

Virtual Memory II

CSE 351 Autumn 2018

Instructor:

Justin Hsia

Teaching Assistants:

Akshat Aggarwal

An Wang

Andrew Hu

Brian Dai

Britt Henderson

James Shin

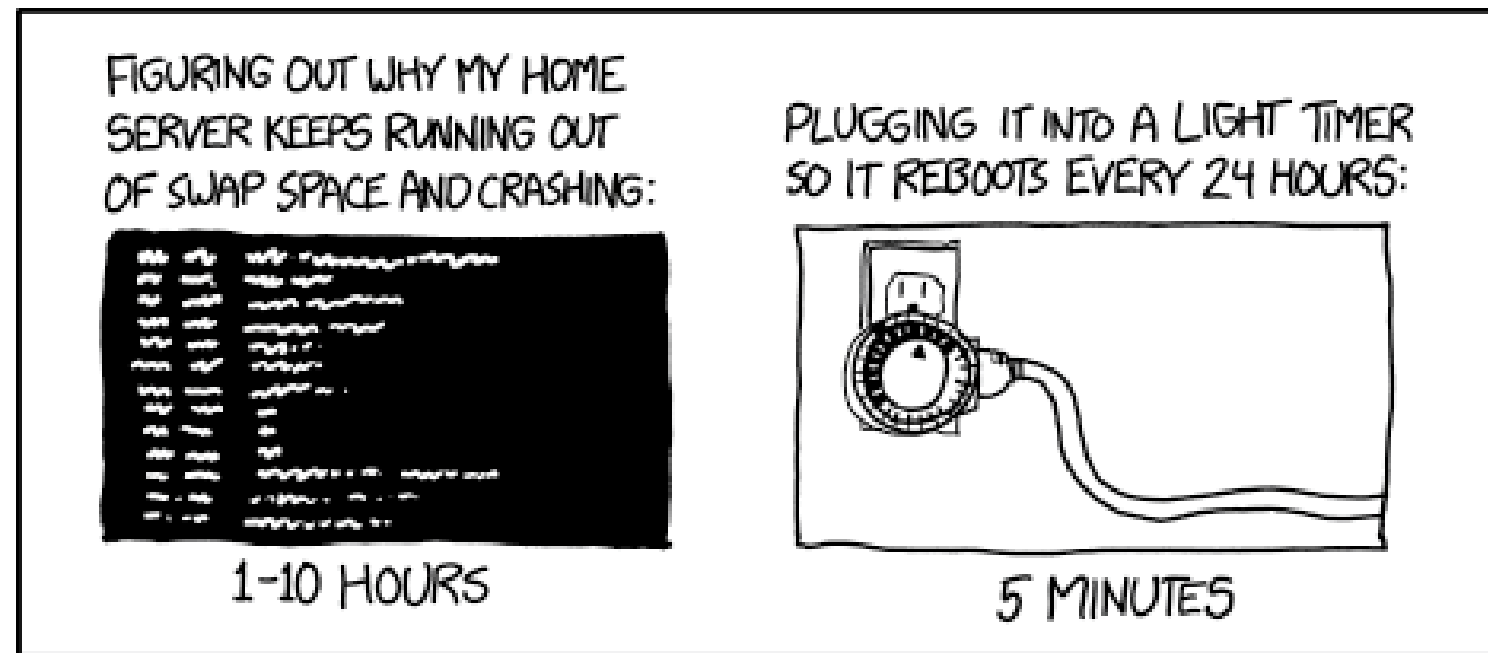
Kevin Bi

Kory Watson

Riley Germundson

Sophie Tian

Teagan Horkan



WHY EVERYTHING I HAVE IS BROKEN

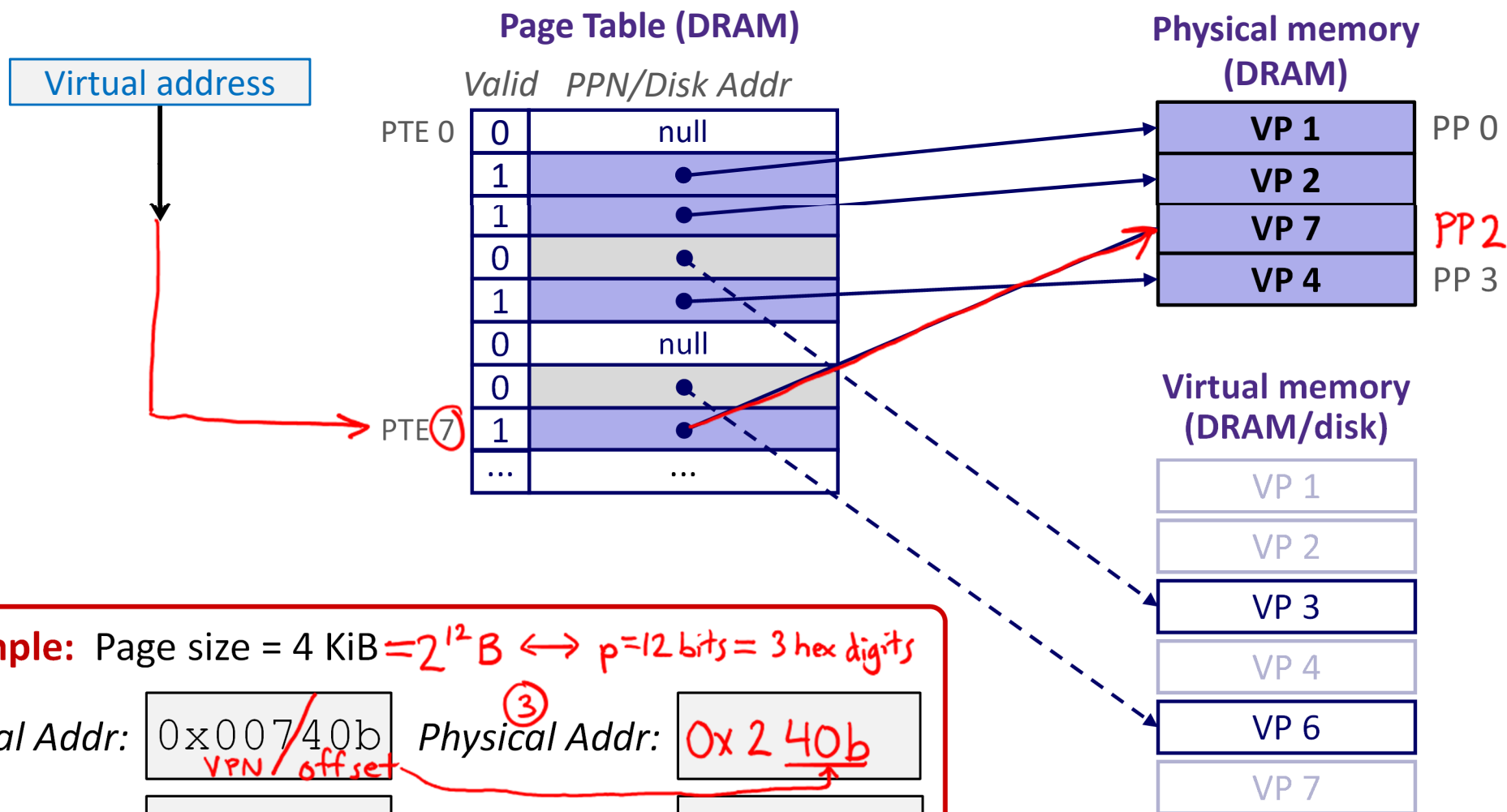
<https://xkcd.com/1495/>

Administrivia

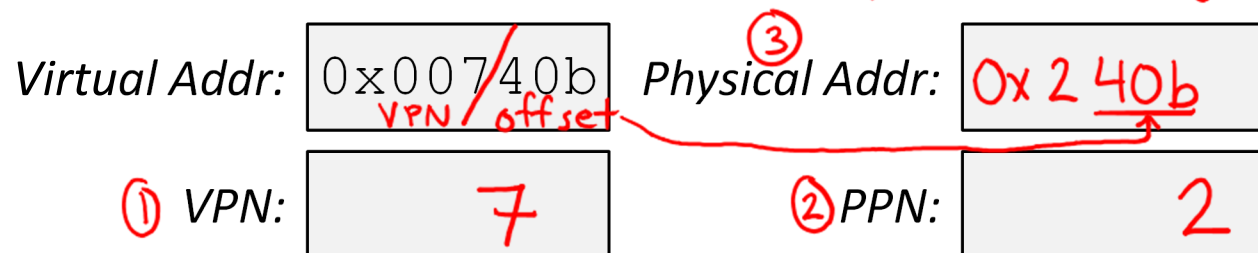
- ❖ Lab 4 due next Monday (11/26)
- ❖ Homework 5 released tomorrow (11/19)
 - Processes and Virtual Memory
- ❖ There is lecture on Wednesday
 - VM wrap-up, VM review problems, plus fun slides on a VM exploit
- ❖ “Virtual Section” on Virtual Memory
 - Worksheet and solutions released on Wednesday
 - Videos of Justin working through problems

Page Hit

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

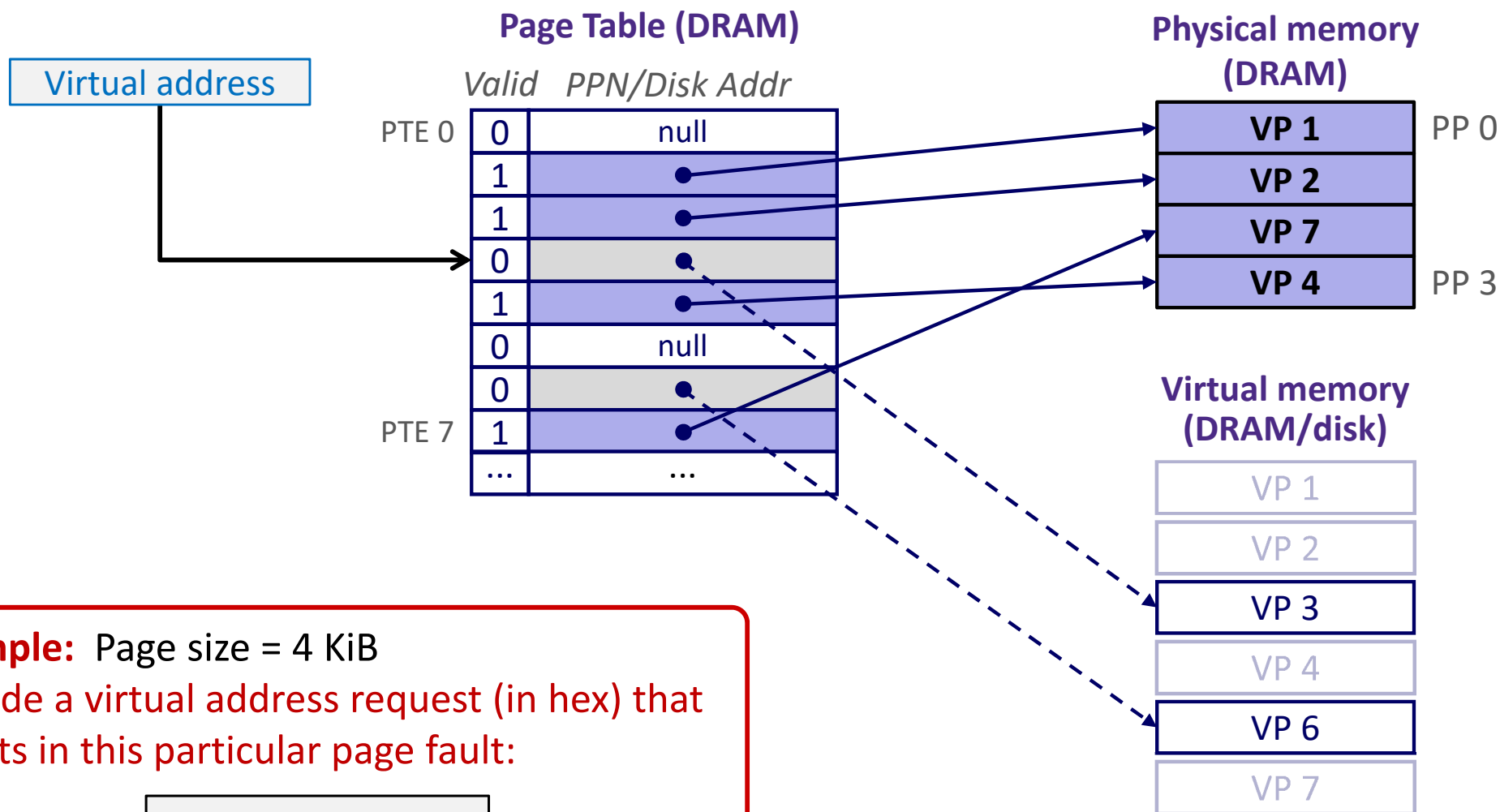


Example: Page size = 4 KiB = 2^{12} B \leftrightarrow $p=12$ bits = 3 hex digits



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:

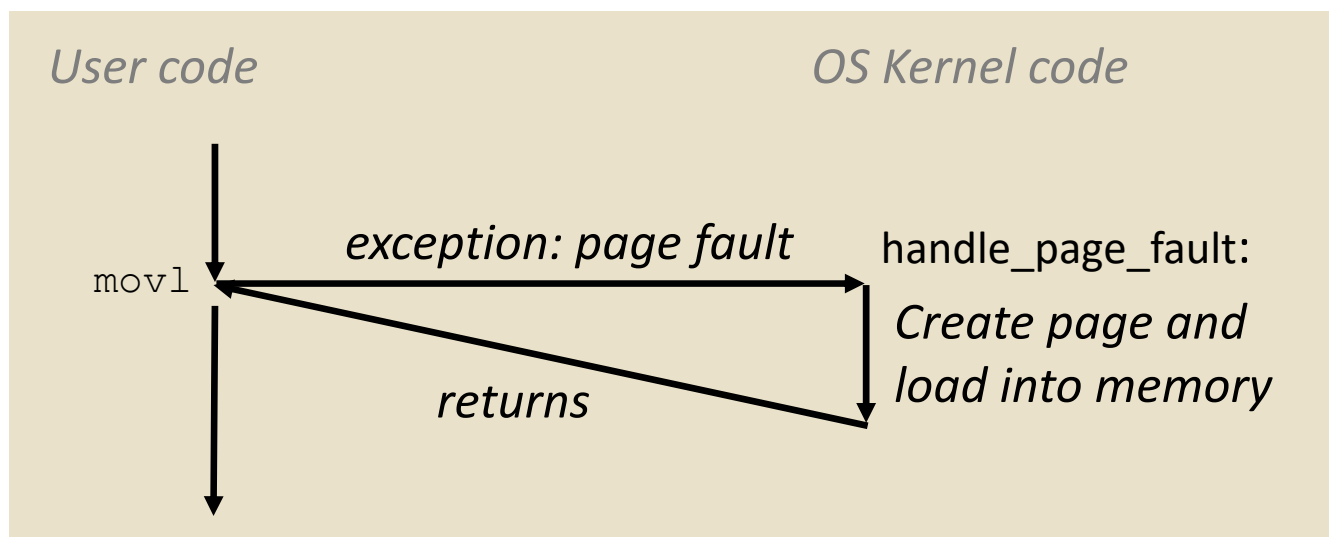
Virtual Addr:

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;
}
```

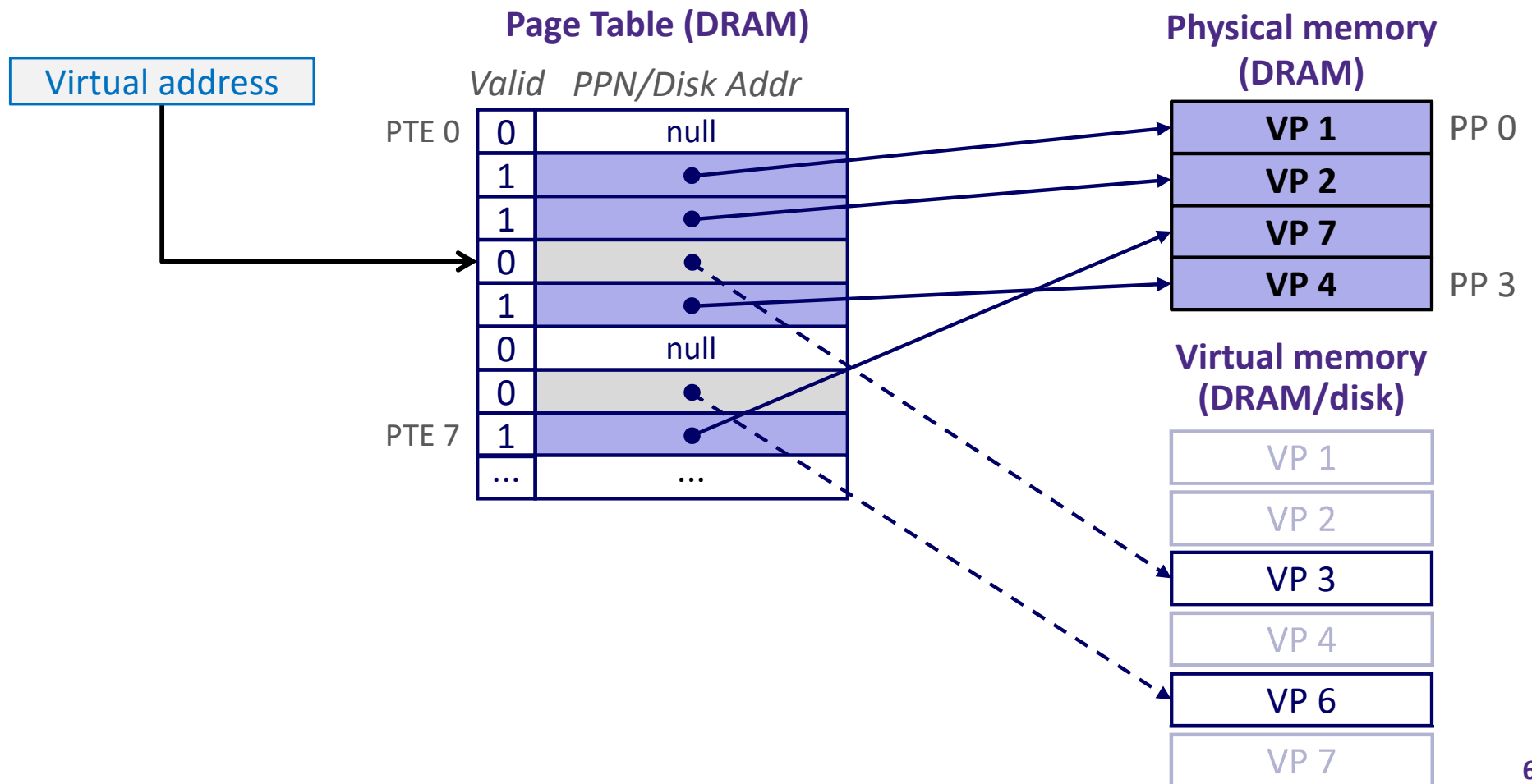
```
80483b7:      c7 05 10 9d 04 08 0d  movl   $0xd,0x8049d10
```



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

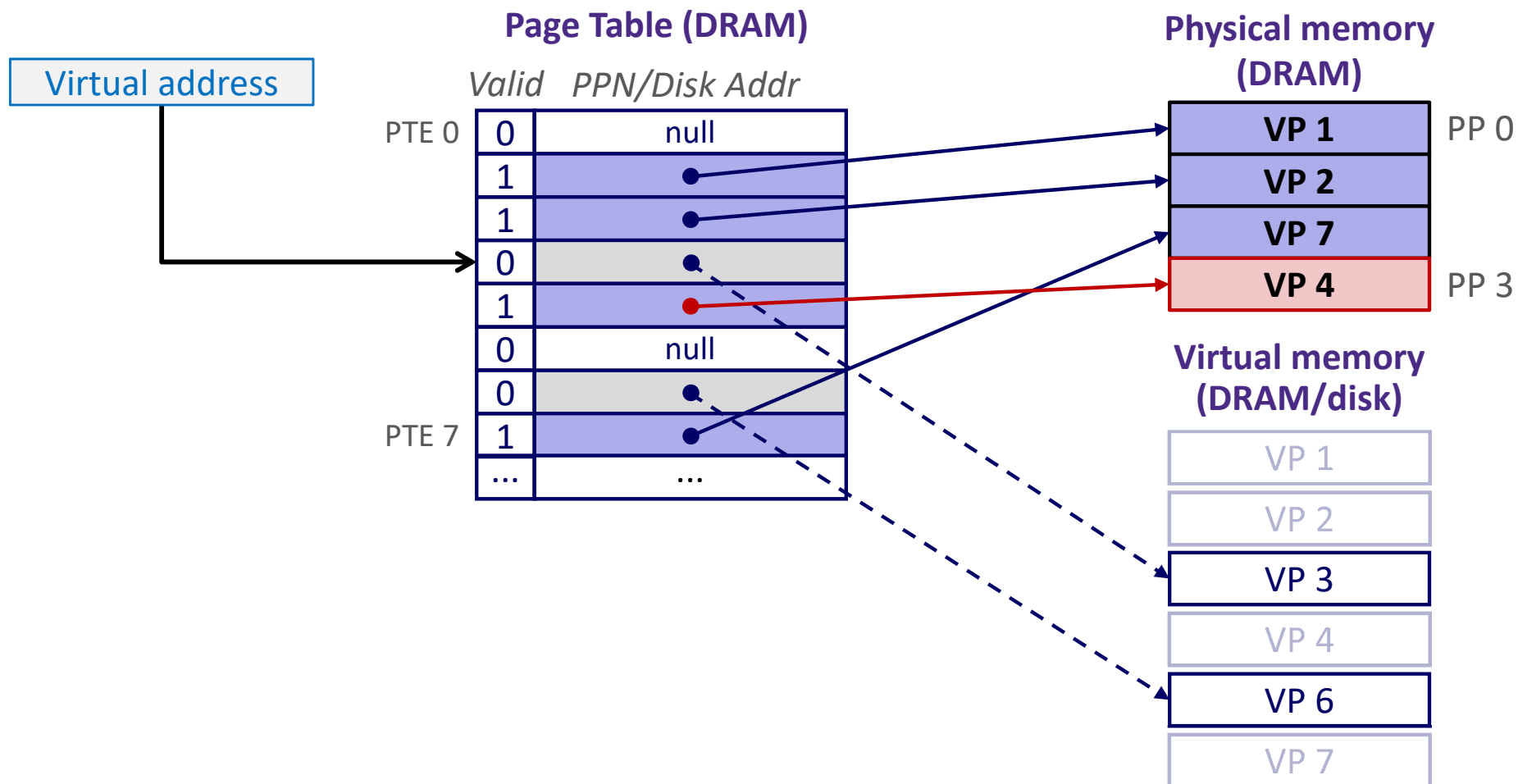
Handling a Page Fault

- ❖ Page miss causes page fault (an exception)



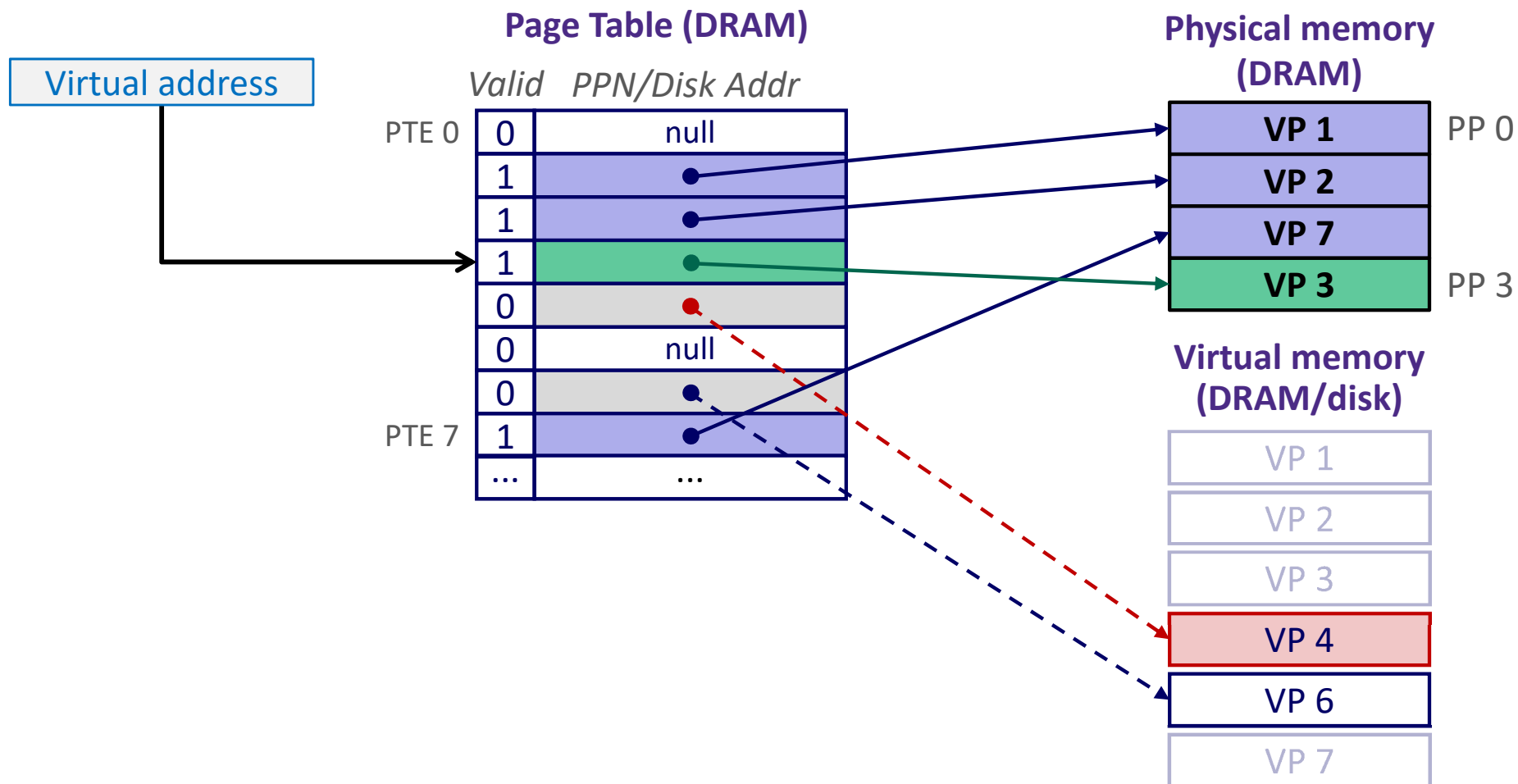
Handling a Page Fault

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



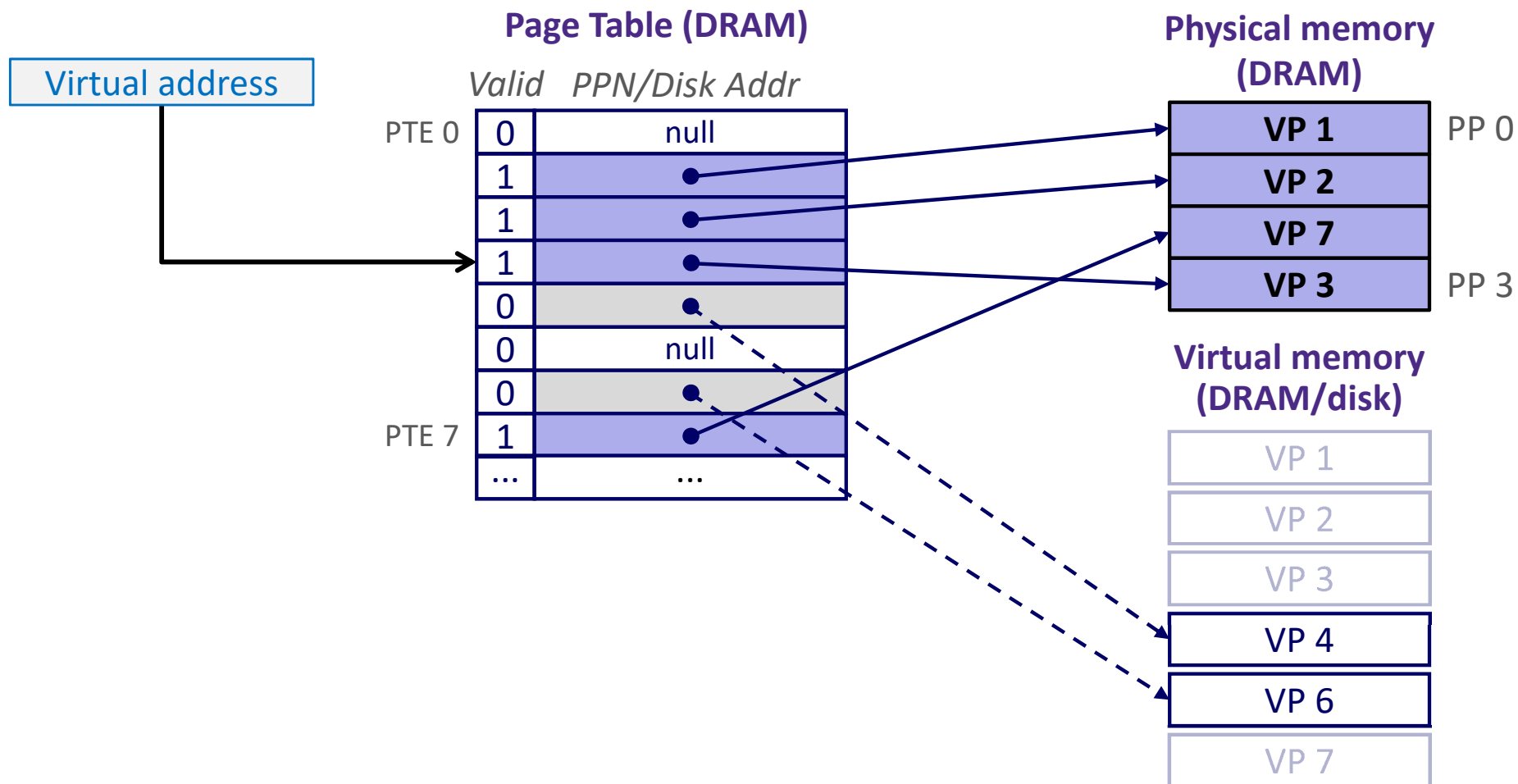
Handling a Page Fault

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



Handling a Page Fault

- ❖ 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
 - 48-bit virtual addresses
 - 16 GiB physical memory
 - Vote at: <http://PollEv.com/justinh>

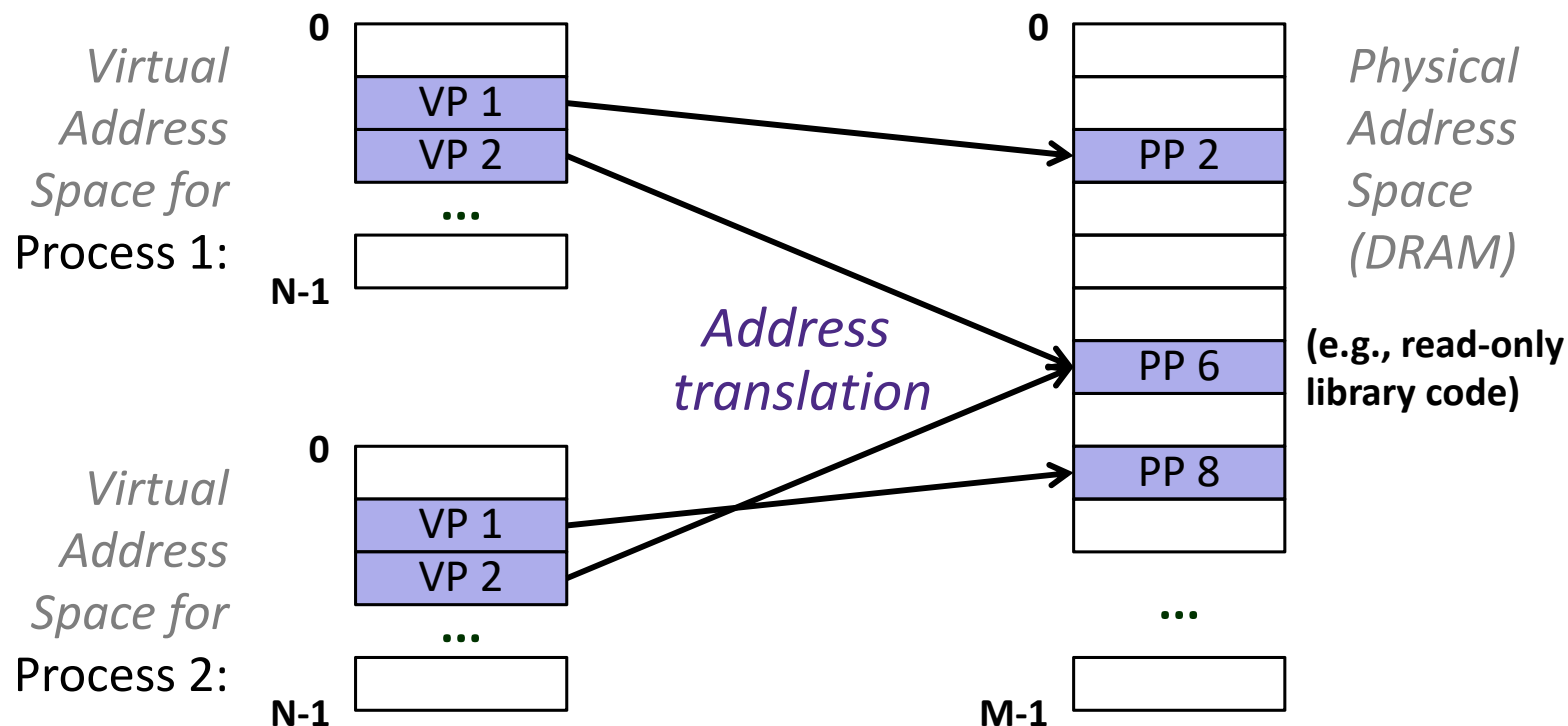
	VPN	PPN
(A)	34	24
(B)	32	18
(C)	30	20
(D)	34	20

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!



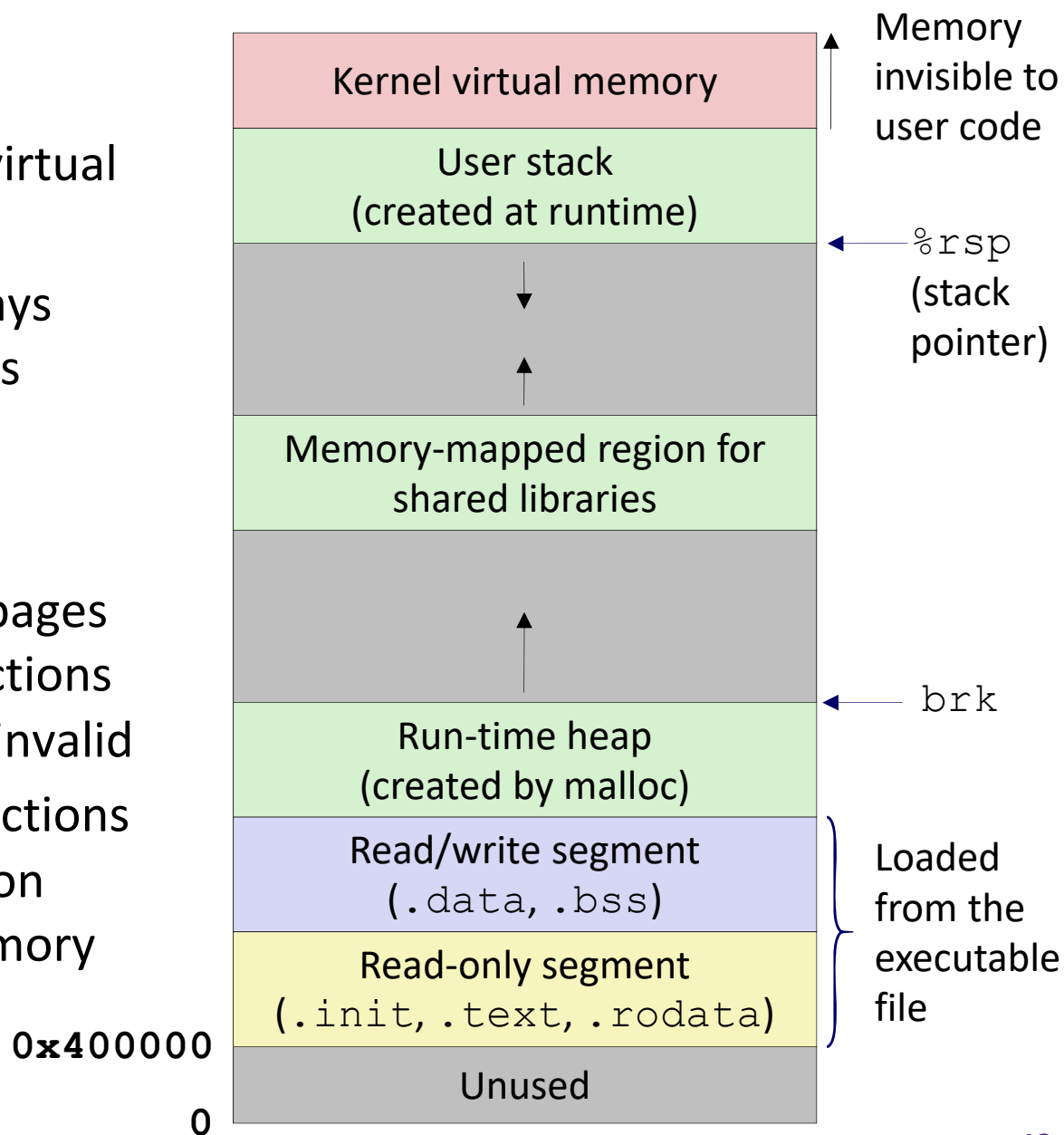
Simplifying Linking and Loading

❖ Linking

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

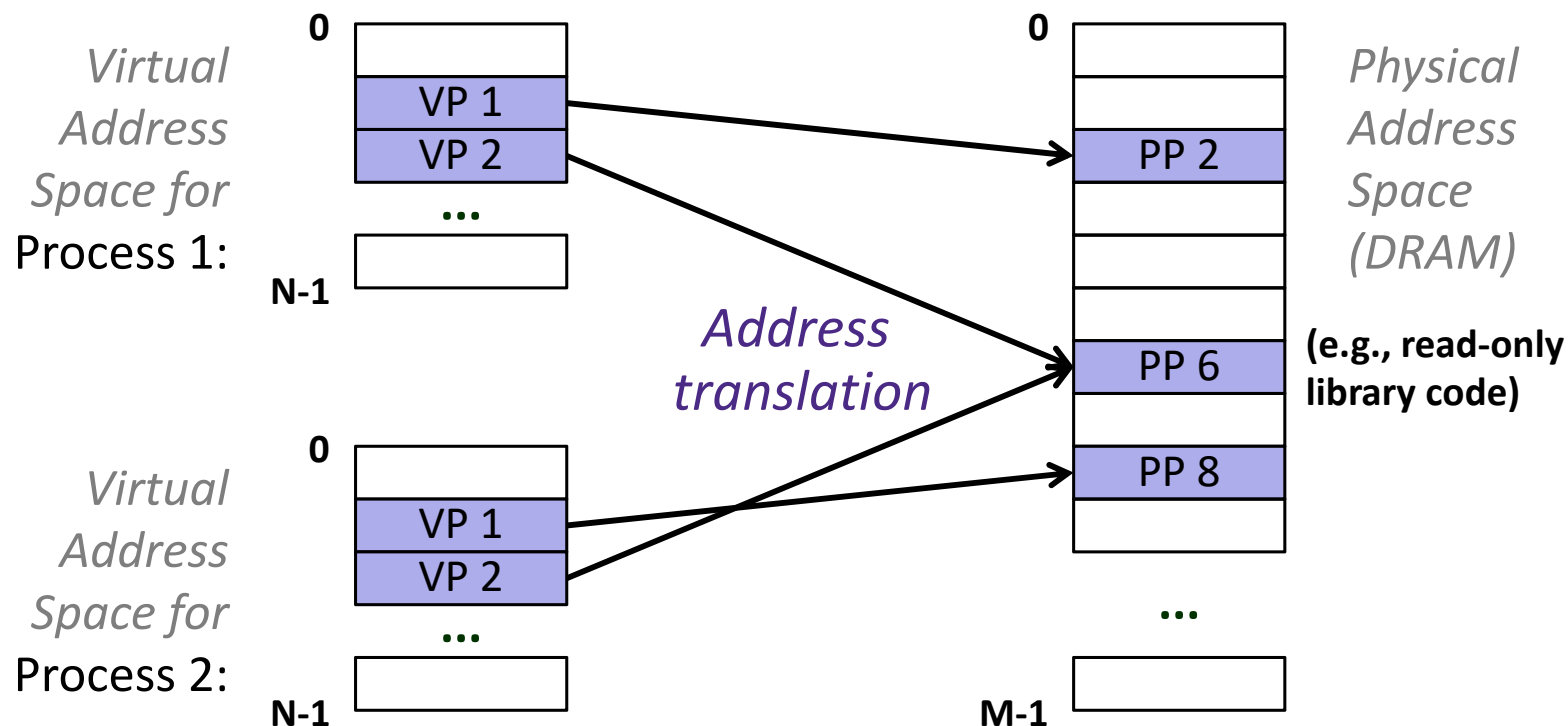
❖ Loading

- `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



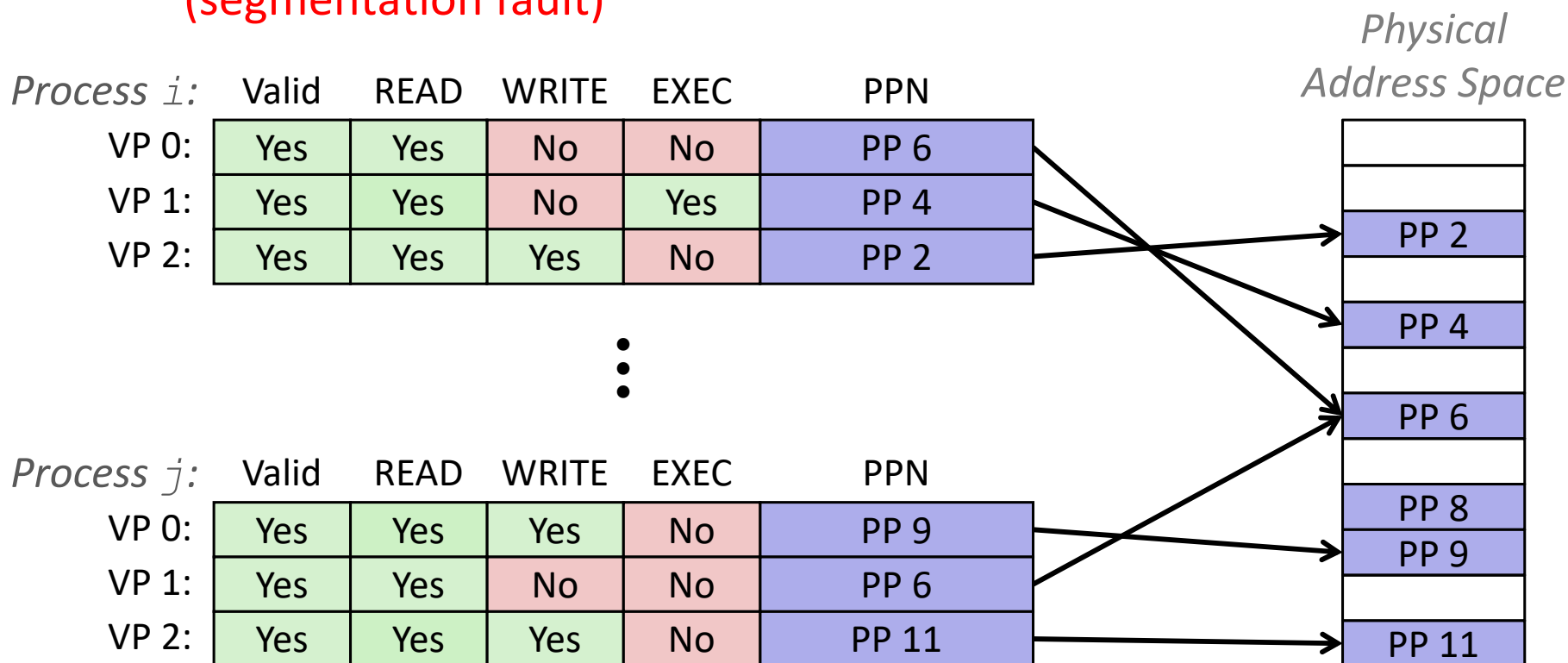
VM for Protection and Sharing

- ❖ 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)

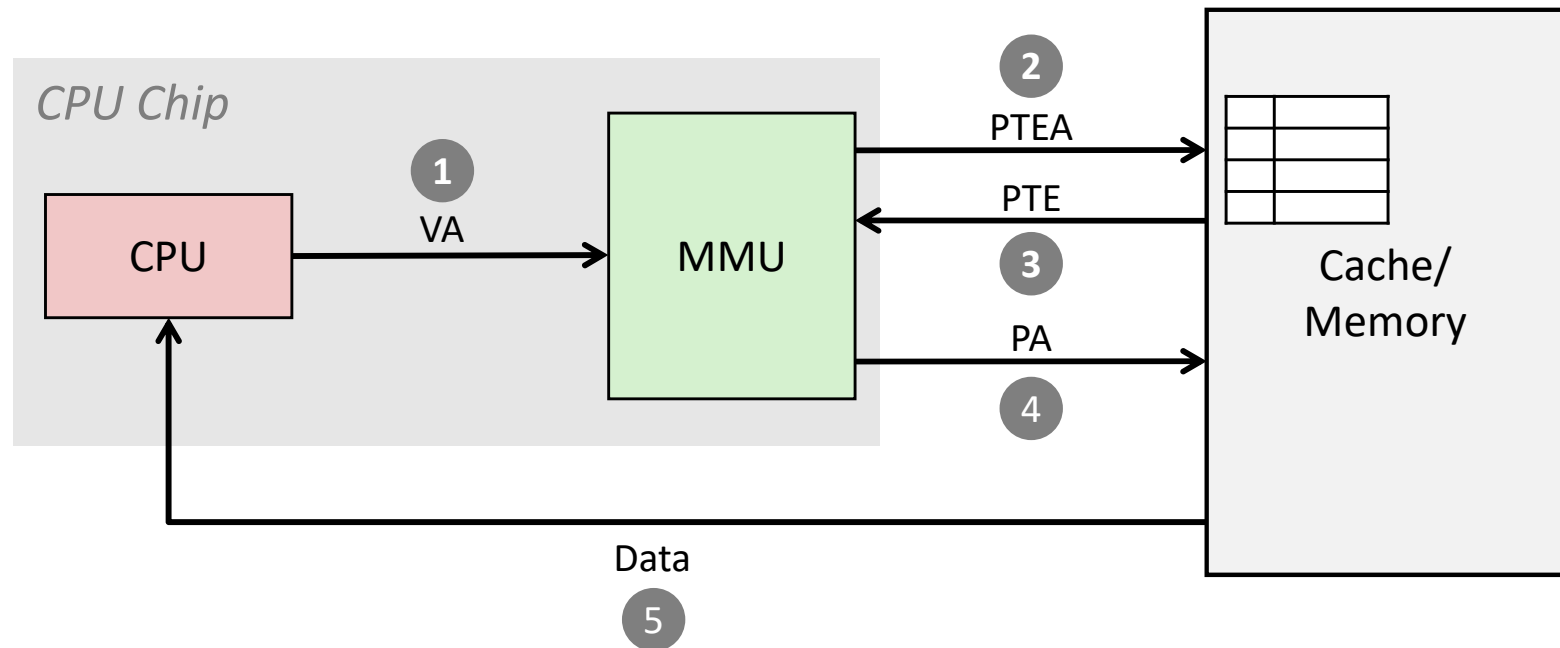


Review Question

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

Section	Read	Write	Execute
Stack			
Heap			
Static Data			
Literals			
Instructions			

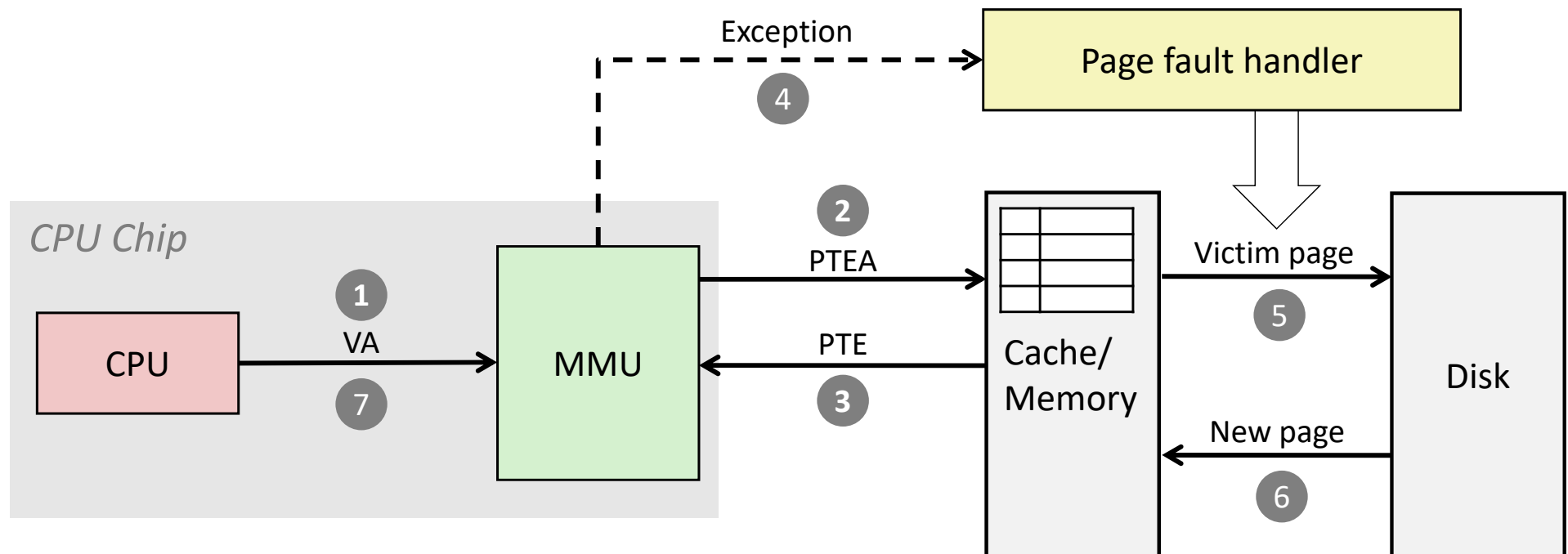
Address Translation: Page Hit



- 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



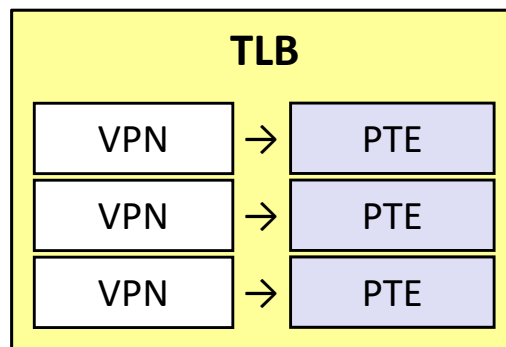
- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in cache/memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim (and, if dirty, pages it out to disk)
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns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

Hmm... Translation Sounds Slow

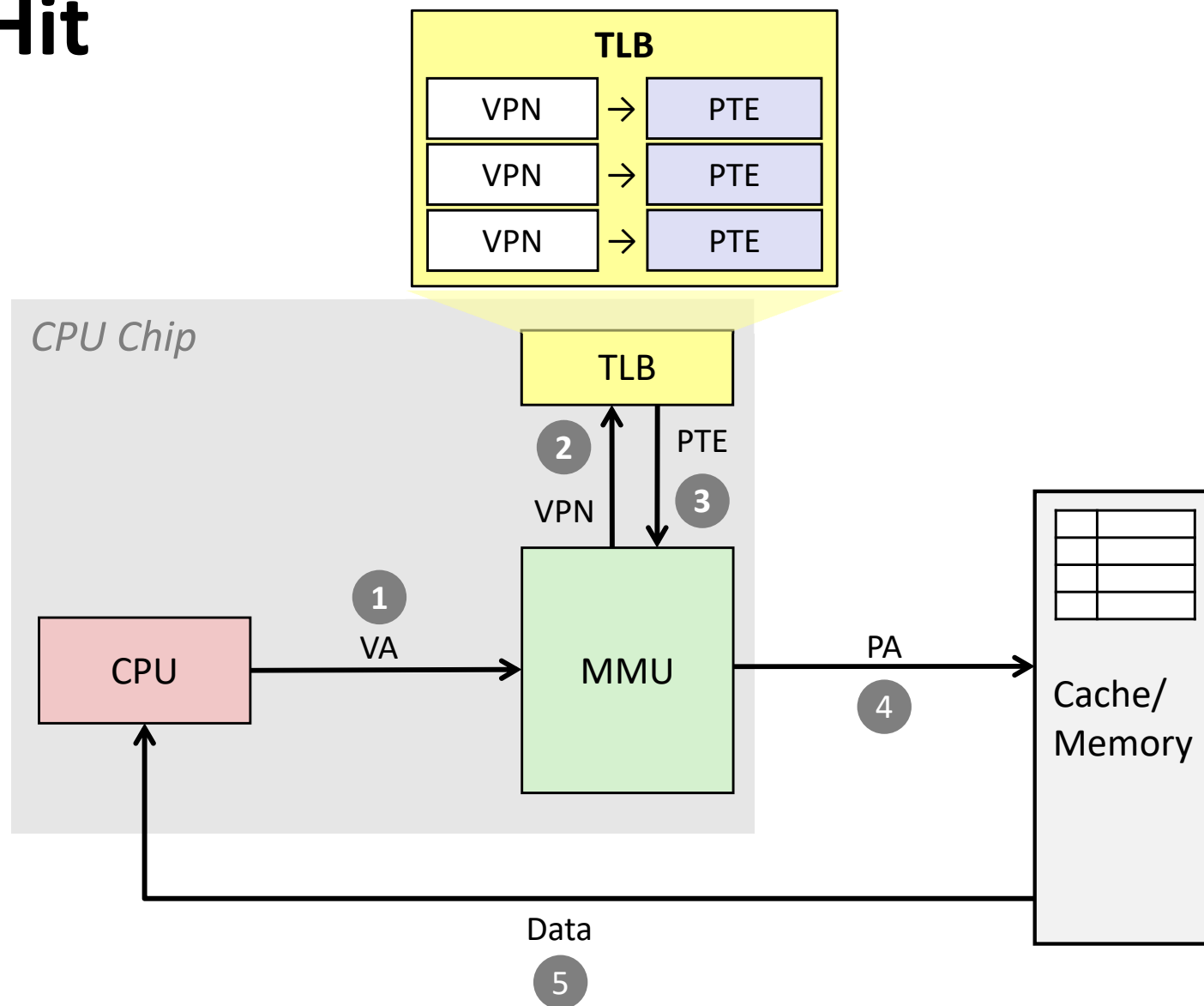
- ❖ The MMU accesses memory *twice*: once to get the PTE for translation, and then again for the actual memory request
 - The PTEs *may* be cached in L1 like any other memory word
 - But they may be evicted by other data references
 - And a hit in the L1 cache still requires 1-3 cycles
- ❖ *What can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

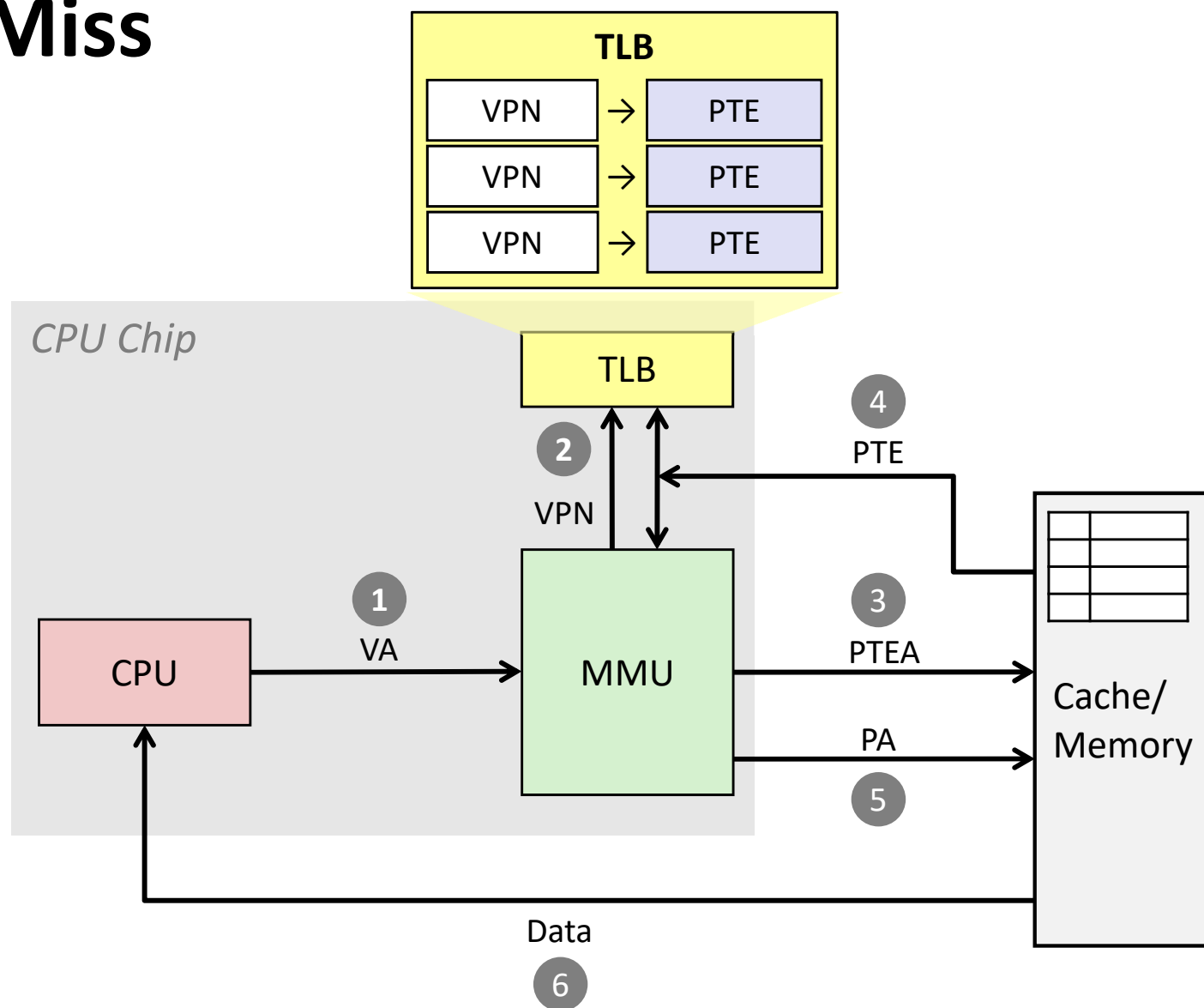


TLB Hit



❖ 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

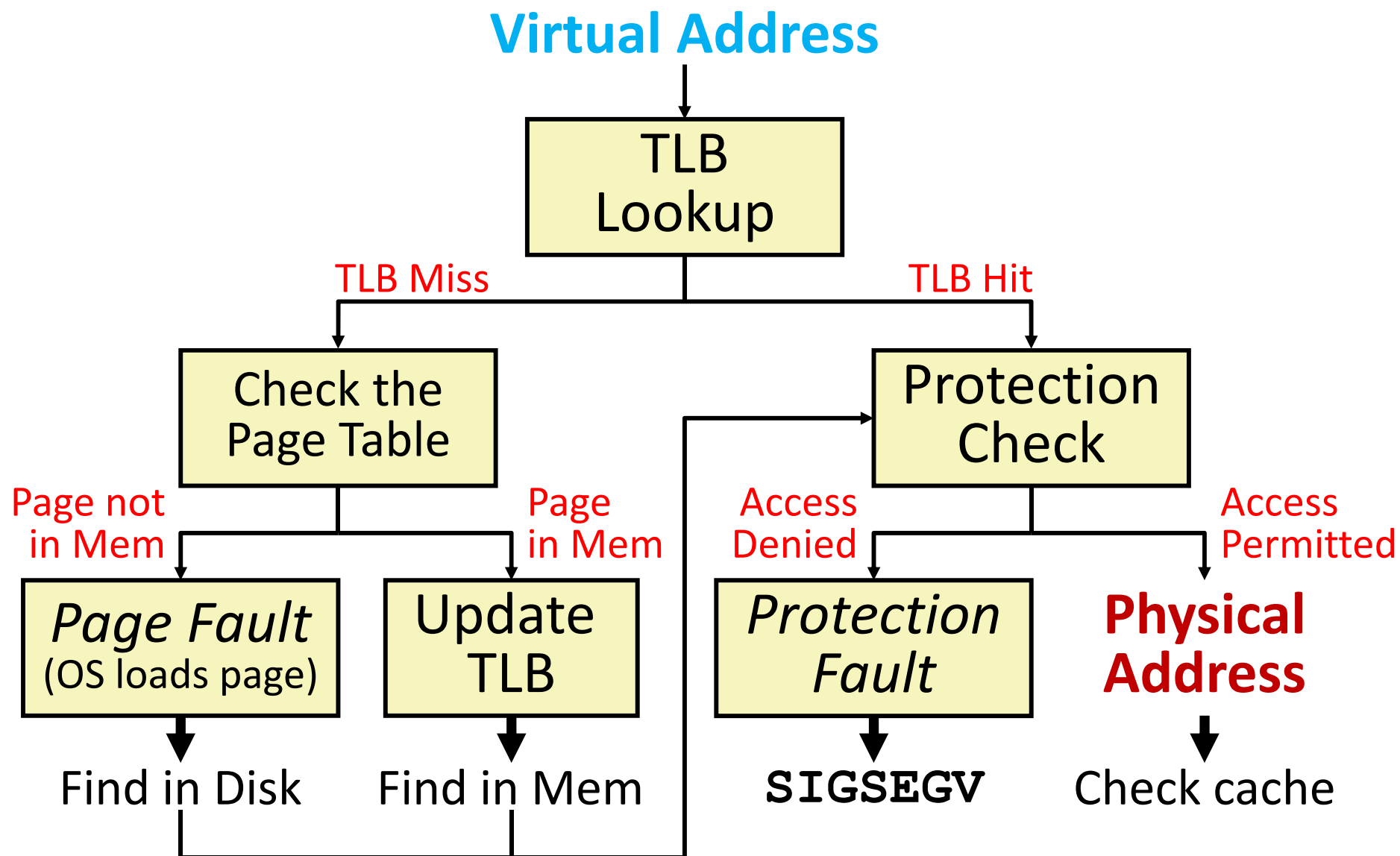
1) Check TLB

- 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

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



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

- $N = 2^n$ 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)