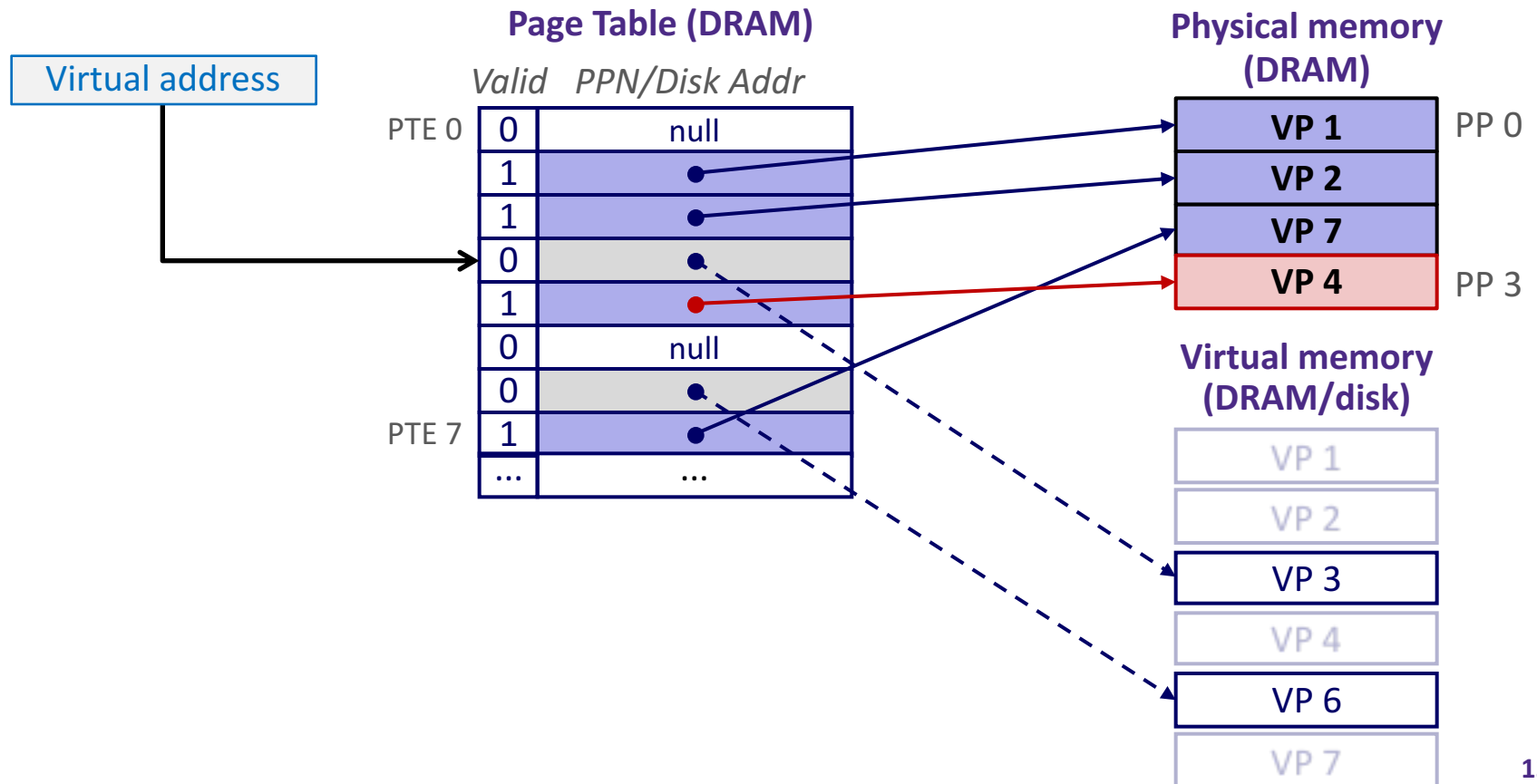
# Handling a Page Fault

- ❖ Page miss causes page fault (an exception)



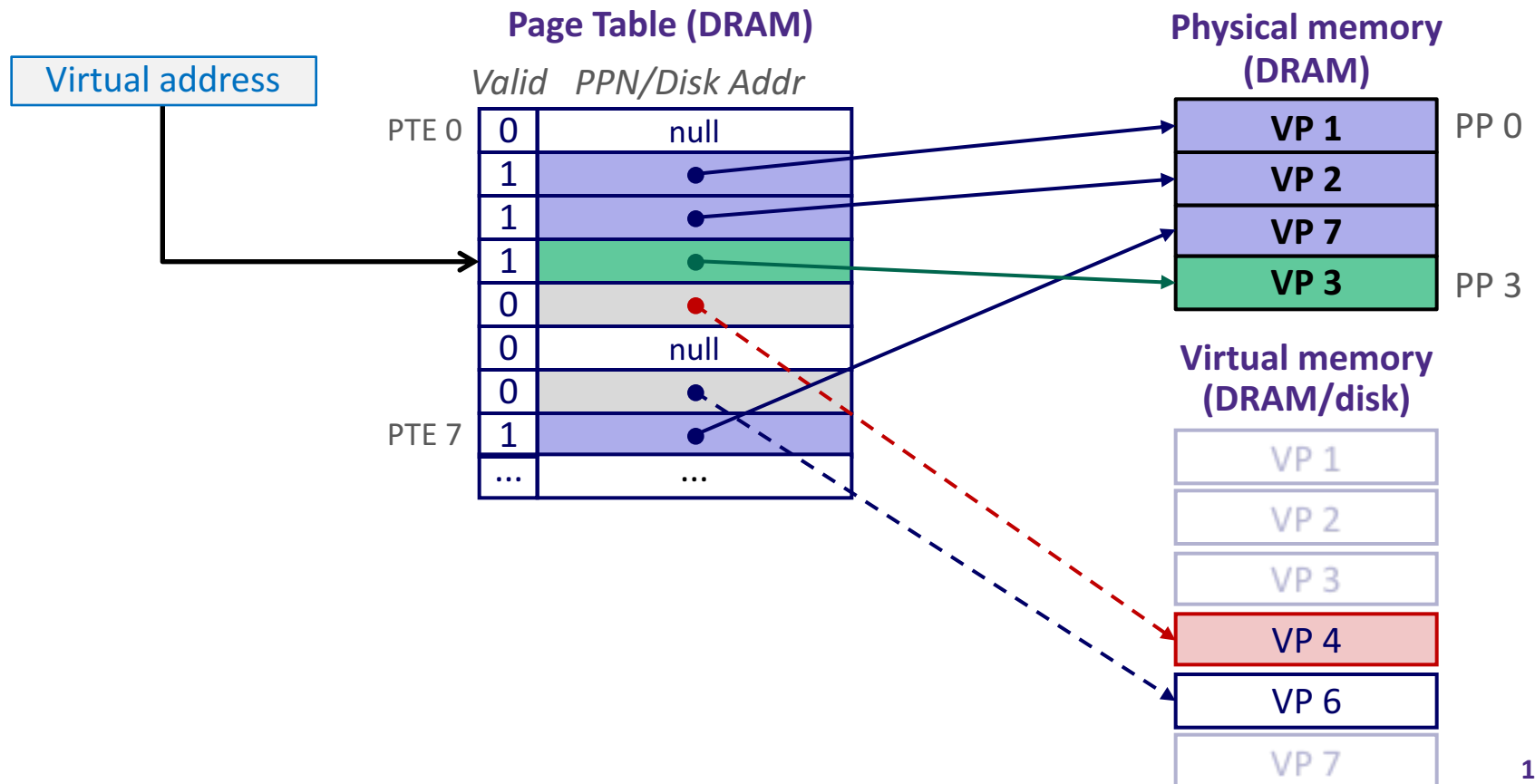
# Handling a Page Fault

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



# Handling a Page Fault

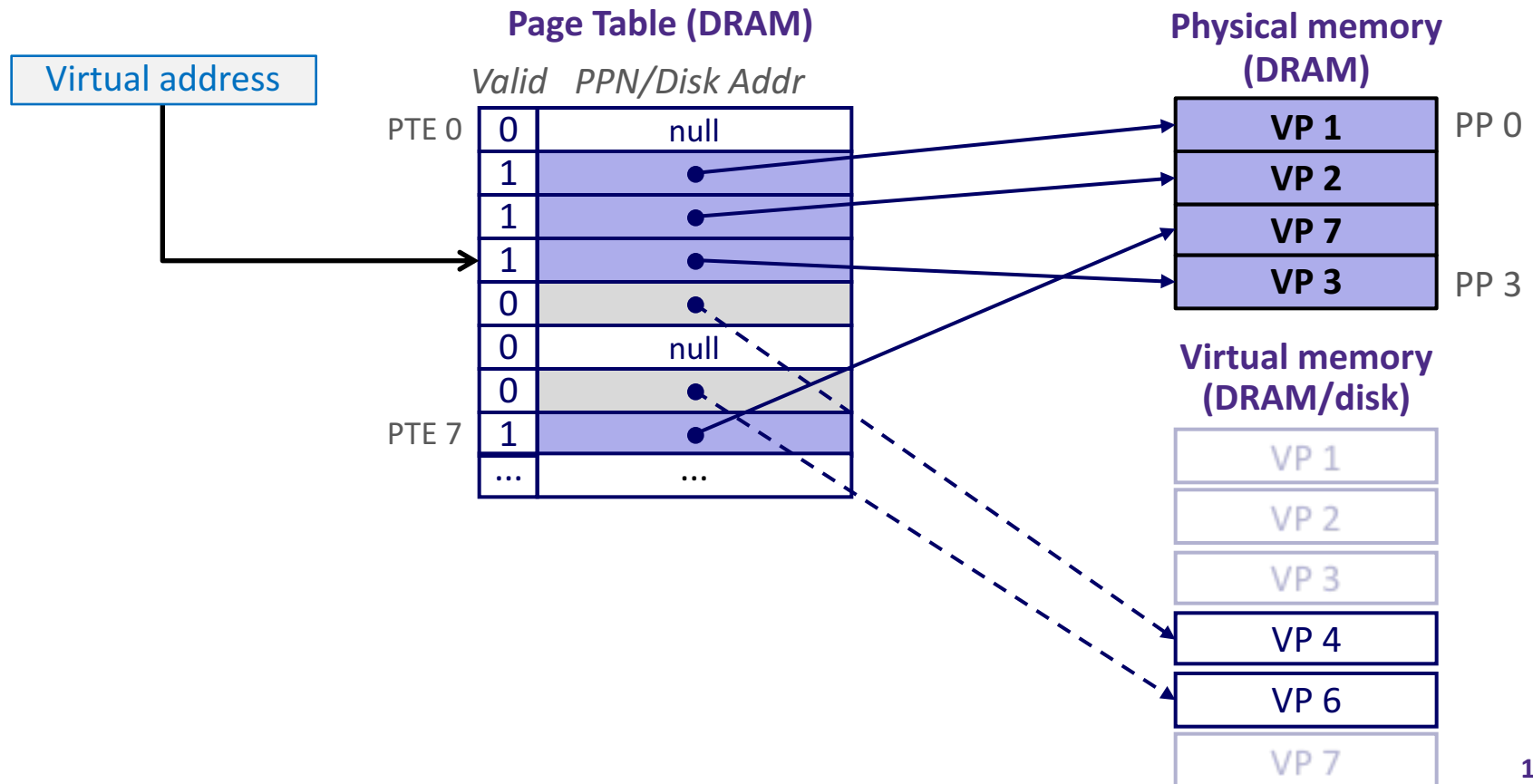
- ❖ Page miss causes page fault (an exception)
- ❖ Page fault handler selects a *victim* to be evicted (here VP 4)



# Handling a Page Fault

*OS chooses victim!*

- ❖ 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!**



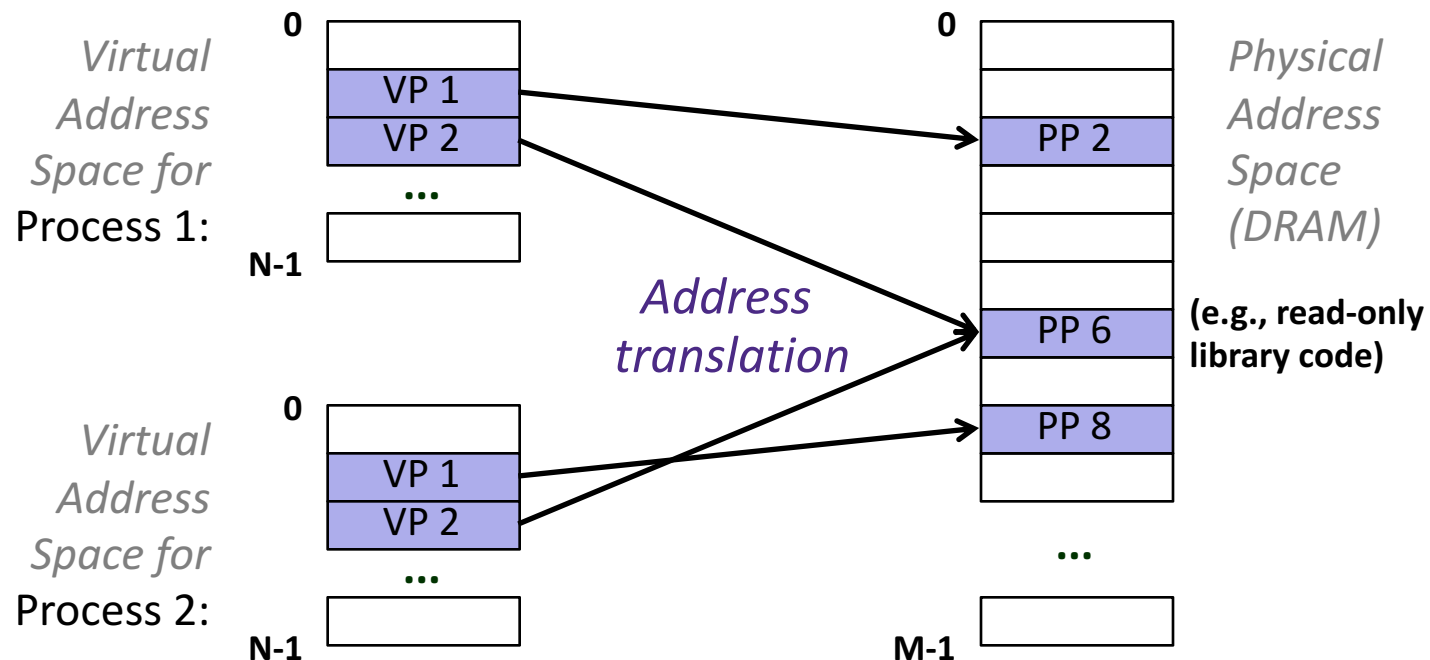
# Virtual Memory (VM)

- ❖ 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

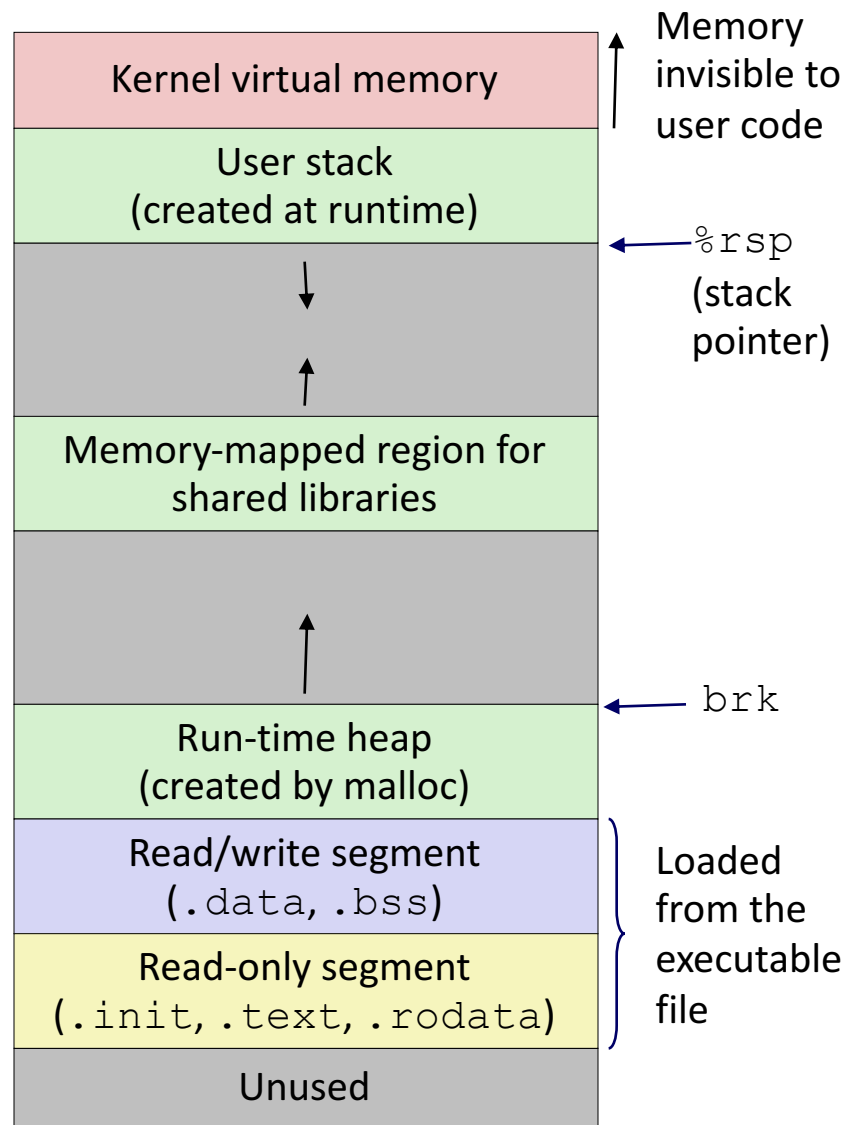
- Each program has similar virtual address space
- Code, Data, and Heap always start at the same addresses

## ❖ 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

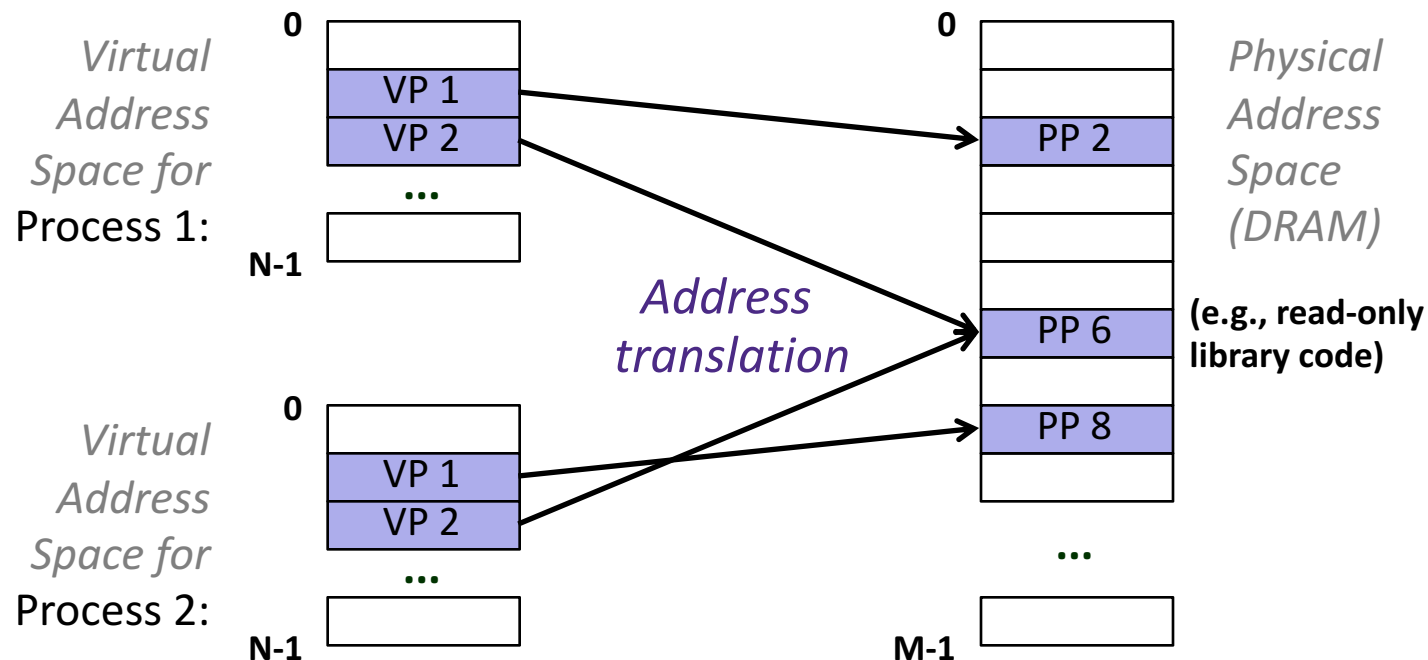
0x400000

0



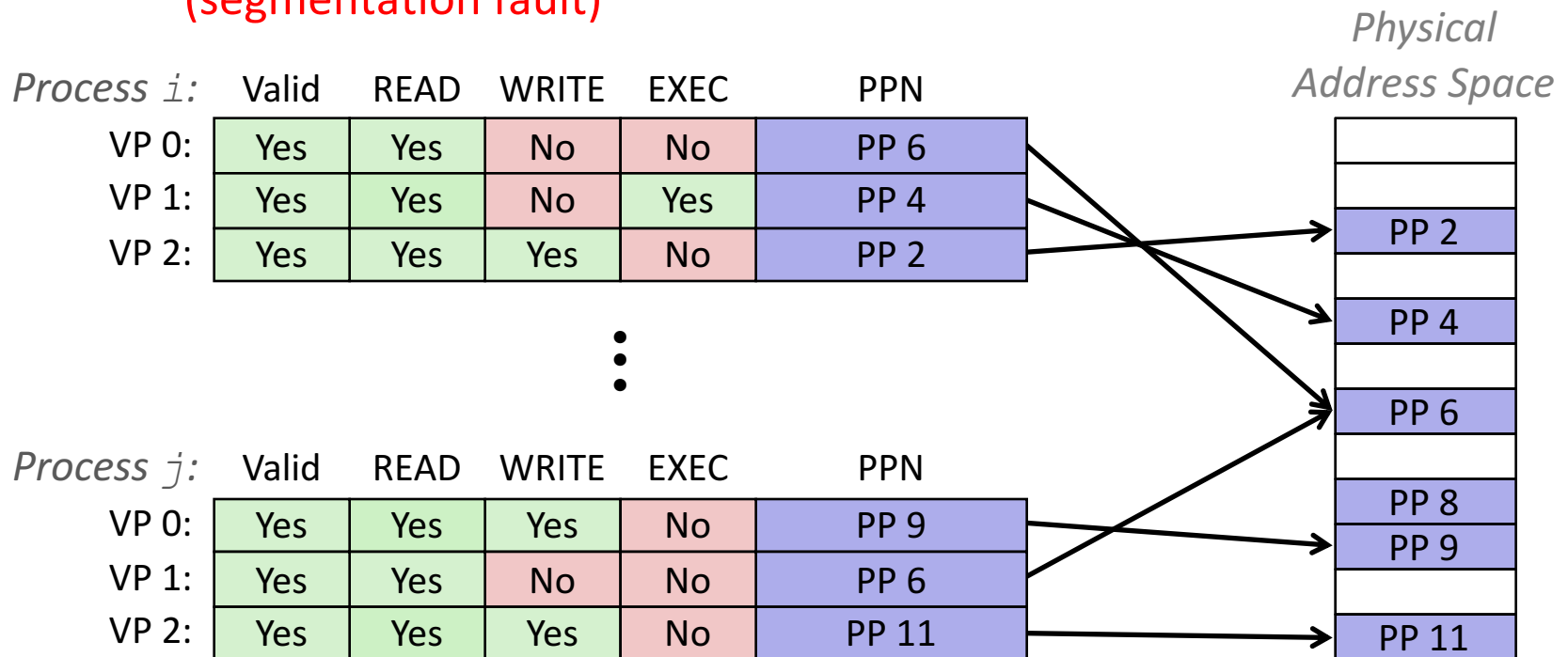
# VM for Protection and Sharing

- ❖ 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)



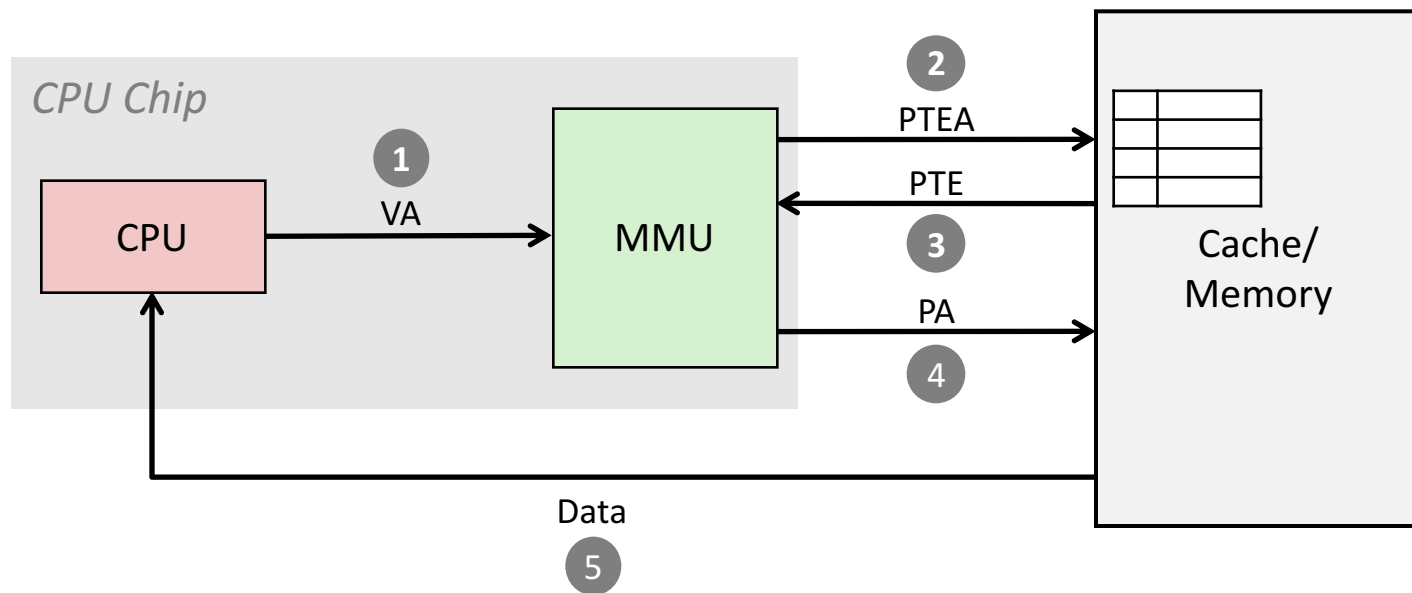
# Review Question

- ❖ 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                             |
| <u>Static Data</u> | 1    | 1                    | 0                             |
| Literals           | 1    | 0 (constant)         | 0                             |
| Instructions       | 1    | 0 (don't alter code) | 1 (only execute instructions) |

Static  
in  
size

# Address Translation: Page Hit *(Page is in physical memory)*

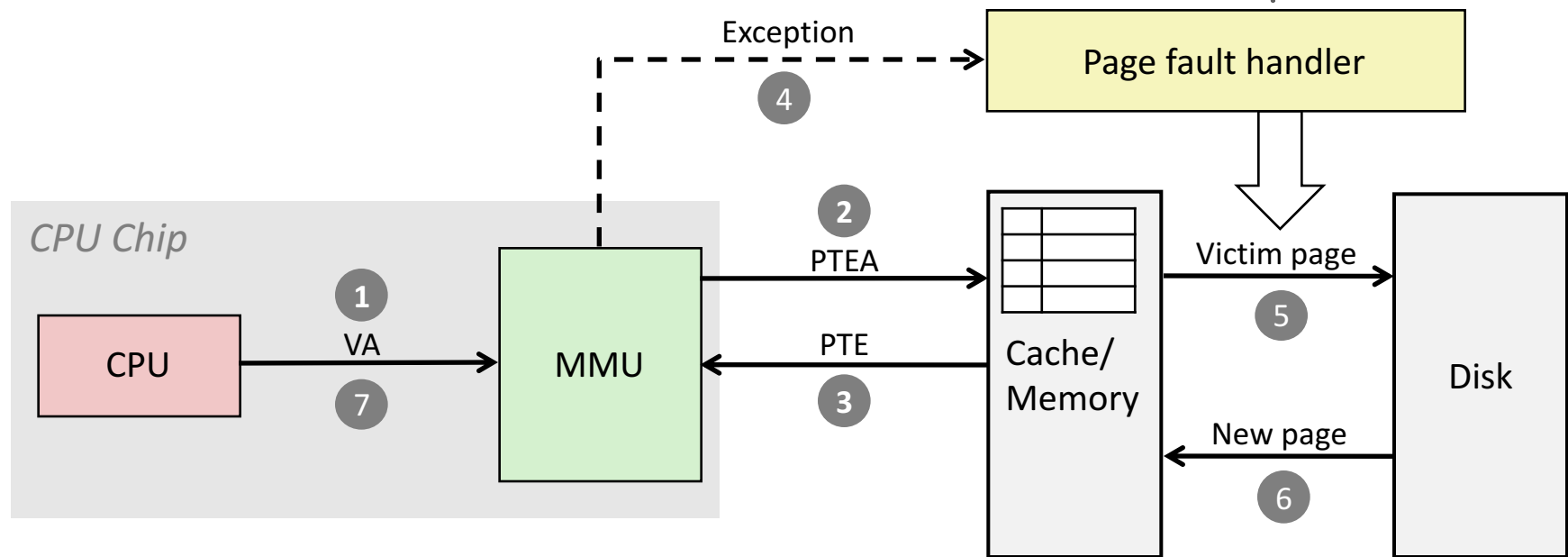


- 1) Processor sends *virtual* address to MMU (*memory management unit*)
- 2-3) MMU fetches PTE from page table in cache/memory  
(Uses PTBR to find beginning of page table for current process)
- 4) MMU sends *physical* address to cache/memory requesting data
- 5) Cache/memory sends data to processor

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

*(Page is not in physical memory)*



- 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! 🎉

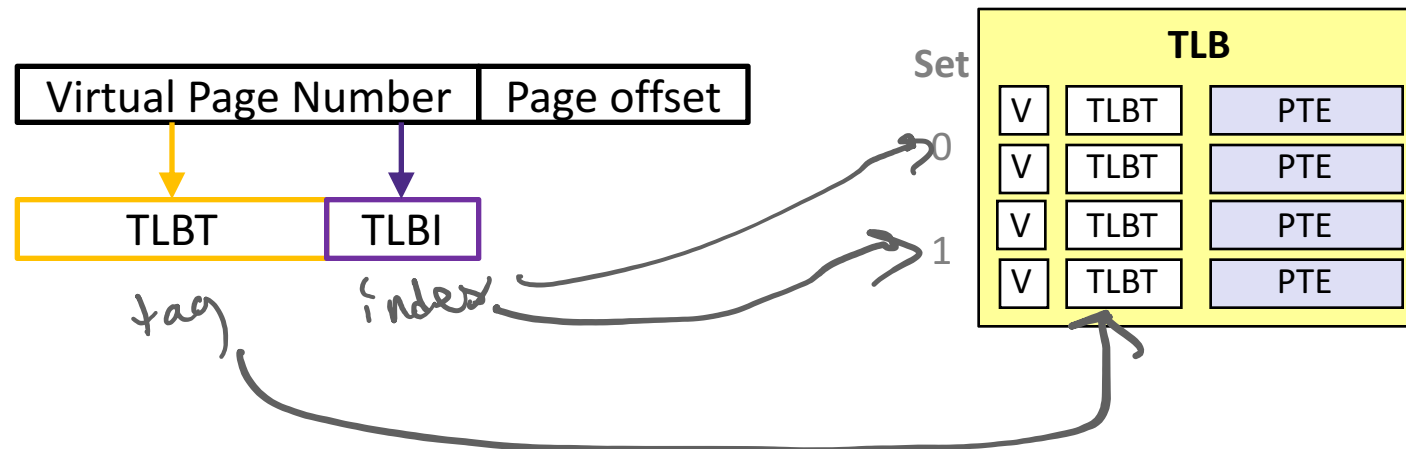


# Speeding up Translation with a TLB

"cache"

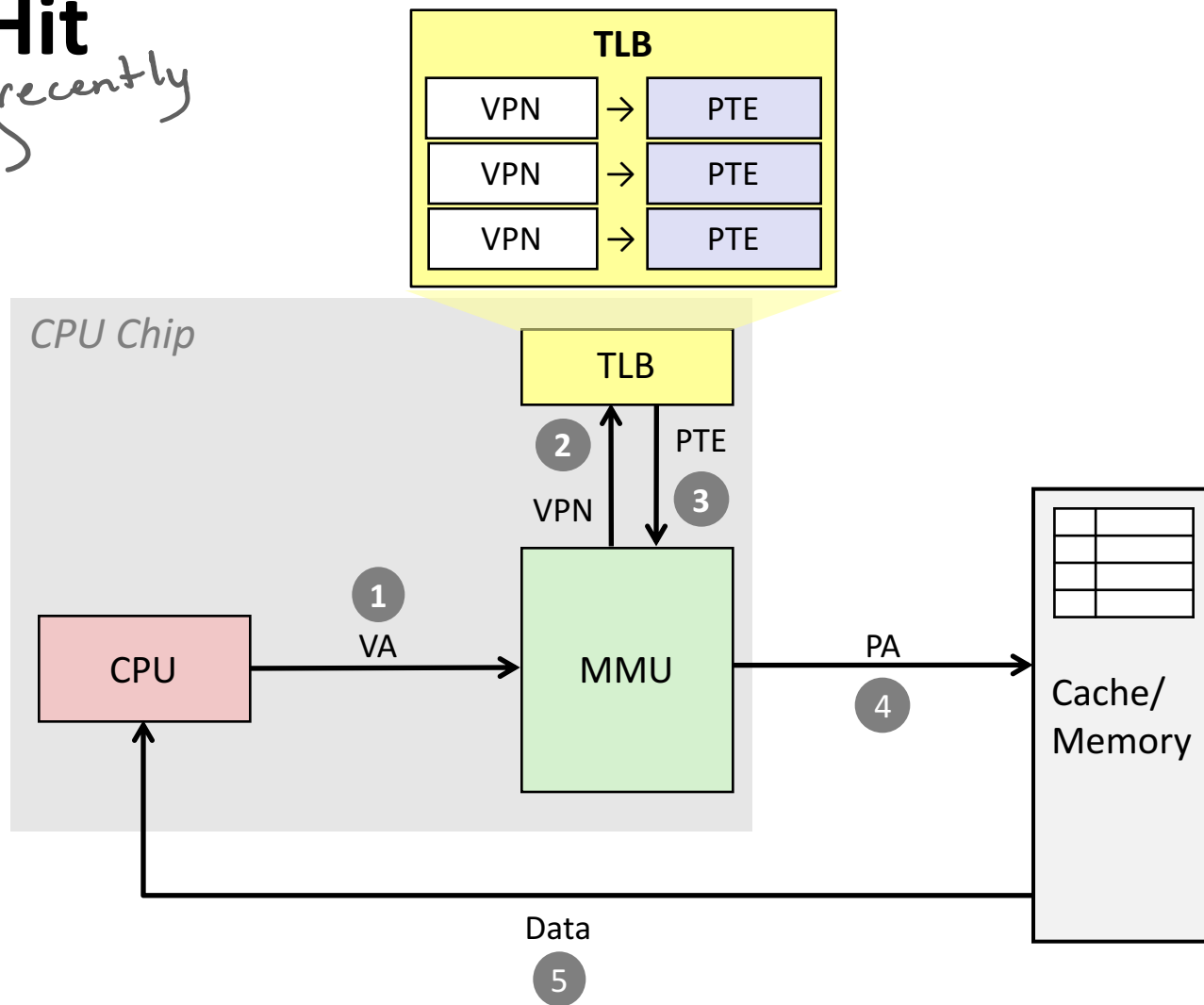
## ❖ *Translation Lookaside Buffer (TLB):*

- 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

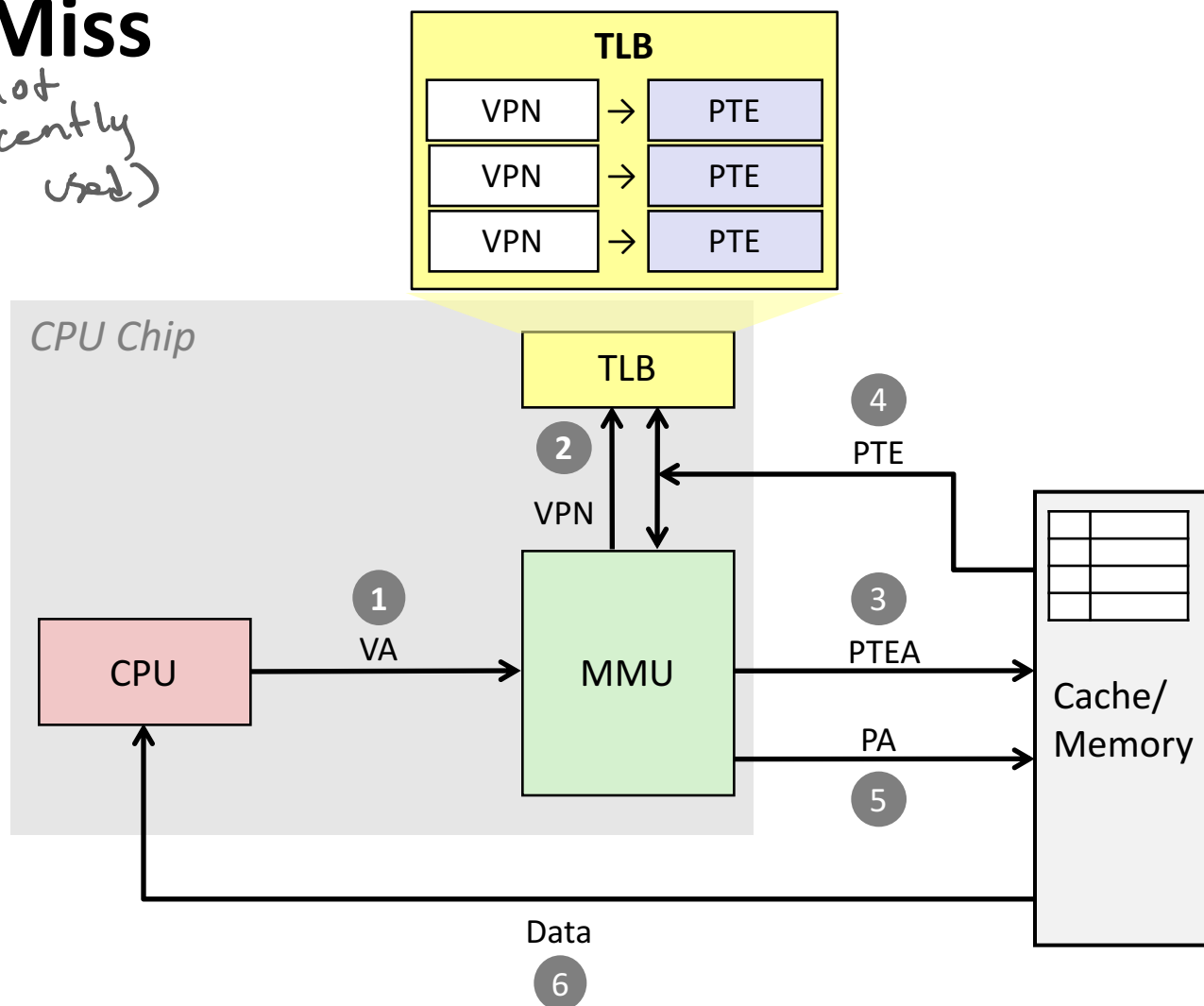
*(page recently used)*



- ❖ A TLB hit eliminates a memory access!

# TLB Miss

(page not recently used)



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

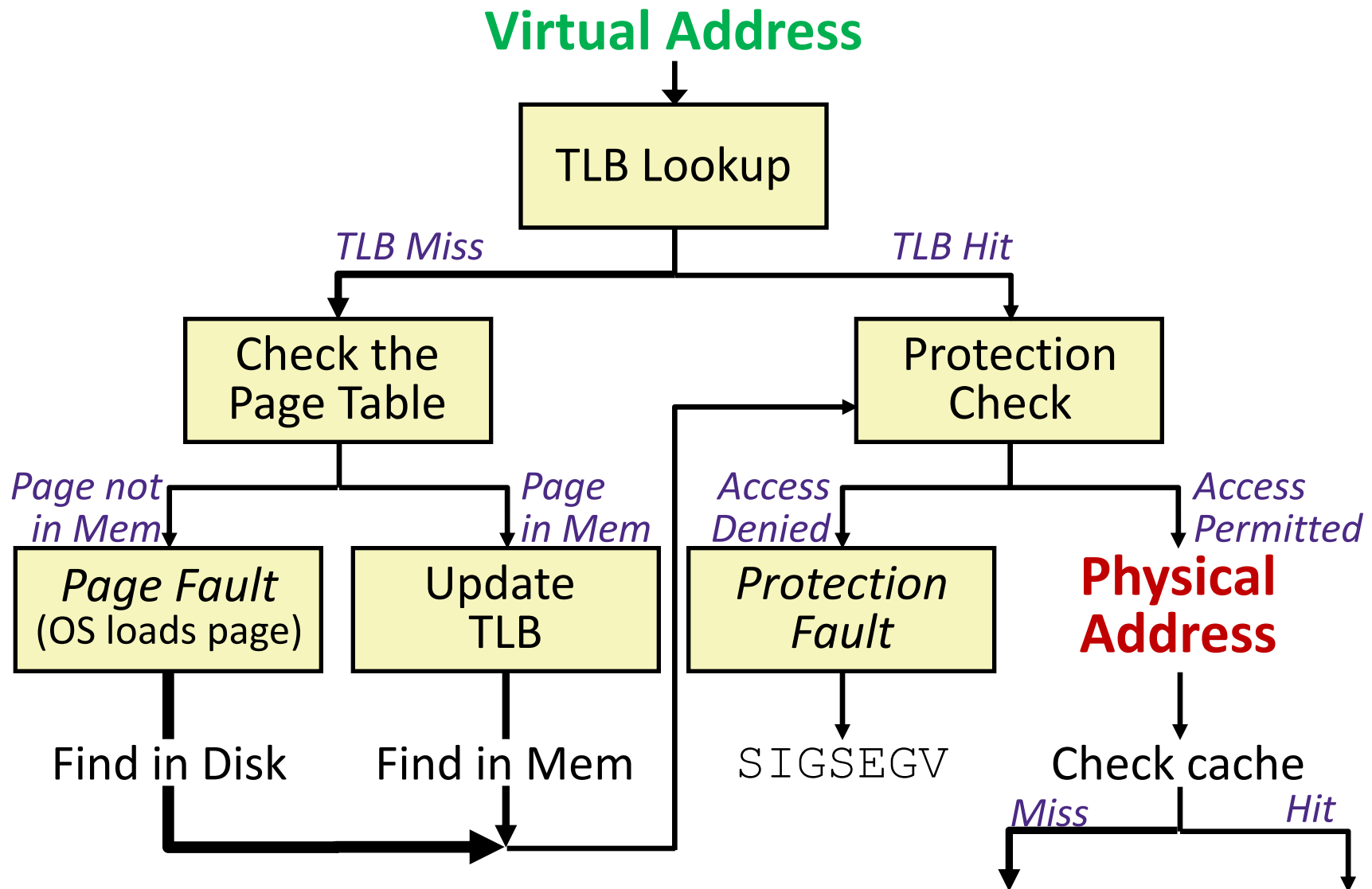
# Fetching Data on a Memory Read

- 1) Check TLB *translate VA  $\rightarrow$  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

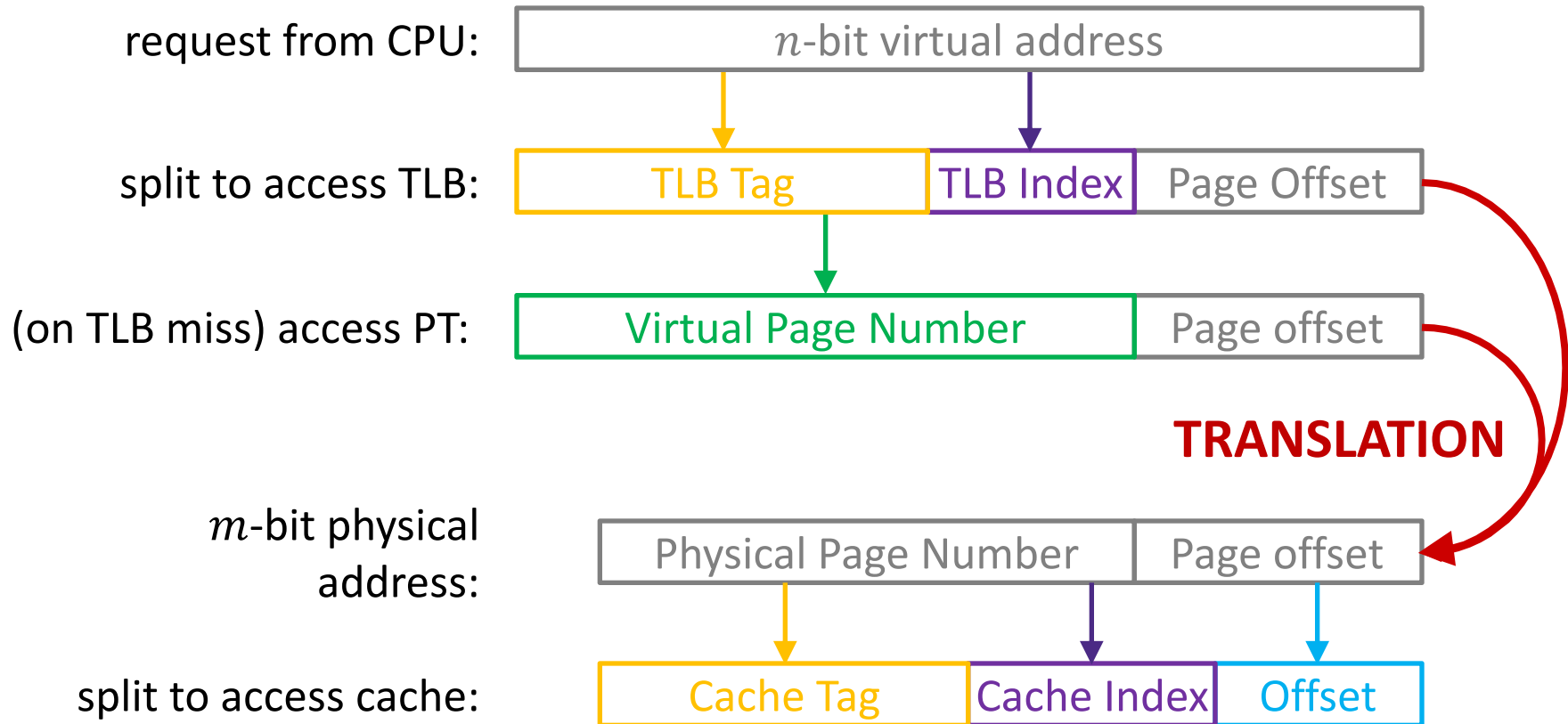
---

- 2) Check cache *use PA to get 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

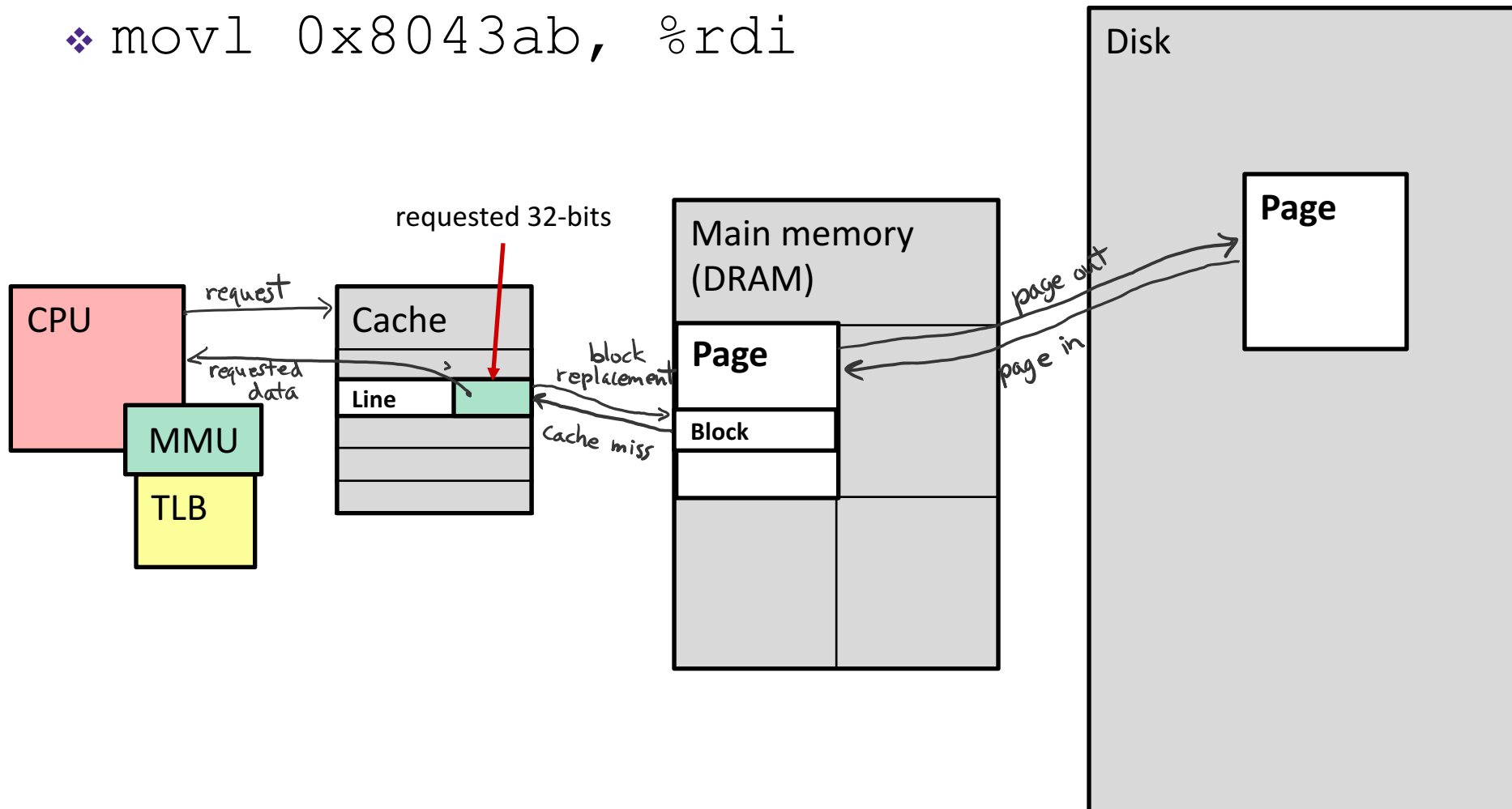


# Context Switching Revisited

- ❖ 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)
  - <sup>physical</sup>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

# Memory Overview *(data flow)*

❖ `movl 0x8043ab, %rdi`





# Summary of Address Translation Symbols

## ❖ Basic Parameters

- $N = 2^n$  Number of addresses in virtual address space
- $M = 2^m$  Number of addresses in physical address space
- $P = 2^p$  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

# 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

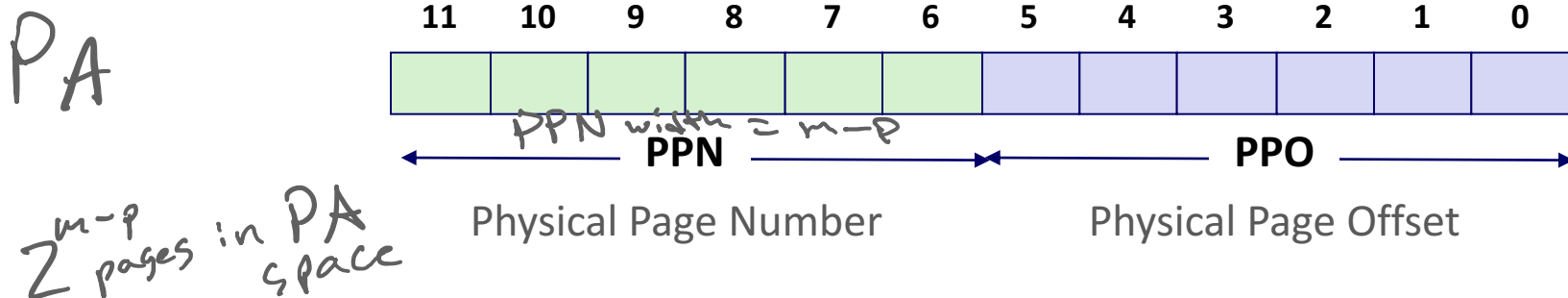
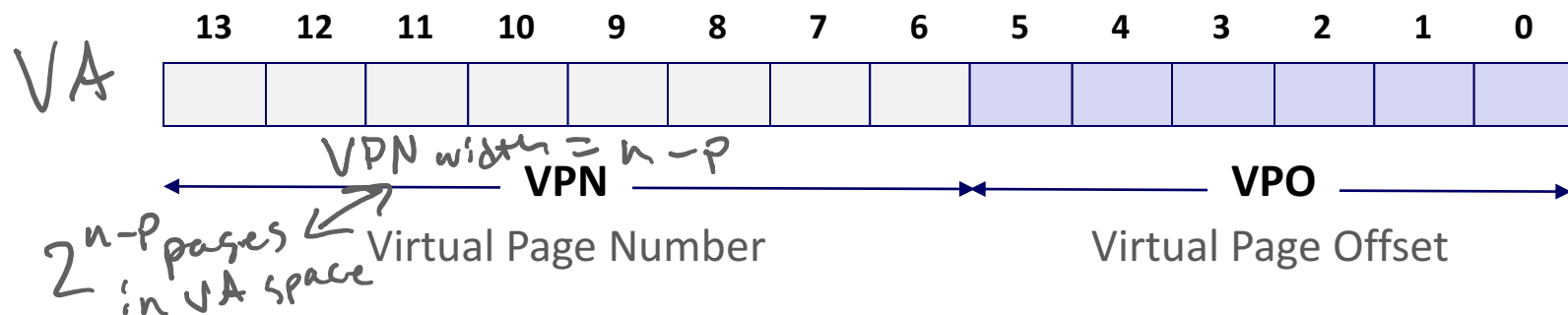
- ❖ Memory Caches (L1/L2/L3)
  - Purely a speed-up technique
  - Behavior invisible to application programmer and (mostly) OS
  - Implemented totally in hardware
- ❖ Virtual Memory
  - Supports many OS-related functions
    - Process creation, task switching, protection
  - Operating System (software)
    - 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)

# Simple Memory System Example (small)

## ❖ Addressing

- 14-bit virtual addresses  $n=14$
- 12-bit physical address  $m=12$
- Page size = 64 bytes  $P=64$

$\leftrightarrow N = 16 \text{ K}; B \text{ VA space}$   
 $\leftrightarrow M = 4 \text{ K}; B \text{ PA space}$   
 $\leftrightarrow p = 6 \text{ bits}$



# Simple Memory System: Page Table

- ❖ Only showing first 16 entries (out of  $2^8 = 256$ ) *one for every virtual page*
- **Note:** showing 2 hex digits for PPN even though only 6 bits
- **Note:** other management bits not shown, but part of PTE

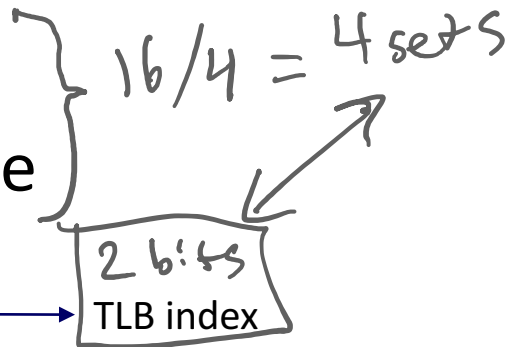
*(D, R, W, X)*

| <i>VPN</i> | <i>PPN</i> | <i>Valid</i> |
|------------|------------|--------------|
| <b>0</b>   | 28         | 1            |
| <b>1</b>   | –          | 0            |
| <b>2</b>   | 33         | 1            |
| <b>3</b>   | 02         | 1            |
| <b>4</b>   | –          | 0            |
| <b>5</b>   | 16         | 1            |
| <b>6</b>   | –          | 0            |
| <b>7</b>   | –          | 0            |

| <i>VPN</i> | <i>PPN</i> | <i>Valid</i> |
|------------|------------|--------------|
| <b>8</b>   | 13         | 1            |
| <b>9</b>   | 17         | 1            |
| <b>A</b>   | 09         | 1            |
| <b>B</b>   | –          | 0            |
| <b>C</b>   | –          | 0            |
| <b>D</b>   | 2D         | 1            |
| <b>E</b>   | –          | 0            |
| <b>F</b>   | 0D         | 1            |

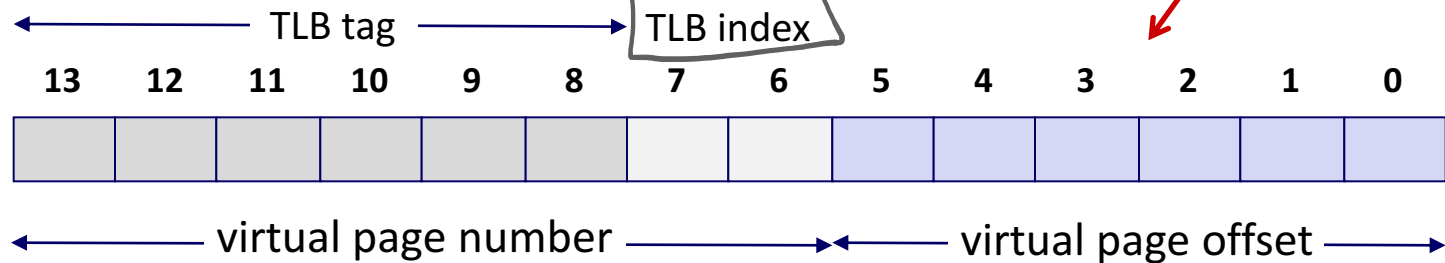
# Simple Memory System: TLB

- ❖ 16 entries total
- ❖ 4-way set associative



Why does the TLB ignore the page offset?

VA



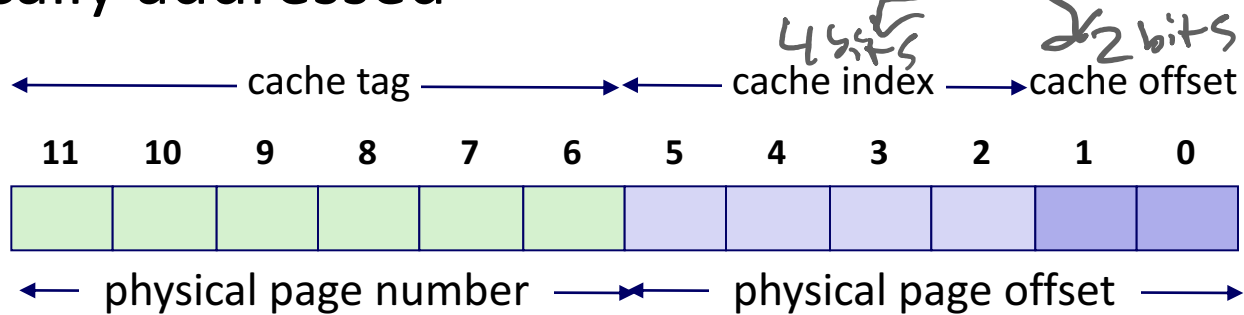
Handwritten labels for the TLB table: way (block) 0 ; way (block) 1 ; way (block) 2 ; way (block) 3

| 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  $K = 4 \text{ B}$ ,  $C/K = 16$
- ❖ Physically addressed



PA

| 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

*Circled #s refer to memory request example #*

## TLB:

| Set | Tag             | PPN | V              | Tag             | PPN | V              | Tag | PPN | V              | Tag | PPN | V              |
|-----|-----------------|-----|----------------|-----------------|-----|----------------|-----|-----|----------------|-----|-----|----------------|
| ③ 0 | 03              | -   | 0              | 09              | 0D  | 1              | 00  | -   | 0 <del>X</del> | 07  | 02  | 1              |
| ④ 1 | 03 <sup>✓</sup> | 2D  | 1 <sup>✓</sup> | 02              | -   | 0              | 04  | -   | 0              | 0A  | -   | 0              |
| ② 2 | 02              | -   | 0              | 08              | -   | 0              | 06  | -   | 0              | 03  | -   | 0 <del>X</del> |
| ① 3 | 07              | -   | 0              | 03 <sup>✓</sup> | 0D  | 1 <sup>✓</sup> | 0A  | 34  | 1              | 02  | -   | 0              |

## Page table (partial):

| VPN | PPN | V              | VPN | PPN | V              |
|-----|-----|----------------|-----|-----|----------------|
| ③ 0 | 28  | 1 <sup>✓</sup> | 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              | ② E | -   | 0 <del>X</del> |
| 7   | -   | 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 |
| ① 5   | 0D <sup>✓</sup> | 1 <sup>✓</sup> | 36 | 72 | F0 | 1D |
| 6     | 31              | 0              | -  | -  | -  | -  |
| 7     | 16              | 1              | 11 | C2 | DF | 03 |

| Index | Tag             | V              | B0 | B1 | B2 | B3 |
|-------|-----------------|----------------|----|----|----|----|
| ③ 8   | 24 <del>X</del> | 1 <sup>✓</sup> | 3A | 00 | 51 | 89 |
| 9     | 2D              | 0              | -  | -  | -  | -  |
| ④ A   | 2D <sup>✓</sup> | 1 <sup>✓</sup> | 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              | -  | -  | -  | -  |

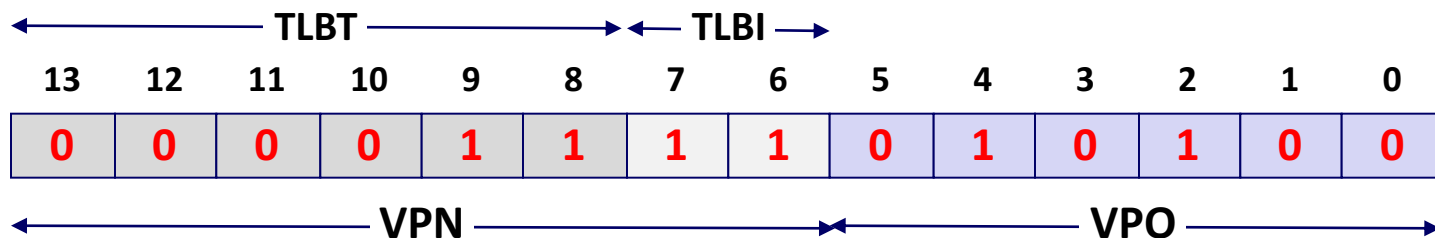


# Polling Question [VM III]

## Memory Request Example #1

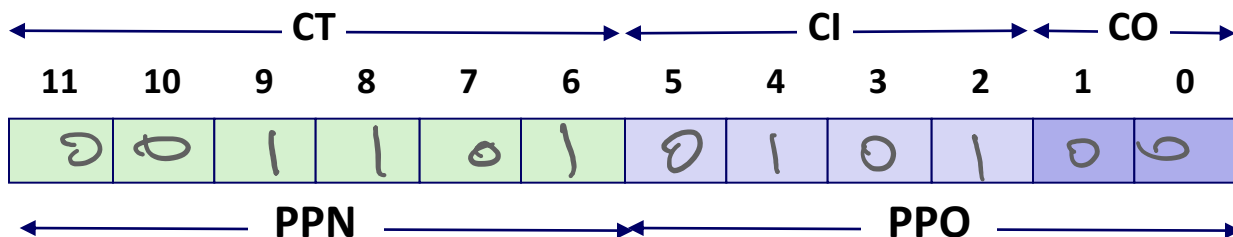
**Note:** It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: 0x03D4



VPN 0xF *Which page table entry to check*  
 TLBT 0x3 *look for this in TLB set*  
 TLBI 3 *TLB set to look in*  
 TLB Hit? Y  
 Page Fault? N  
 PPN 0x0D

❖ Physical Address:



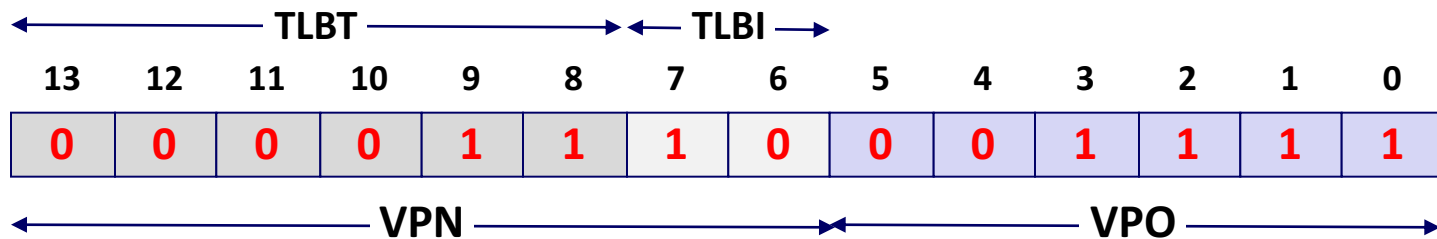
CT 0x0D *check this in cache set*  
 CI 5 *cache set to look in*  
 CO 0 *block offset of data*  
 Cache Hit? Y  
 Data (byte) 0x36

Give your answer for Data(byte) at: <http://pollev.com/pbjones>

# Memory Request Example #2

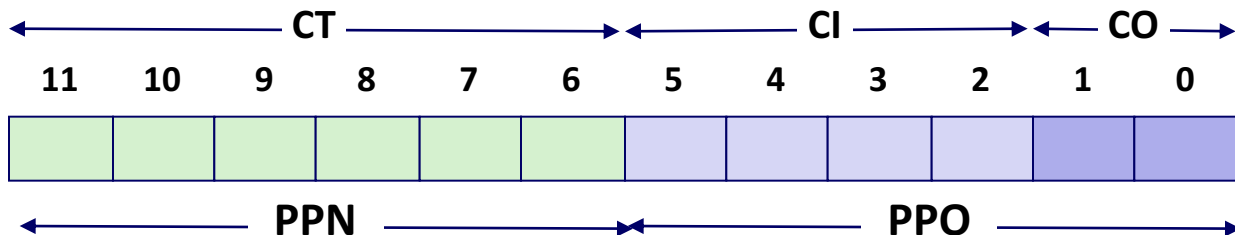
**Note:** It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: 0x038F



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

❖ Physical Address:

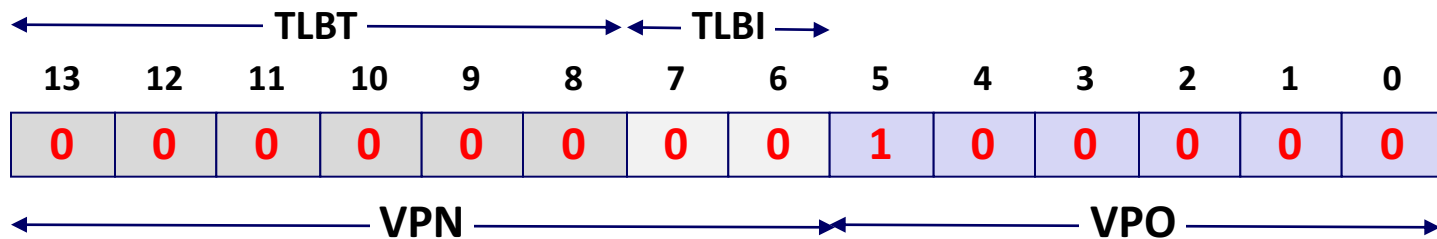


CT n/a CI n/a CO n/a Cache Hit? n/a Data (byte) n/a

# Memory Request Example #3

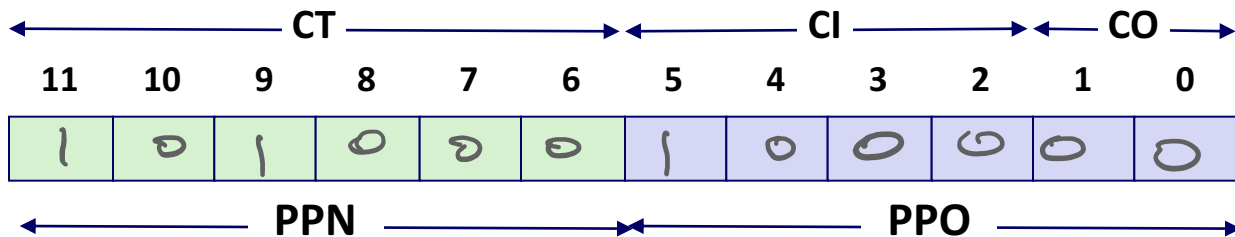
**Note:** It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: 0x0020



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

❖ Physical Address:

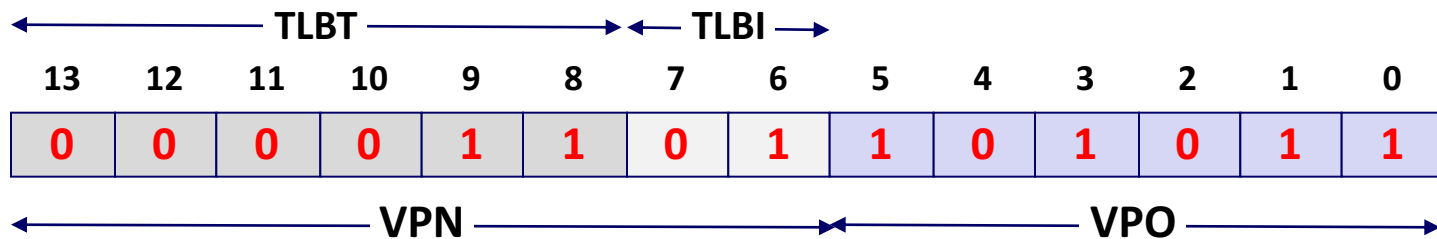


CT 0x28 CI 8 CO 0 Cache Hit? N Data (byte) n/a

# Memory Request Example #4

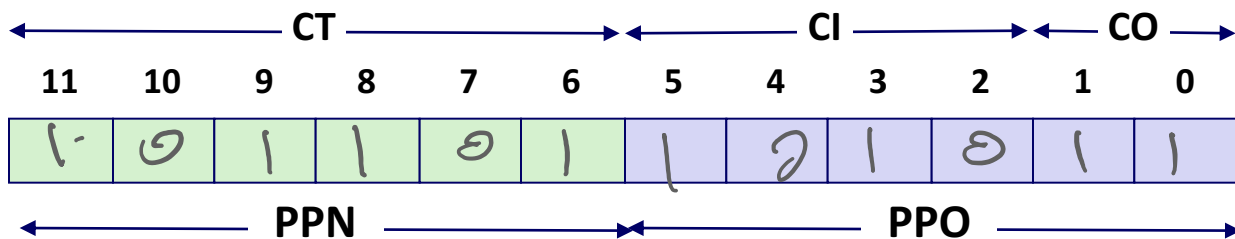
**Note:** It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: 0x036B



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

❖ Physical Address:



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

# 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

1 set

a) Fill in the following blanks:

$$\frac{2^{17}}{2^{n-p}} \leftarrow \begin{matrix} \text{Entries in a page table} \\ \# \text{ of virtual pages} \end{matrix}$$

$$\frac{20}{\text{Minimum bit-width of PTBR physical address}}$$

$$\frac{17}{\text{TLBT bits}} \rightarrow \text{VPN} \rightarrow \text{TLBT/TLBI}$$

here TLBI = 0

$$\frac{2^5}{\text{Max \# of valid entries in a page table}}$$

$2^{m-p}$

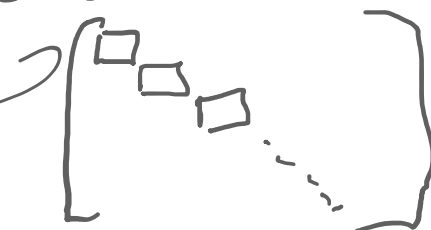
# Practice VM Question

*start address is at page offset of 0*

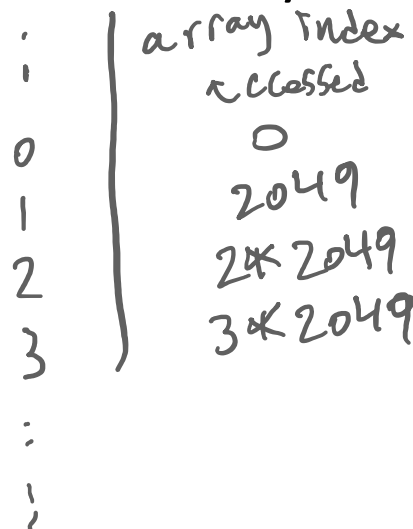
- ❖ One process uses a page-aligned square matrix `mat[]` of 32-bit integers in the code shown below:

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

*Updating diagonal entries*



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



*increasing by 2049 ints each time*

Stride is  $2049 \text{ ints} = 2049 * 4 \text{ bytes}$

# Practice VM Question

$$\text{page size} = 32 \text{ K: B} = 2^{15} \text{ B}$$

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

```
#define MAT_SIZE = 2048 211  $\text{rows} = 2^{13} \text{ B}$ 
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?

$$\underline{3/4 = 75\%} \quad \text{TLB Hit Rate}$$

access pattern: single write to index  
never revisit indices  
each row of matrix  
accessed exactly once

$$\text{page holds } 2^{15} / 2^{13} = 4 \text{ rows of matrix}$$

within each page = M H H H

$$\underline{0\%}$$

Page Table Hit Rate

only access PT on  
TLB miss, but each  
first miss to mat is  
a page table miss (since  
it starts on disk) and  
only one miss per page.

# 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  $n = 64$  bits VAs,  $p = 13$  bits pages,  $m = 33$  bits physical memory

- How many page table entries is that?

1 PTE for every virtual page

$$2^{n-p} = 2^{51} \text{ PTEs}$$

- About how long is each PTE?

PPNwidth + management bits =  $20 + 5 = 25$  bits  $\approx 3$  bytes  
 $m-p$  (V,D,R,W,X)

$$\approx 2^{52} + 2^{51} \text{ bytes per page table!}$$

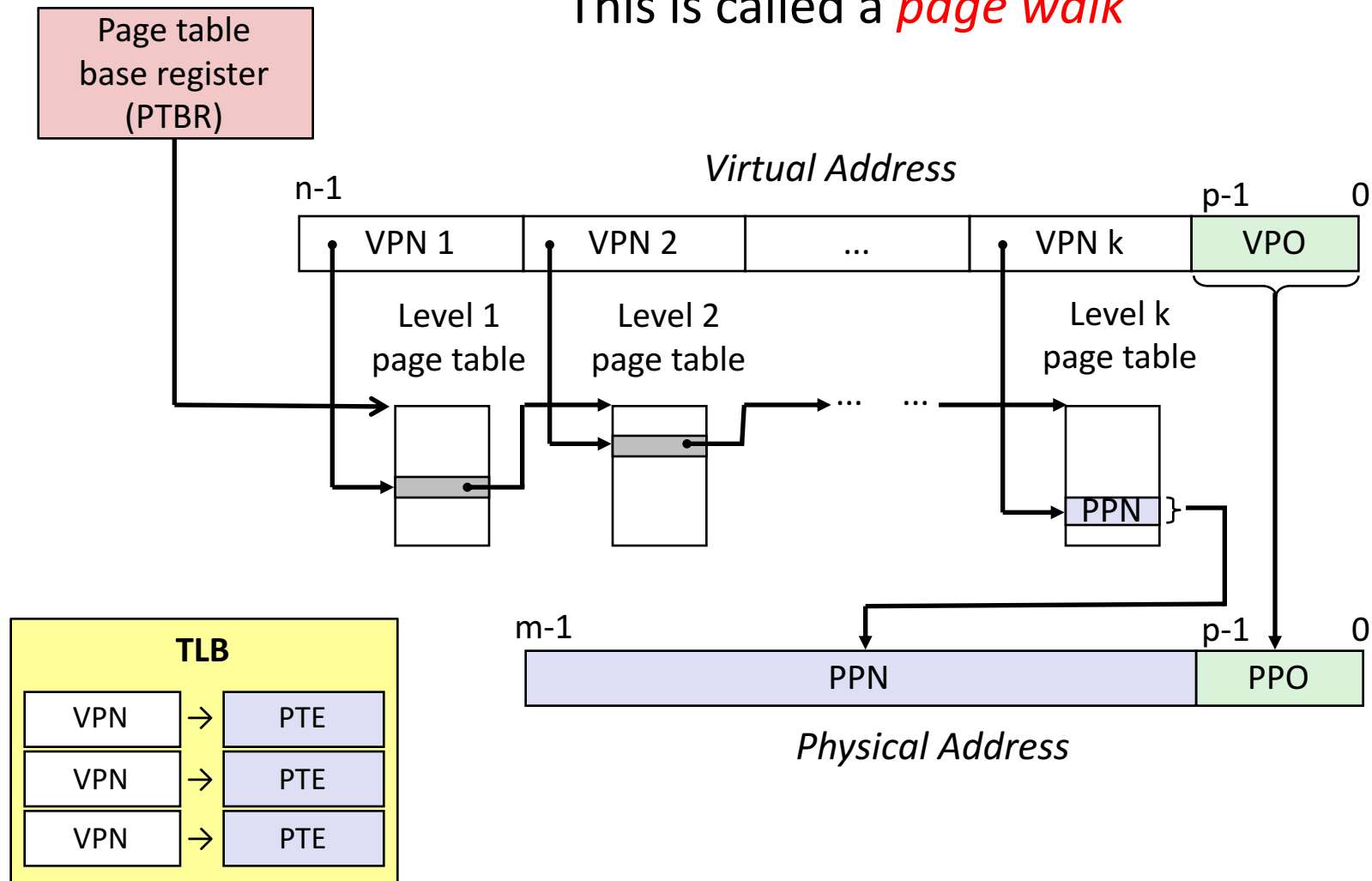
- **Moral:** Cannot use this naïve implementation of the virtual  $\rightarrow$  physical page mapping – it's way too big



# 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  $\sim 1\text{-}100\text{KB}$
- ❖ 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

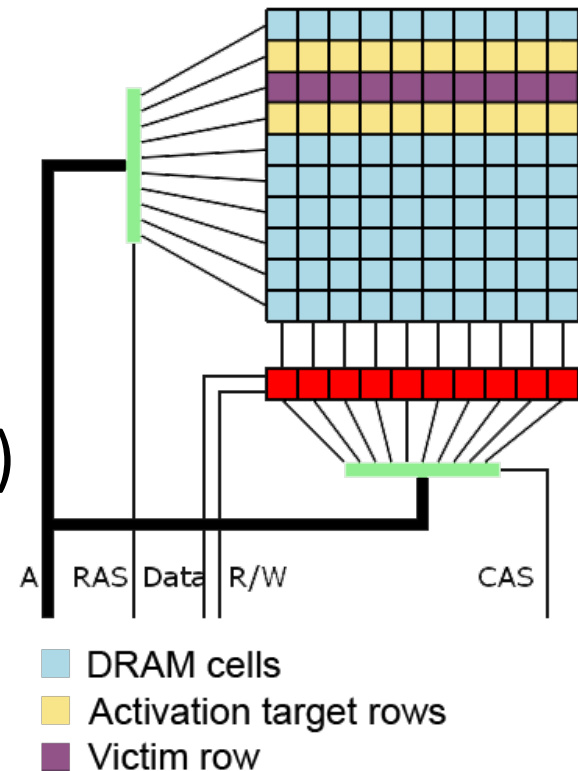
# BONUS SLIDES

## For Fun: **DRAMMER Security Attack**

- ❖ Why are we talking about this?
  - **Recent:** 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



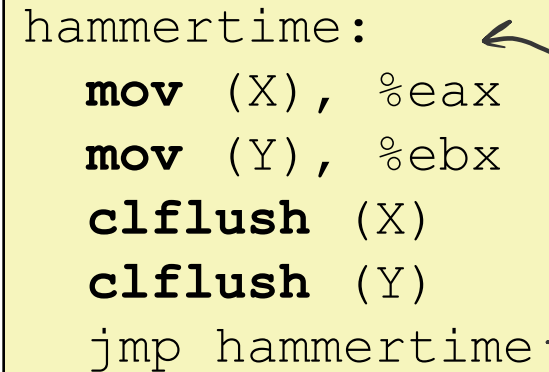
By Dsimic (modified), CC BY-SA 4.0,  
<https://commons.wikimedia.org/w/index.php?curid=38868341>

# 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

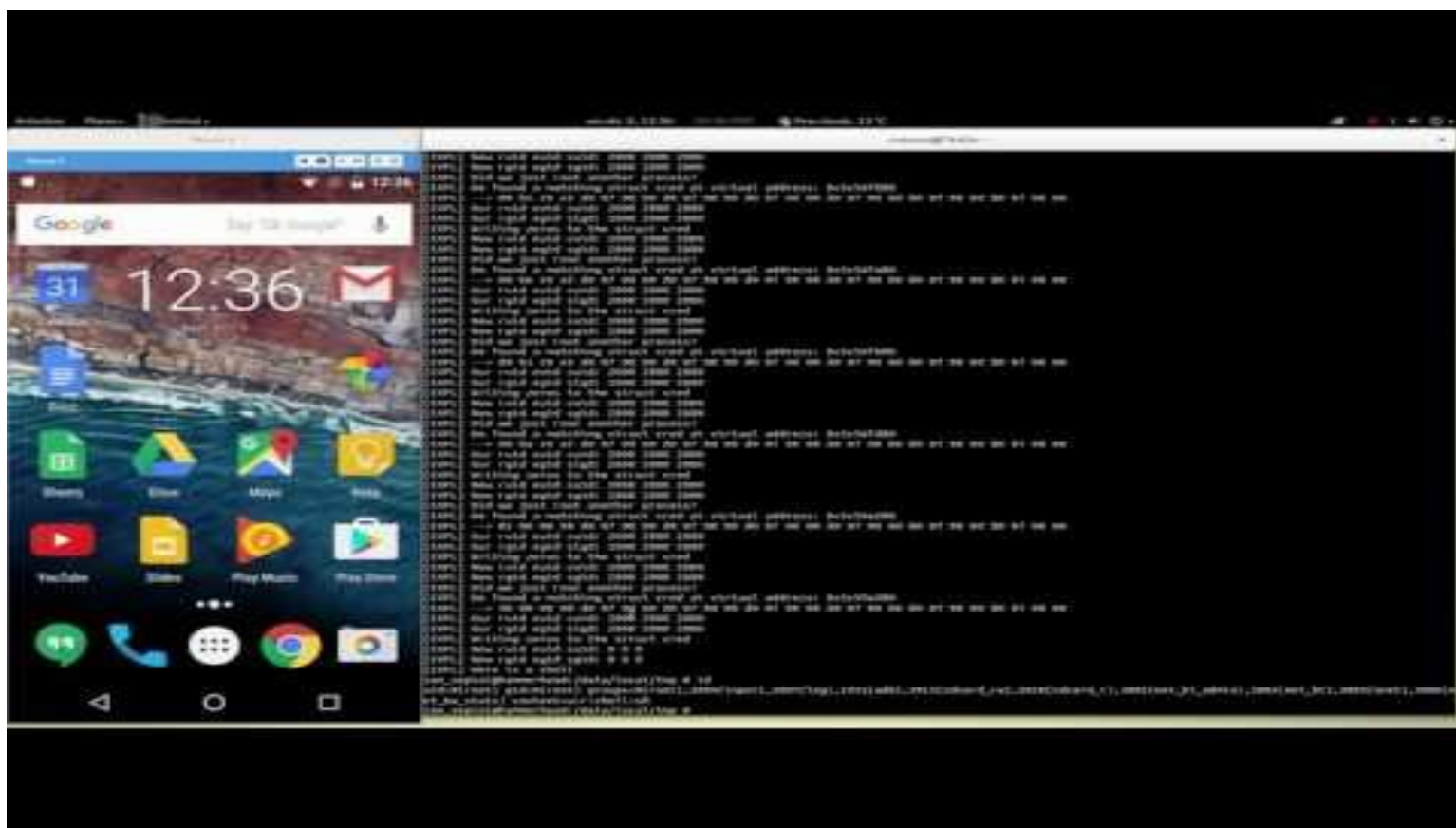
# What's DRAMMER?

- ❖ 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 glamorous, 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](https://en.wikipedia.org/wiki/Row_hammer)

# Quick Review

- ❖ What do Page Tables map?

VPN  $\rightarrow$  PPN or disk address

- ❖ Where are Page Tables located?

physical memory

- ❖ How many Page Tables are there?

one per process

- ❖ True / False: Virtual Addresses that are contiguous will always be contiguous in physical memory

Also, MMU/OS throws page fault, process just waits for data

page boundary:  $x \mid x+1$

pages can be mapped to any slot in physical mem

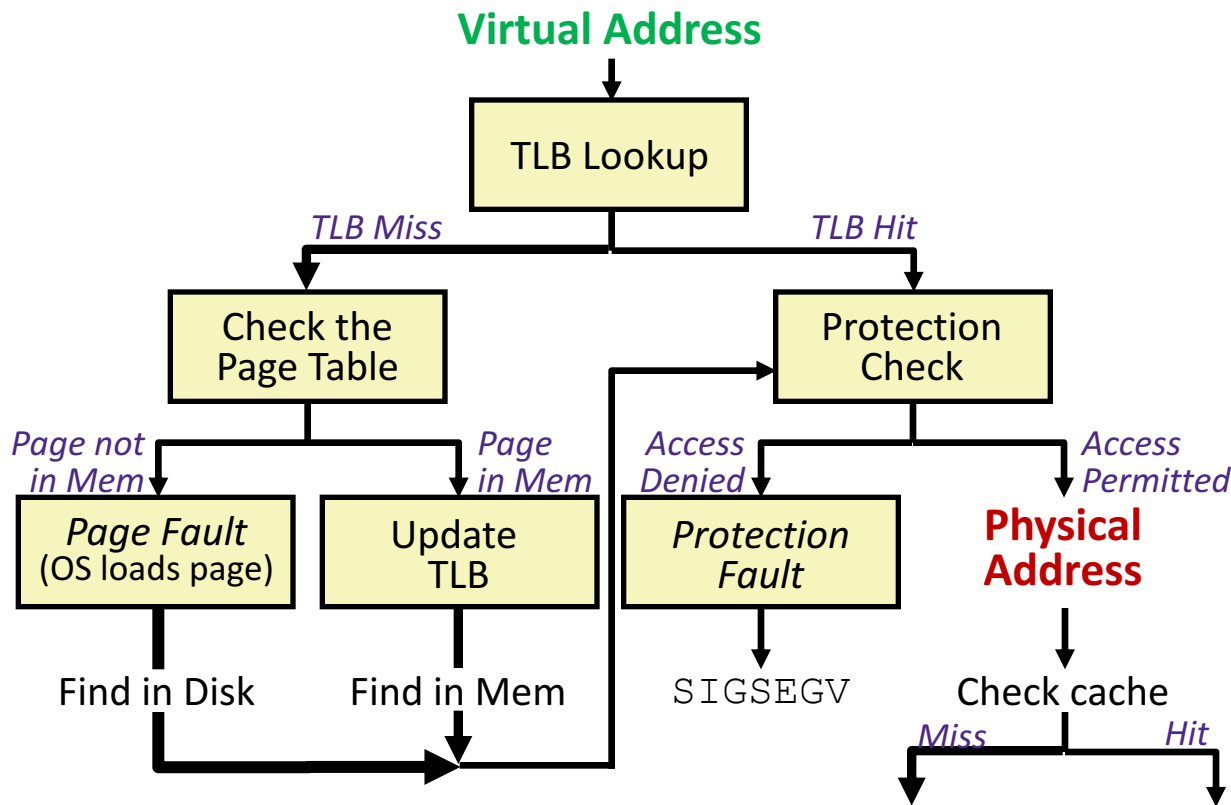
- ❖ TLB stands for translation lookaside buffer and stores translation lookaside buffer  
British for "cache"

page table entries

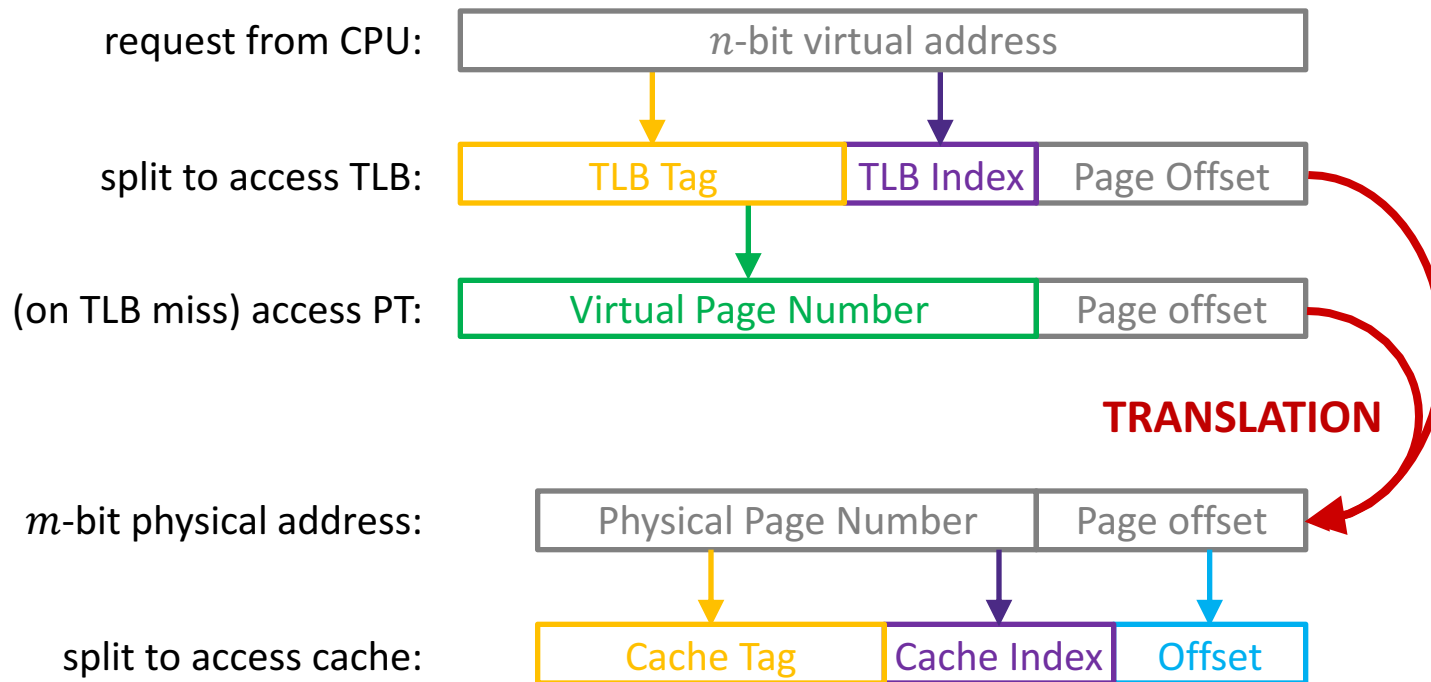
# Quick Review Answers

- ❖ What do Page Tables map?
  - VPN → PPN or disk address
- ❖ Where are Page Tables located?
  - In physical memory
- ❖ How many Page Tables are there?
  - One per process
- ❖ Can your program tell if a page fault has occurred?
  - Nope, but it has to wait a long time
- ❖ What is thrashing?
  - Constantly paging out and paging in
- ❖ True / **False**: Virtual Addresses that are contiguous will always be contiguous in physical memory
  - Could fall across a page boundary
- ❖ TLB stands for Translation Lookaside Buffer and stores page table entries

# Handouts Diagrams



# Handouts Diagrams



# 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

