Virtual Memory

CSCI 237: Computer Organization
30st Lecture, Monday, November 18, 2024

Kelly Shaw

Administrative Details

- Read CSAPP 9.3-9.6 (Ch. 9 sections are short)
- Lab #5 due **Wednesday** at 11pm
- Lab #6 partner signup due Friday at 8am

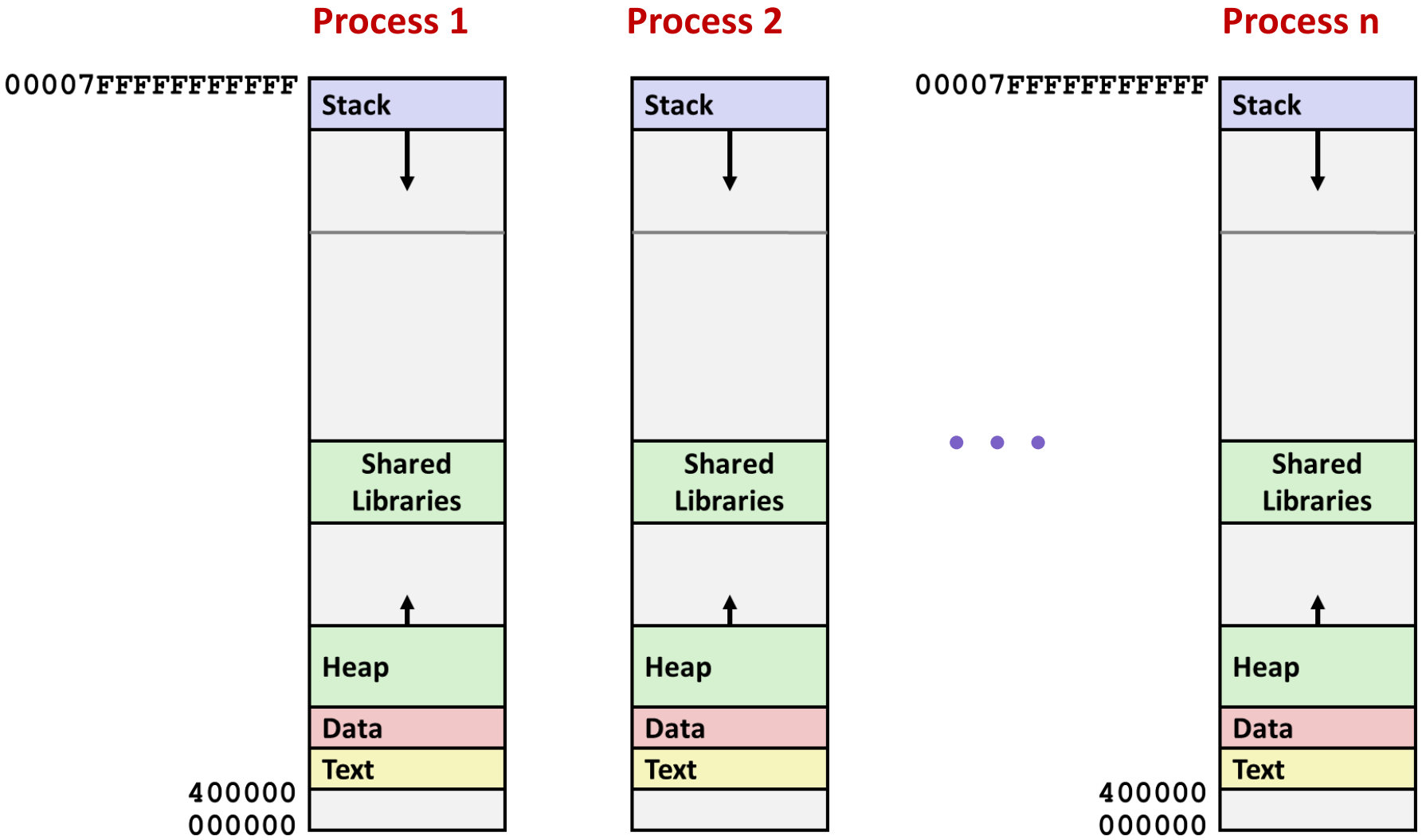
Last Time: Cache Performance

- Caches in real systems
- Performance impact of caches
  - The memory mountain

Today: Virtual Memory

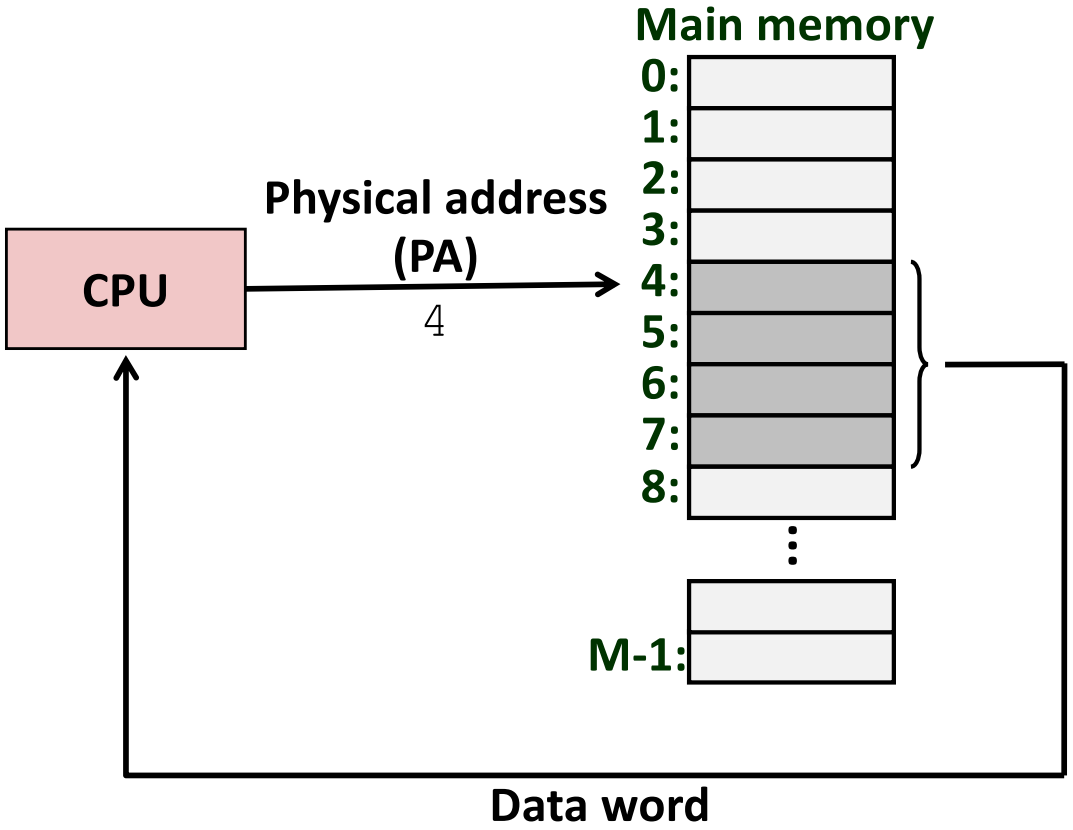
- Address spaces (Ch 9.2)
- VM as a tool for caching (Ch 9.3)
- VM as a tool for memory management (Ch 9.4)
- VM as a tool for memory protection (Ch 9.5)
- Address translation (Ch 9.6)

Question: How Does This Work?!



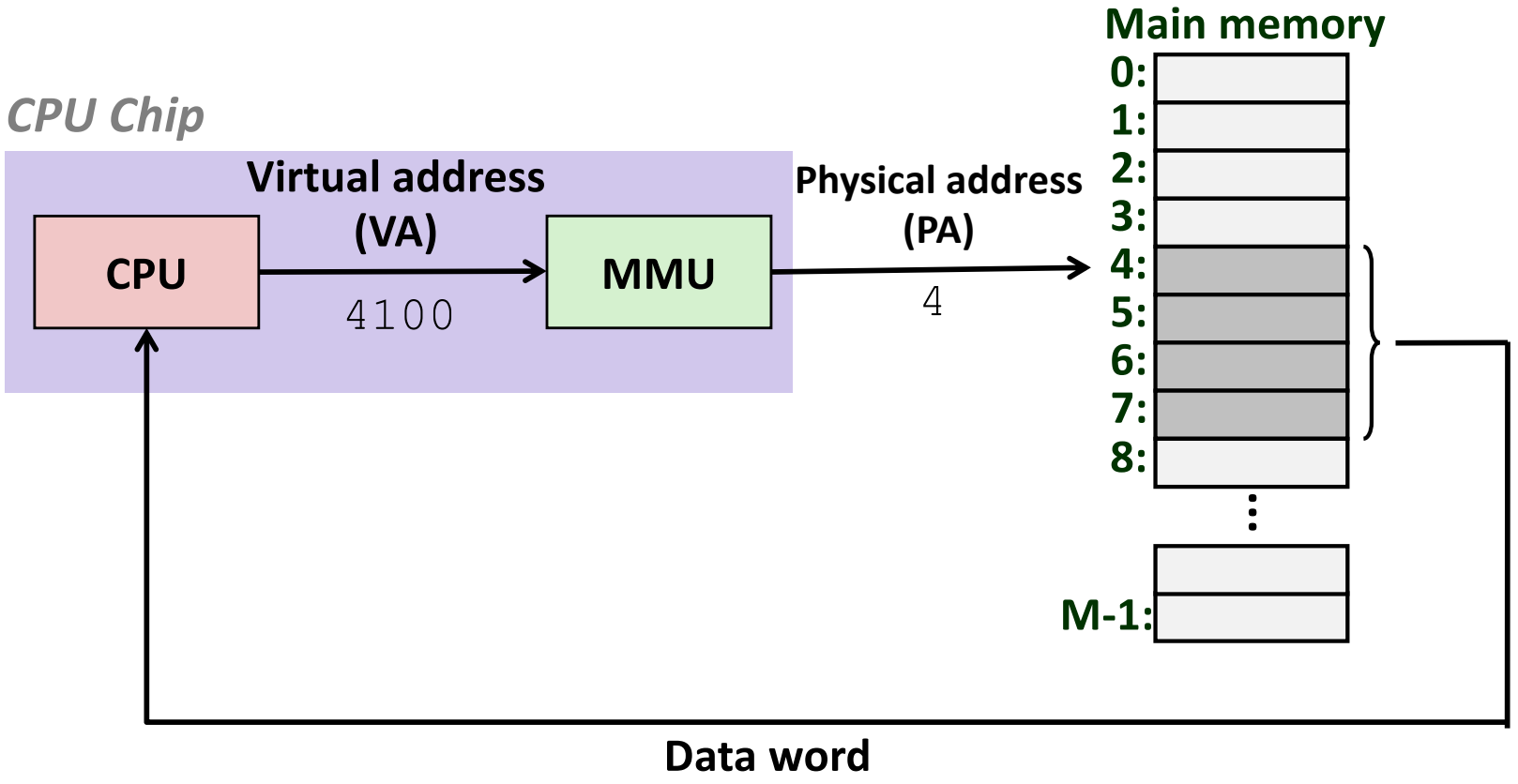
Solution: Virtual Memory

A System Using Physical Addressing



- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing



- Used in all modern servers, laptops, and smart phones
- One of the great ideas in computer science

Address Spaces: Terminology

- Linear address space: Ordered set of contiguous non-negative integer addresses (we always assume this):

  {0, 1, 2, 3 … }

- Virtual address space: Set of $N = 2^n$ virtual addresses

  {0, 1, 2, 3, …, N-1}

- Physical address space: Set of $M = 2^m$ physical addresses

  {0, 1, 2, 3, …, M-1}

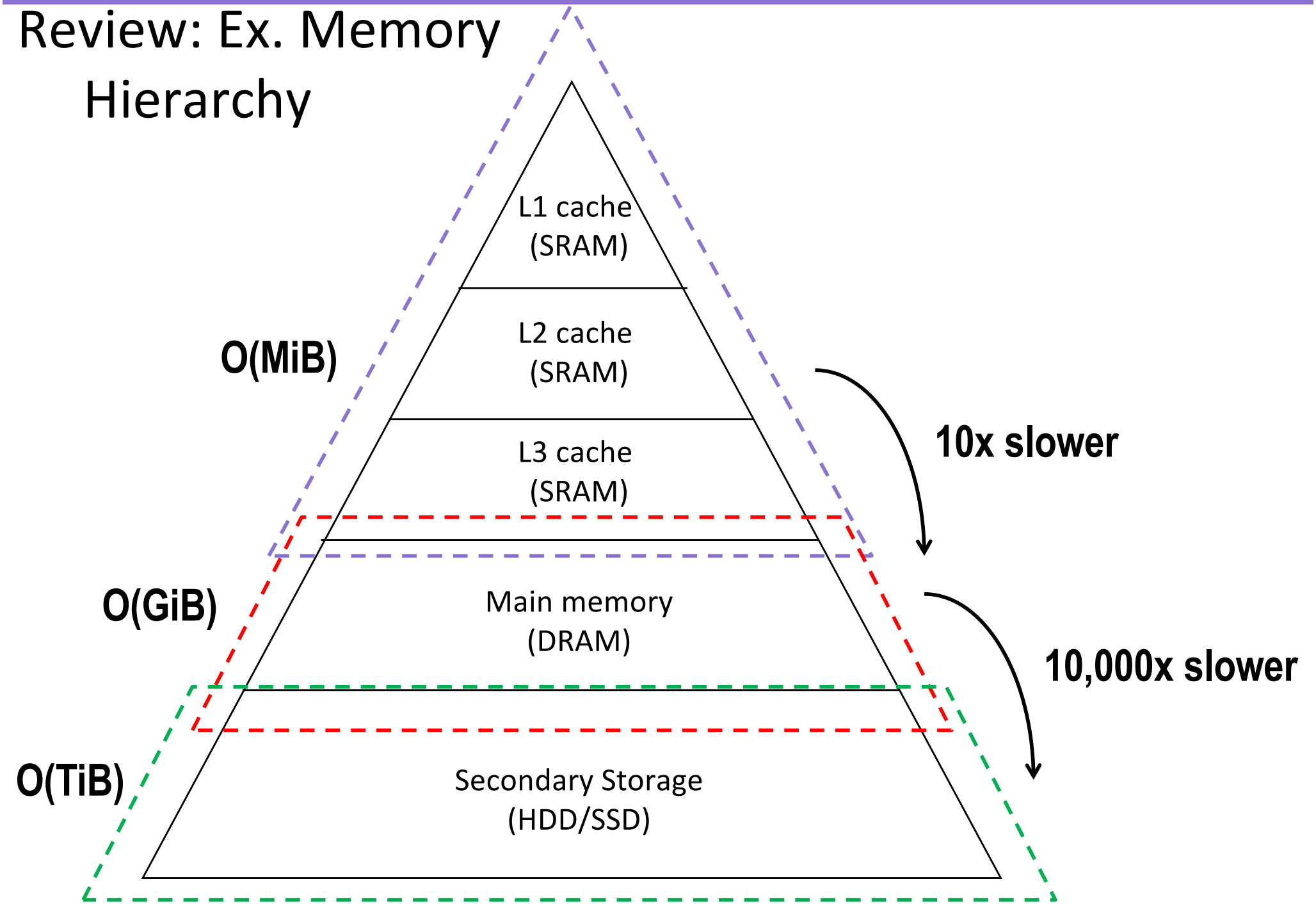
Why Virtual Memory (VM)?

- Uses main memory efficiently
  - Use DRAM as a cache for parts of a virtual address space
- Simplifies memory management
  - Each process gets the same uniform linear address space
- Isolates address spaces
  - Enables multiple processes to execute simultaneously
  - One process can't interfere with another's memory
  - Processes can allocate memory dynamically
  - *User* program cannot access privileged *kernel* information and code
- Total virtual memory in use can exceed physical memory

Today: Virtual Memory

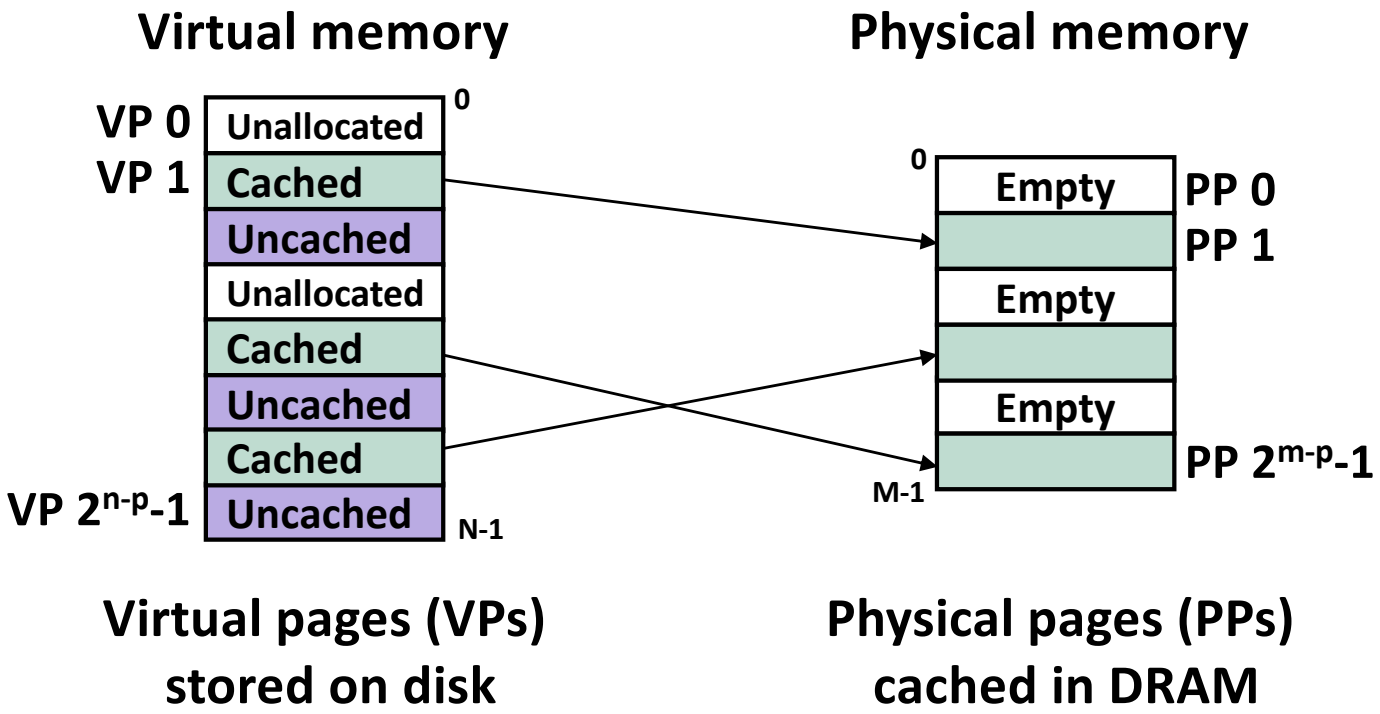
- Address spaces (Ch 9.2)
- VM as a tool for caching (Ch 9.3)
- VM as a tool for memory management (Ch 9.4)
- VM as a tool for memory protection (Ch 9.5)
- Address translation (Ch 9.6)

Review: Ex. Memory Hierarchy



VM as a Tool for Caching

- Conceptually, *virtual memory* is an array of N contiguous bytes stored **on disk**.
- The contents of the array on disk are cached in *physical memory* (*DRAM cache*)
  - These cache blocks are called *pages* (size is P = 2^p bytes)



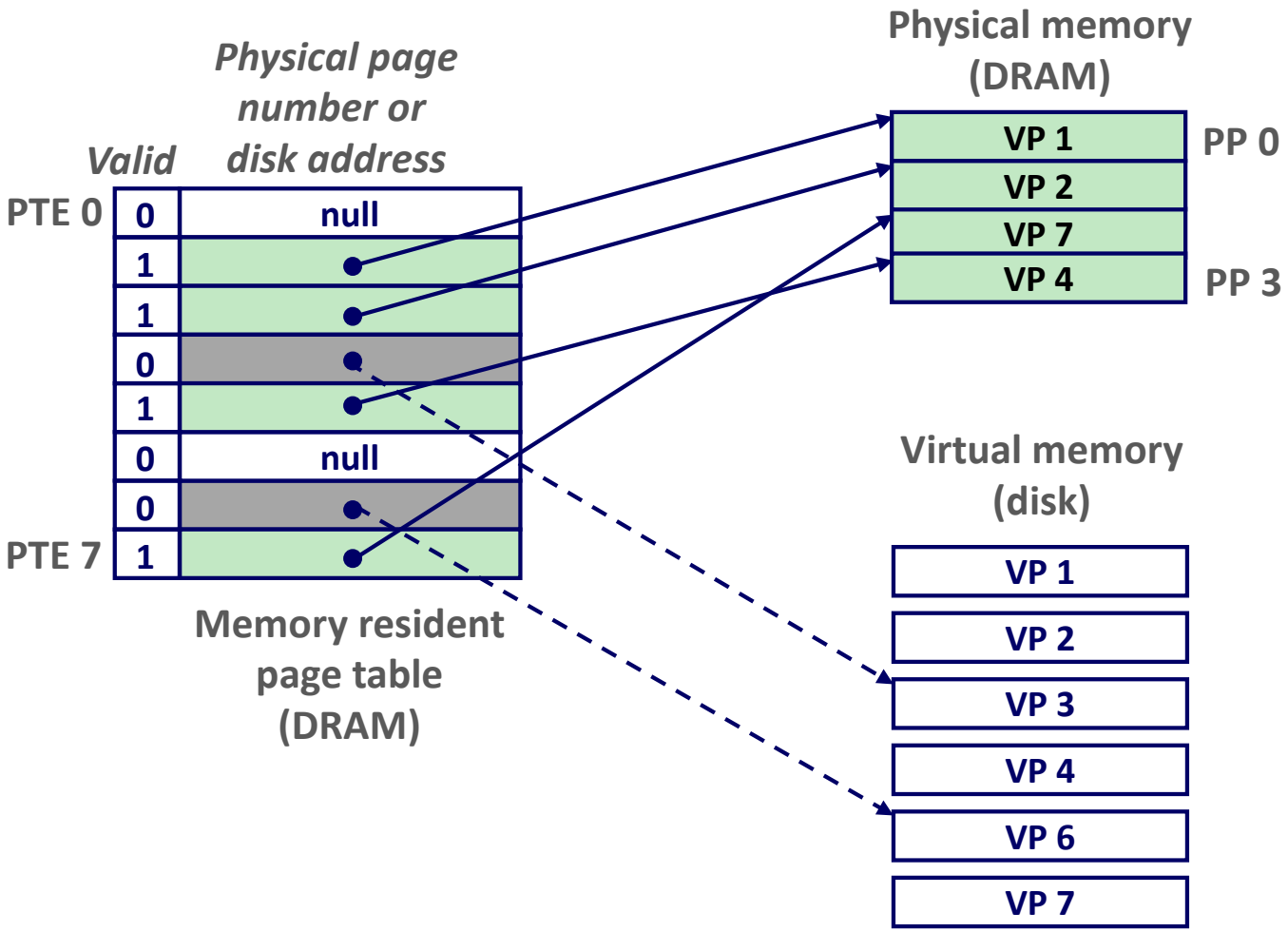
DRAM Cache Organization

- DRAM cache organization driven by the enormous miss penalty
  - DRAM is about ***10x*** slower than SRAM
  - Disk is about ***10,000x*** slower than DRAM

- Consequences
  - Large page (block) size: typically 4 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a “large” mapping function – different from cache memories
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through

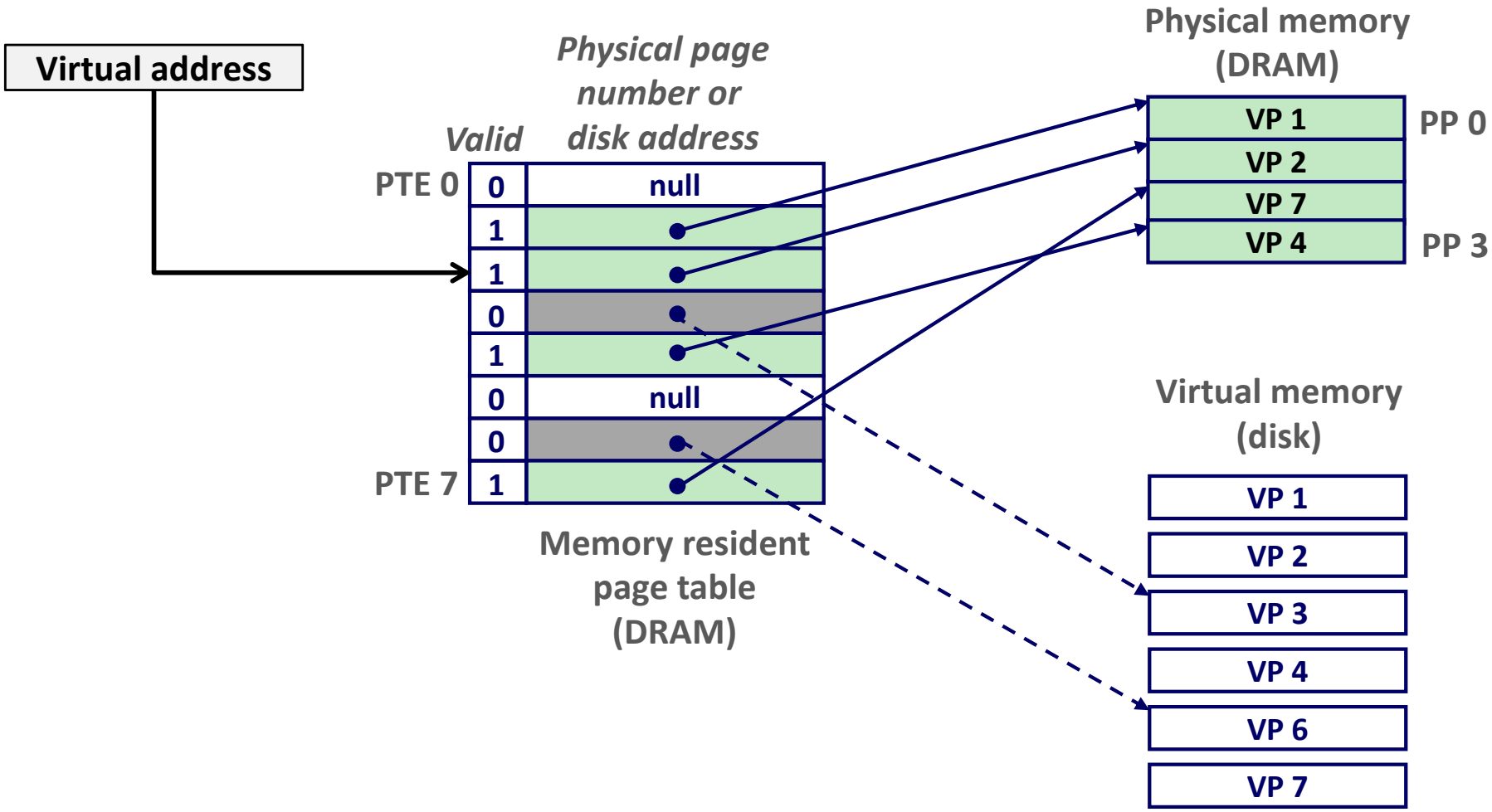
Enabling Data Structure: Page Table

- A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages.
  - Per-process kernel (i.e., OS) data structure in DRAM



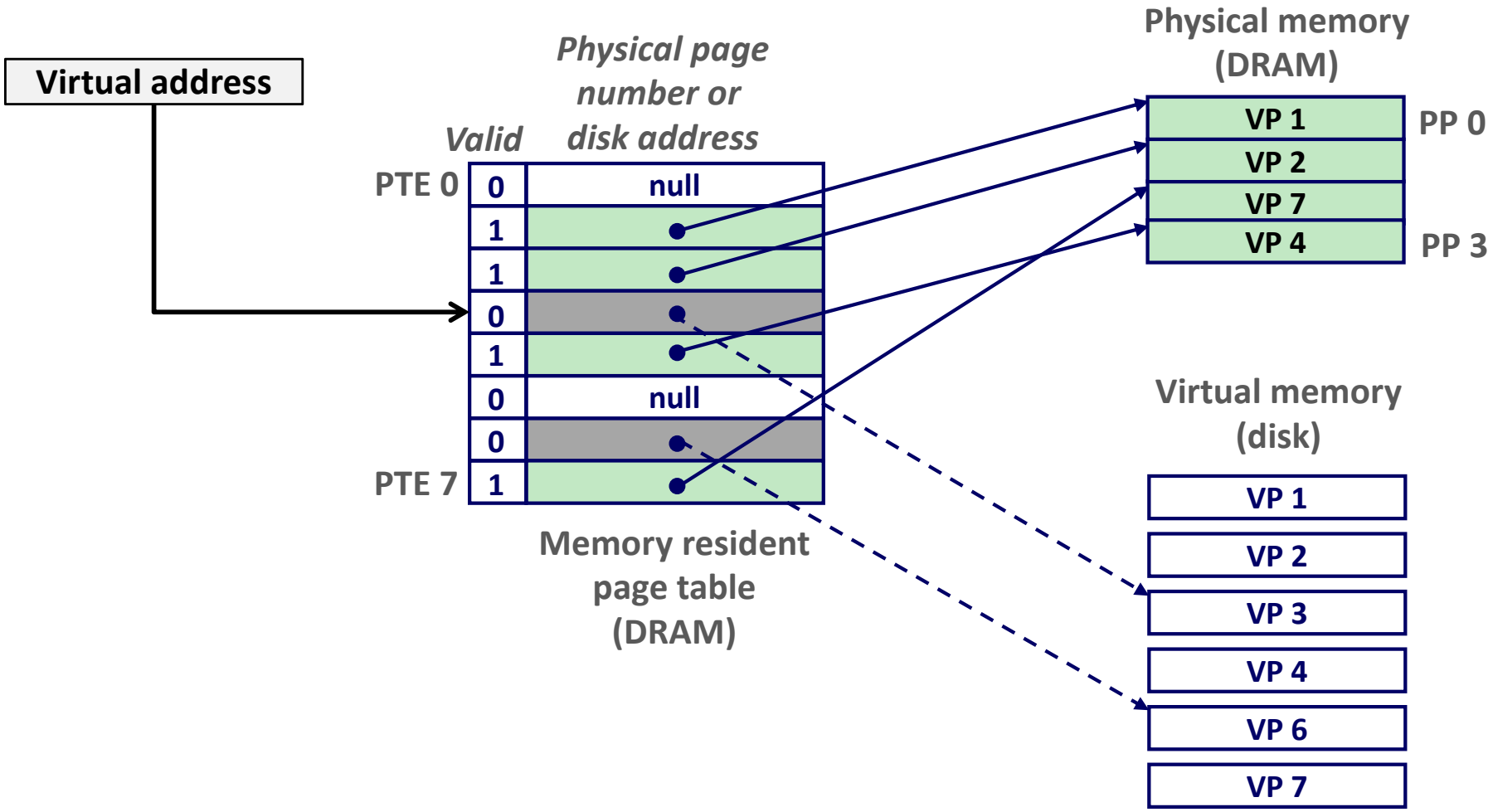
Page Hit

- *Page hit:* reference to VM word that is in physical memory (DRAM cache hit)



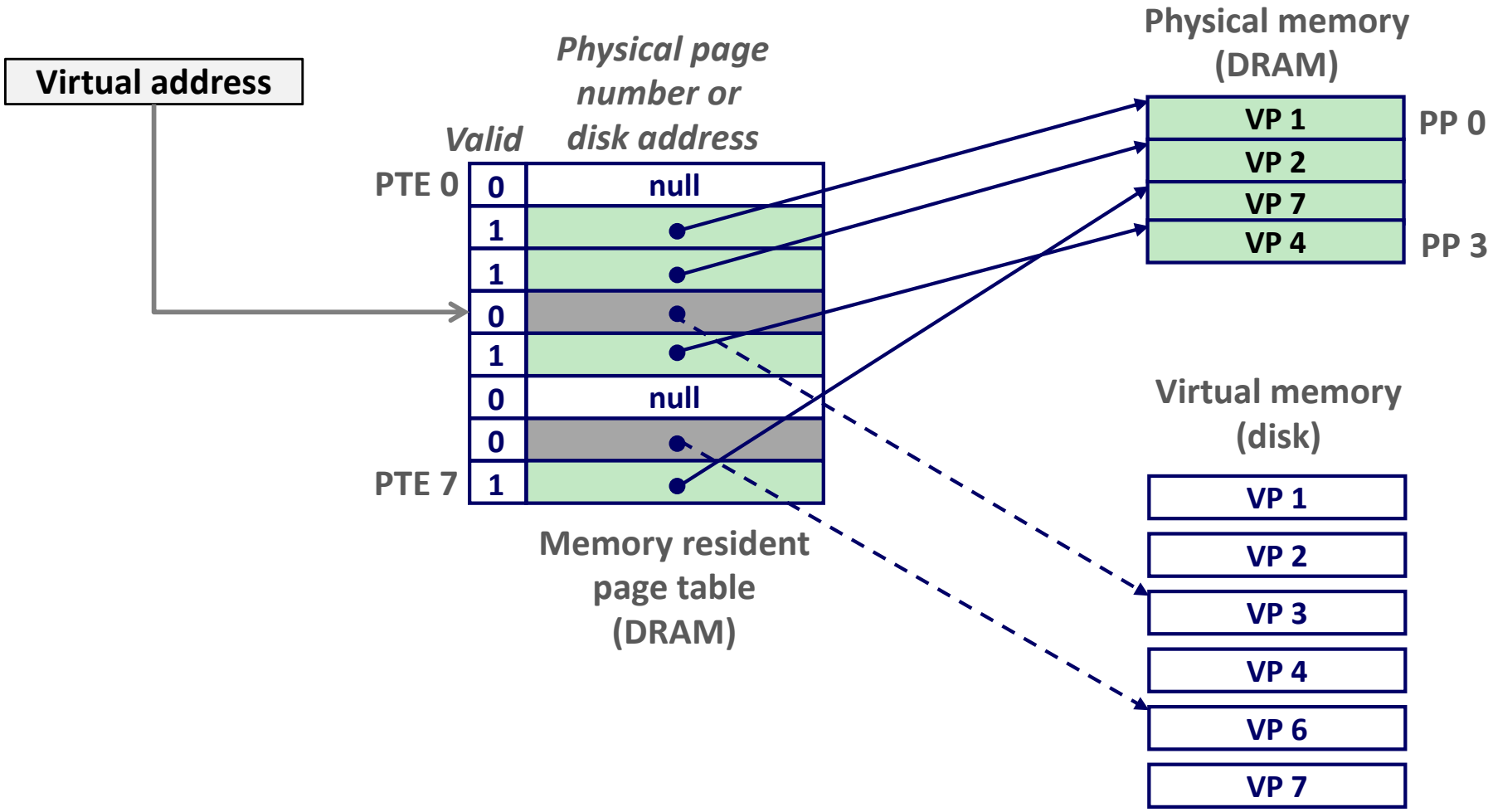
Page Fault

- *Page fault:* reference to VM word that is not in physical memory (DRAM cache miss)



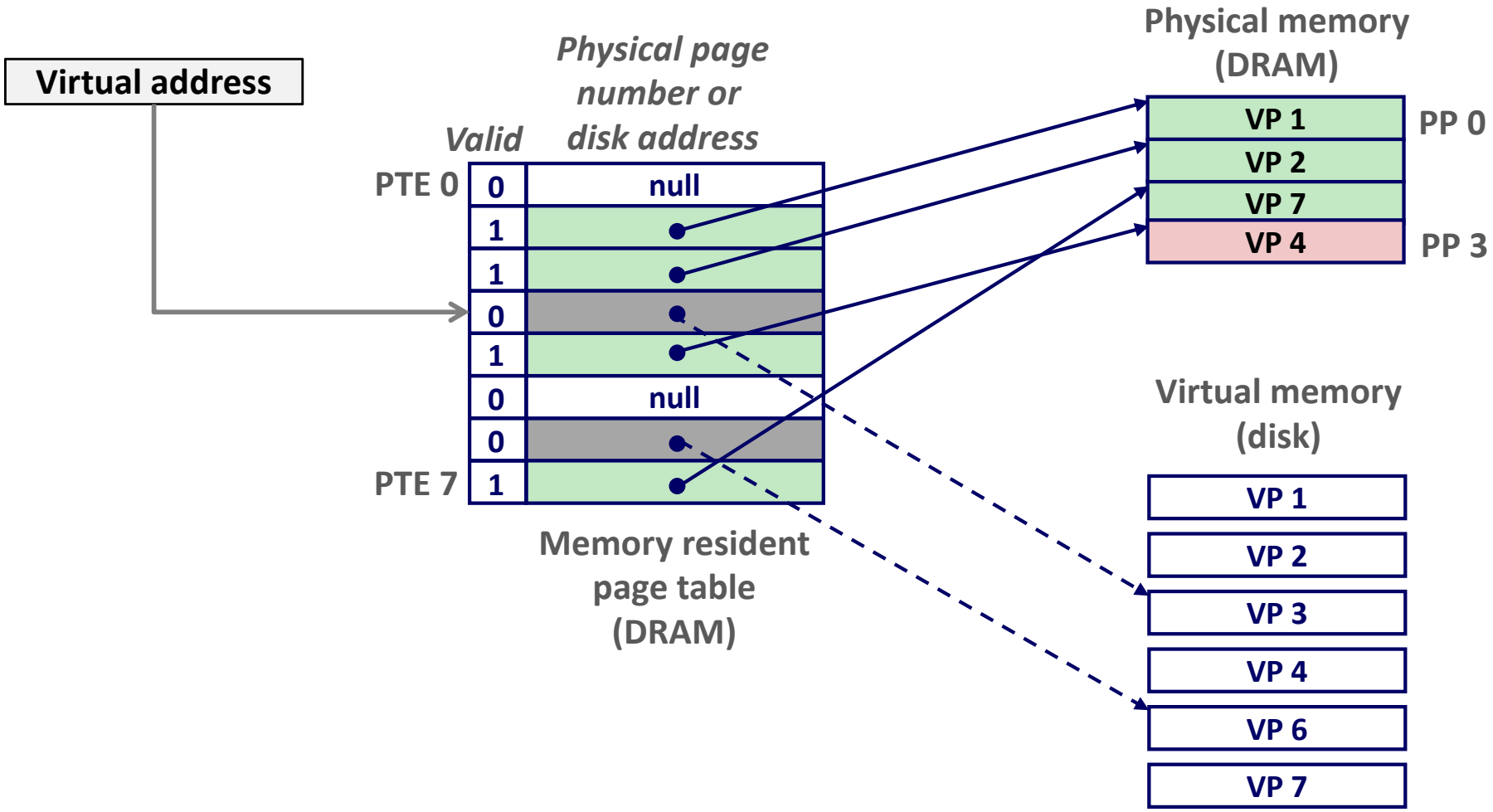
Handling Page Fault

- Page miss causes page fault (an exception)



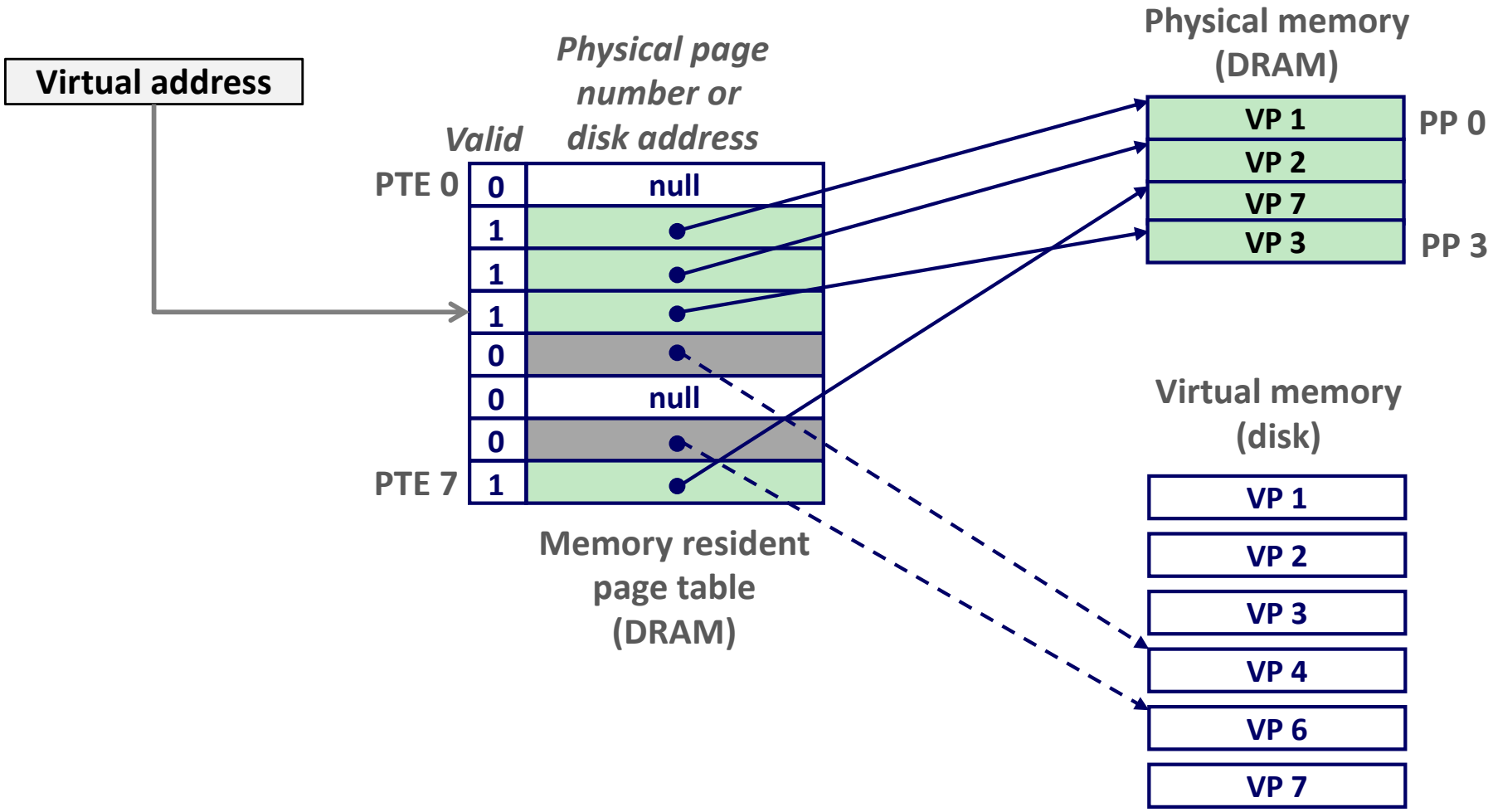
Handling Page Fault

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



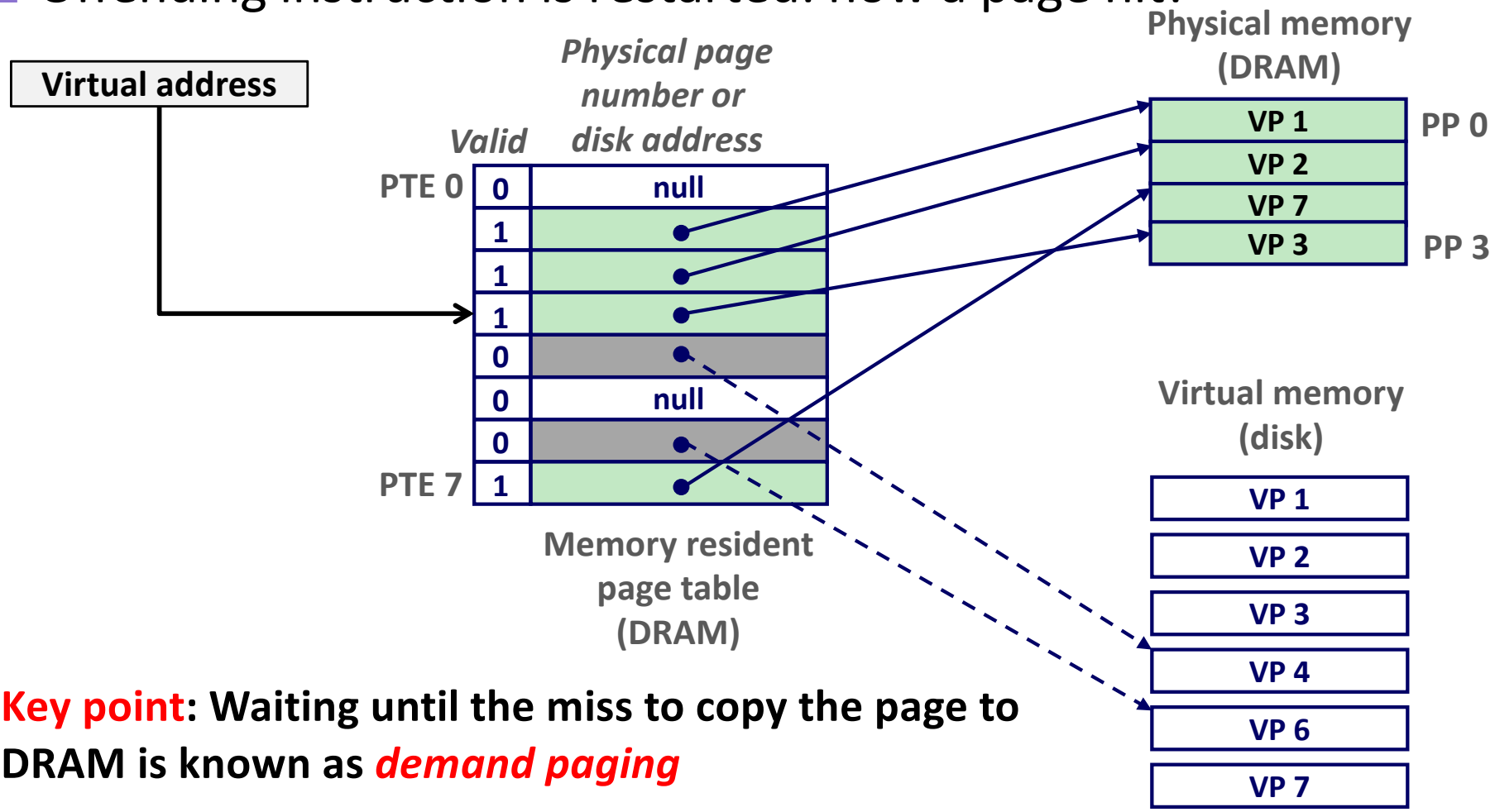
Handling Page Fault

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



Handling Page Fault

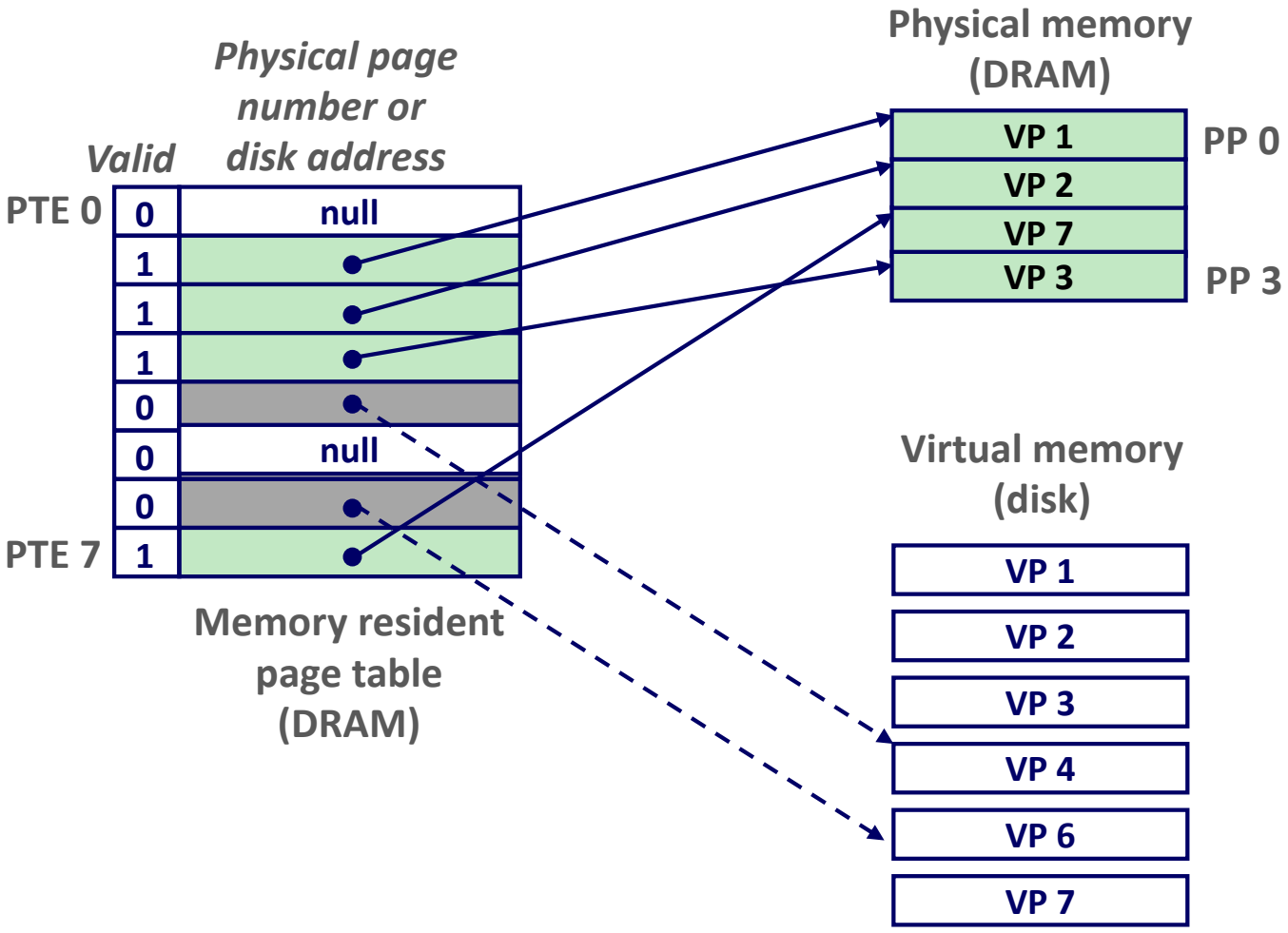
- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: now a page hit!



Key point: Waiting until the miss to copy the page to DRAM is known as *demand paging*

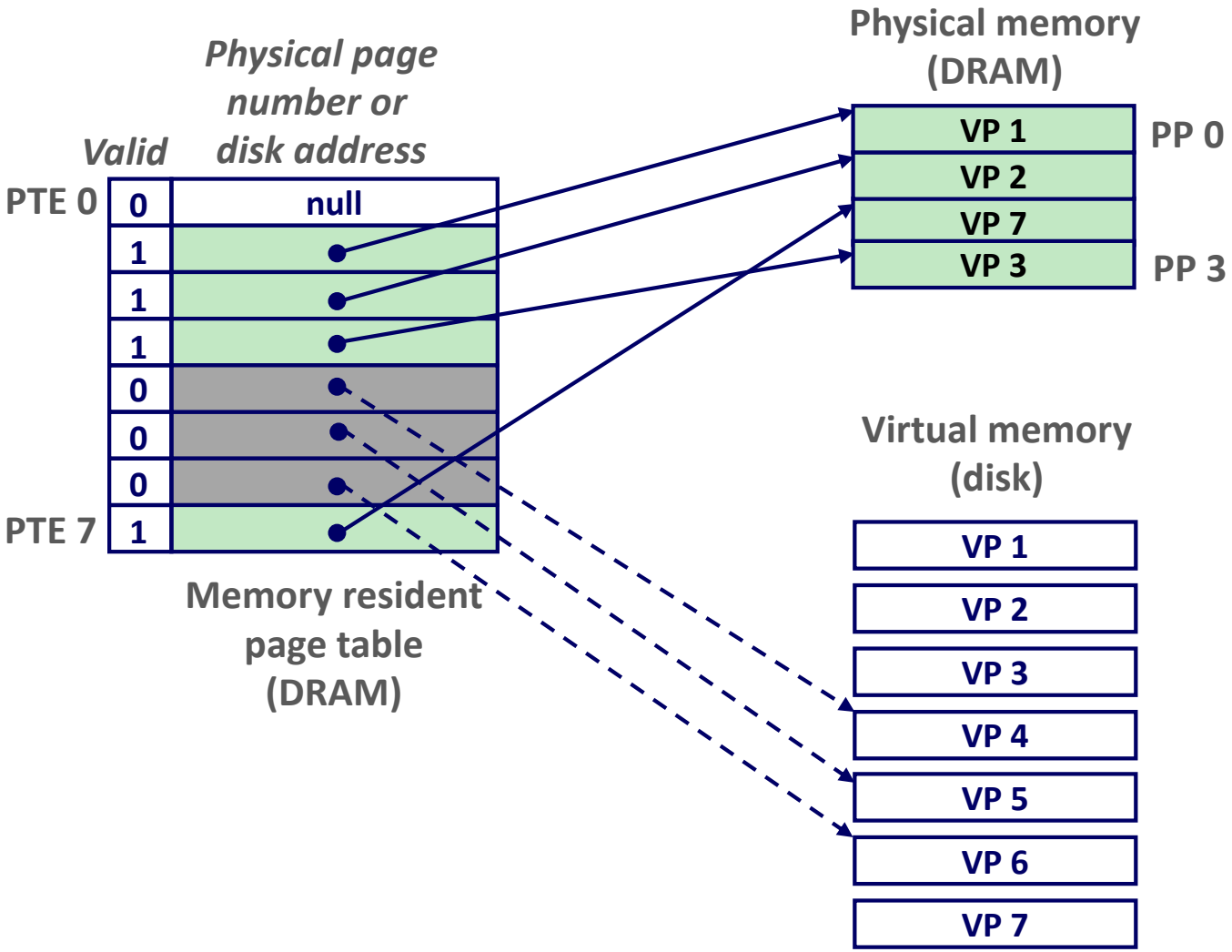
Allocating Pages

- Allocating a new page (VP 5) of virtual memory.



Allocating Pages

- Allocating a new page (VP 5) of virtual memory.



Locality to the Rescue Again!

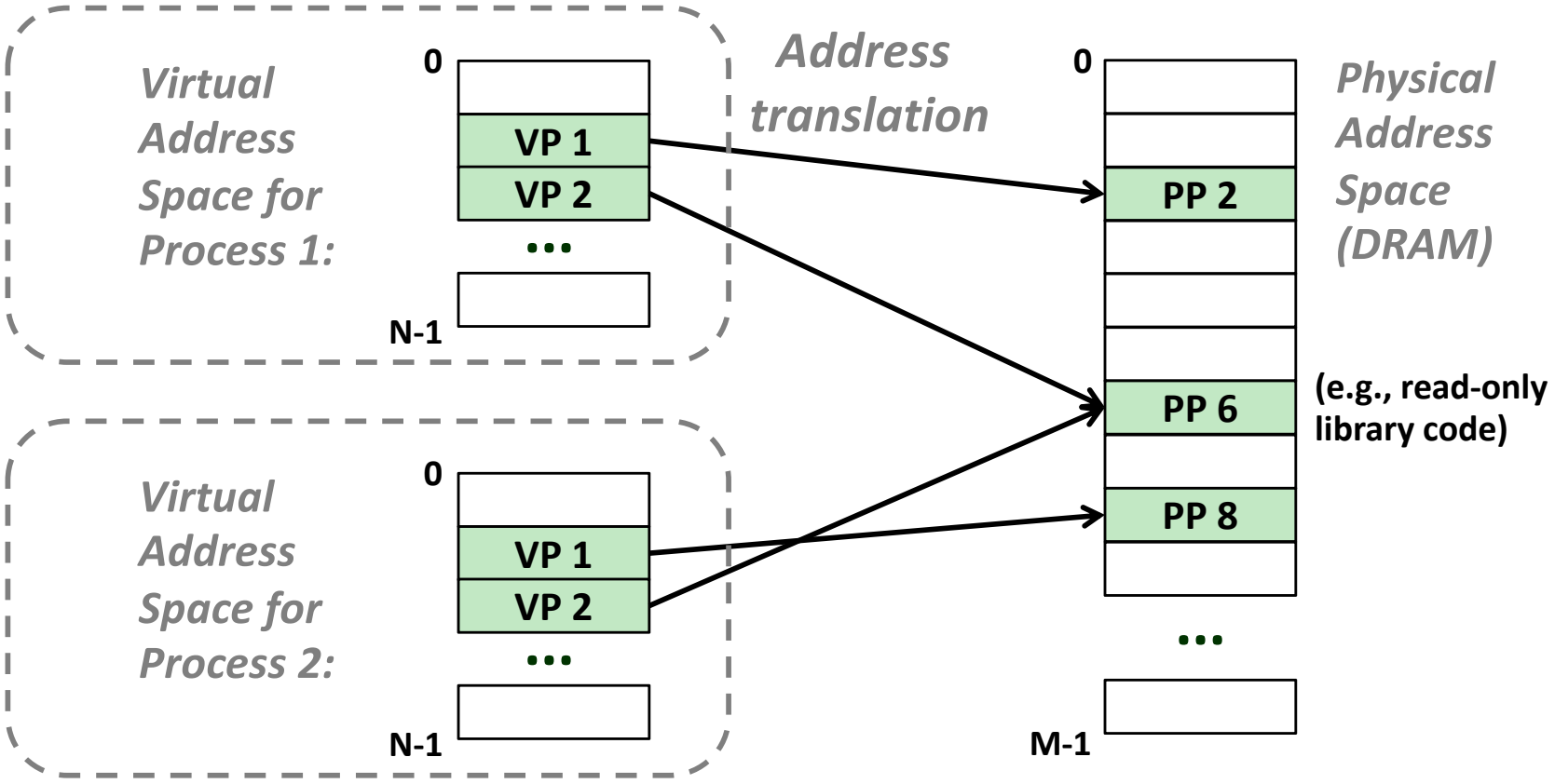
- Virtual memory seems terribly inefficient, but it works because of locality.
- At any point in time, programs tend to access a set of active virtual pages called the *working set*
  - Programs with better temporal locality will have smaller working sets
- If (working set size < main memory size)
  - Good performance for one process *after* initial compulsory misses
- If (SUM(working set sizes) > main memory size)
  - *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously

Today: Virtual Memory

- Address spaces (Ch 9.2)
- VM as a tool for caching (Ch 9.3)
- VM as a tool for memory management (Ch 9.4)
- VM as a tool for memory protection (Ch 9.5)
- Address translation (Ch 9.6)

VM as a Tool for Memory Management

- Key idea: each process has its own virtual address space
  - A process can view memory as a simple linear array
  - Mapping function scatters virtual addresses through physical memory

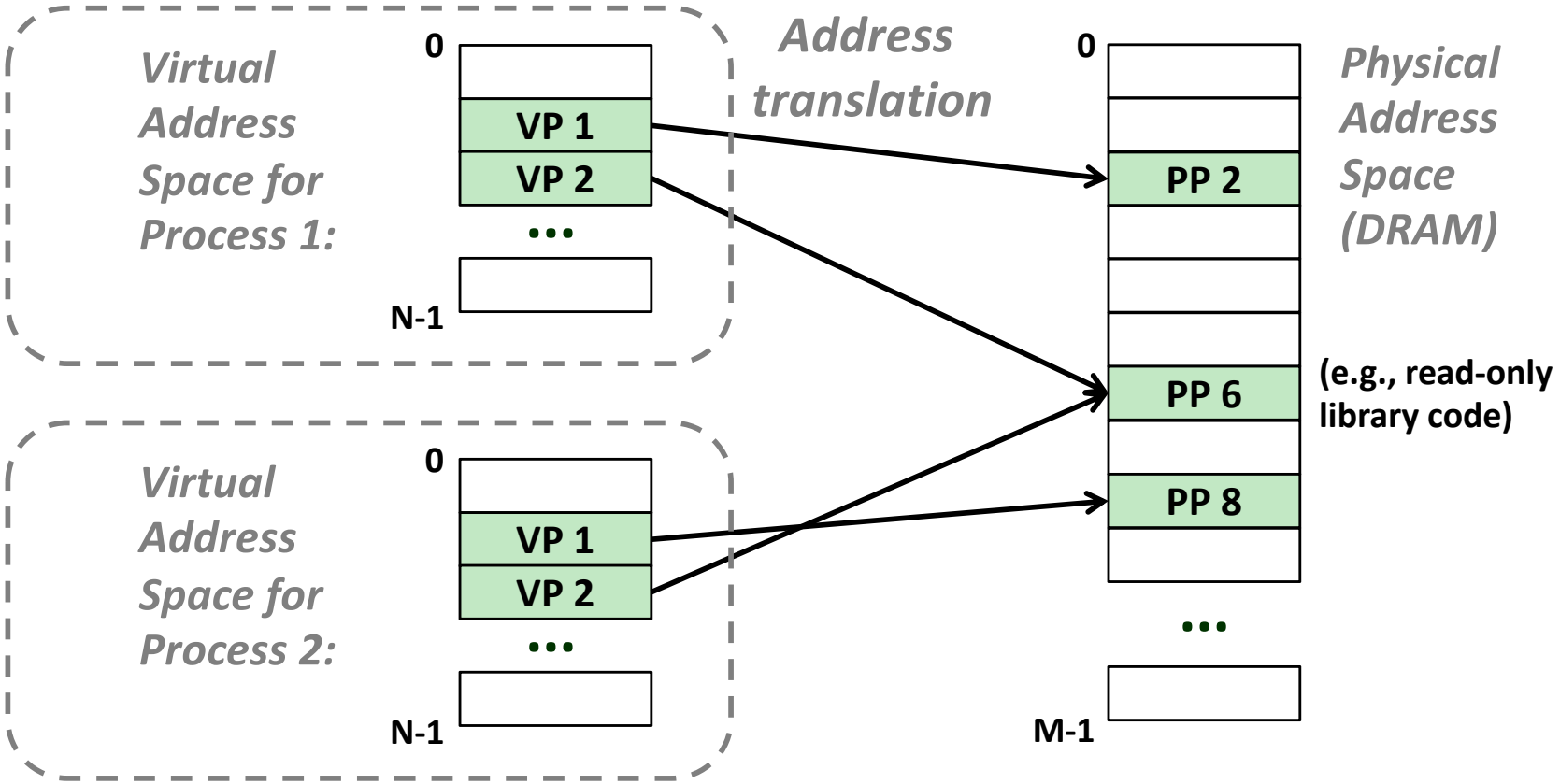


VM as a Tool for Memory Management

- Simplifies memory allocation
  - Each virtual page can be mapped to any physical page
  - A virtual page can be stored in different physical pages at different times
- Allows sharing code and data among processes

Why?

  - Can map virtual pages to the same physical page (here: PP 6)

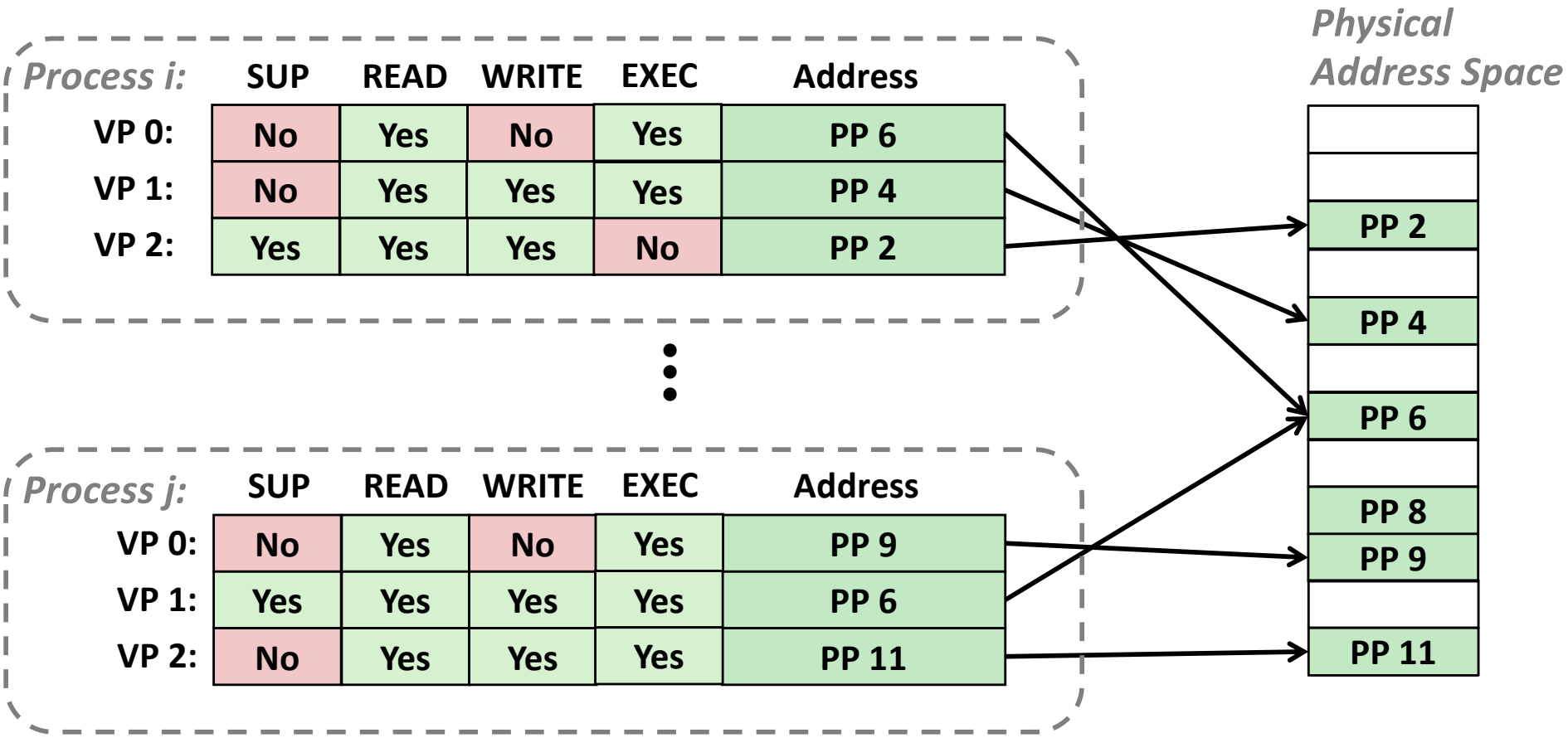


Today: Virtual Memory

- Address spaces (Ch 9.2)
- VM as a tool for caching (Ch 9.3)
- VM as a tool for memory management (Ch 9.4)
- VM as a tool for memory protection (Ch 9.5)
- Address translation (Ch 9.6)

VM as a Tool for Memory Protection

- General Idea: Extend PTEs with permission bits
- MMU checks these bits on each access



Today: Virtual Memory

- Address spaces (Ch 9.2)
- VM as a tool for caching (Ch 9.3)
- VM as a tool for memory management (Ch 9.4)
- VM as a tool for memory protection (Ch 9.5)
- Address translation (Ch 9.6)

VM Address Translation (formally)

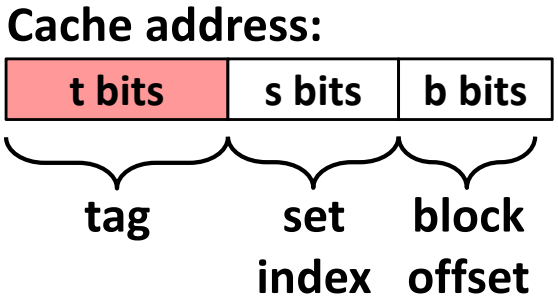
- Virtual Address Space
  - $V = \{0, 1, \ldots, N-1\}$
- Physical Address Space
  - $P = \{0, 1, \ldots, M-1\}$
- Address Translation. **MAP:** $V \rightarrow P \cup \{\varnothing\}$
  - For virtual address ***a***:
    - ***MAP(a) = a'*** if data at virtual address ***a*** in ***V*** is at physical address ***a'*** in ***P***
    - ***MAP(a) =*** $\varnothing$ if data at virtual address ***a*** is not in physical memory (either unallocated or stored on disk)

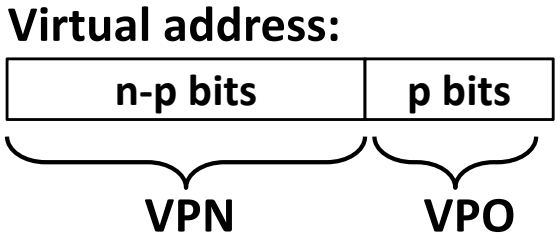
Summary of Address Translation Jargon

- 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 (in bytes) of physical and virtual pages
- Components of the virtual address (VA)
  - **VPN**: Virtual page number
  - **VPO**: Virtual page offset
  - **TLBI**: TLB (Translation Lookaside Buffer) index
  - **TLBT**: TLB tag
- Components of the physical address (PA)
  - **PPN:** Physical page number
  - **PPO**: Physical page offset (same as VPO)

Translating Virtual to Physical Addresses

- Control register (CR3) stores physical address of Page Table
- Given the page table's location, how does lookup work?
  - Familiar approach: break up the address and index into our cache



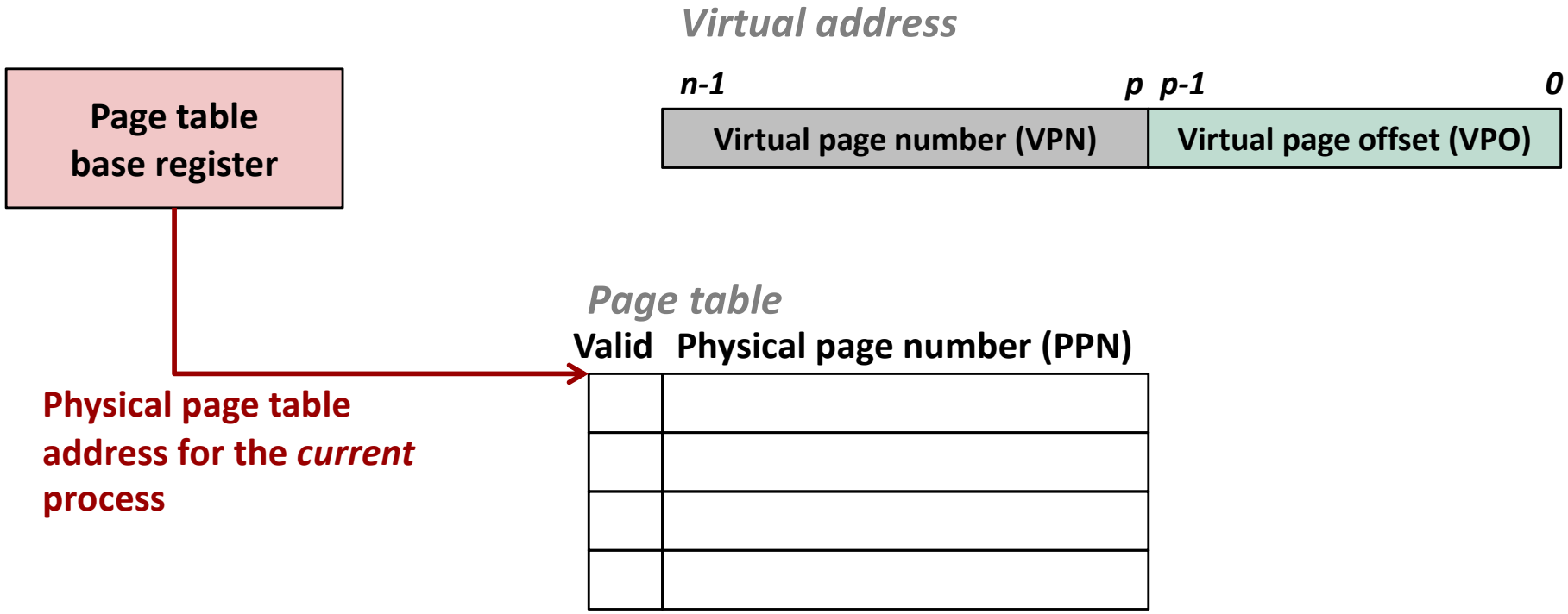


Typically, how many bits is p? **$\log_2(4096)$ => 12 bits**

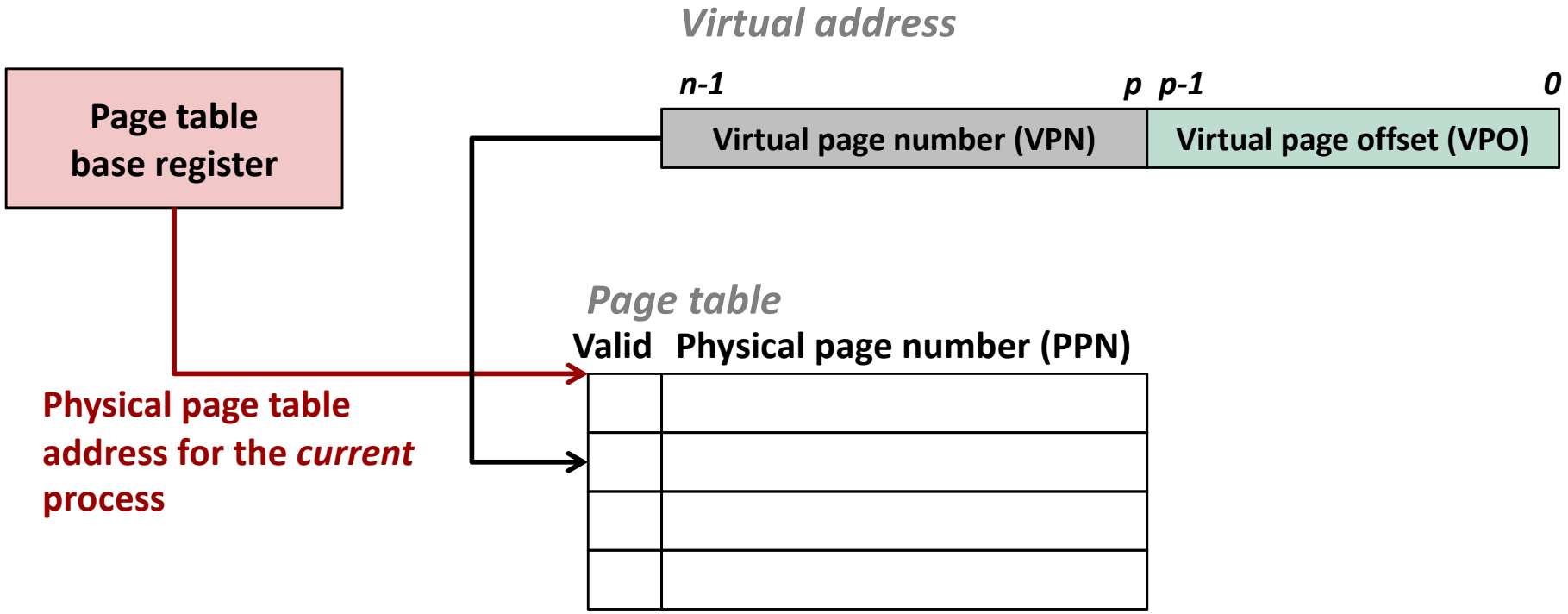
What does the VPN tell us? **Offset of our PTE in the page table**

What does the VPO tell us? **Offset of our data within the page**

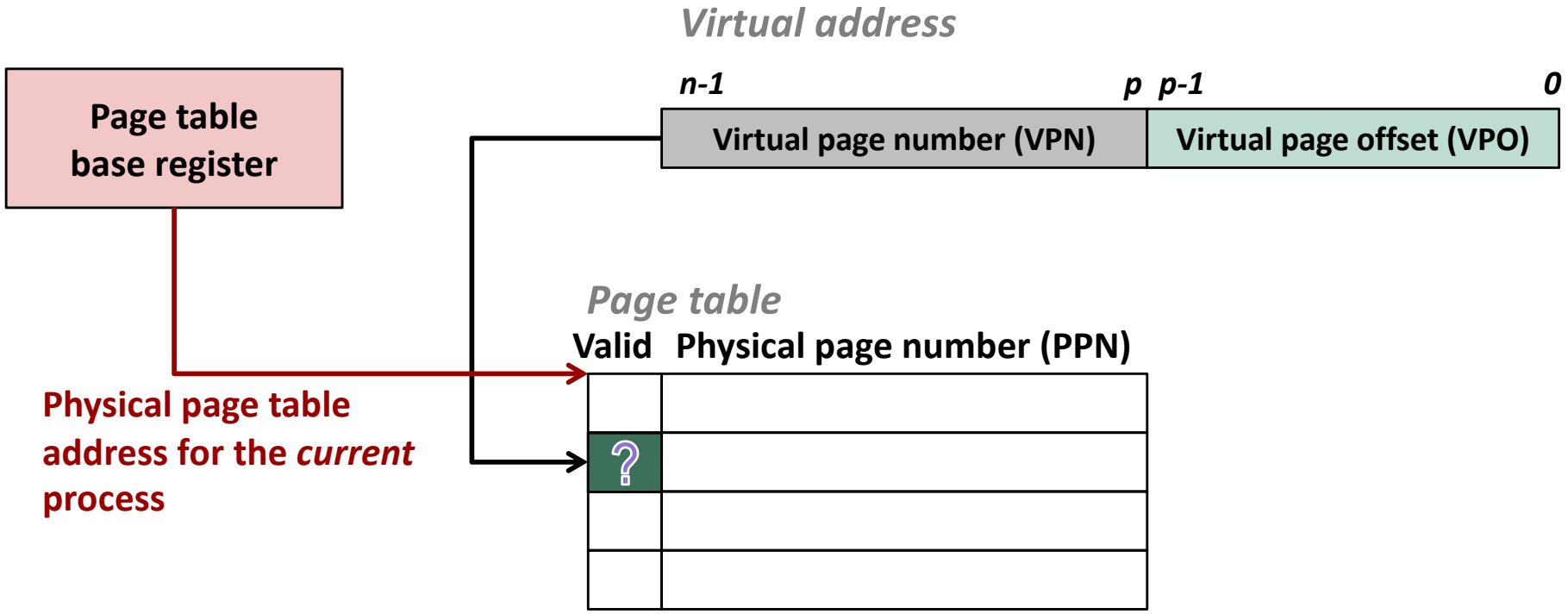
Address Translation With a Page Table



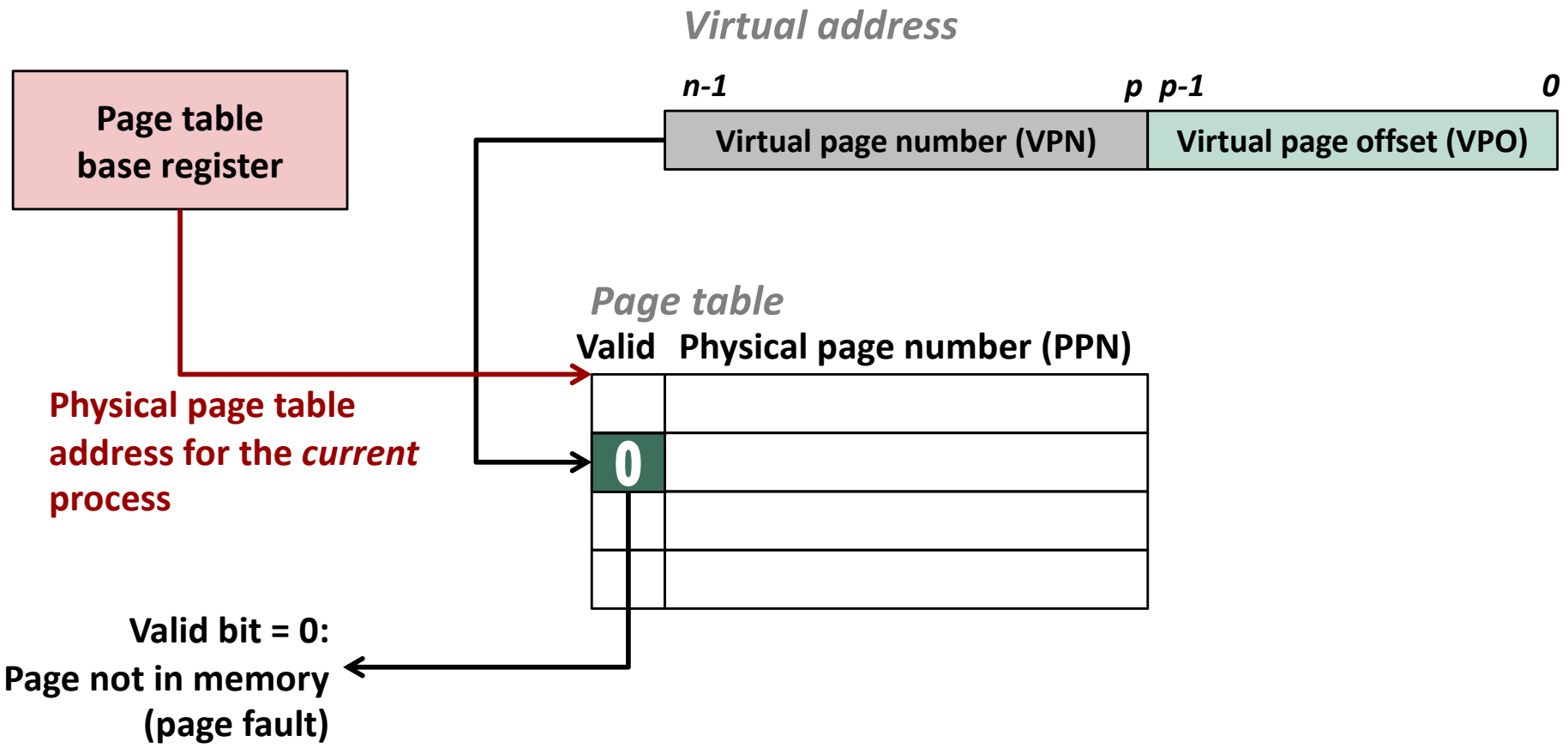
Address Translation With a Page Table



Address Translation With a Page Table

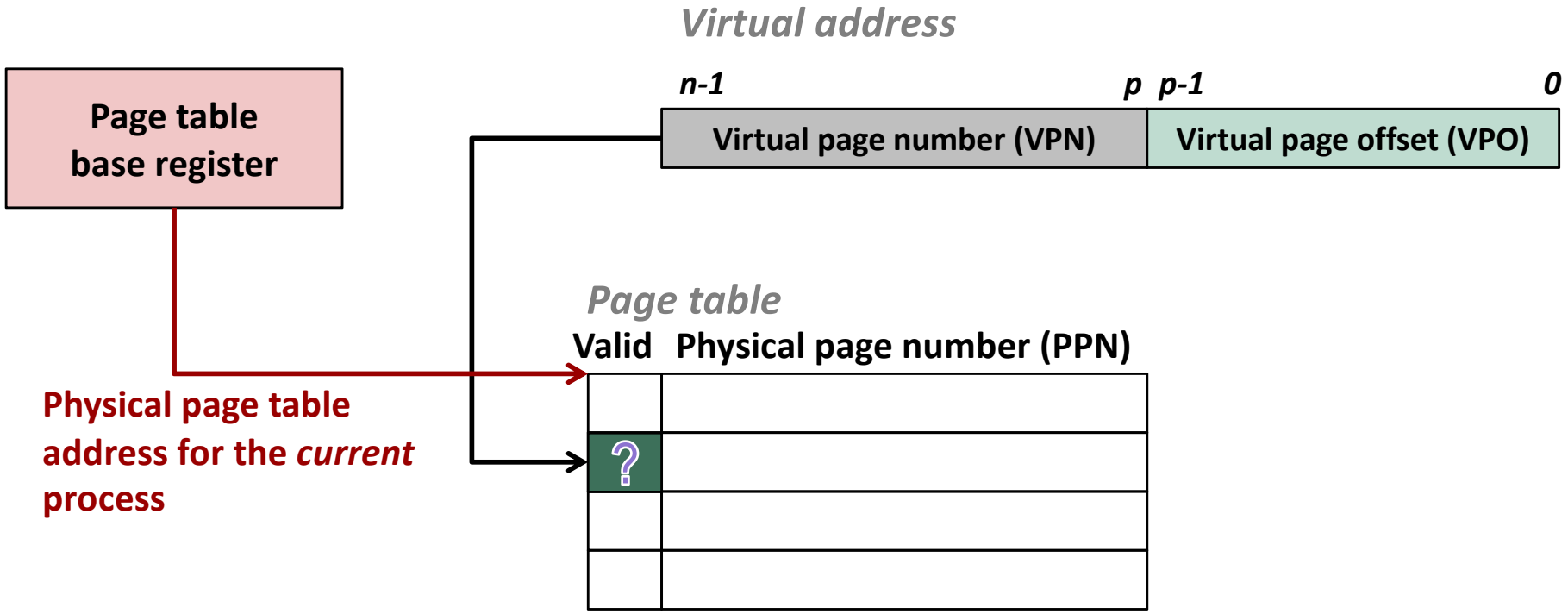


Address Translation With a Page Table

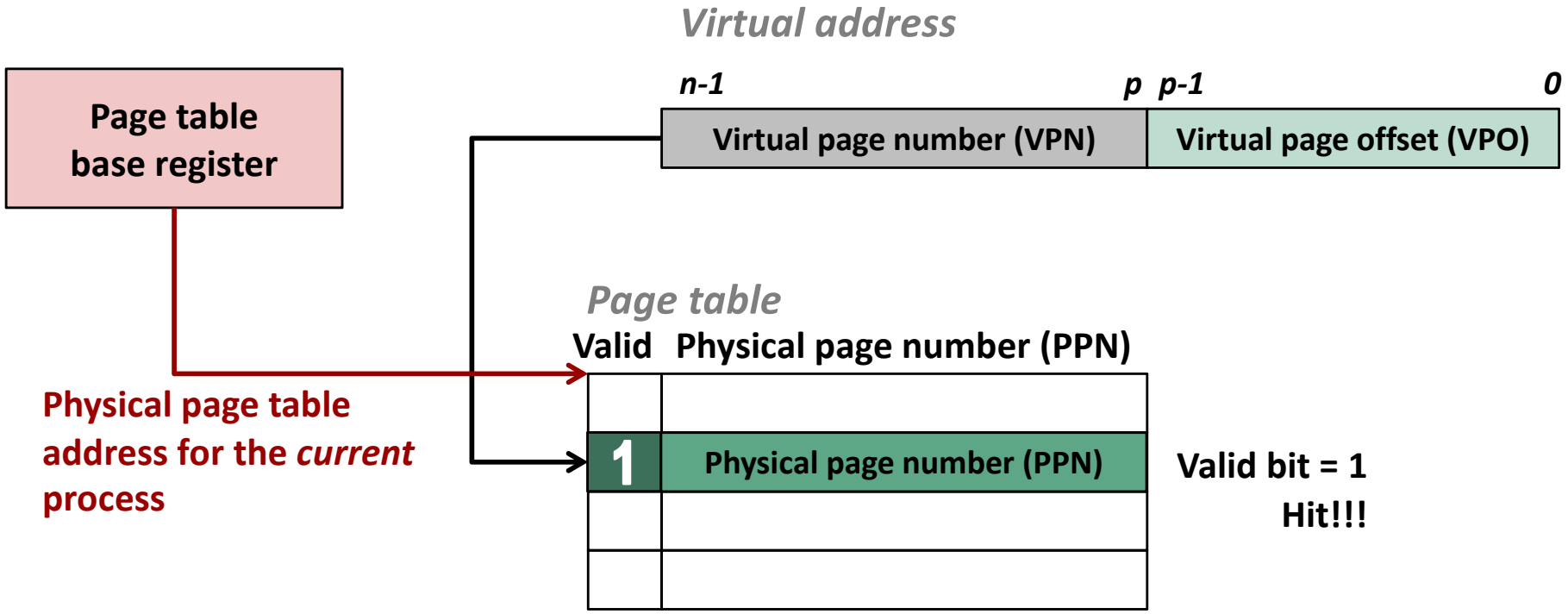


What do we do on a page fault? **Page fault exception handler reads physical page from disk and updates PTE for this VA.**

Address Translation With a Page Table



Address Translation With a Page Table



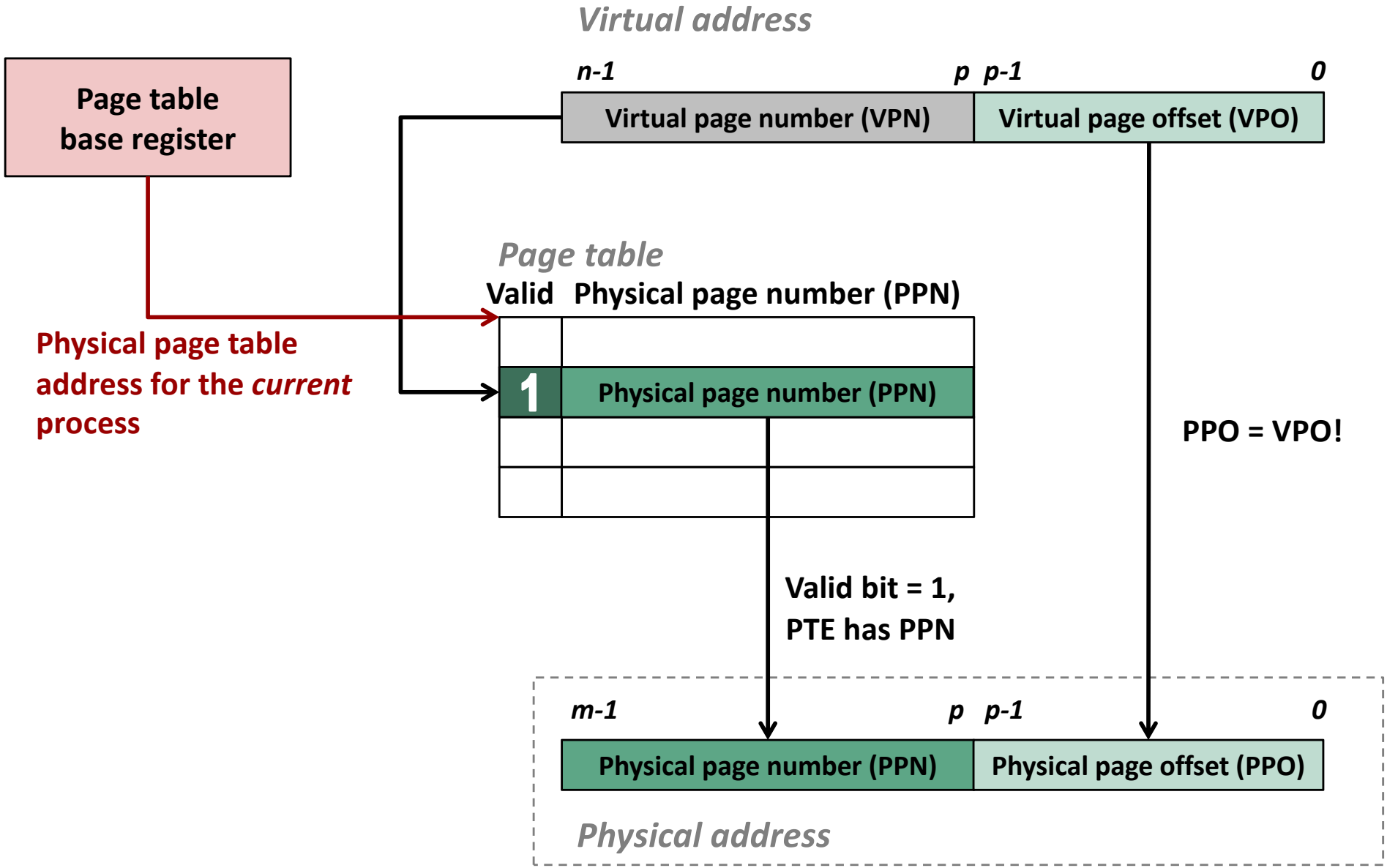
What does a hit mean?

The PTE contains the PPN.

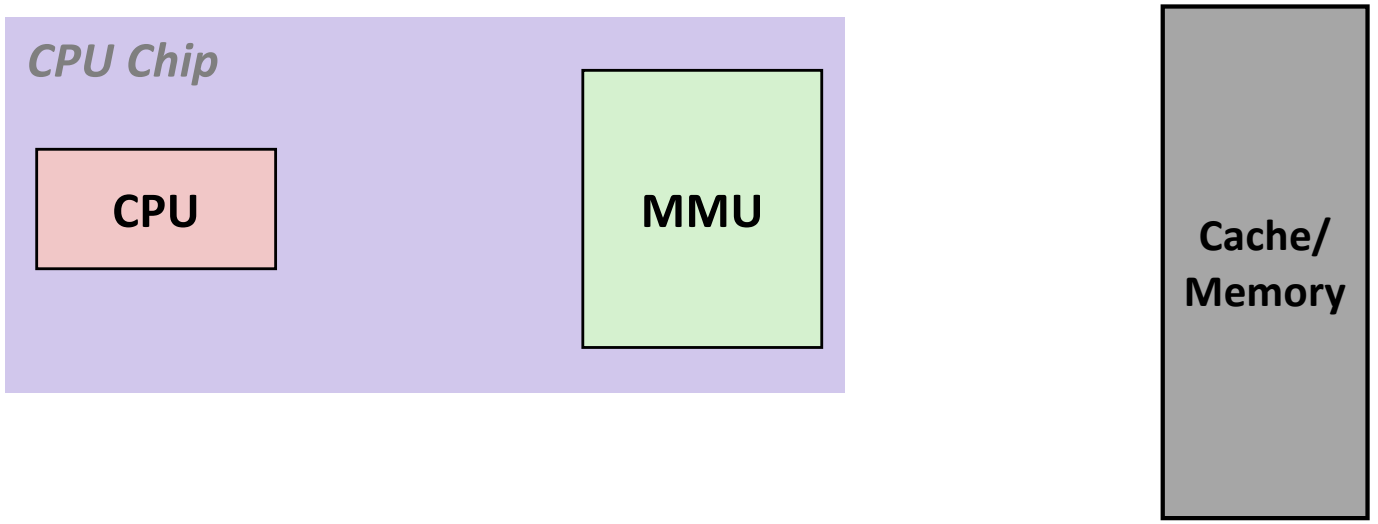
Are we done?

No. A physical address consists of a PPN and a PPO. We still need the PPO…

Address Translation With a Page Table

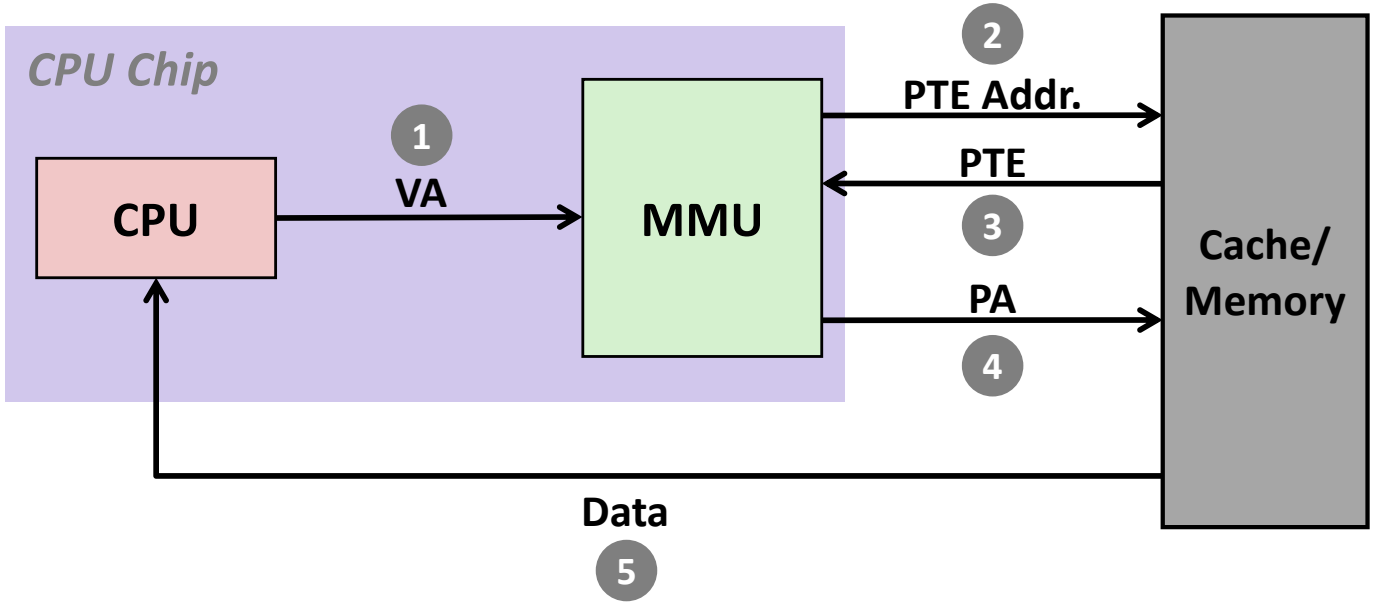


Putting it all together: Address Translation



- The CPU generates a request for a virtual address
- The MMU drives the page translation
- Let's explore the interaction between the CPU, MMU, and Cache/memory on:
  - A page table hit
  - A page fault (miss)

Address Translation: Page Hit



1) Processor sends virtual address to MMU

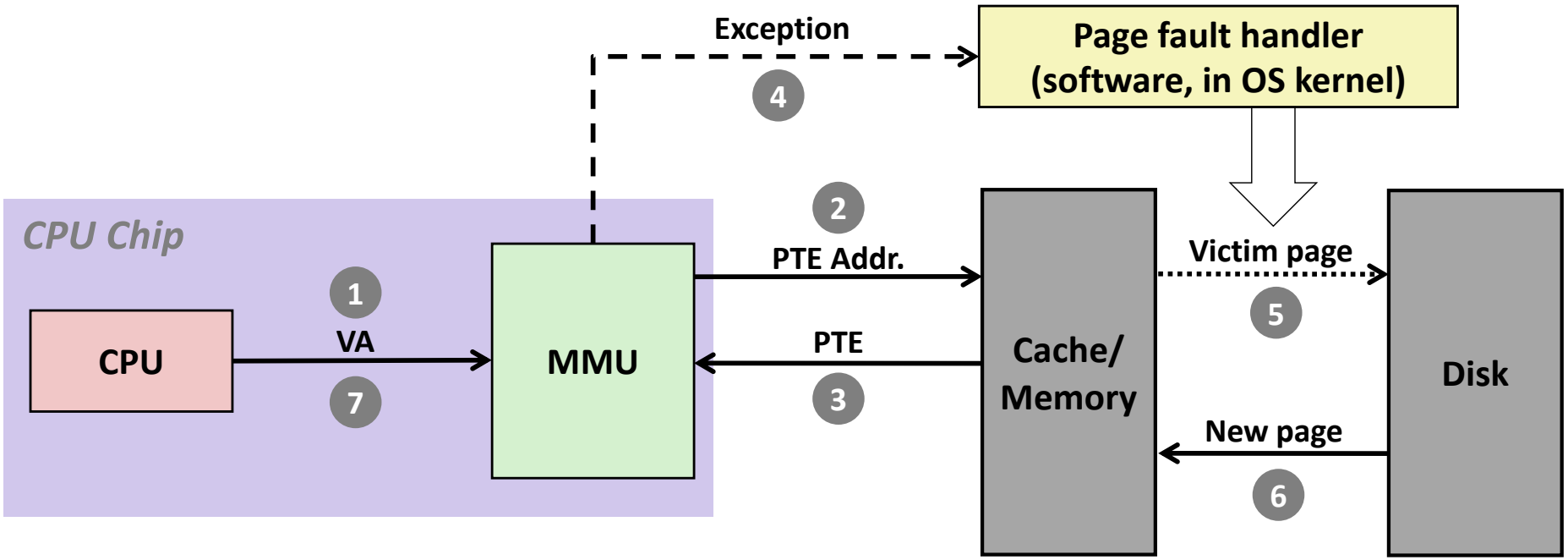
2-3) MMU fetches PTE from page table in cache/memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

Page hit handled entirely by hardware!

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 PTE (and, if dirty, swaps it out to disk)

6) Handler pages in new page and updates PTE in memory

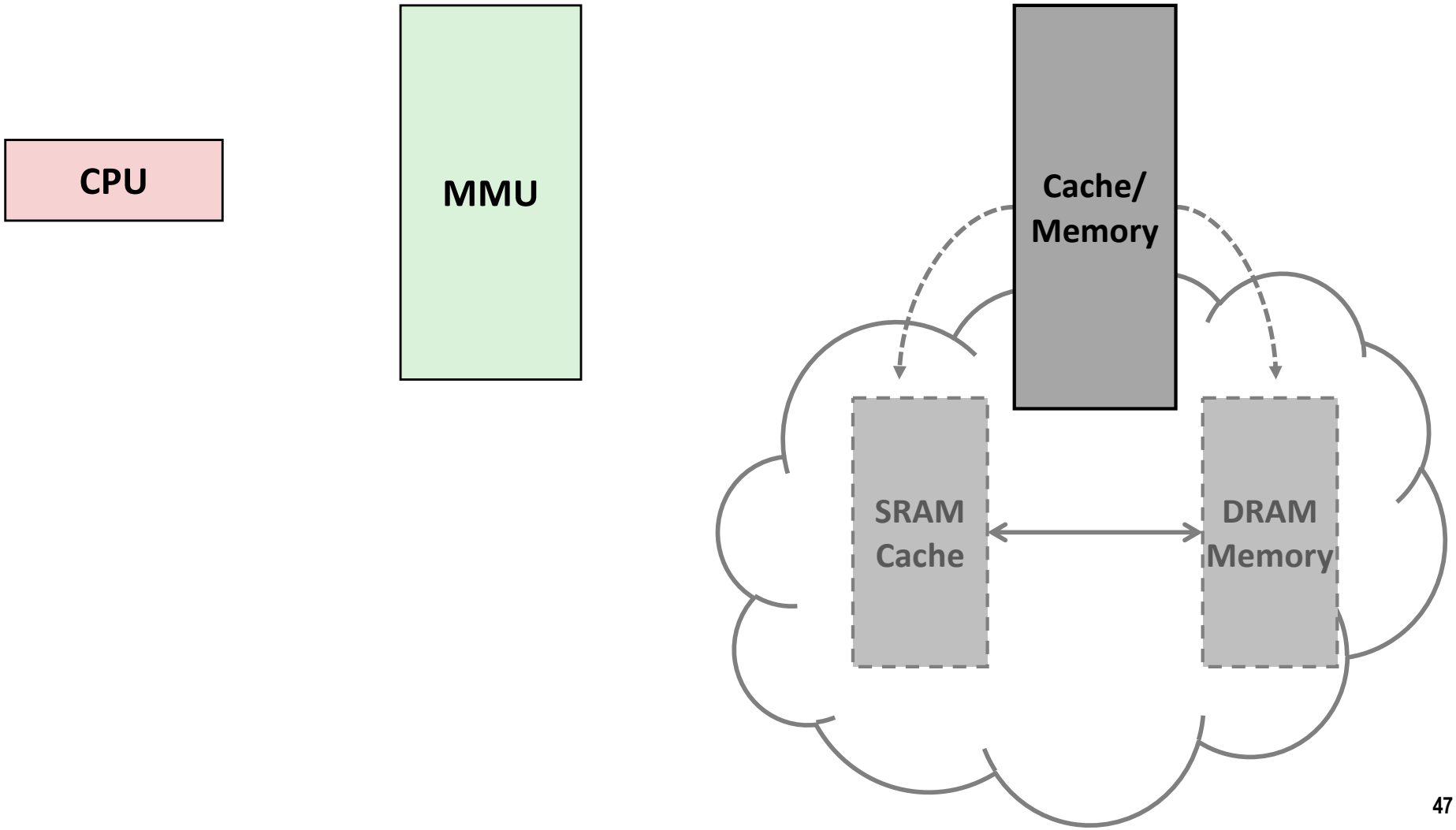
7) Handler returns to original process. Original process restarts faulting instruction

Restarted instruction should now hit.

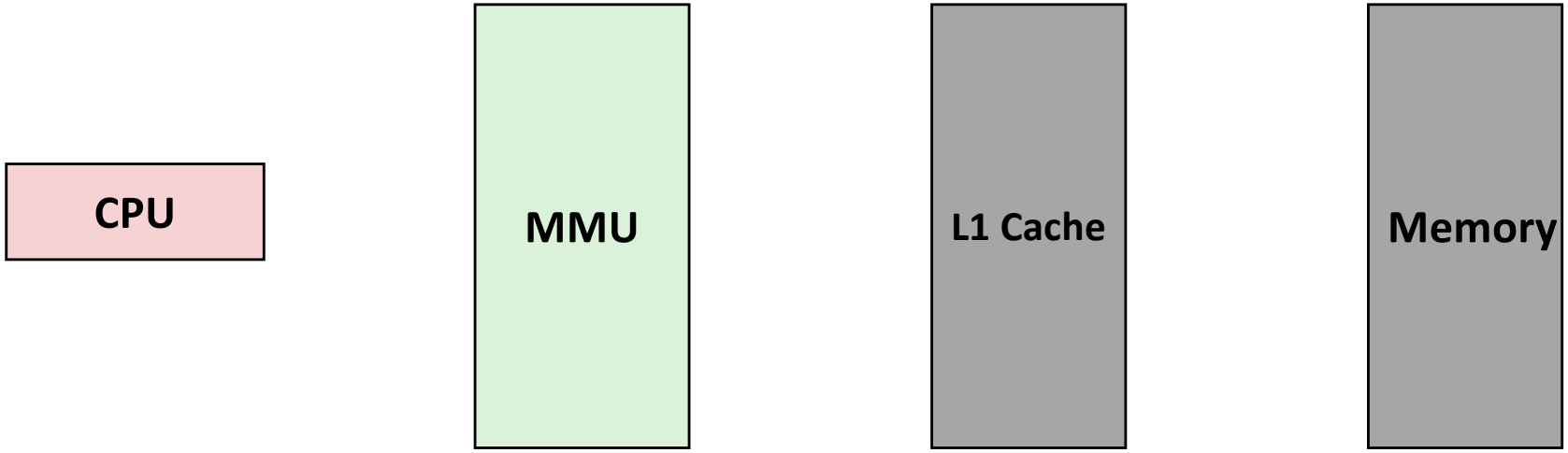
Practice on Your Own

- Suppose a system uses 2048B sized pages and addresses are specified using 32 bits. How many PTEs would be needed in the page table for a process? How many bits would be need to specify the virtual page offset (i.e., which byte in the page is being accessed)?

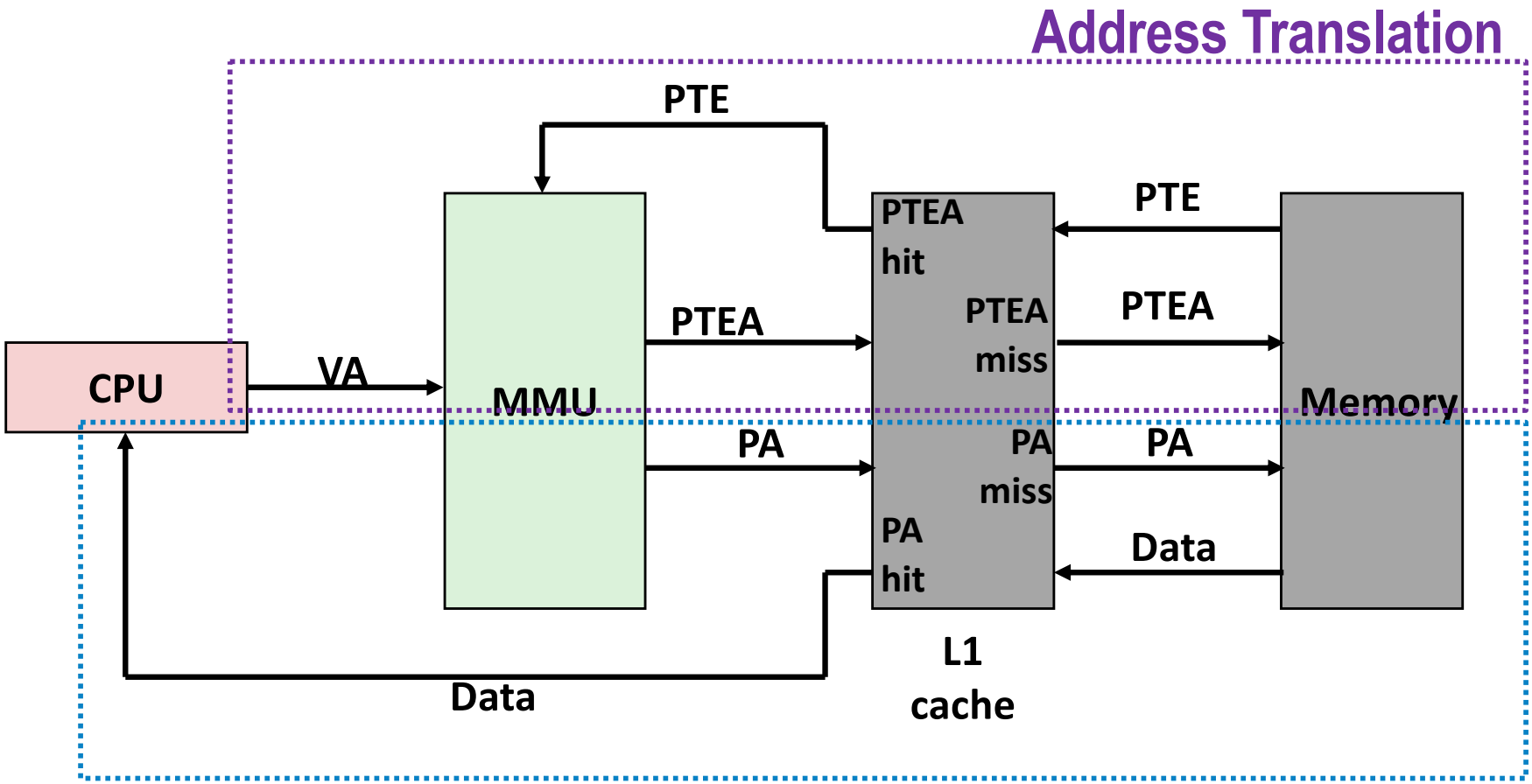
Digging Deeper: Integrating VM and Caching



Integrating VM and Cache: “The players”



Integrating VM and Cache: "The game"

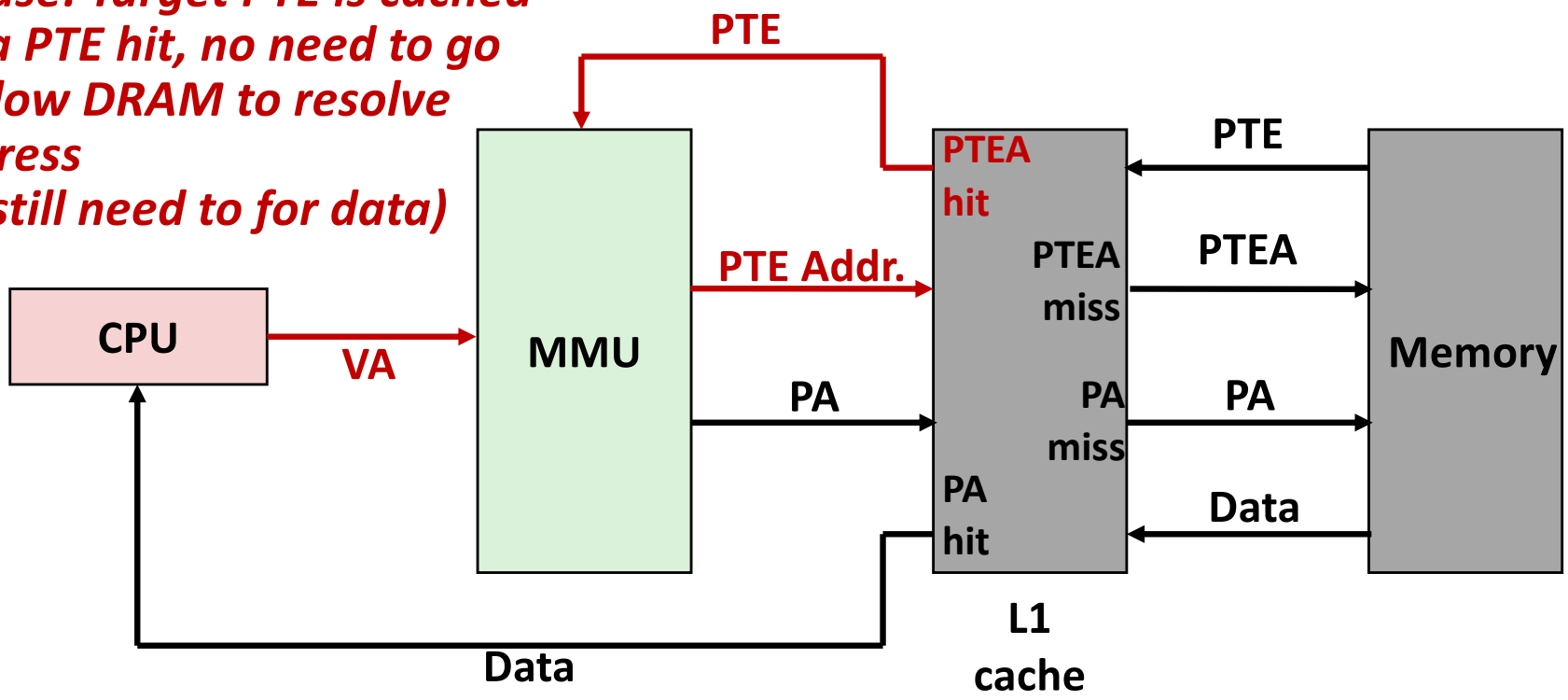


VA: virtual address, PA: physical address,
PTE: page table entry, PTEA = PTE address

Data transfer depends on address translation

Integrating VM and Cache

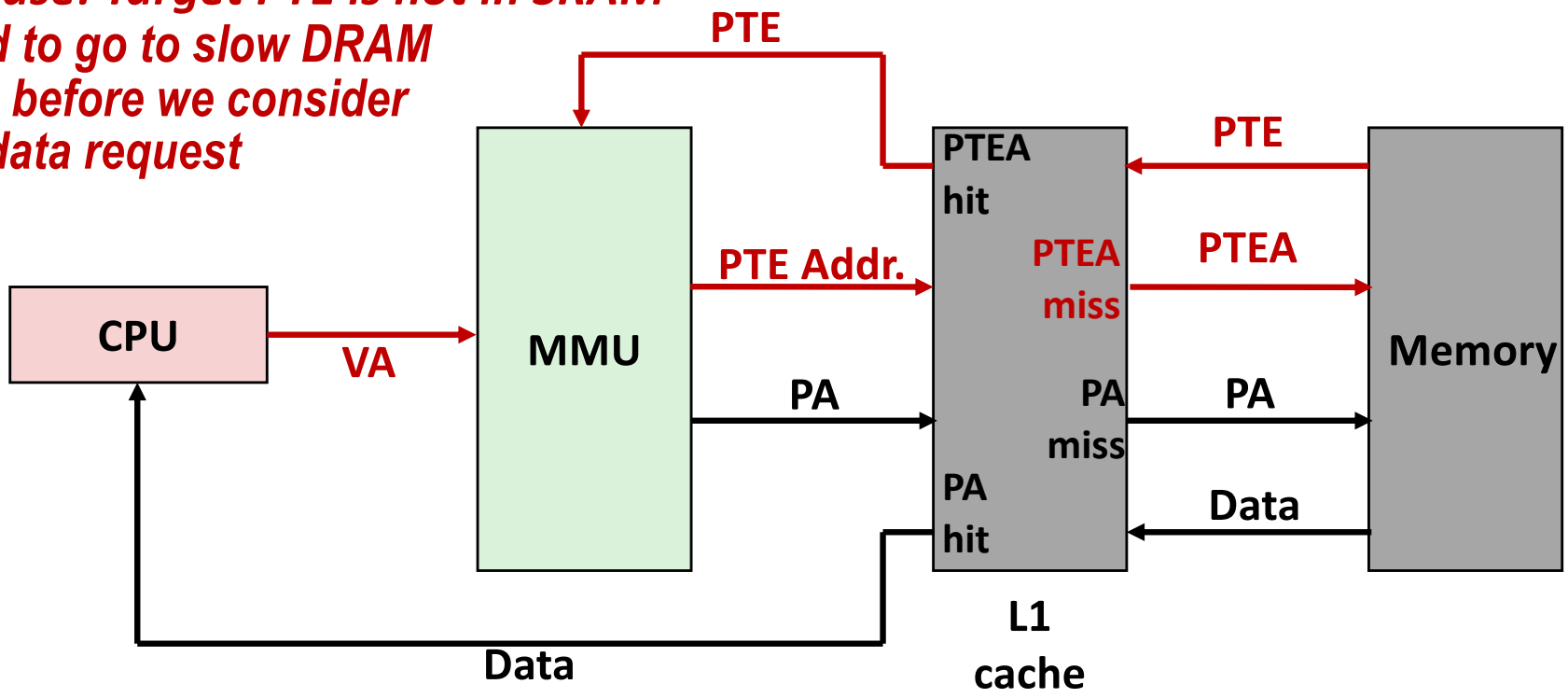
Fast case: Target PTE is cached
- on a PTE hit, no need to go to slow DRAM to resolve address
(may still need to for data)



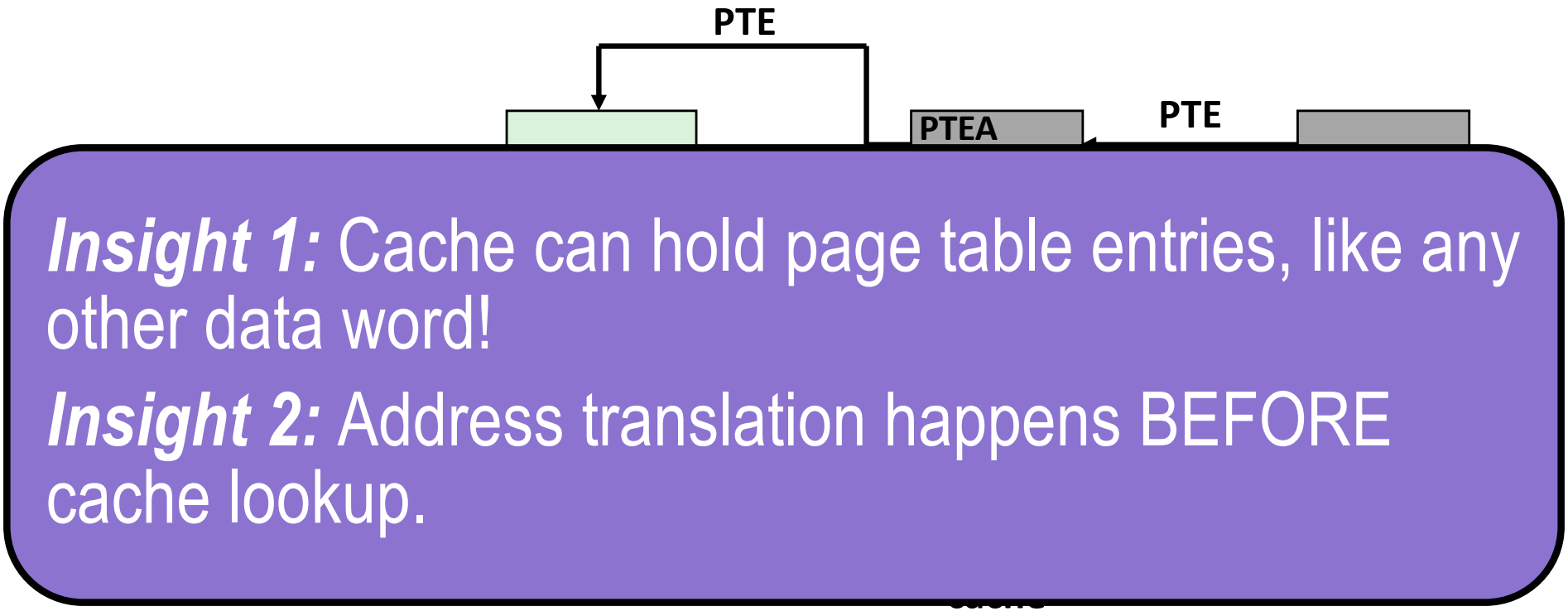
VA: virtual address, PA: physical address,
PTE: page table entry, PTEA = PTE address

Integrating VM and Cache

Slow case: Target PTE is not in SRAM
- Need to go to slow DRAM even before we consider our data request



Integrating VM and Cache



Insight 1: Cache can hold page table entries, like any other data word!

Insight 2: Address translation happens BEFORE cache lookup.

VA: virtual address, PA: physical address,
PTE: page table entry, PTEA = PTE address

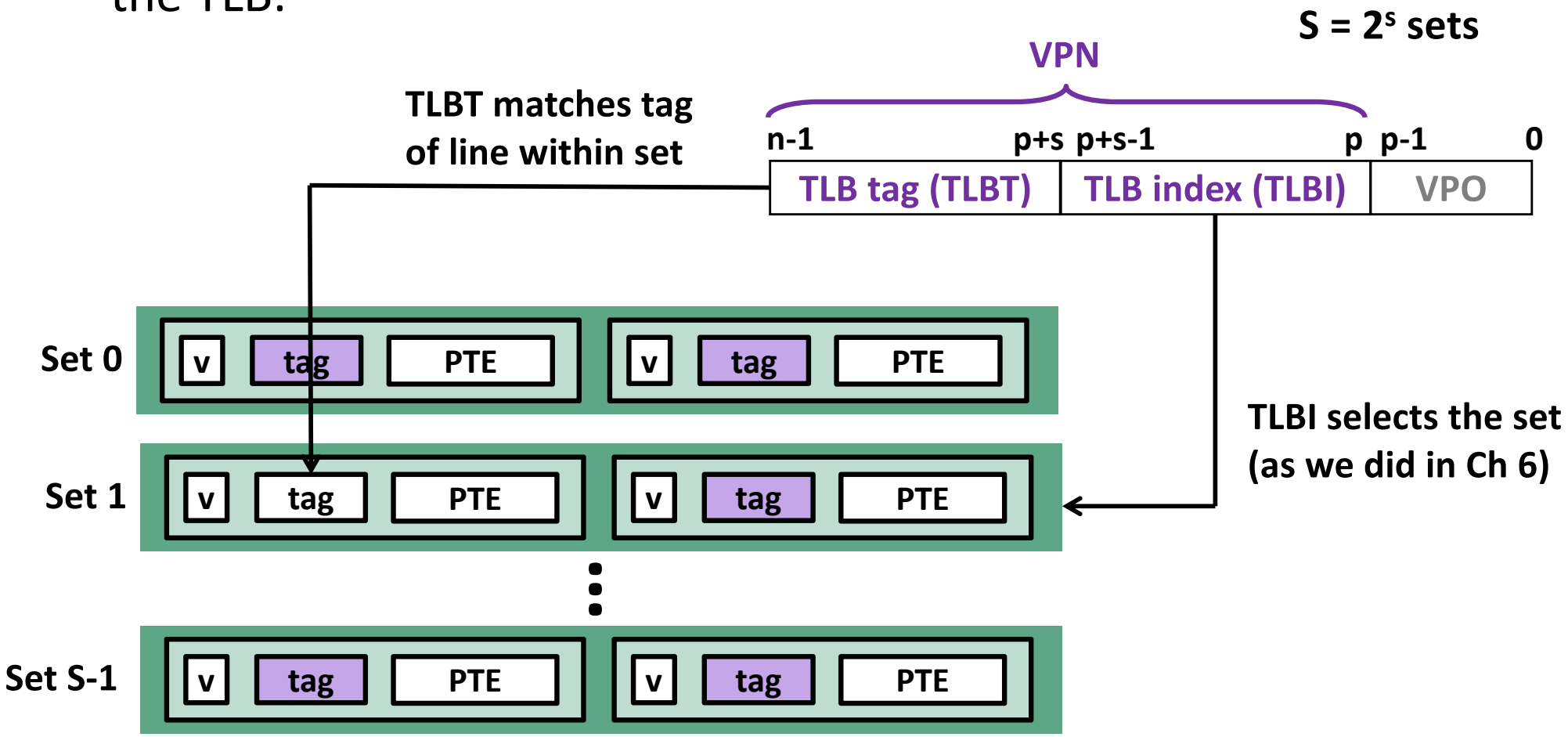
Speeding up Translation with a TLB

- **Problem**: Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - Even a PTE hit pays a small L1 delay
- **Solution**: *Translation Lookaside Buffer* (TLB)
  - Small set-associative **hardware cache** *in* MMU (part of CPU chip)
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for a small number of pages
  - Each cache line holds one block consisting of a single PTE

Similar to Instruction and Data cache separation.

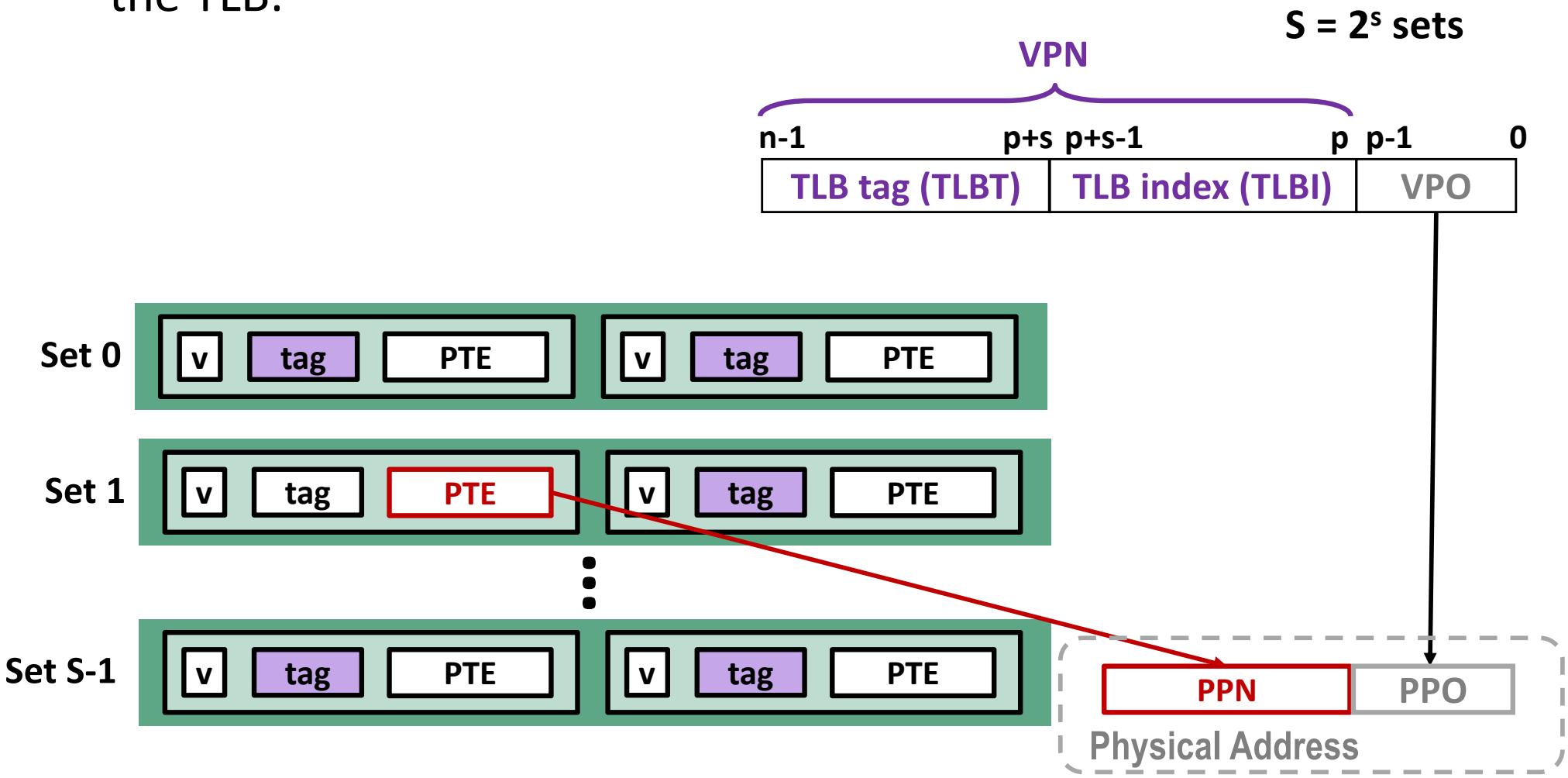
Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB:

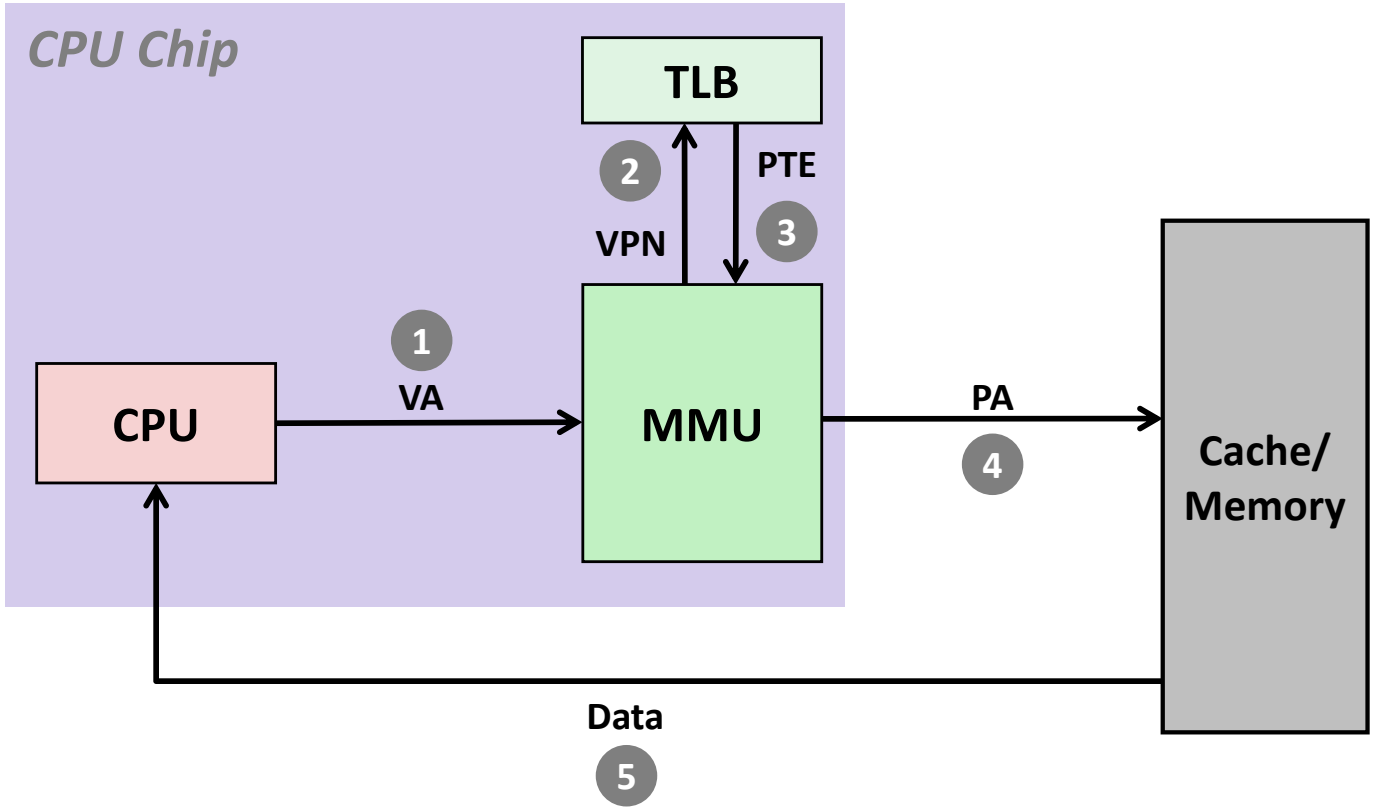


Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB:



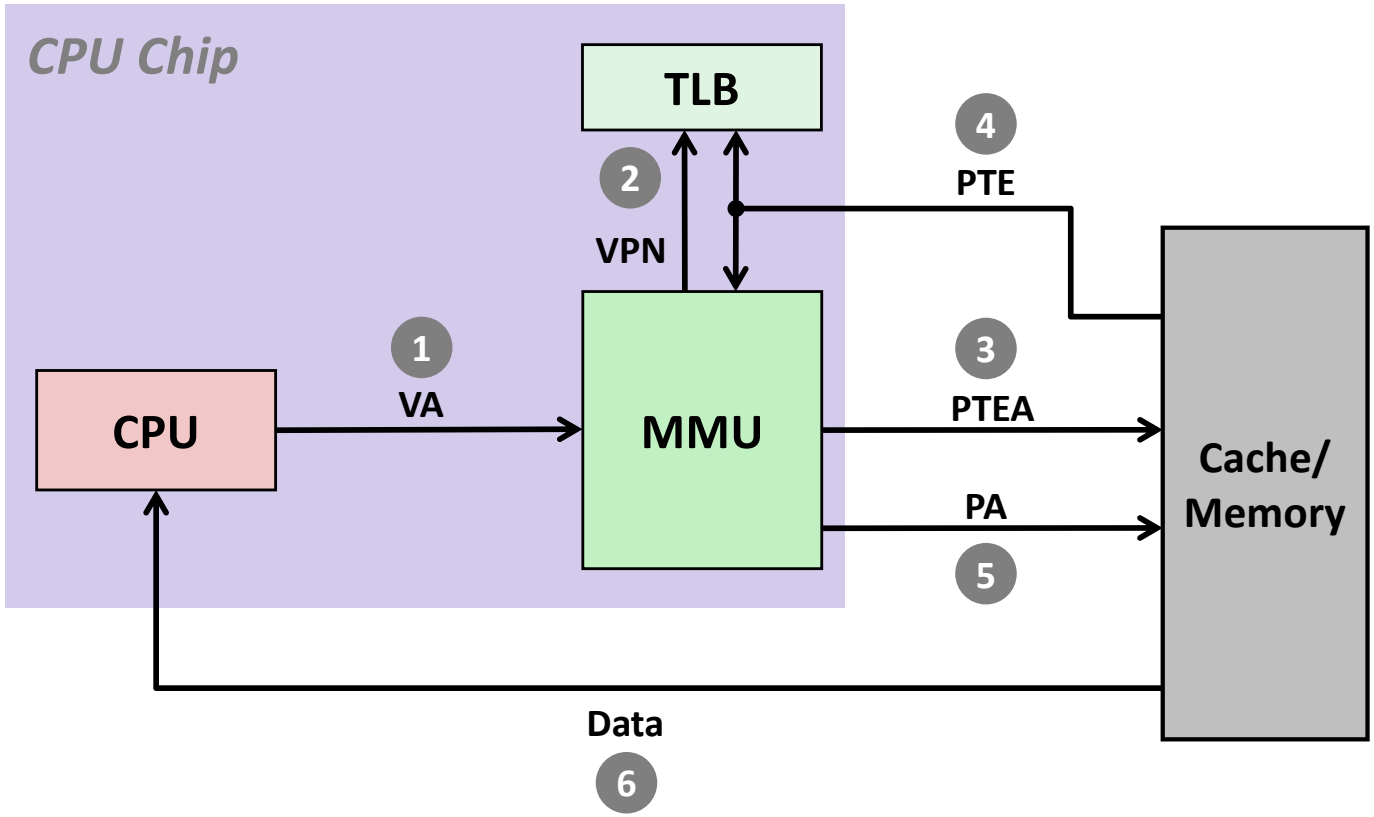
TLB Hit



A TLB hit eliminates a memory access.

All steps in address *translation* happen inside MMU and are fast.

TLB Miss



A TLB miss incurs an additional memory access (the PTE).

Fortunately, TLB misses are rare. Thanks, locality!

TLB Fun Facts

- May have separate instruction and data TLBs
- May have multiple levels of TLBs
- Who loads the TLB with entries?
  - Hardware-managed:
    - Page table walkers walk page table and update TLB
  - Software-managed TLB:
    - On TLB miss, OS walks page tables and loads TLB

Practice on Your Own

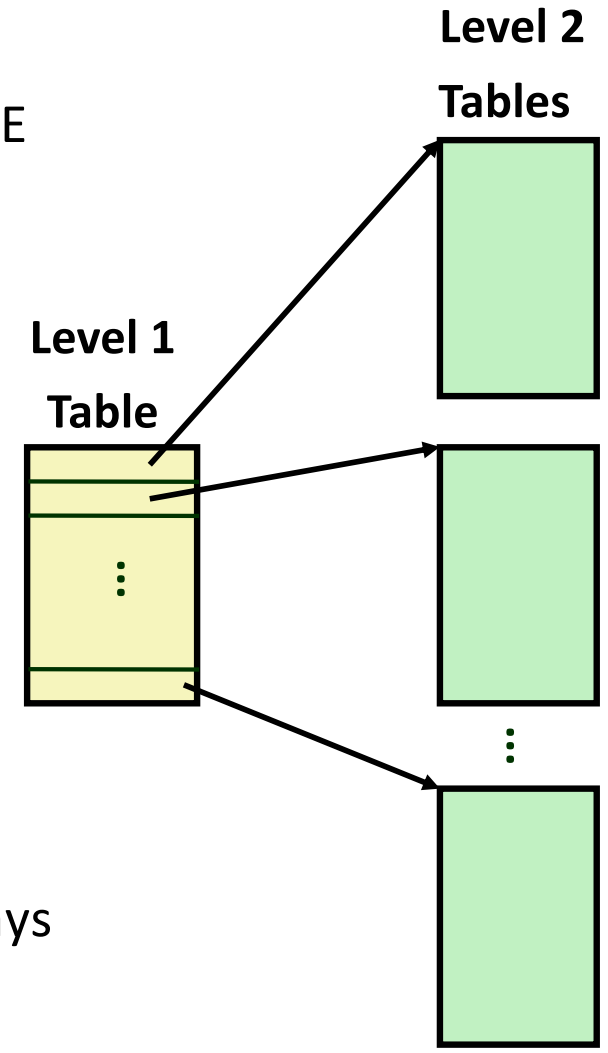
- How many times might a TLB need to be accessed when executing a single instruction (from fetch to write back)?

Page Table Structure

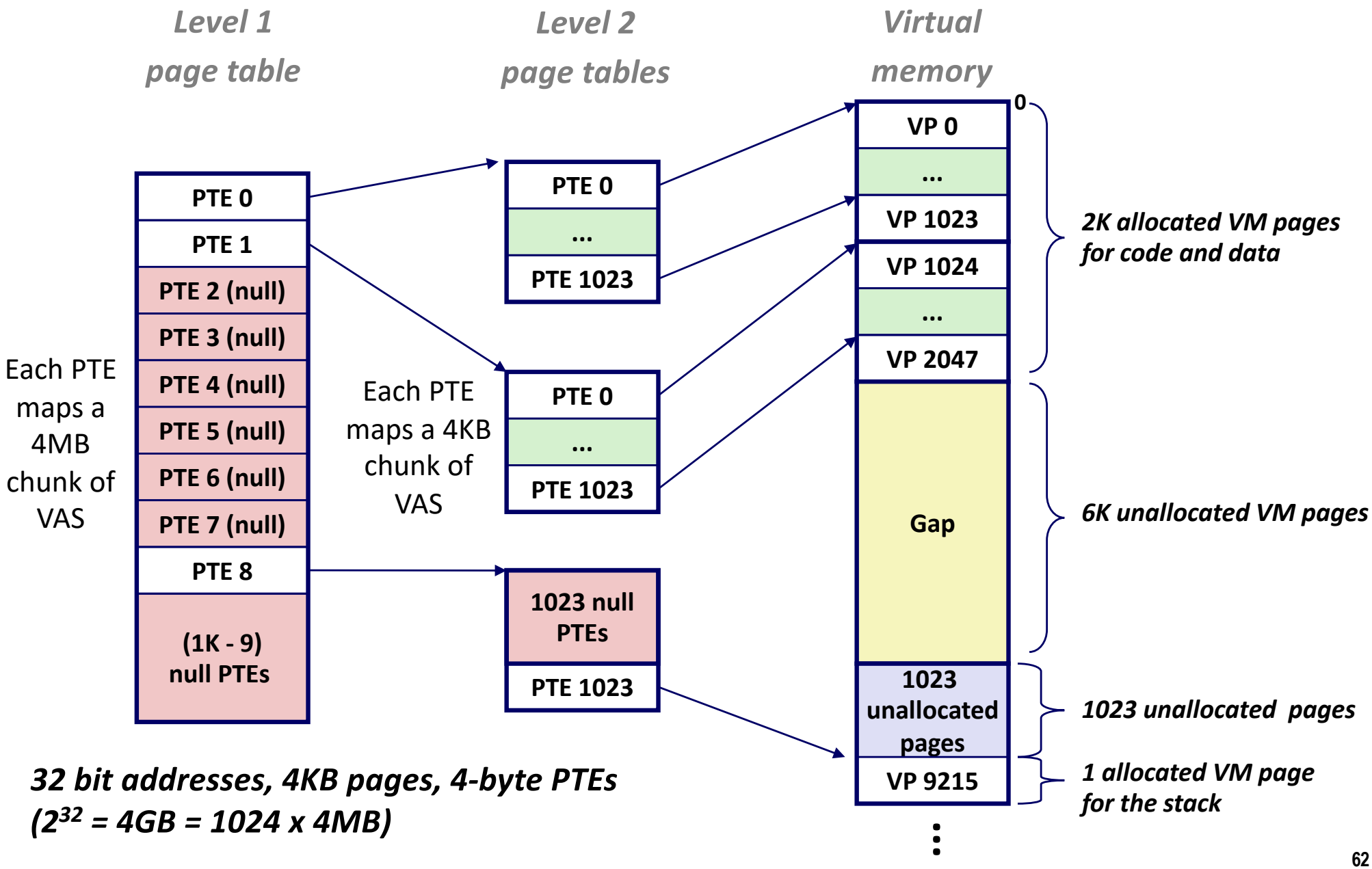
- Recall: A control register (CR3) holds the starting address of a process's page table
  - How big is a process's page table?
    - Size of a PTE (what is stored)?
    - Number of PTEs?
  - How big is a process's working set (roughly speaking)?
    - Stack size?
    - Heap size?
    - Code/text?
- Observations:
  - Page table is HUGE, but sparsely populated
- What data structures might we use to represent our page table?

Multi-Level Page Tables: Concrete Example

- Suppose:
  - 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE
- Problem?
  - Would need a 512 GB page table!
    - $2^{48} / 2^{12} = 2^{36}$ = # entries in page table
    - 2^3 bytes per entry
    - $2^{48} * 2^{-12} * 2^3 = 2^{39}$ bytes in every page table
- Common solution: Multi-level page tables
  - No need to waste space for unallocated pages!
- Example: 2-level page table
  - Level 1 table: each PTE points to a page table (always memory resident)
  - Level 2 table: each PTE points to a page (paged in and out like any other data)



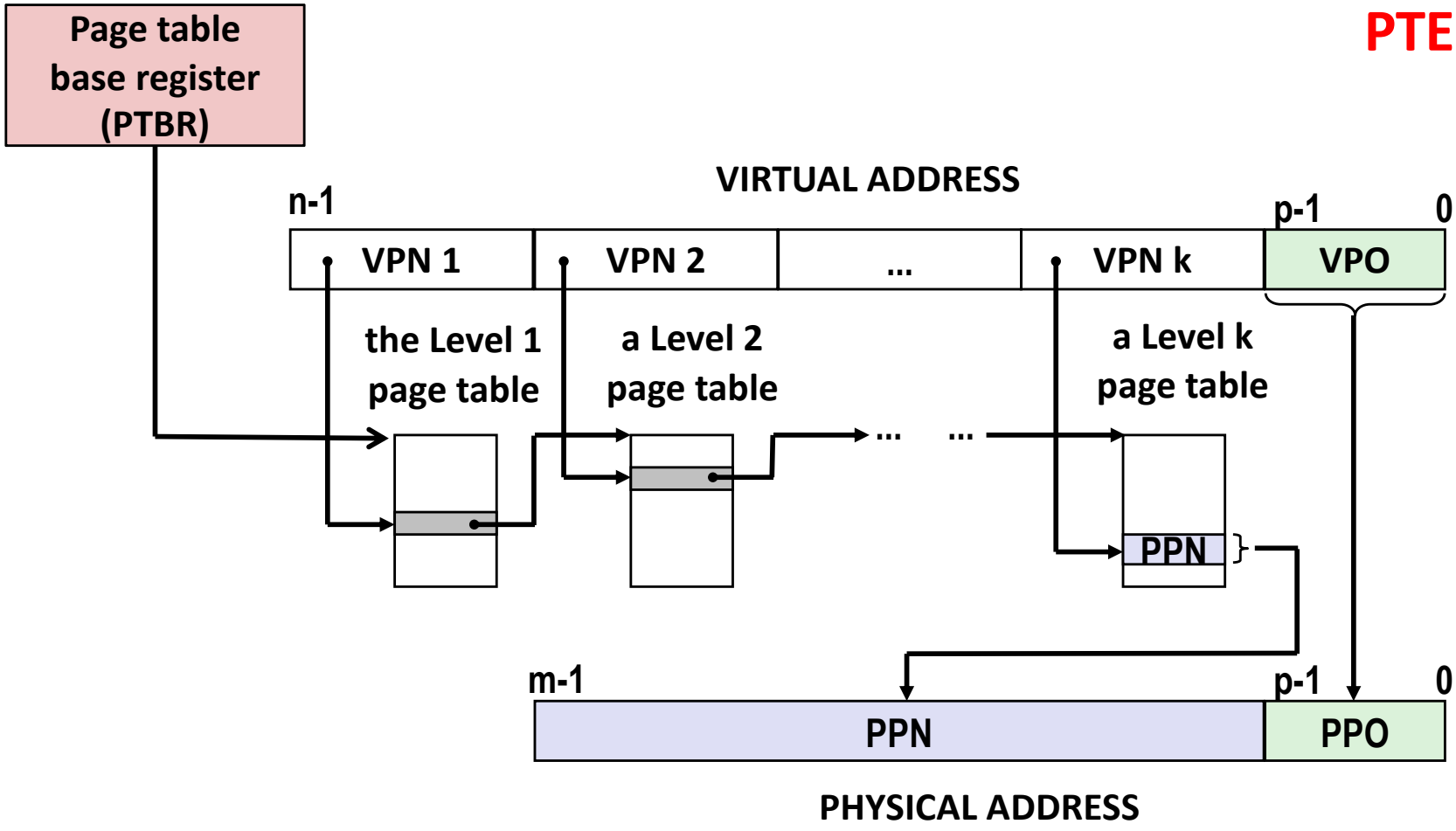
A Two-Level Page Table Hierarchy



32 bit addresses, 4KB pages, 4-byte PTEs
(2^{32} = 4GB = 1024 x 4MB)

Translating with a k-level Page Table

TLB caches PTEs from all levels



Address Translation Summary (Ch 9.6)

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

- System's view of virtual memory:
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient inter-positioning point to check permissions