

# Virtual Memory: Concepts

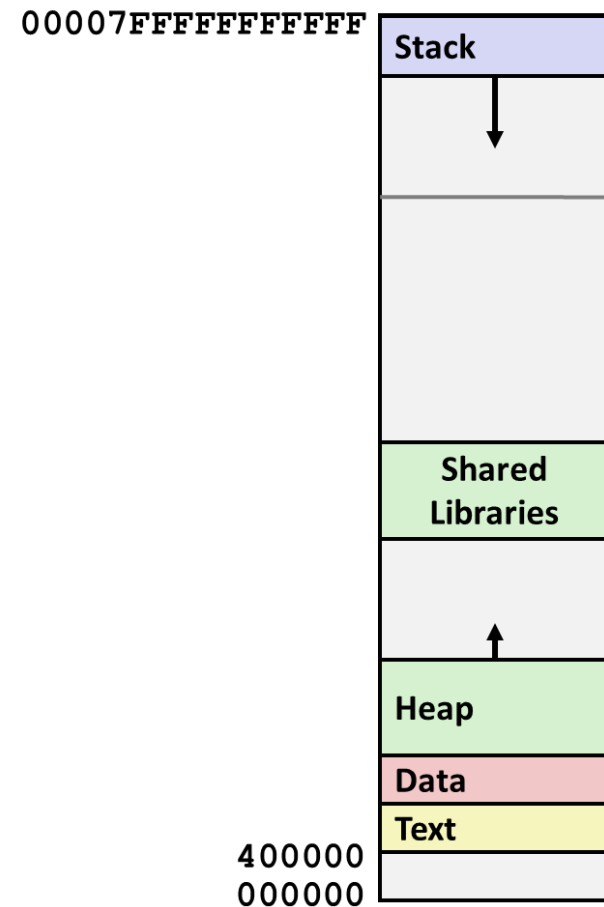
15-213/15-513: Introduction to Computer Systems  
16<sup>th</sup> Lecture, June 13, 2024

## Instructors:

Brian Railing

# This Picture is a Lie

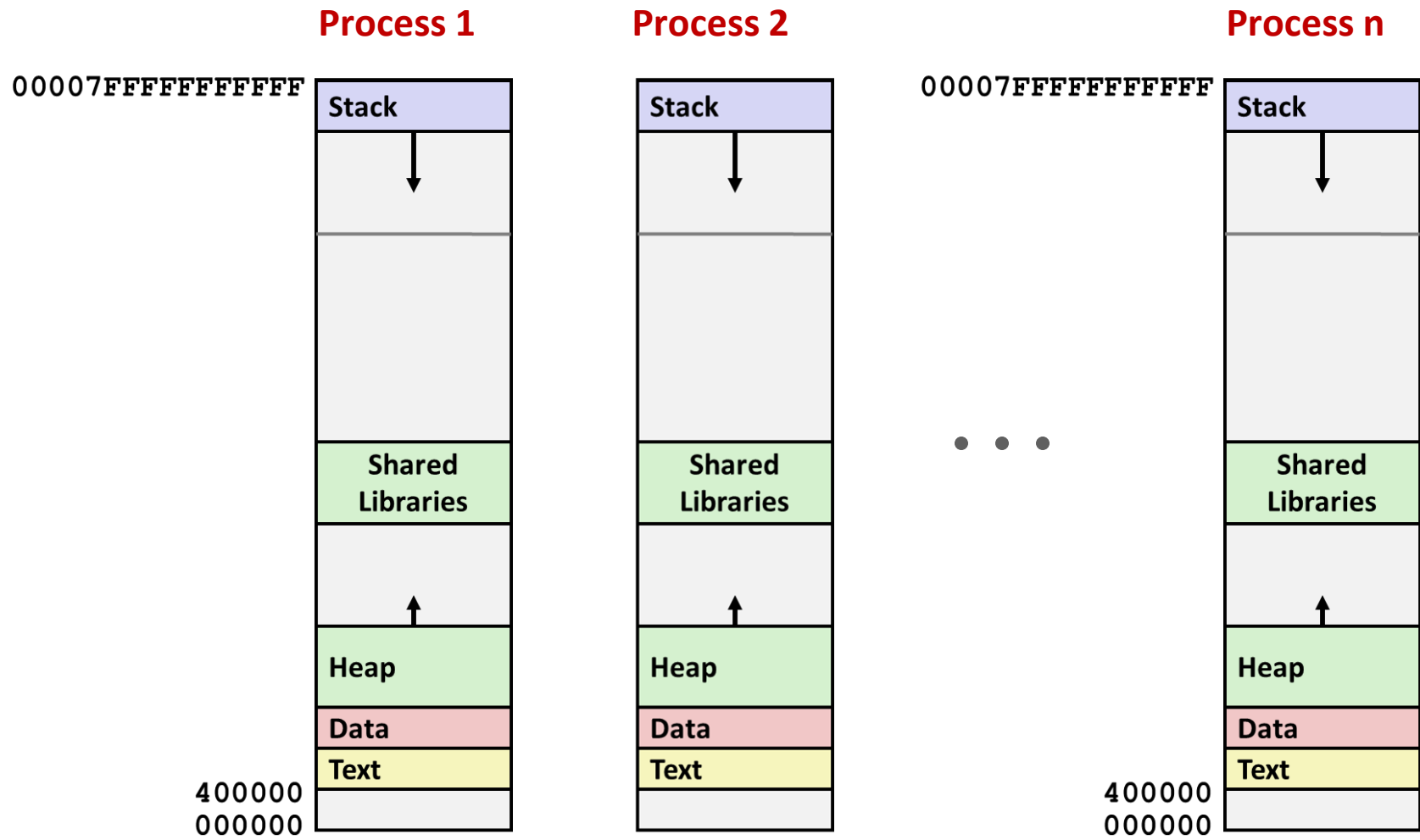
- This is RAM, we said...
- But the computer can run more than one program at a time!
- Where are all the other programs?
- *Let's investigate.*



# Processes (Teaser for July)

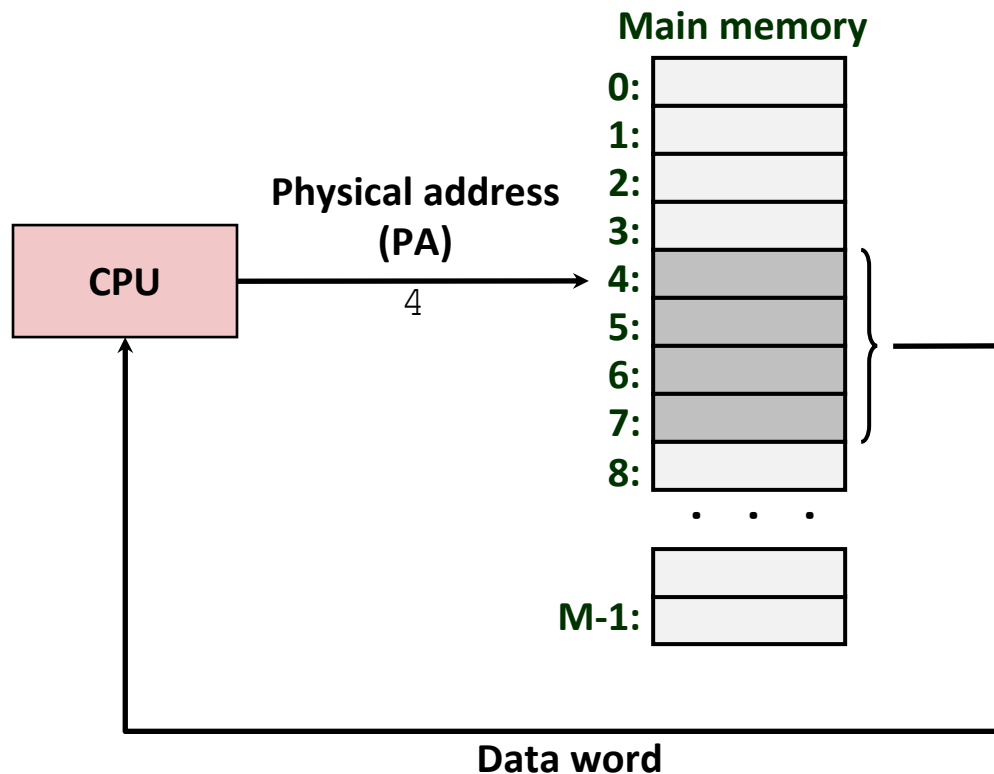
- **Definition:** A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”
- **Unix:** A *parent process* creates a new *child process* by calling `fork`
  - Child is (sort of) a copy of the parent
  - `fork` returns *twice*—once in each process
    - Different return value in each
- **Parent can wait for child to finish by calling `waitpid`**
  - For now, think of this as “what `main` returns to”

# Hmmm, How Does This Work?!



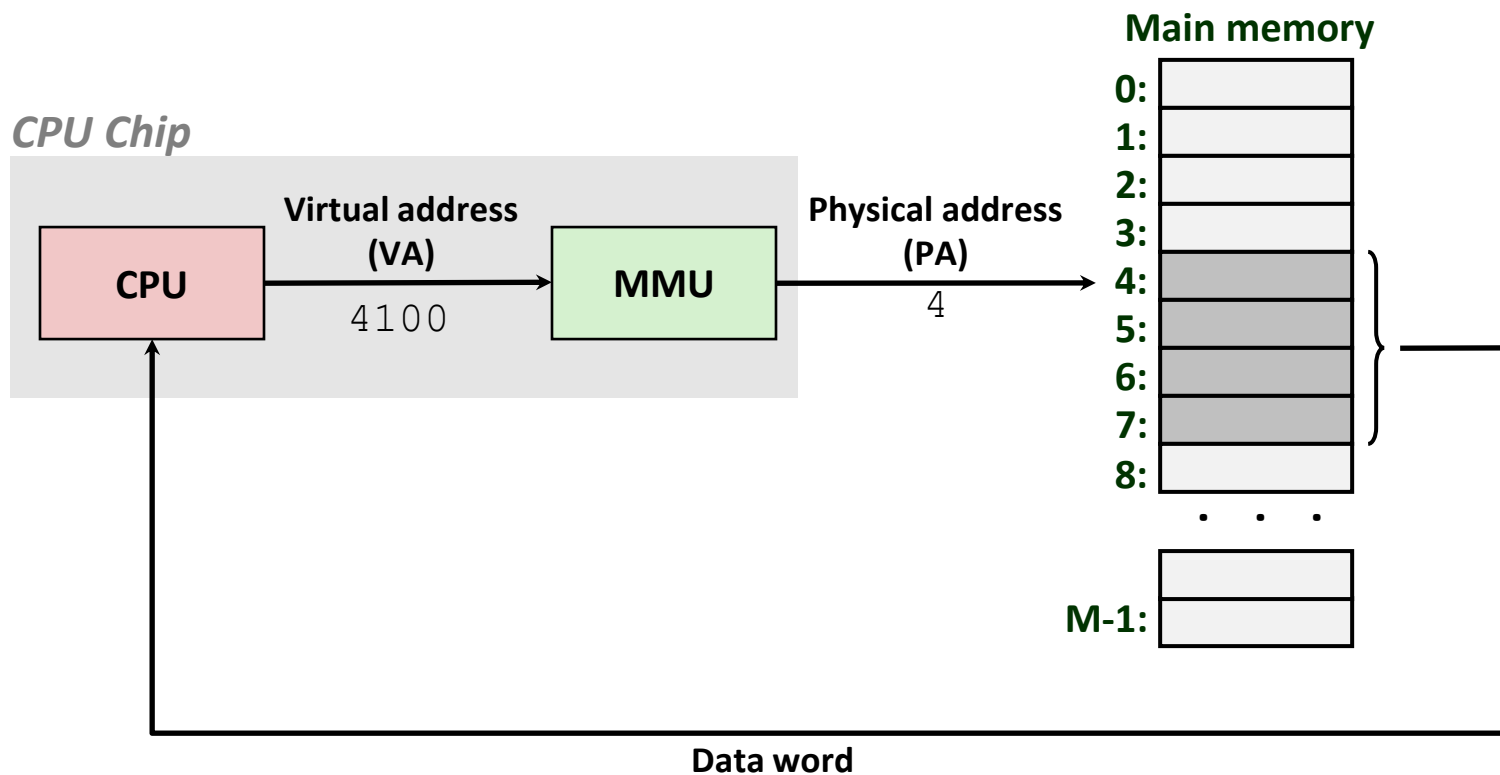
***Solution: Virtual Memory (today and next lecture)***

# 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

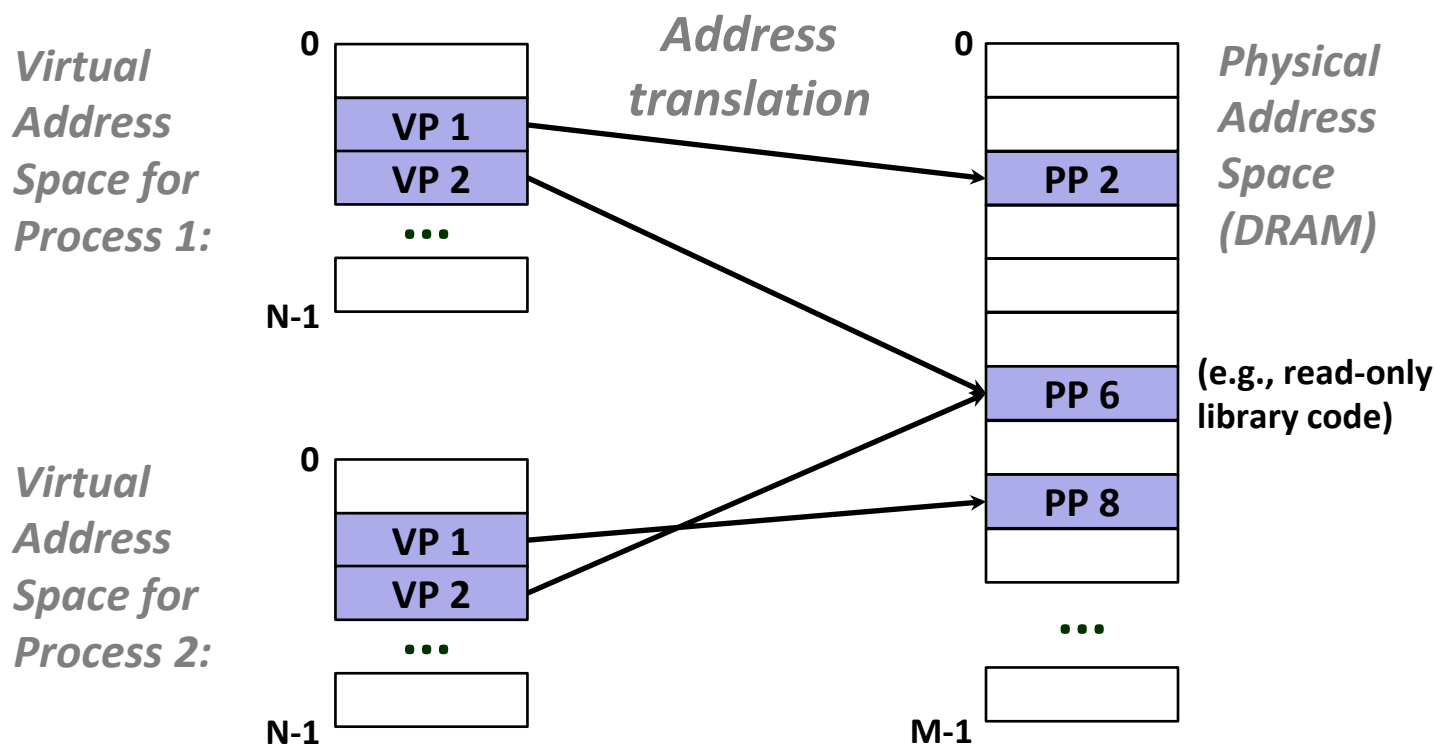


- Used in all modern servers, laptops, and smart phones
- One of the great ideas in computer science

# VM as a Tool for Memory Management

## ■ Key idea: each process has its own virtual address space

- It can view memory as a simple linear array
- Mapping function scatters addresses through physical memory
  - Well-chosen mappings can improve locality



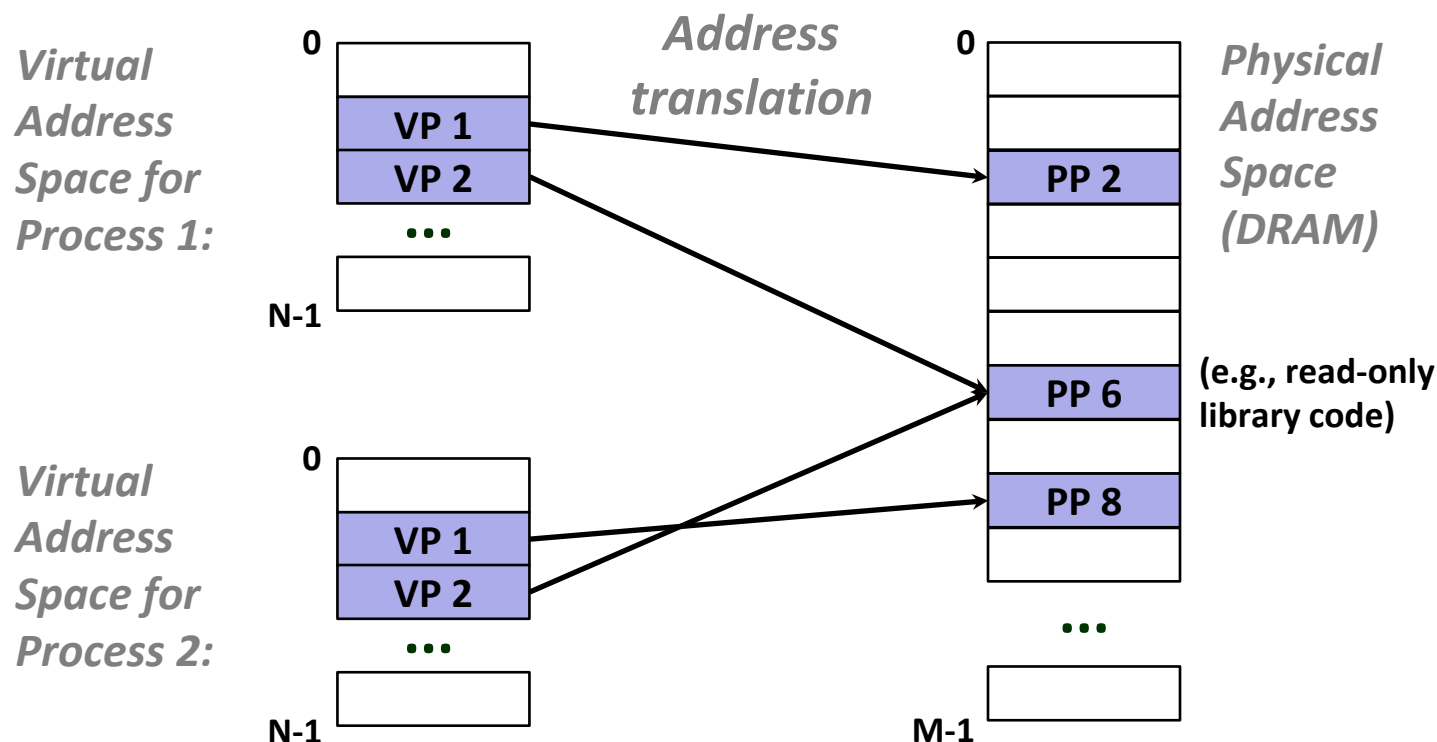
# VM as a Tool for Memory Management

## ■ Simplifying 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

## ■ Sharing code and data among processes

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





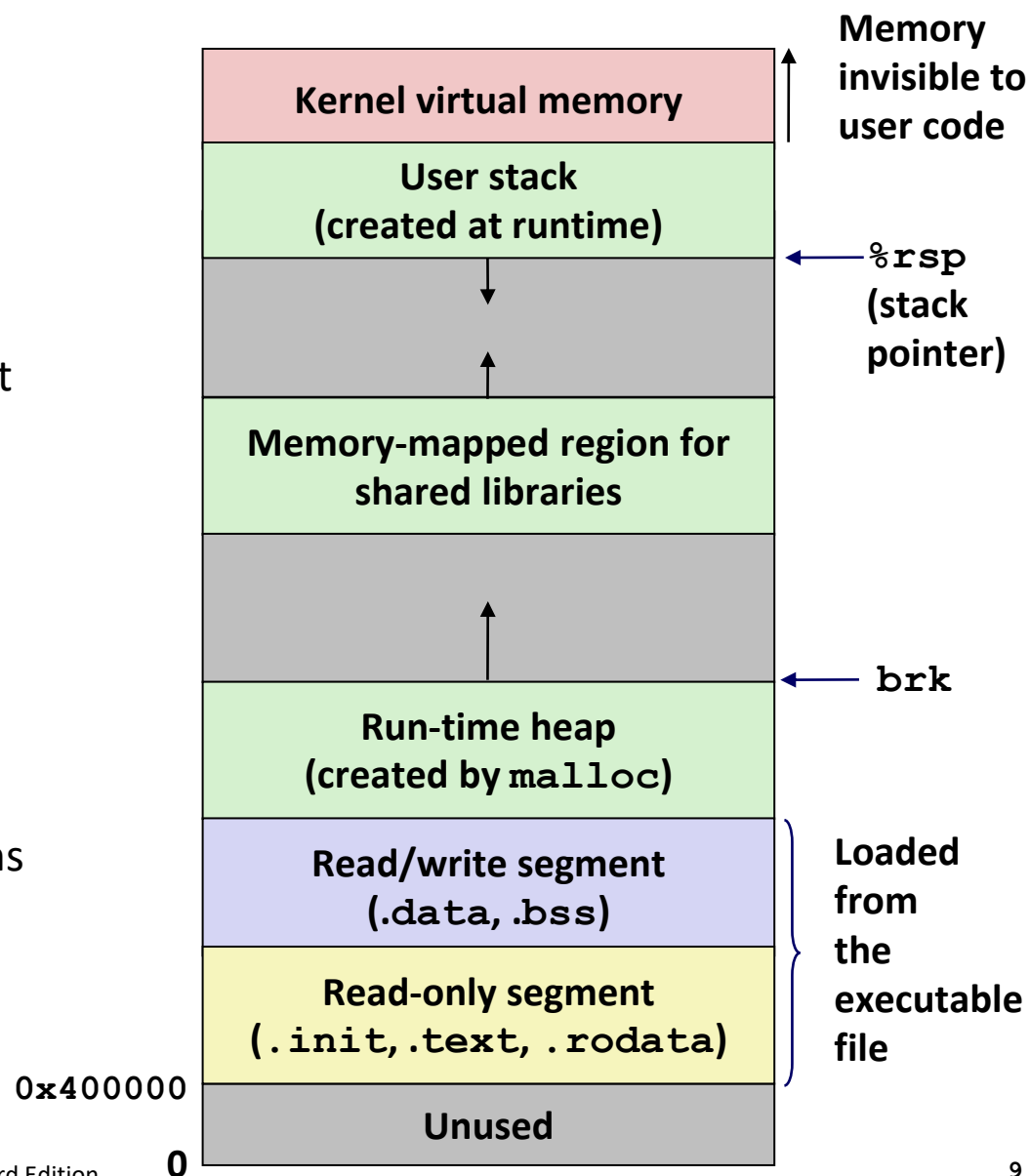
# Simplifying Linking and Loading

## ■ 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



# Address Spaces

- **Linear address space:** Ordered set of contiguous non-negative integer addresses:

$\{0, 1, 2, 3 \dots\}$

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

$\{0, 1, 2, 3, \dots, N-1\}$

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

$\{0, 1, 2, 3, \dots, M-1\}$

# 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

- One process can't interfere with another's memory
- User program cannot access privileged kernel information and code

# VM Address Translation

## ■ Virtual Address Space

- $V = \{0, 1, \dots, N-1\}$

## ■ Physical Address Space

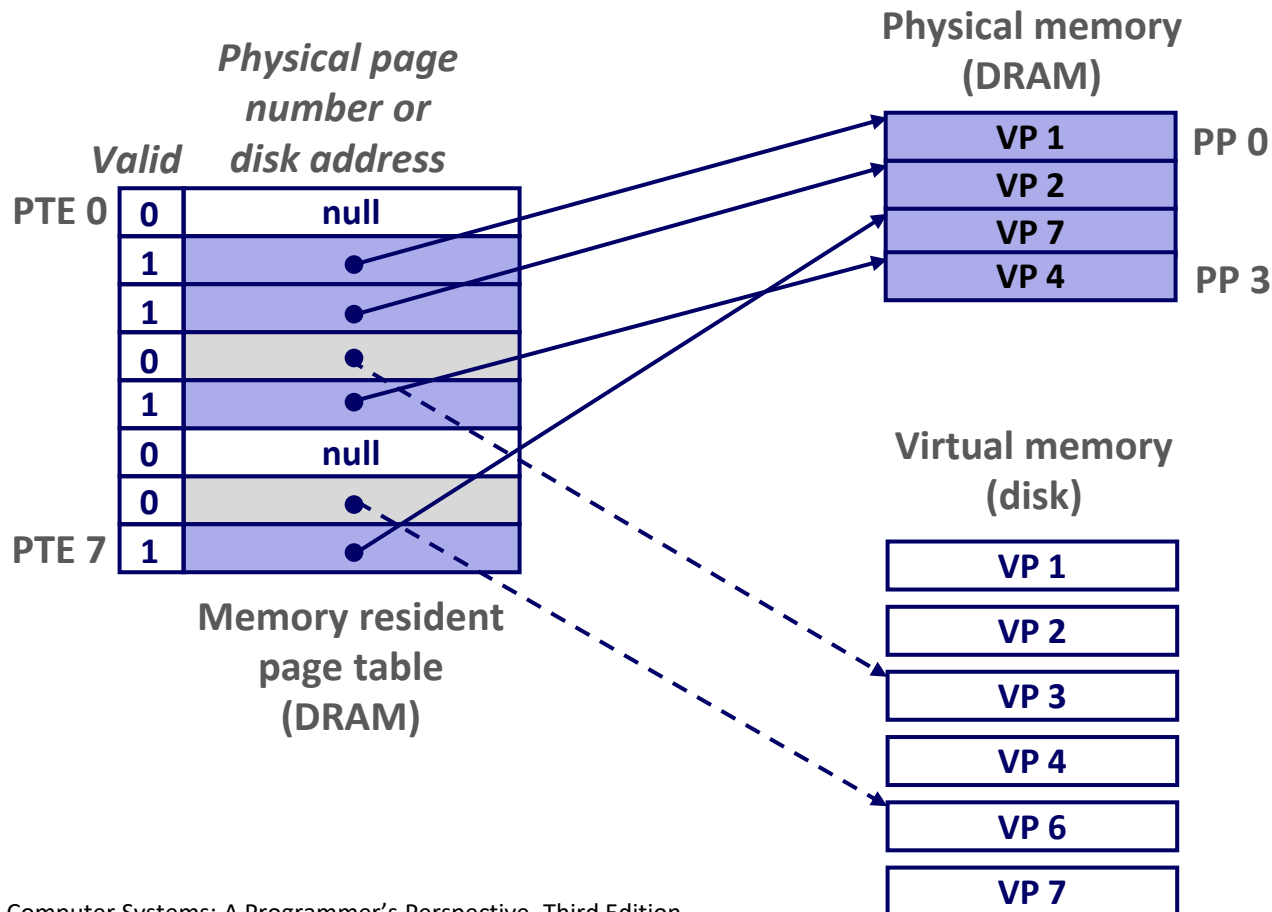
- $P = \{0, 1, \dots, M-1\}$

## ■ Address Translation

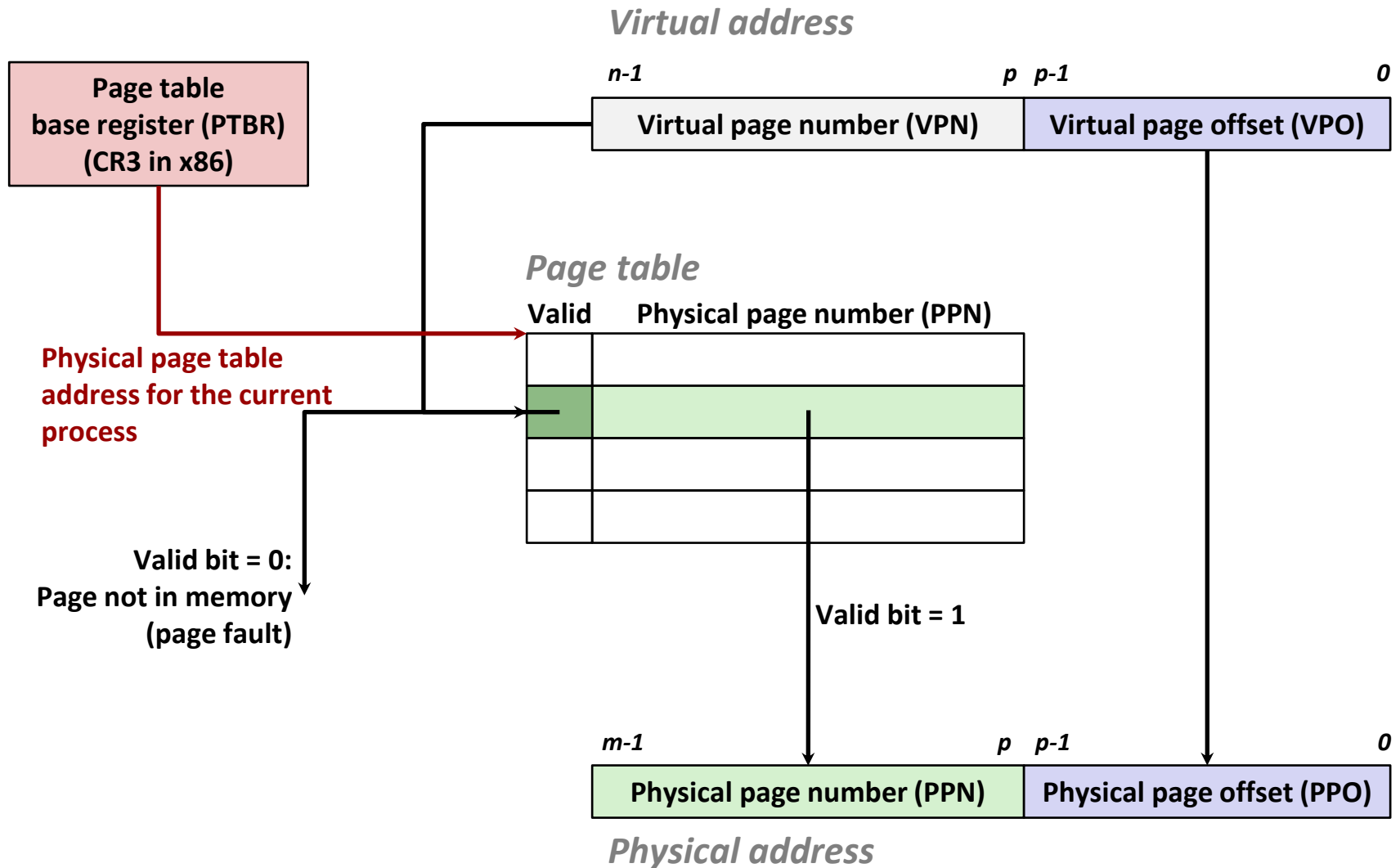
- $MAP: V \rightarrow P \cup \{\emptyset\}$
- For virtual address  $a$ :
  - $MAP(a) = a'$  if data at virtual address  $a$  is at physical address  $a'$  in  $P$
  - $MAP(a) = \emptyset$  if data at virtual address  $a$  is not in physical memory
    - Either invalid or stored on disk

# 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 data structure in DRAM

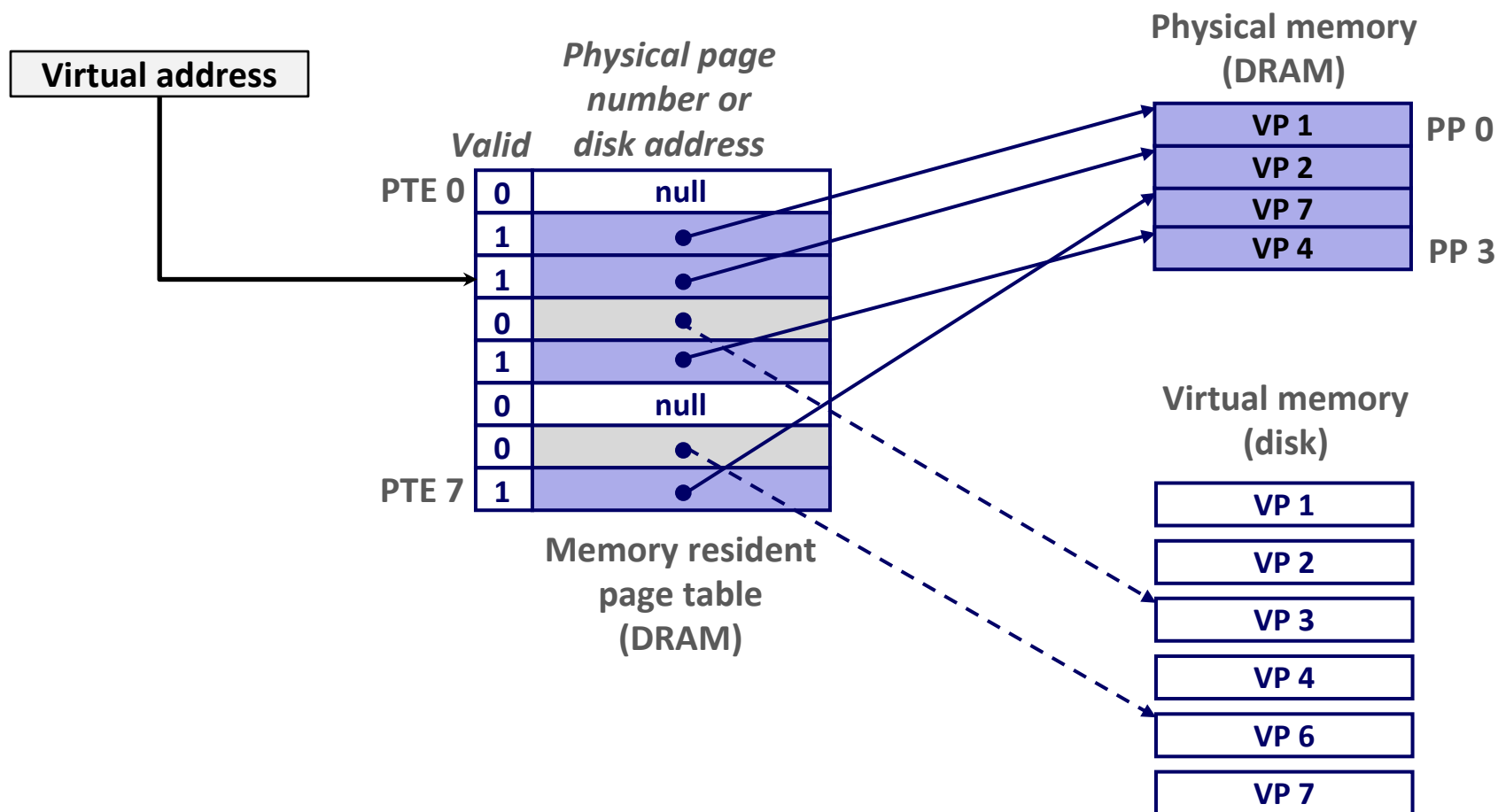


# Address Translation With a Page Table

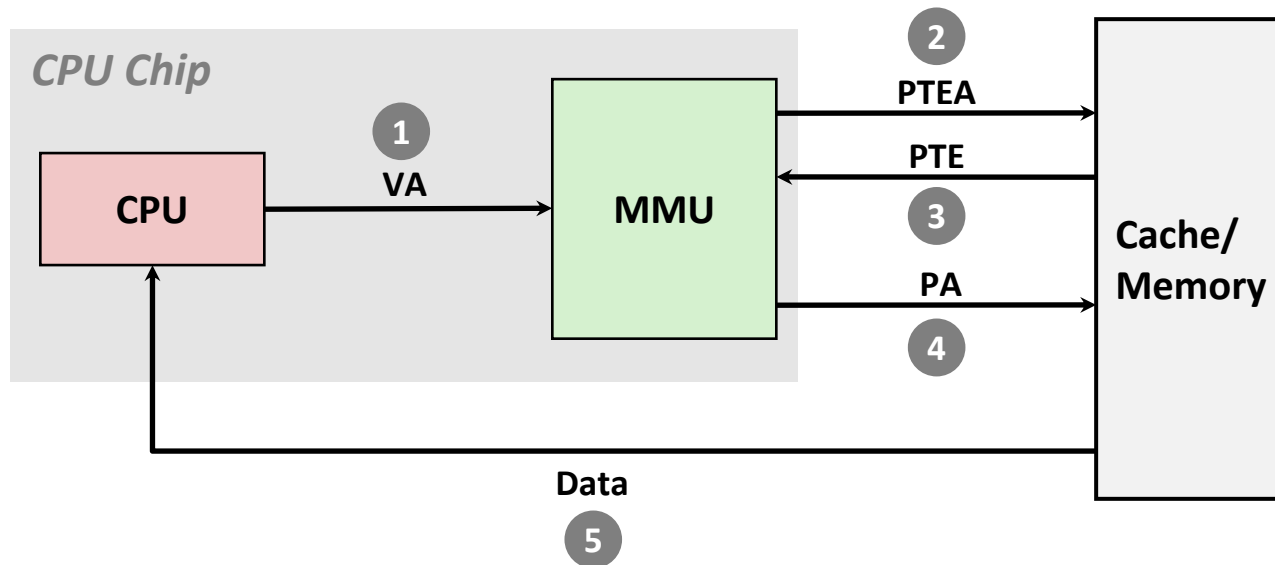


# Page Hit

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



# Address Translation: Page Hit

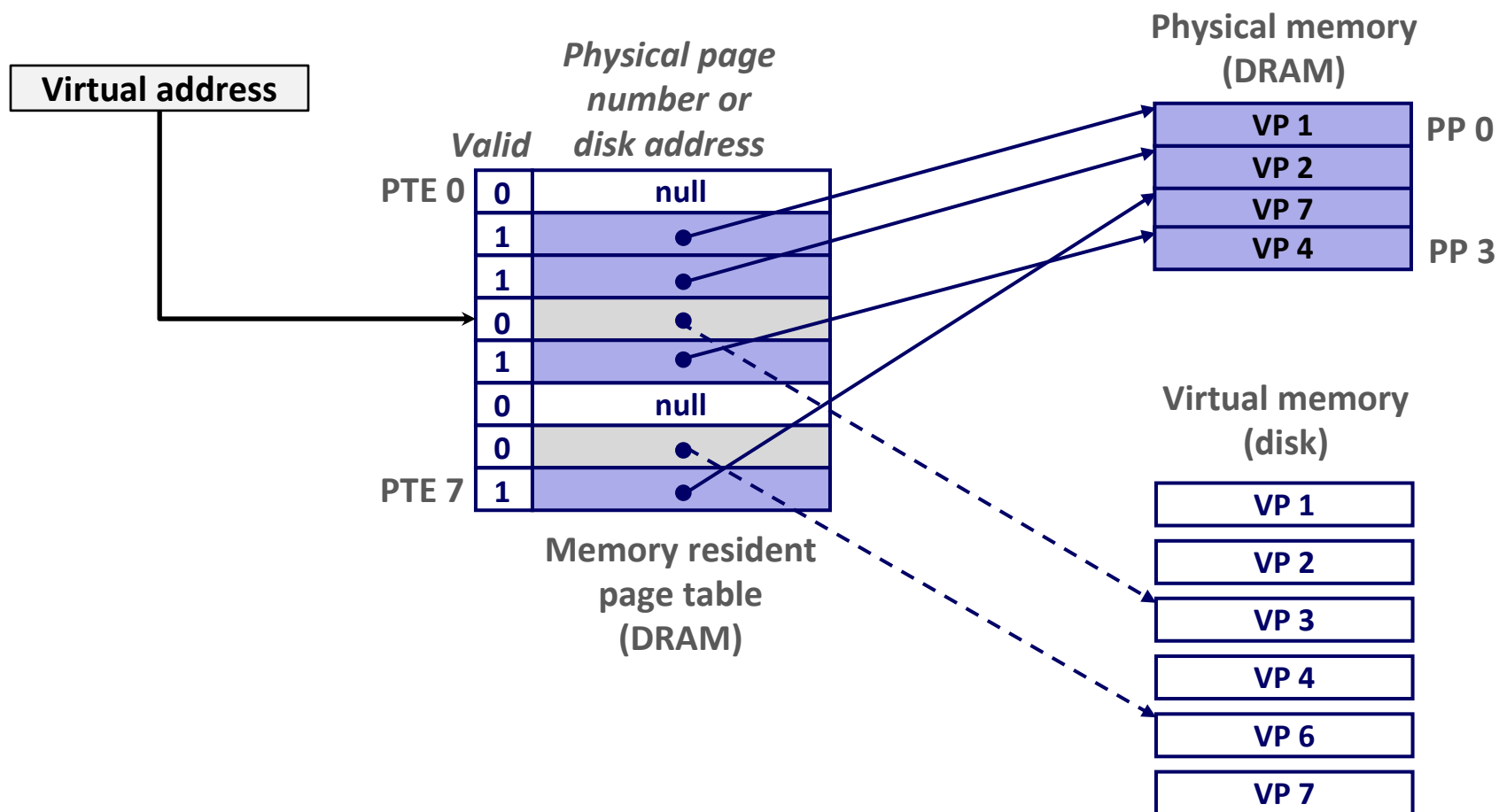


- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

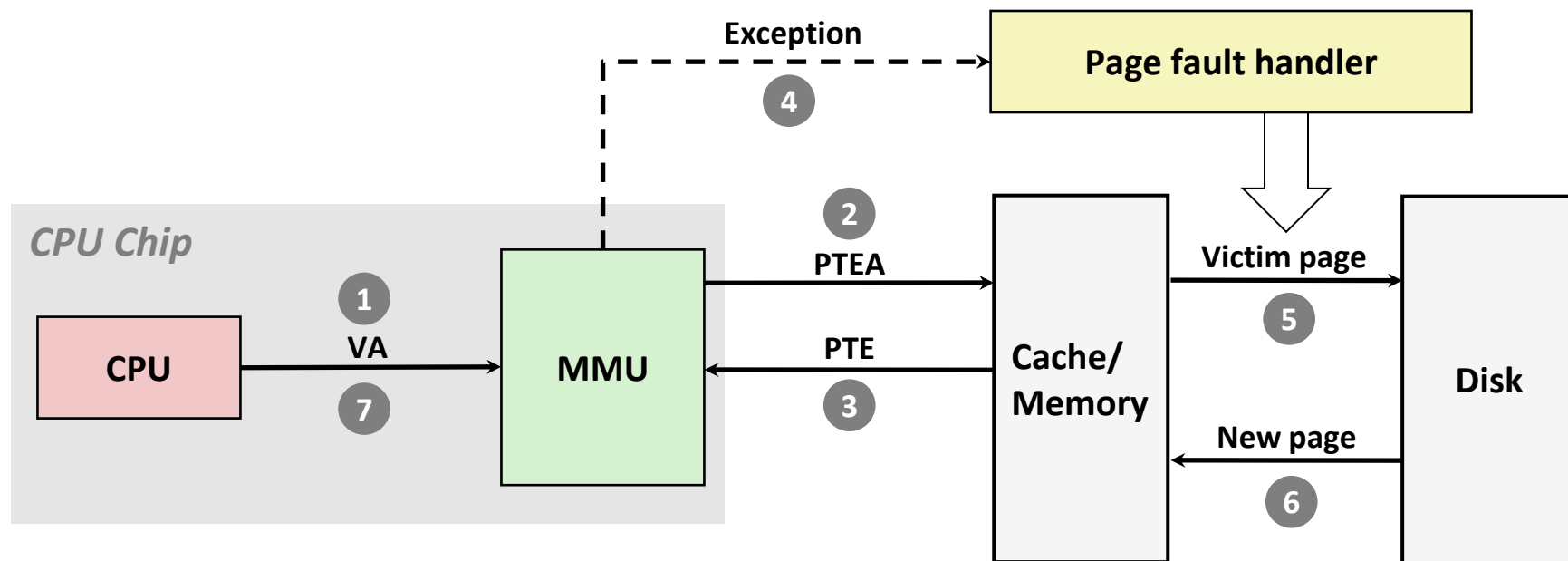


# Page Fault

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



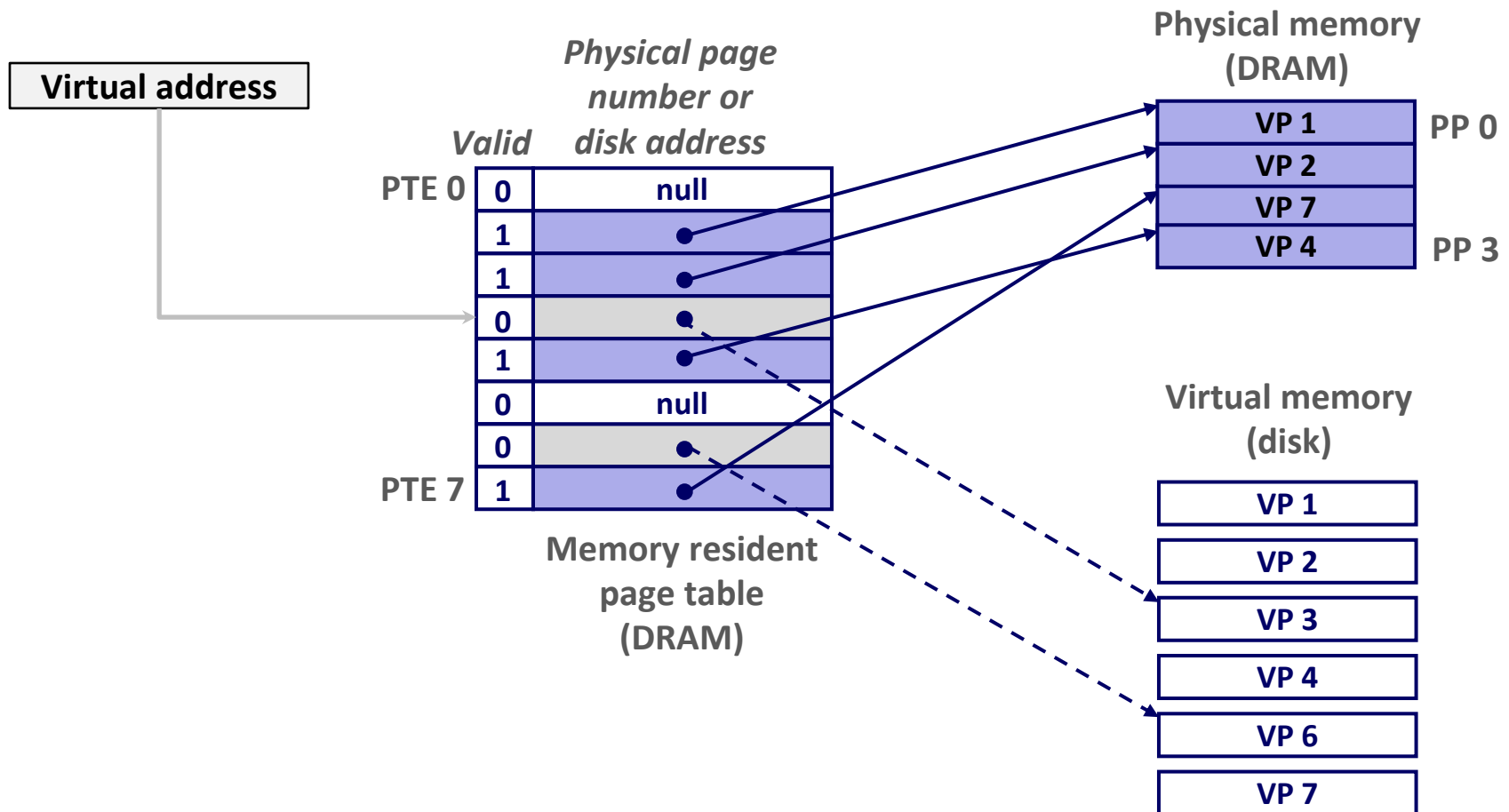
# Address Translation: Page Fault



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in 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

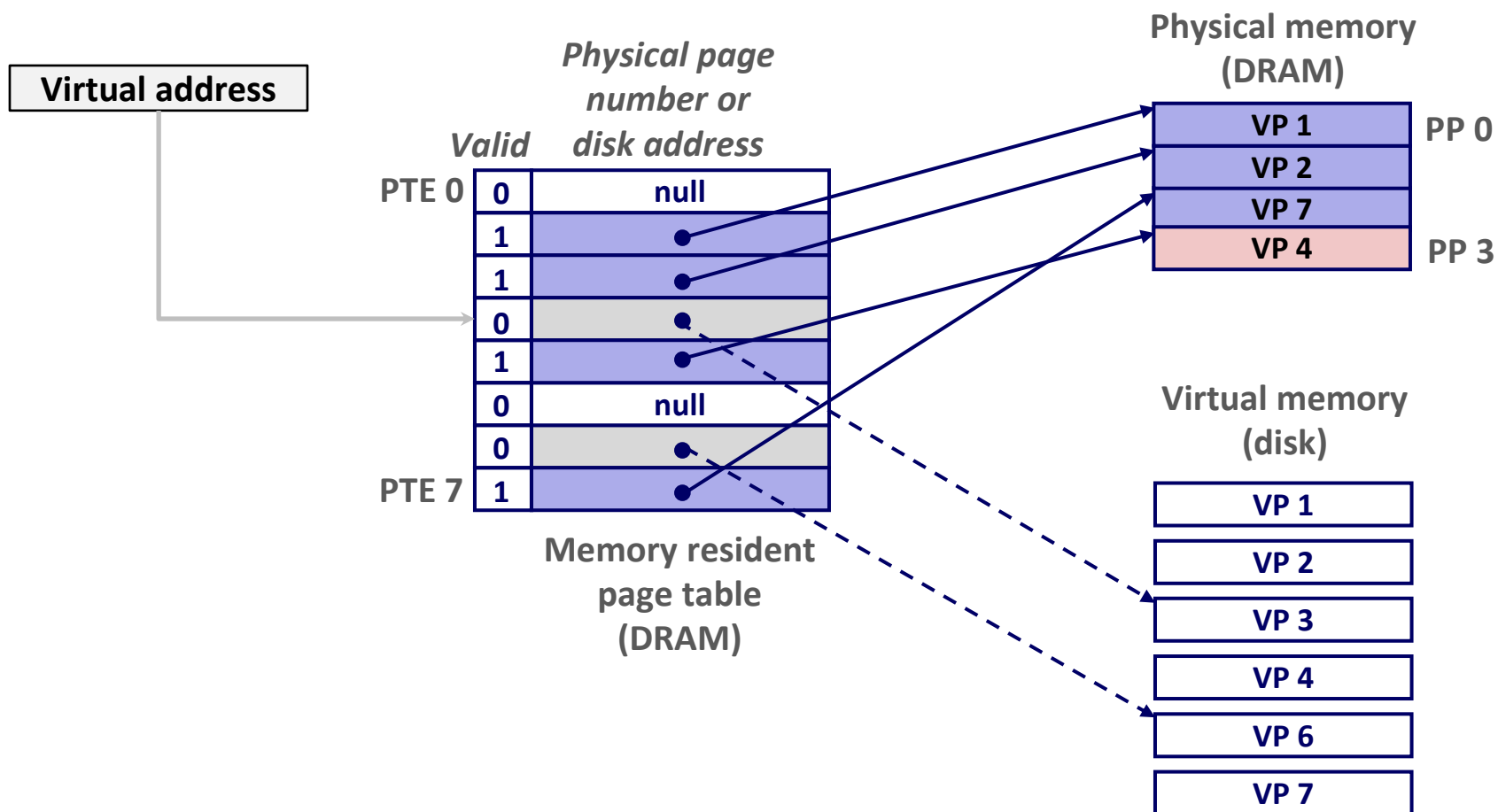
# Handling Page Fault

- Page miss causes page fault (an exception)



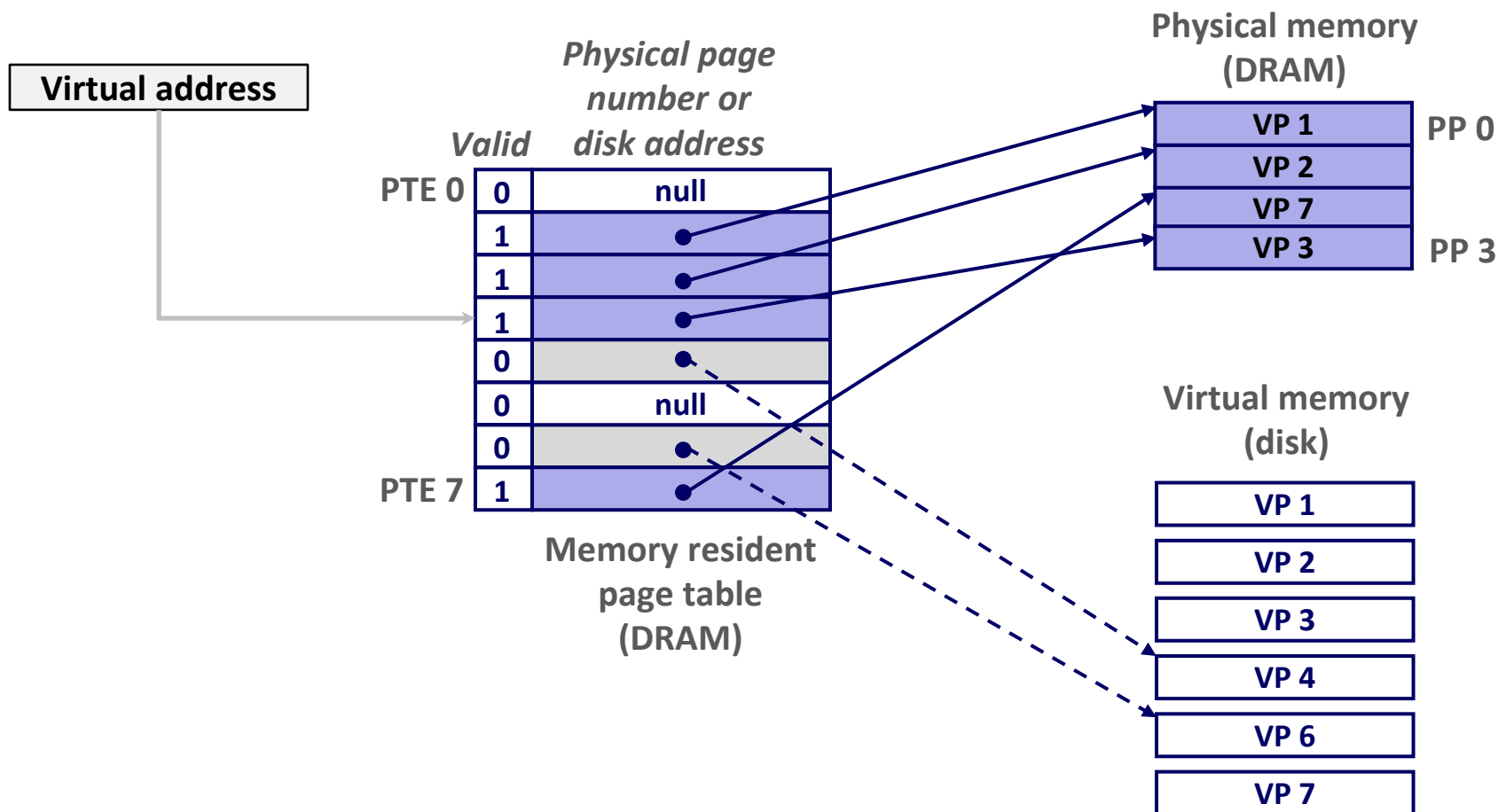
# Handling Page Fault

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



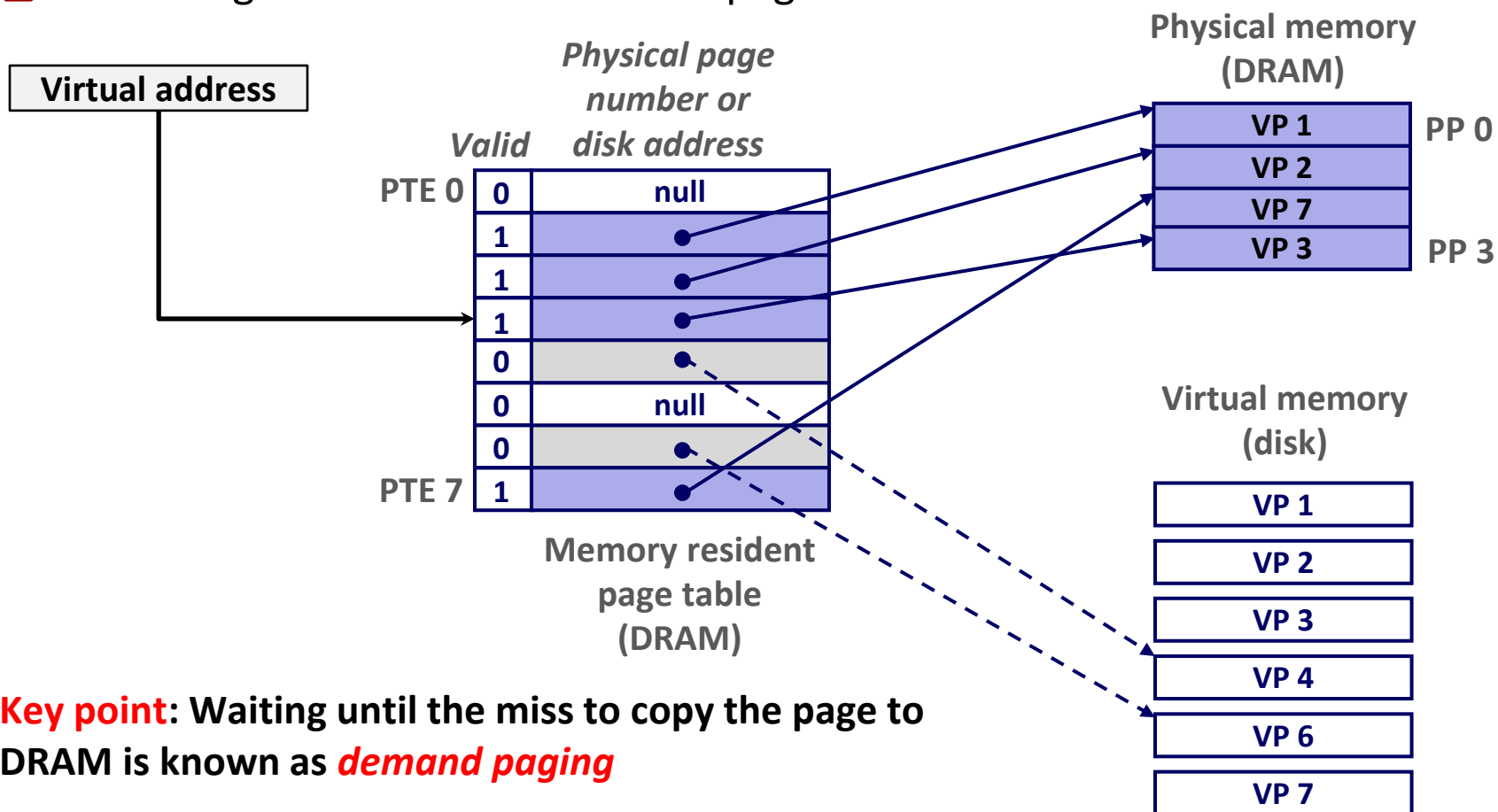
# Handling Page Fault

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



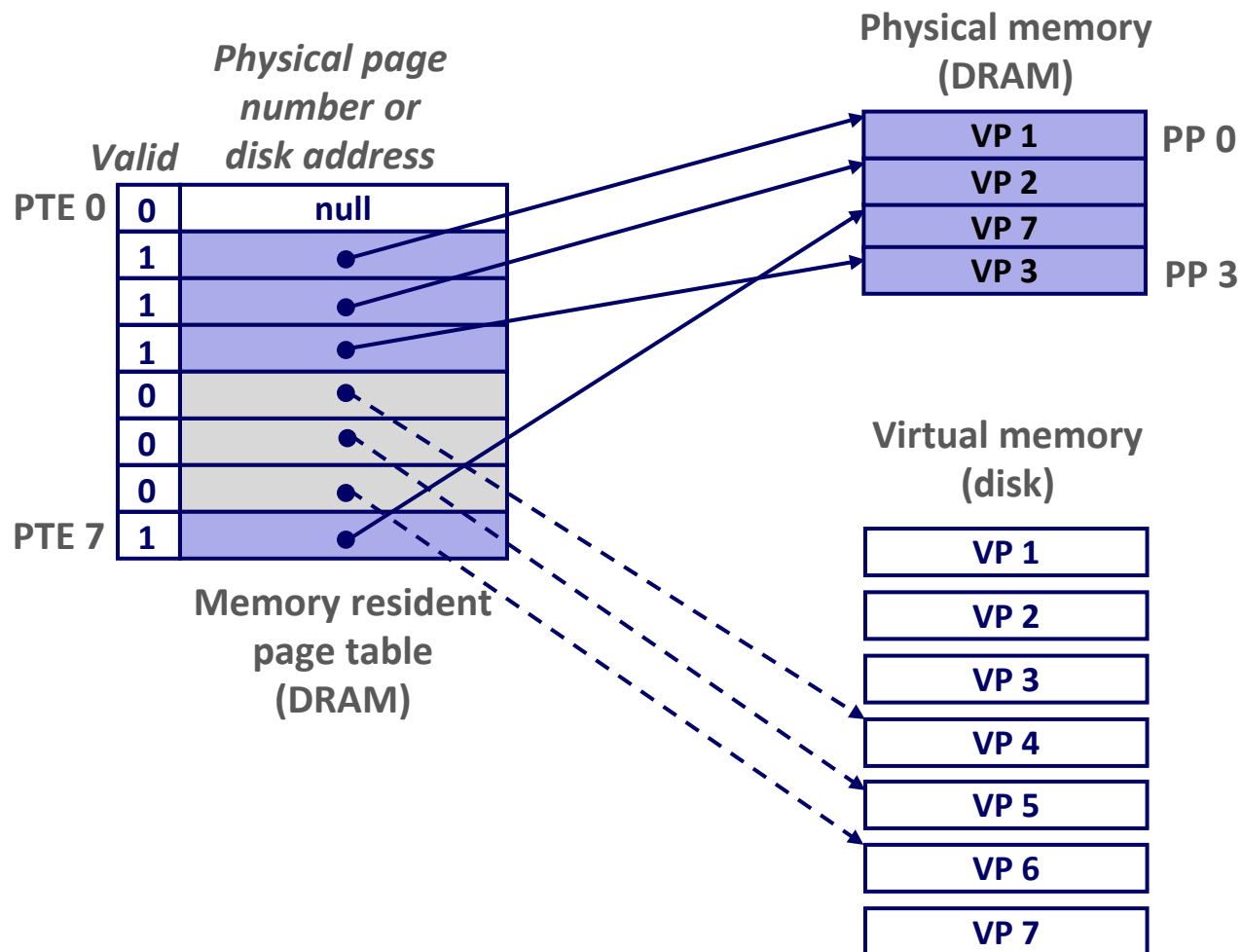
# Handling Page Fault

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



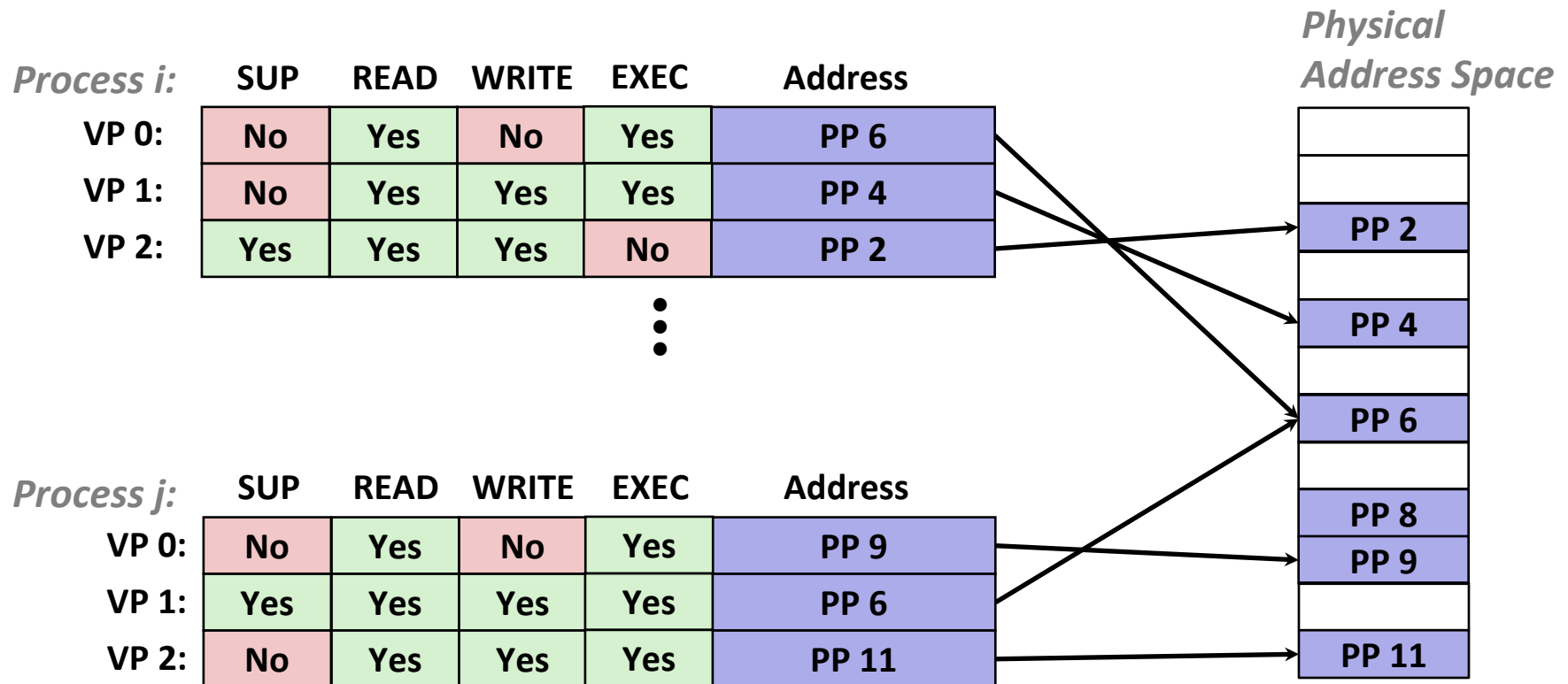
# Allocating Pages

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



# VM as a Tool for Memory Protection

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



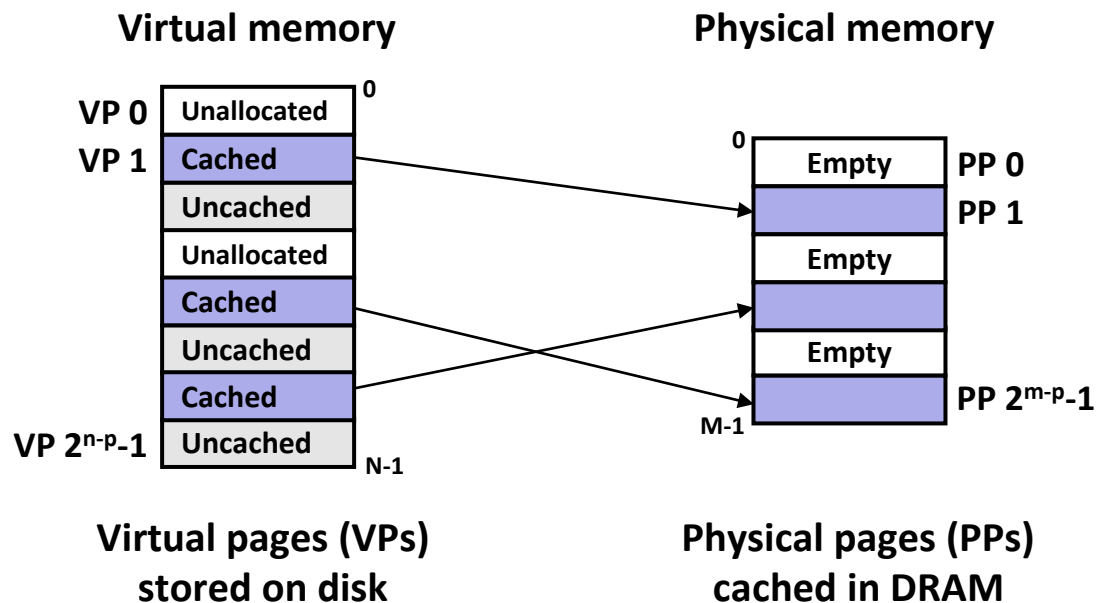


# VM is also Caching

- **Programs allocate virtual address ranges**
  - Implicitly via binaries / libraries
  - Explicitly through heap / stack
  
- **The operating system decides which virtual pages should be **resident** (i.e., in physical memory)**
  - OS manages the placement / replacement policies between DRAM and disk

# 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

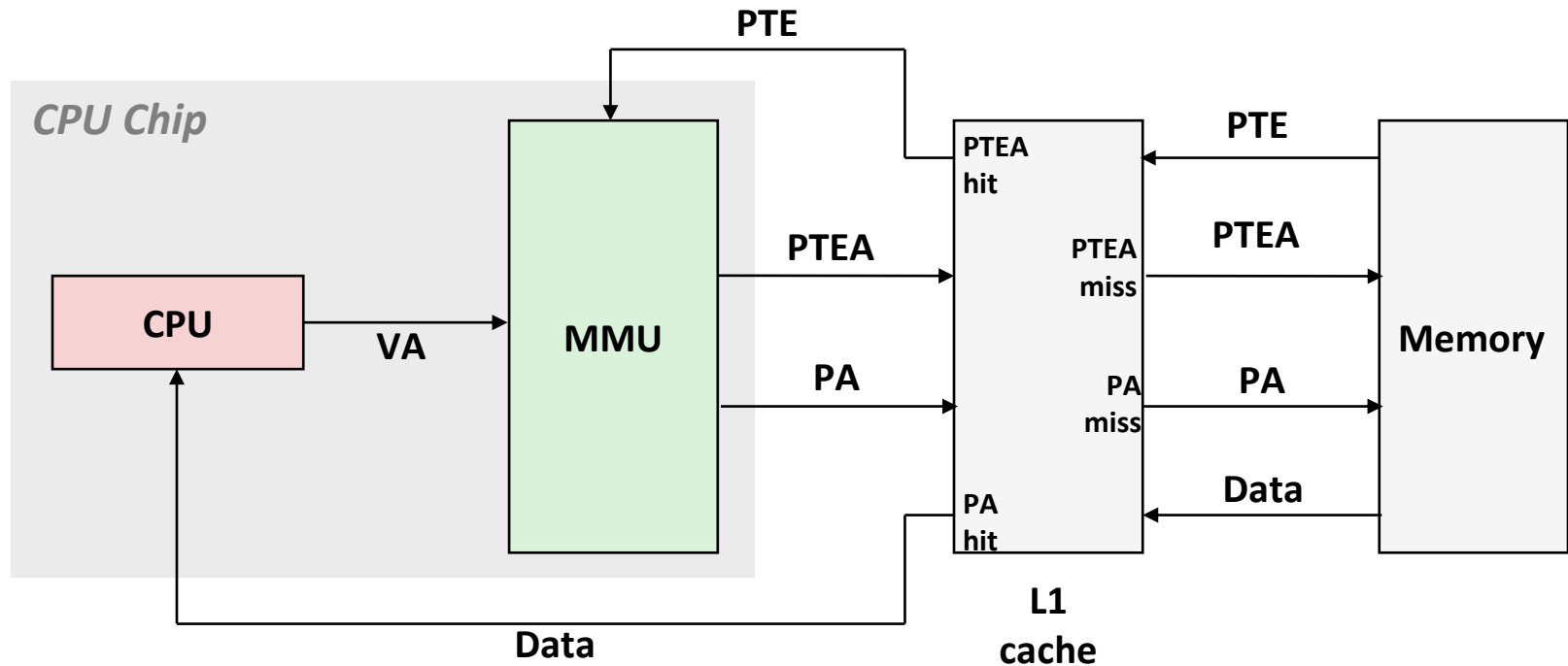
## ■ DRAM cache organization driven by the enormous miss penalty

- 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

# Integrating VM and Cache



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

# Locality to the Rescue Again!

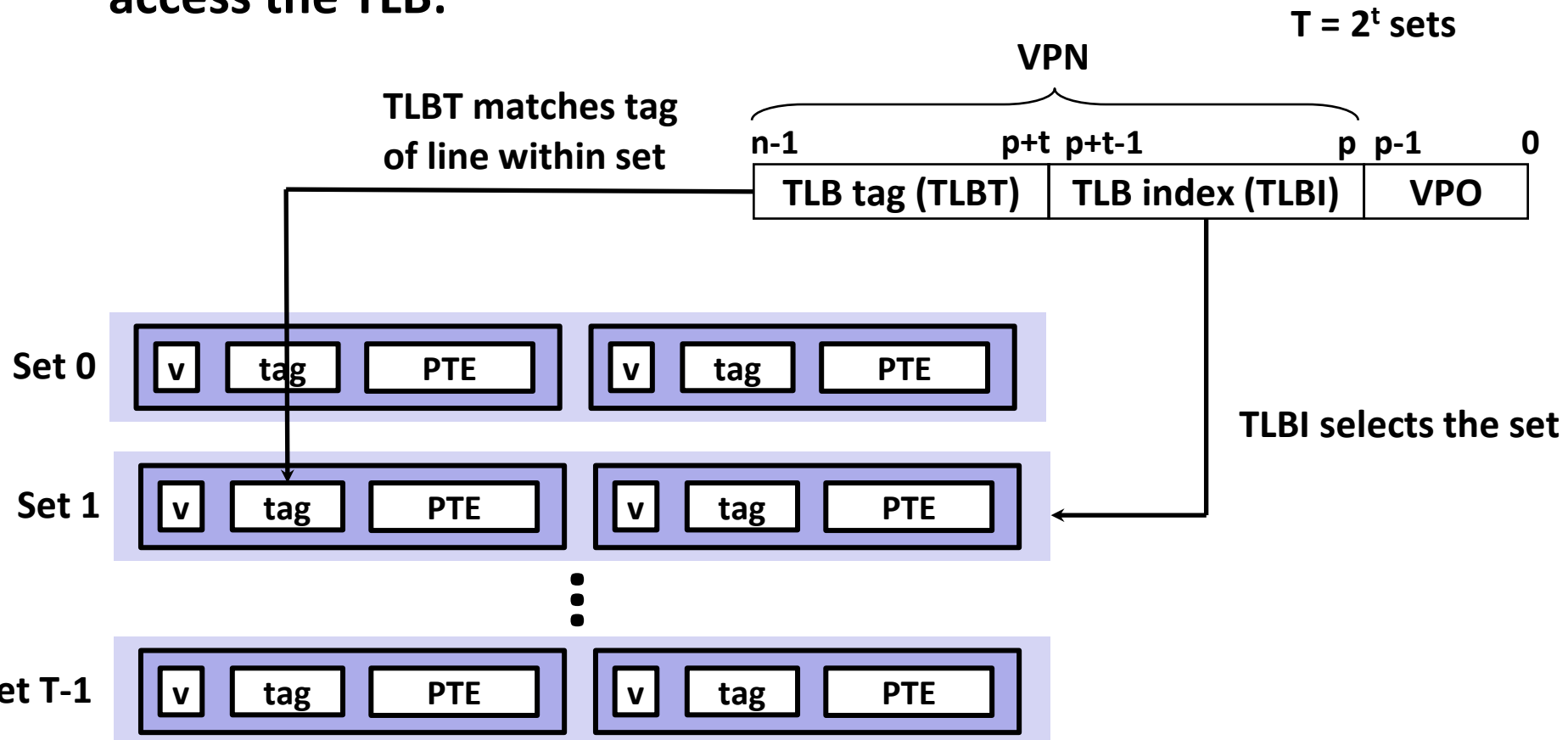
- Virtual memory seems terribly inefficient, but it works because of locality.
- At any point in time, programs tend to access a set of active virtual pages called the *working set*
  - Programs with better temporal locality will have smaller working sets
- If (working set size < main memory size)
 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

# Speeding up Translation with a TLB

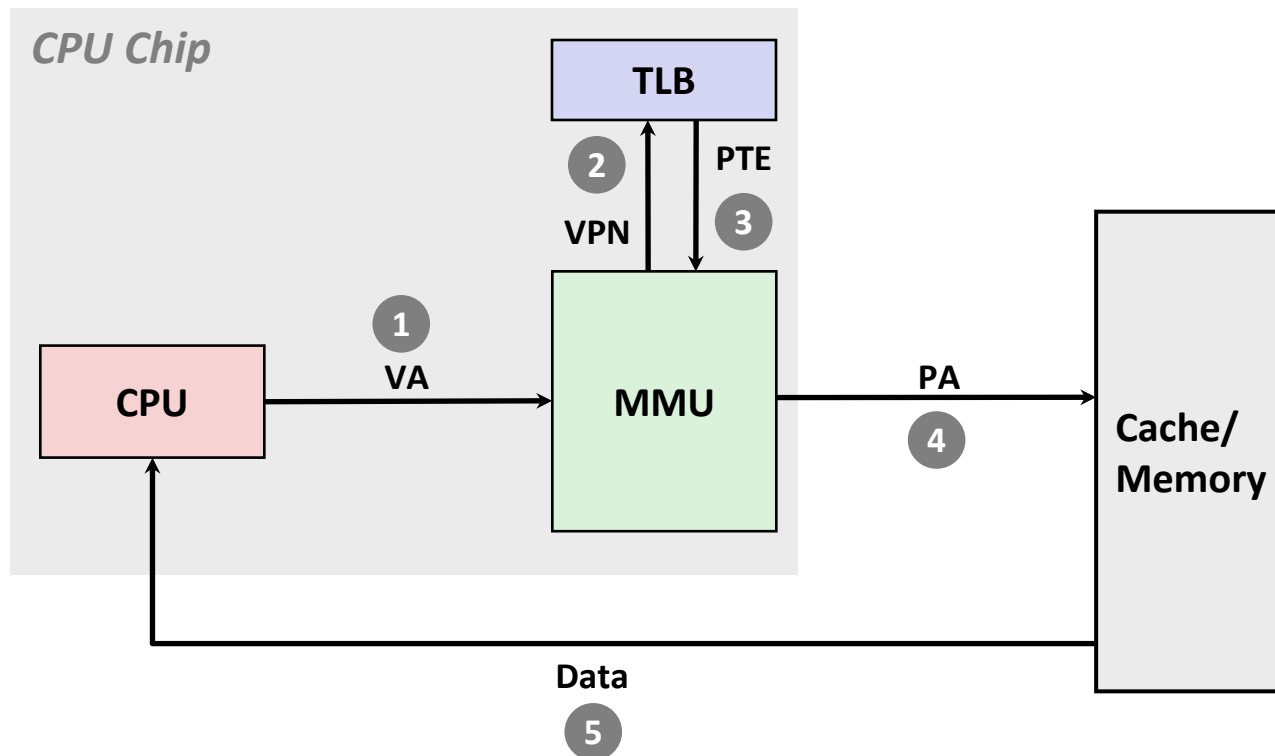
- **Page table entries (PTEs) are cached in L1 like any other memory word**
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay
- **Solution: *Translation Lookaside Buffer* (TLB)**
  - Small set-associative hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

# Accessing the TLB

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



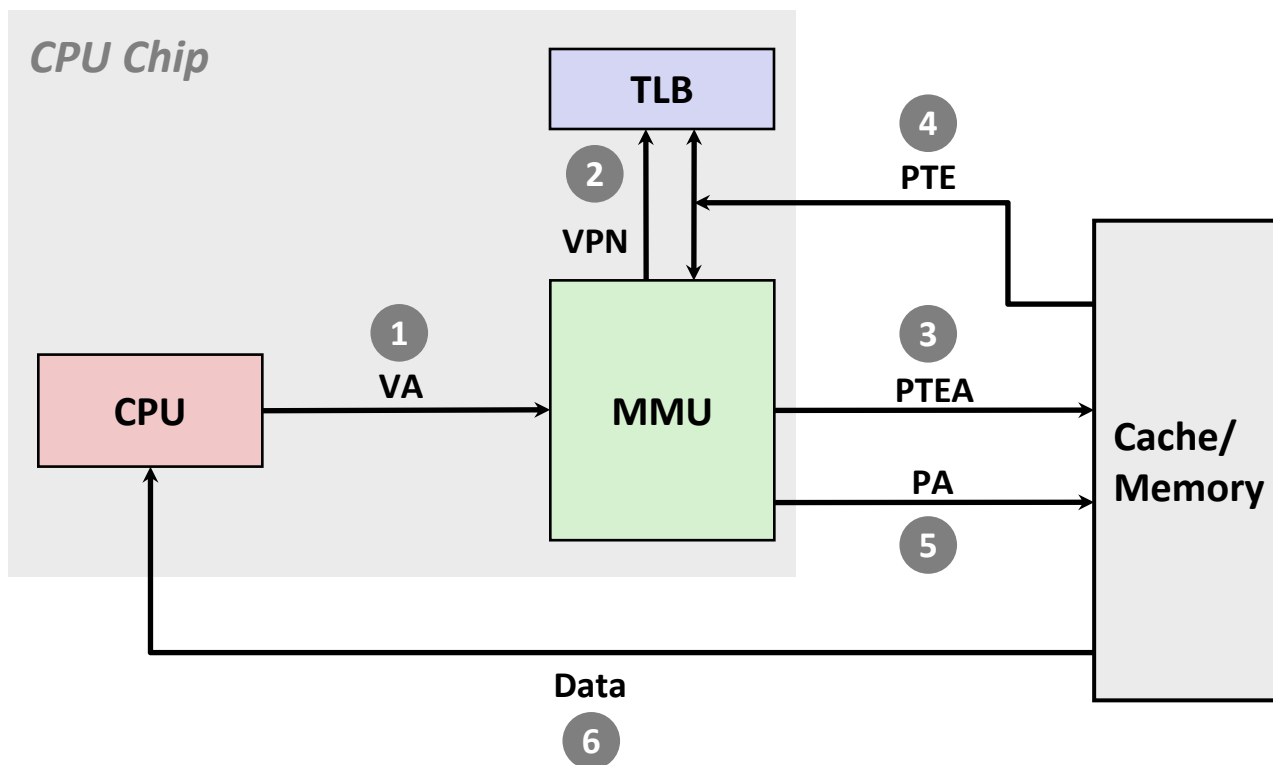
# TLB Hit



**A TLB hit eliminates a memory access**



# TLB Miss



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

Fortunately, TLB misses are rare. Why?

# 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)

- **TLBI**: TLB index
- **TLBT**: TLB tag
- **VPO**: Virtual page offset
- **VPN**: Virtual page number

## ■ Components of the physical address (PA)

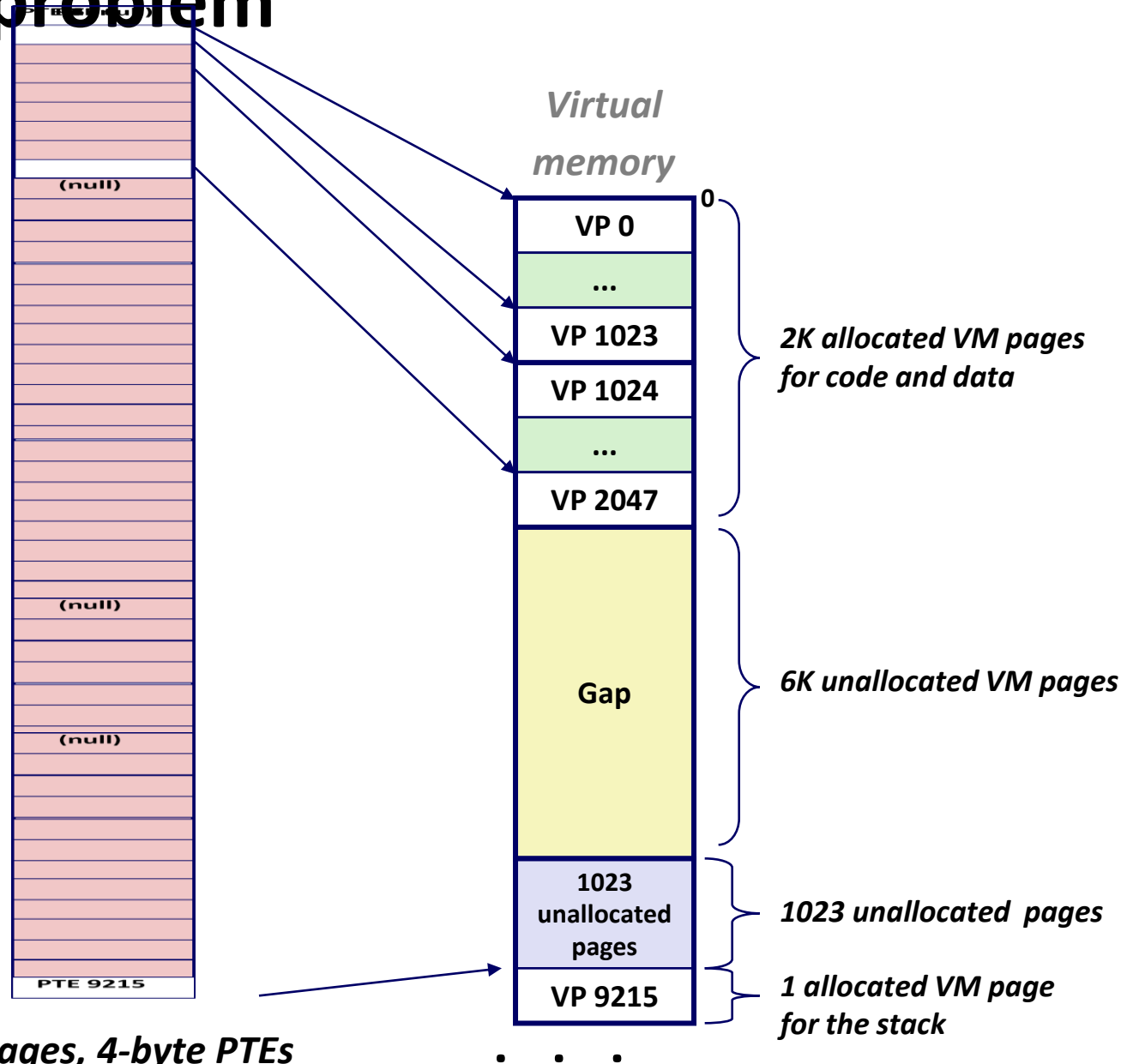
- **PPO**: Physical page offset (same as VPO)
- **PPN**: Physical page number

# Quiz

<https://canvas.cmu.edu/courses/40739/quizzes/123397>

# We have a problem

$2^{20}$  Entries of  
4 bytes each



**32 bit addresses, 4KB pages, 4-byte PTEs**

# Multi-Level Page Tables

## ■ Suppose:

- 4KB ( $2^{12}$ ) page size, 48-bit address space, 8-byte PTE

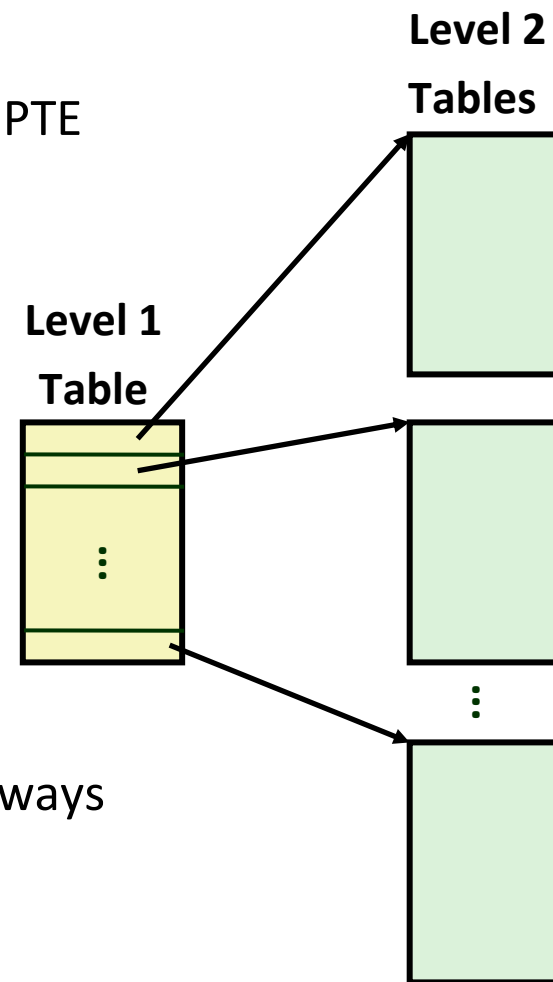
## ■ Problem:

- Would need a 512 GB page table!
  - $2^{48} * 2^{-12} * 2^3 = 2^{39}$  bytes

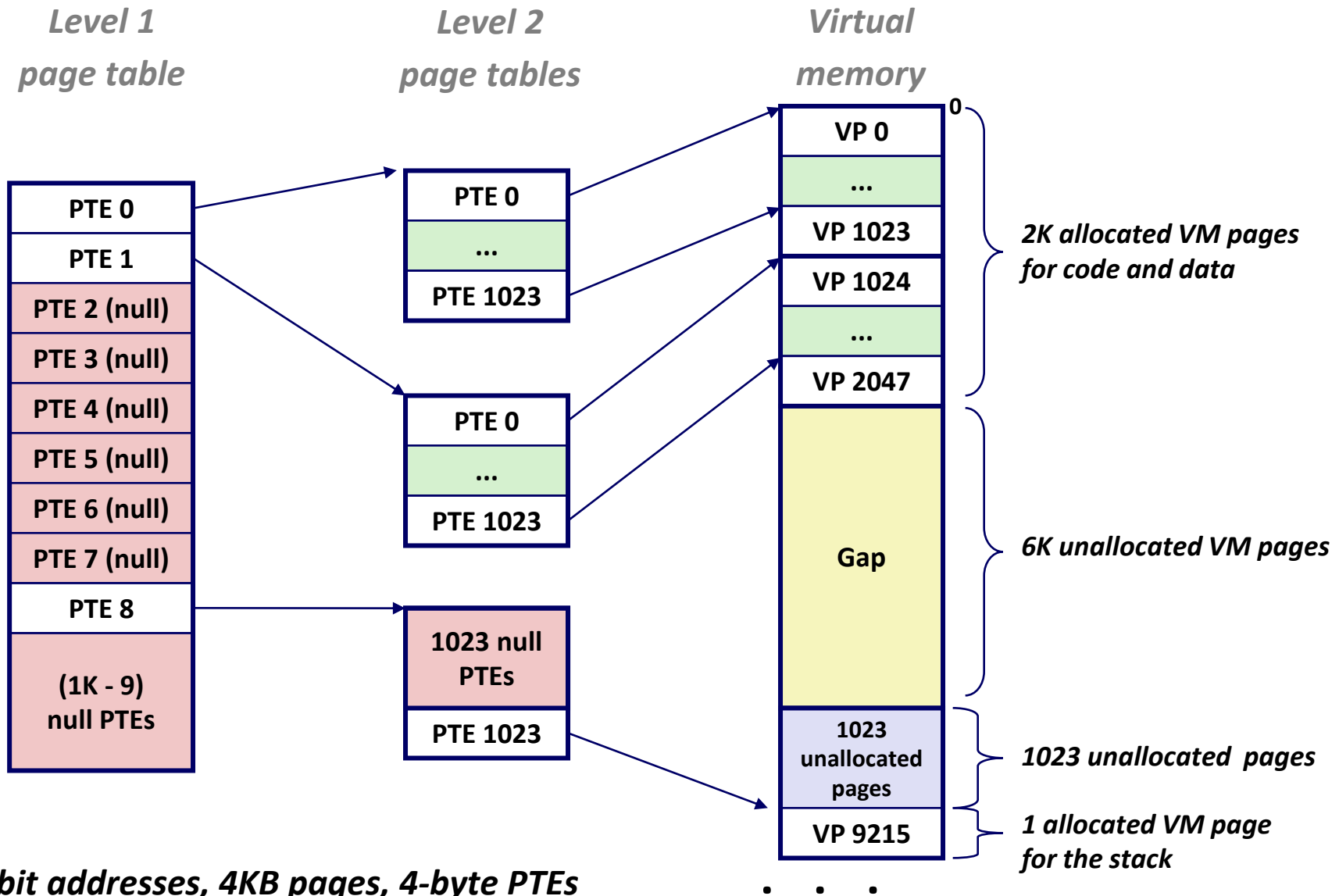
## ■ Common solution: Multi-level page table

## ■ Example: 2-level page table

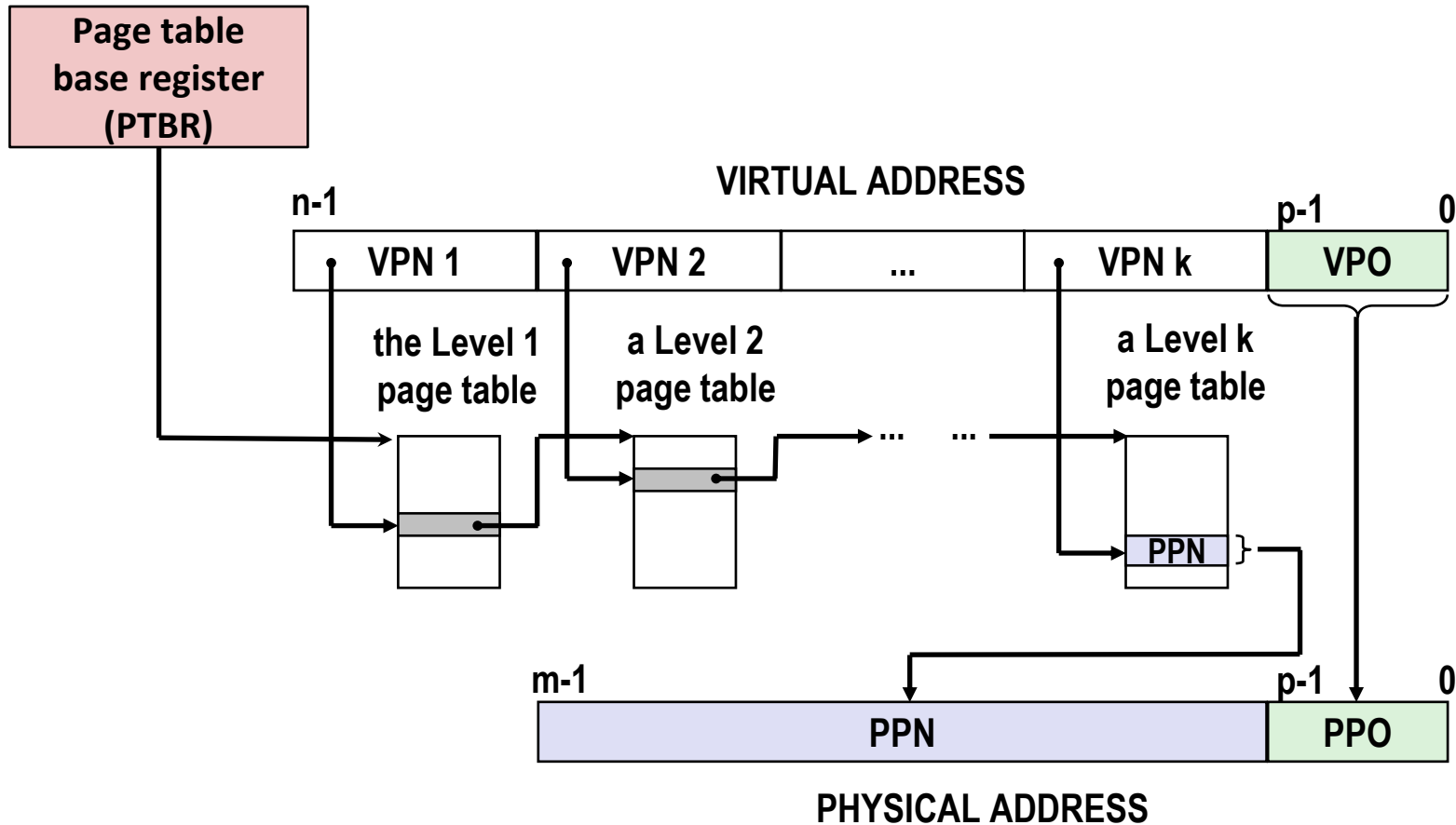
- Level 1 table: each PTE points to a page table (always memory resident)
- Level 2 table: each PTE points to a page (paged in and out like any other data)



# A Two-Level Page Table Hierarchy



# Translating with a k-level Page Table



# TLBs and k-level Page Tables

- **TLBs cache the complete virtual to physical mapping**
  - Regardless of the levels of page tables, the TLB stores the VPN  $\rightarrow$  PPN



# 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 programming
- Simplifies protection by providing a convenient interpositioning point to check permissions

# Activity Part 1

```
wget http://www.cs.cmu.edu/~213/activities/vm-concepts.tar  
tar xf vm-concepts.tar  
cd vm-concepts  
less addr.c
```

... further instructions in handout ...

**Stop after part 1 (end of page 2)**

**Caution: problems 3-5 involve deliberately running the sharks out of memory**

# Activity Part 2 through 4

- Now you have some idea *what* is going on
- Let's look at *how* it's done
- Details aren't *supposed* to be visible
  - We can get some clues via performance monitoring
  
- Do activity part 2 through 4 now
  - Stop at the end of page 5

# Activity Part 5 and 6

- So far we've only been looking at well-behaved programs
- What if they misbehave?
- Wouldn't it be nice if a misbehaving process couldn't interfere with any *other* processes?