

# Virtual Memory: Concepts

15-213 / 18-213: Introduction to Computer Systems  
16<sup>th</sup> Lecture, Oct. 25, 2011

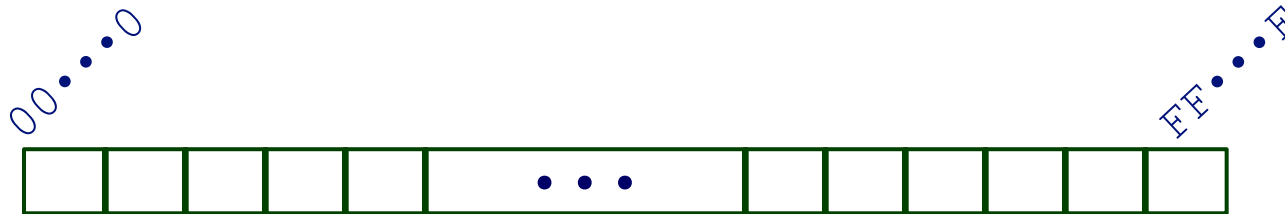
**Instructors:**

Dave O'Hallaron, Greg Ganger, and Greg Kesden

# Today

- **Address spaces**
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation

# Recall: Byte-Oriented Memory Organization



- **Programs refer to data by address**
  - Conceptually, envision it as a very large array of bytes
    - In reality, it's not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address
  
- **Note: system provides private address spaces to each “process”**
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

# Recall: Simple Addressing Modes

## ■ Normal (R) Mem[Reg[R]]

- Register R specifies memory address

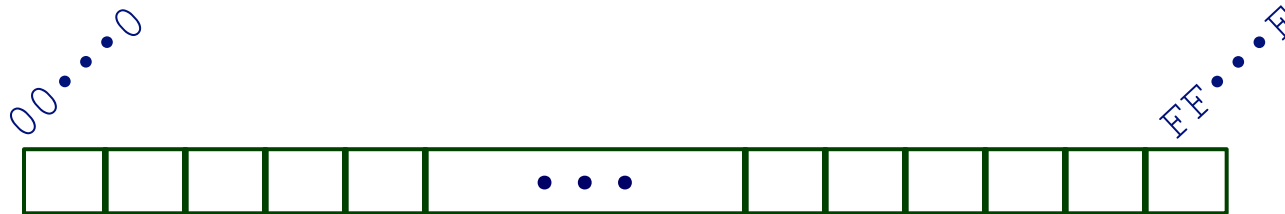
```
movl (%ecx), %eax
```

## ■ Displacement D(R) Mem[Reg[R]+D]

- Register R specifies start of memory region
- Constant displacement D specifies offset

```
movl 8(%ebp), %edx
```

# Lets think about this, a bit



- **How does everything fit?**
  - 32-bit addresses: ~4,000,000,000 (4 billion) bytes
  - 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes
- **How to decide which memory to use in your program?**
  - What about after a `fork()`?
- **What if another process stores data into your memory?**
  - How could you debug your program?

# So, we add a level of indirection

- **One simple trick solves all three problems**
  - Each process gets its own private image of memory
    - appears to be a full-sized private memory range
  - This fixes “how to choose” and “others shouldn’t mess w/yours”
    - surprisingly, it also fixes “making everything fit”
  - Implementation: translate addresses transparently
    - add a mapping function
      - to map private addresses to physical addresses
    - do the mapping on every load or store
- **This mapping trick is the heart of *virtual memory***

# 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

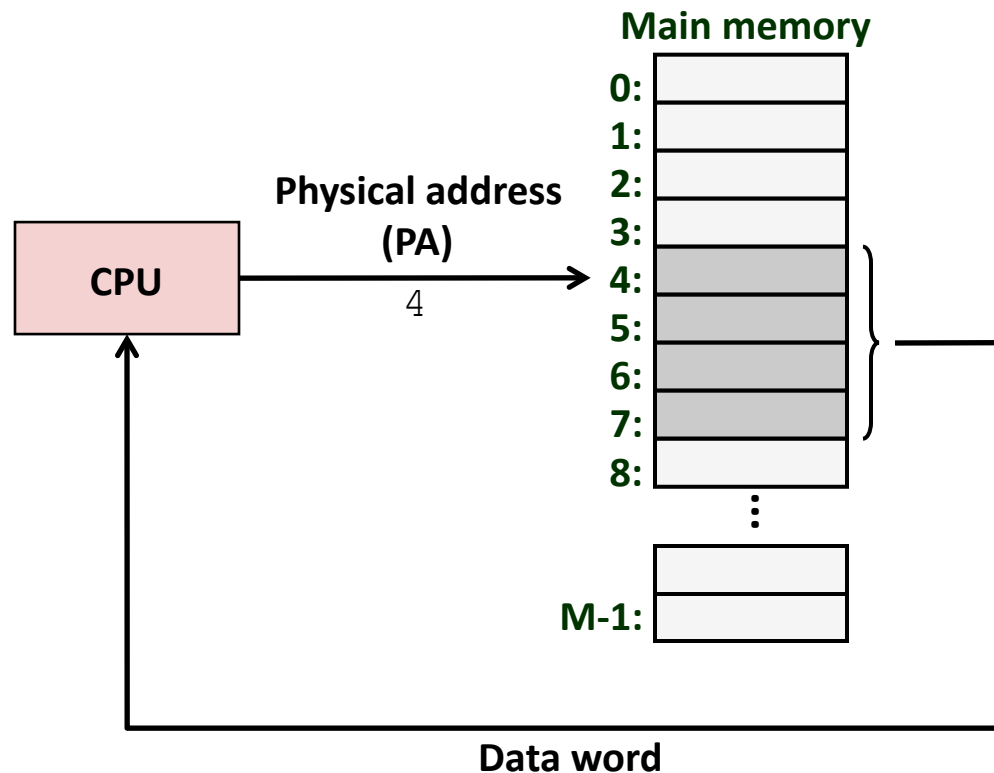
$\{0, 1, 2, 3, \dots, N-1\}$

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

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

- Clean distinction between data (bytes) and their attributes (addresses)
- Each datum can now have multiple addresses
- Every byte in main memory:  
one physical address, one (or more) virtual addresses

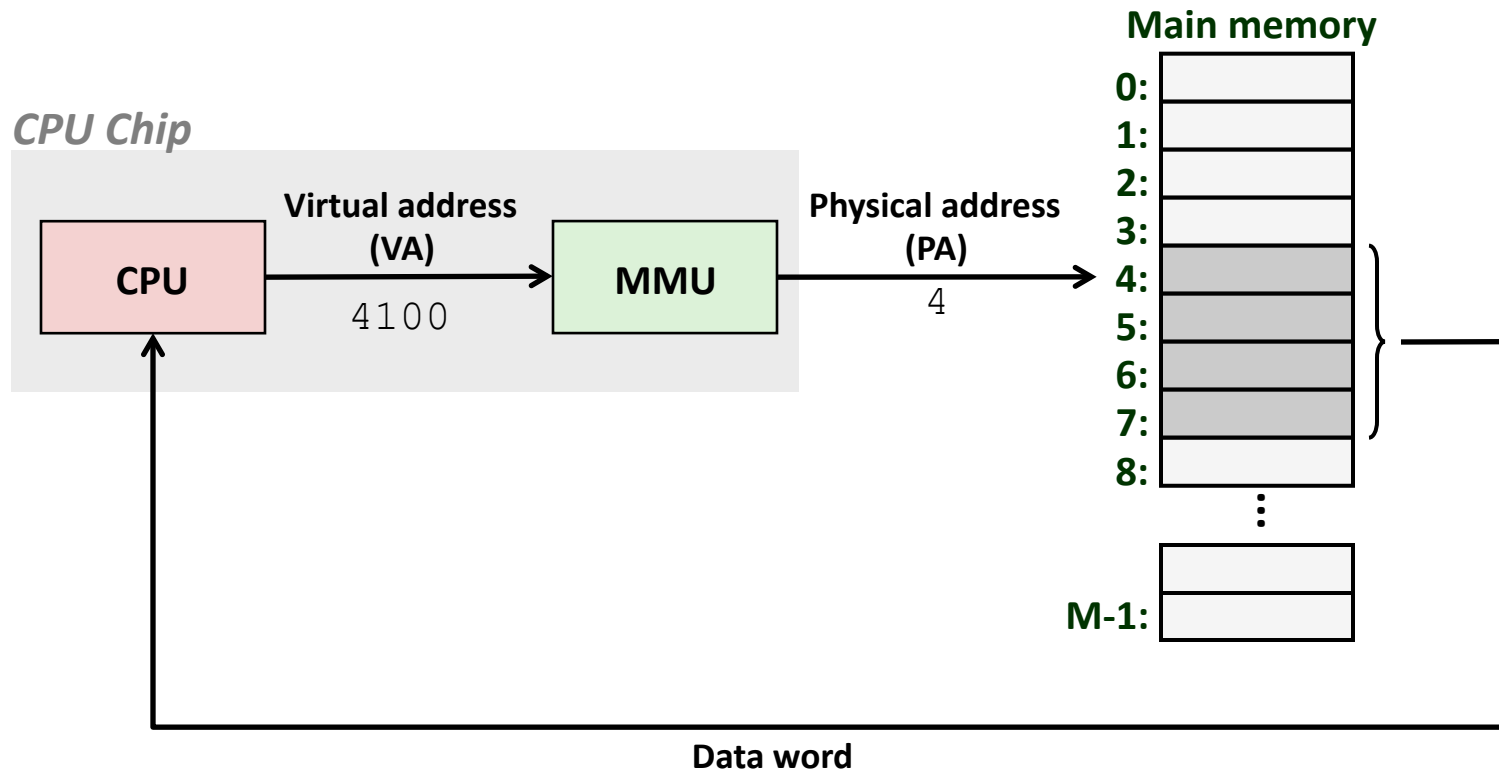
# A System Using Physical Addressing



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



# A System Using Virtual Addressing



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

# Why Virtual Memory?

## **(1) VM allows efficient use of limited main memory (RAM)**

- Use RAM as a cache for the parts of a virtual address space
  - some non-cached parts stored on disk
  - some (unallocated) non-cached parts stored nowhere
- Keep only active areas of virtual address space in memory
  - transfer data back and forth as needed

## **(2) VM simplifies memory management for programmers**

- Each process gets a full, private linear address space

## **(3) VM isolates address spaces**

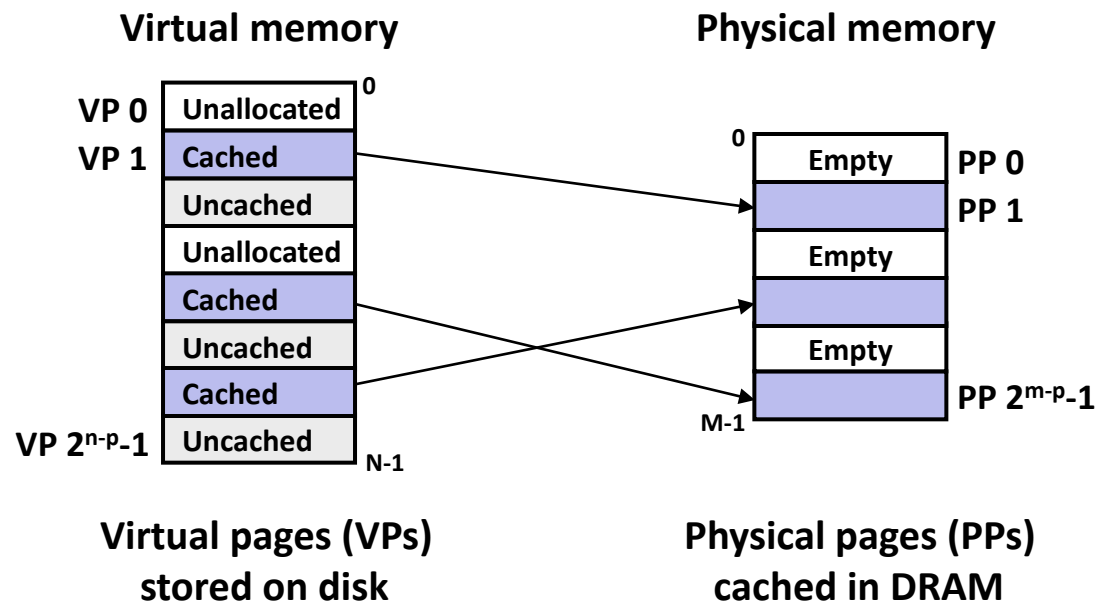
- One process can't interfere with another's memory
  - because they operate in different address spaces
- User process cannot access privileged information
  - different sections of address spaces have different permissions

# Today

- Address spaces
- **(1) VM as a tool for caching**
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- Address translation

# (1) VM as a Tool for Caching

- **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

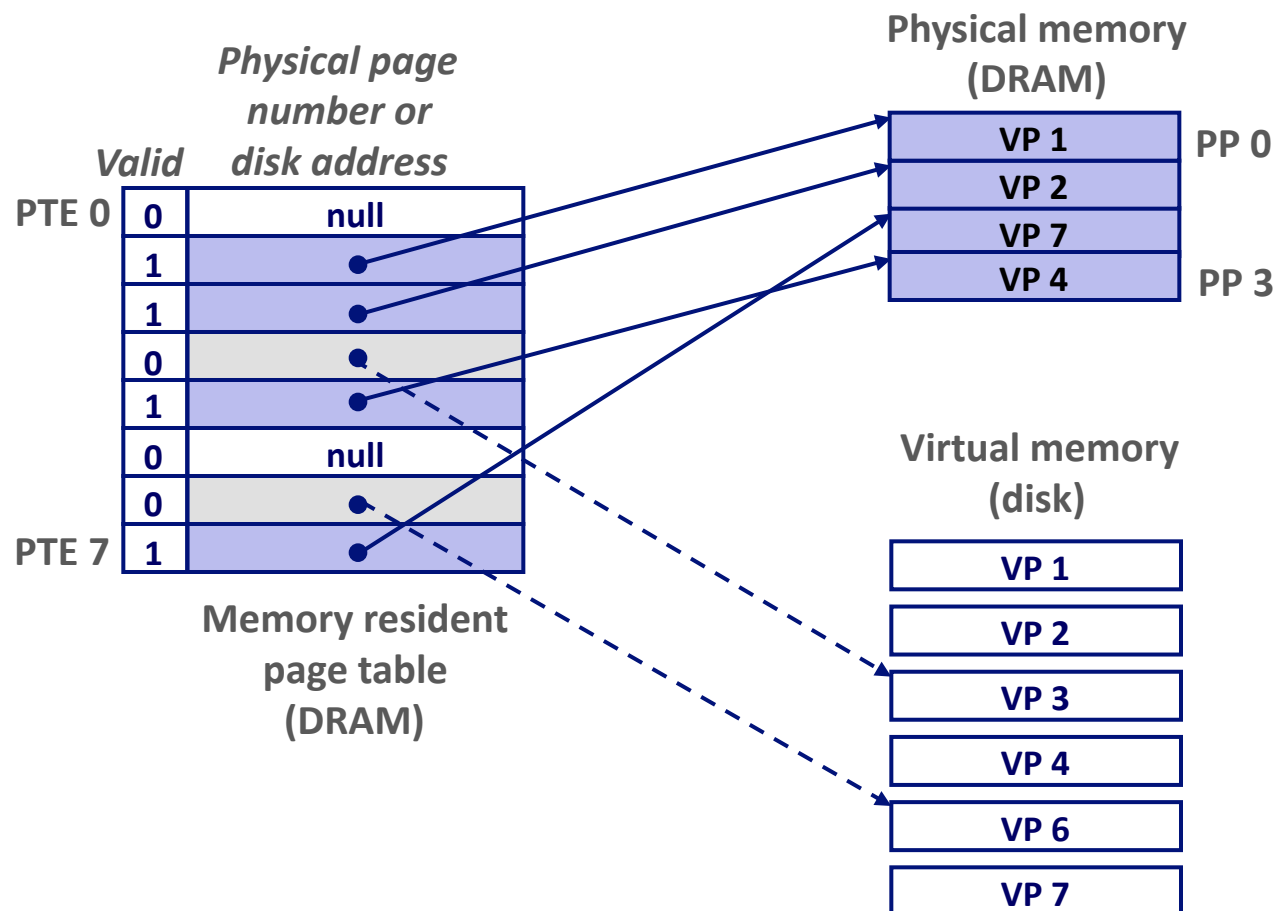
- DRAM is about **10x** slower than SRAM
- Disk is about **10,000x** slower than DRAM

## ■ Consequences

- Large page (block) size: typically 4-8 KB, sometimes 4 MB
- Fully associative
  - Any VP can be placed in any PP
  - Requires a “large” mapping function – different from CPU caches
- Highly sophisticated, expensive replacement algorithms
  - Too complicated and open-ended to be implemented in hardware
- Write-back rather than write-through

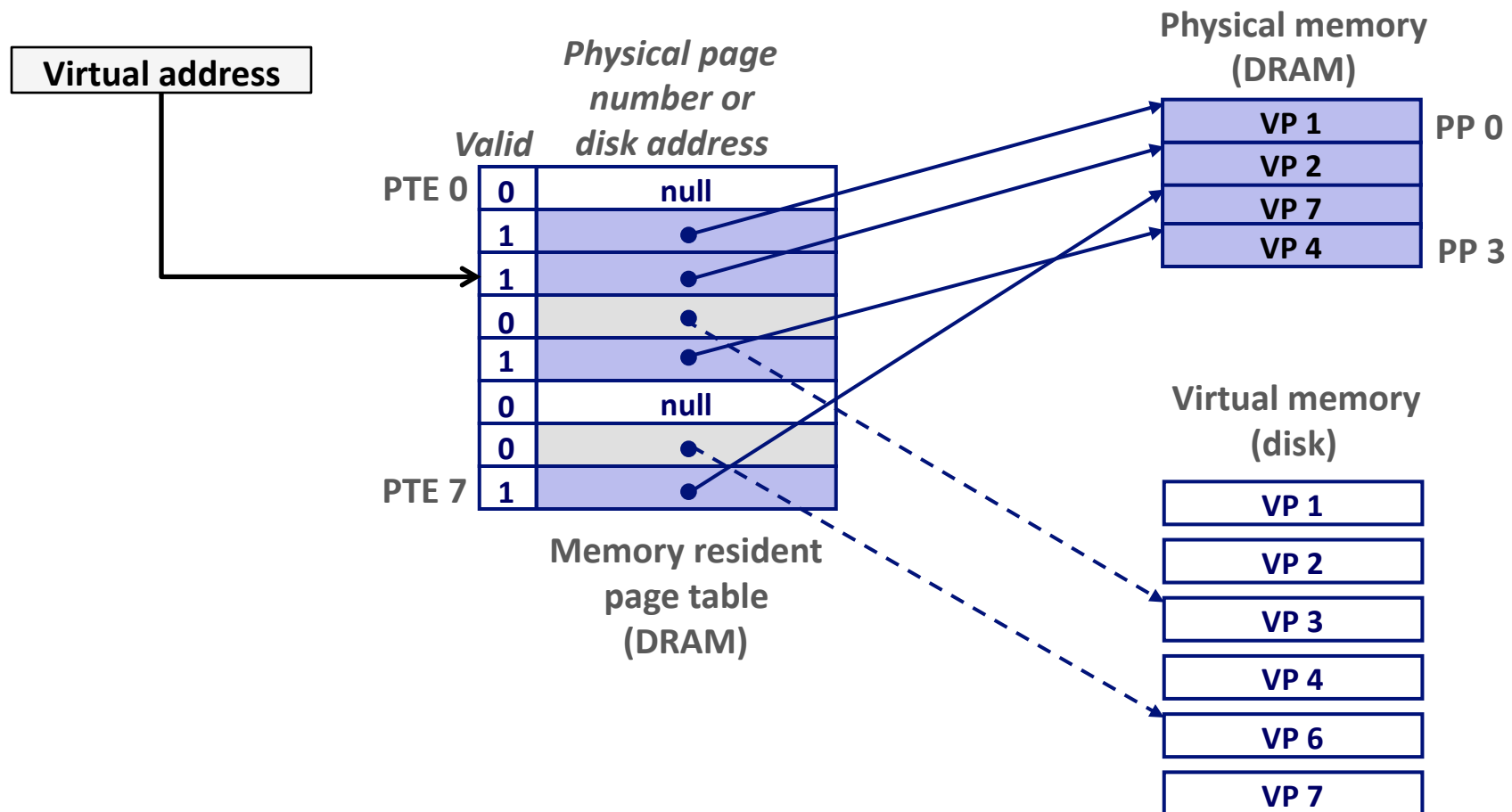
# Enabling data structure: Page Table

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



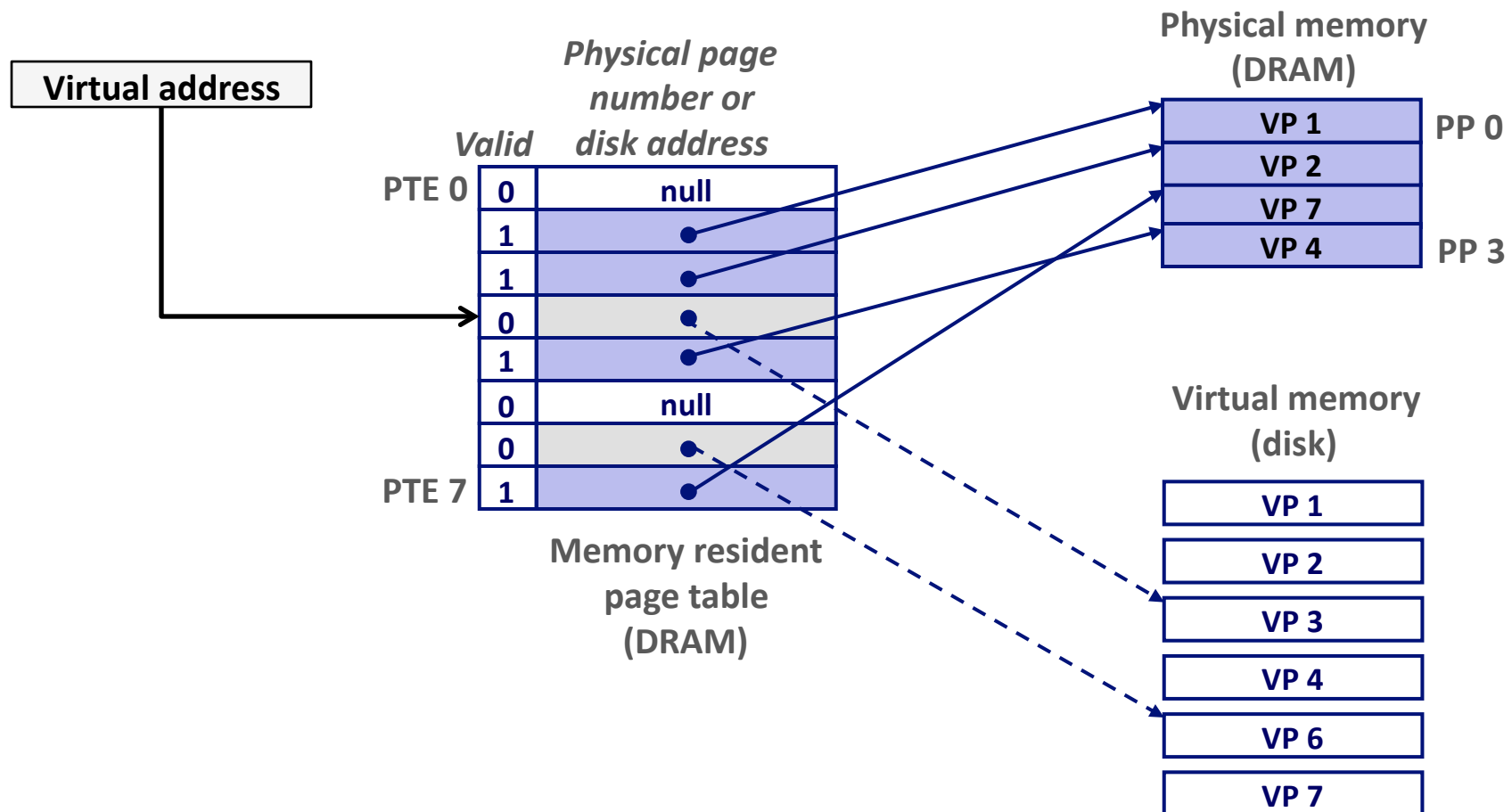
# Page Hit

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



# Page Fault

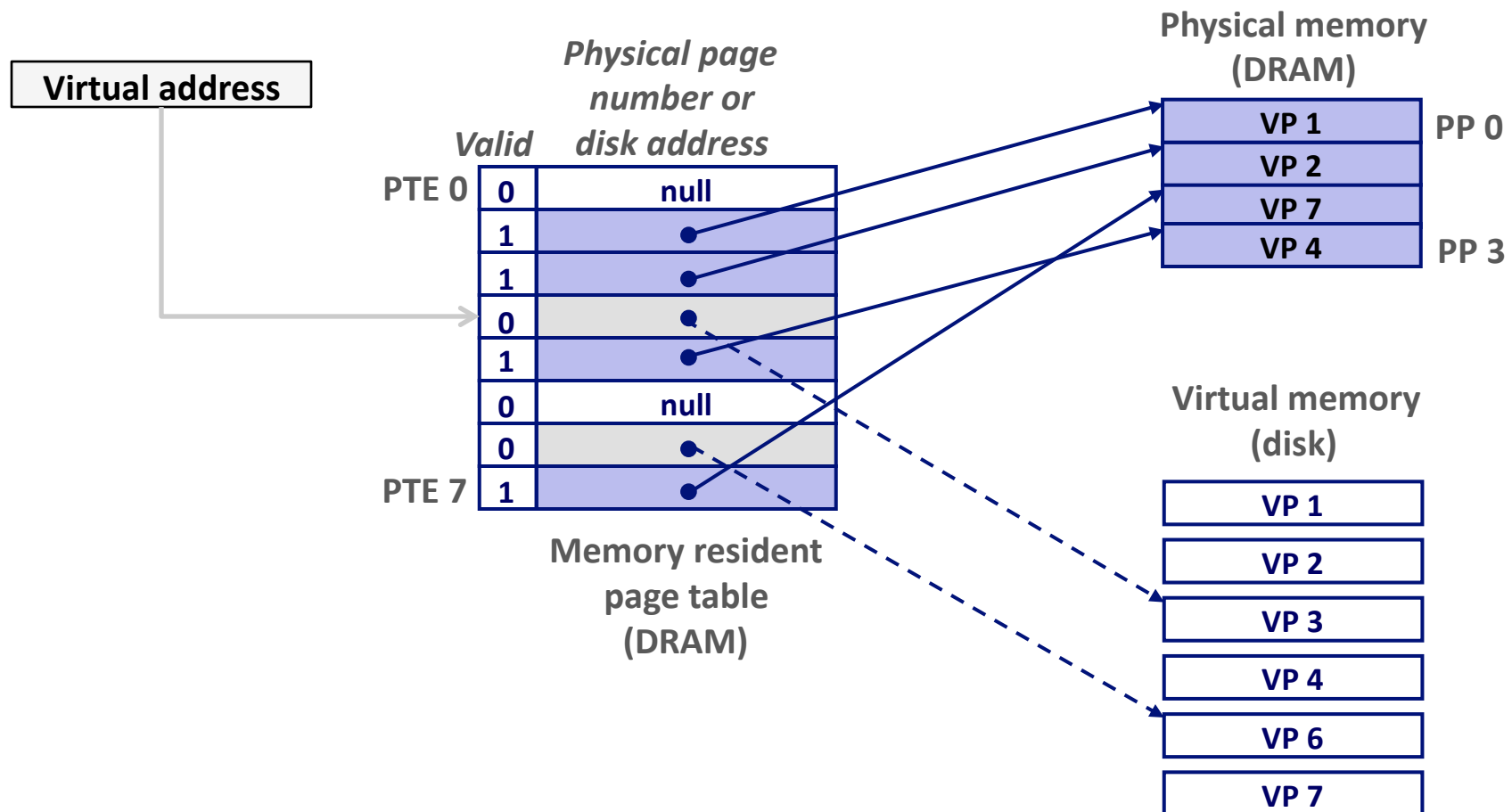
- **Page fault:** reference to VM word that is not in physical memory (DRAM cache miss)





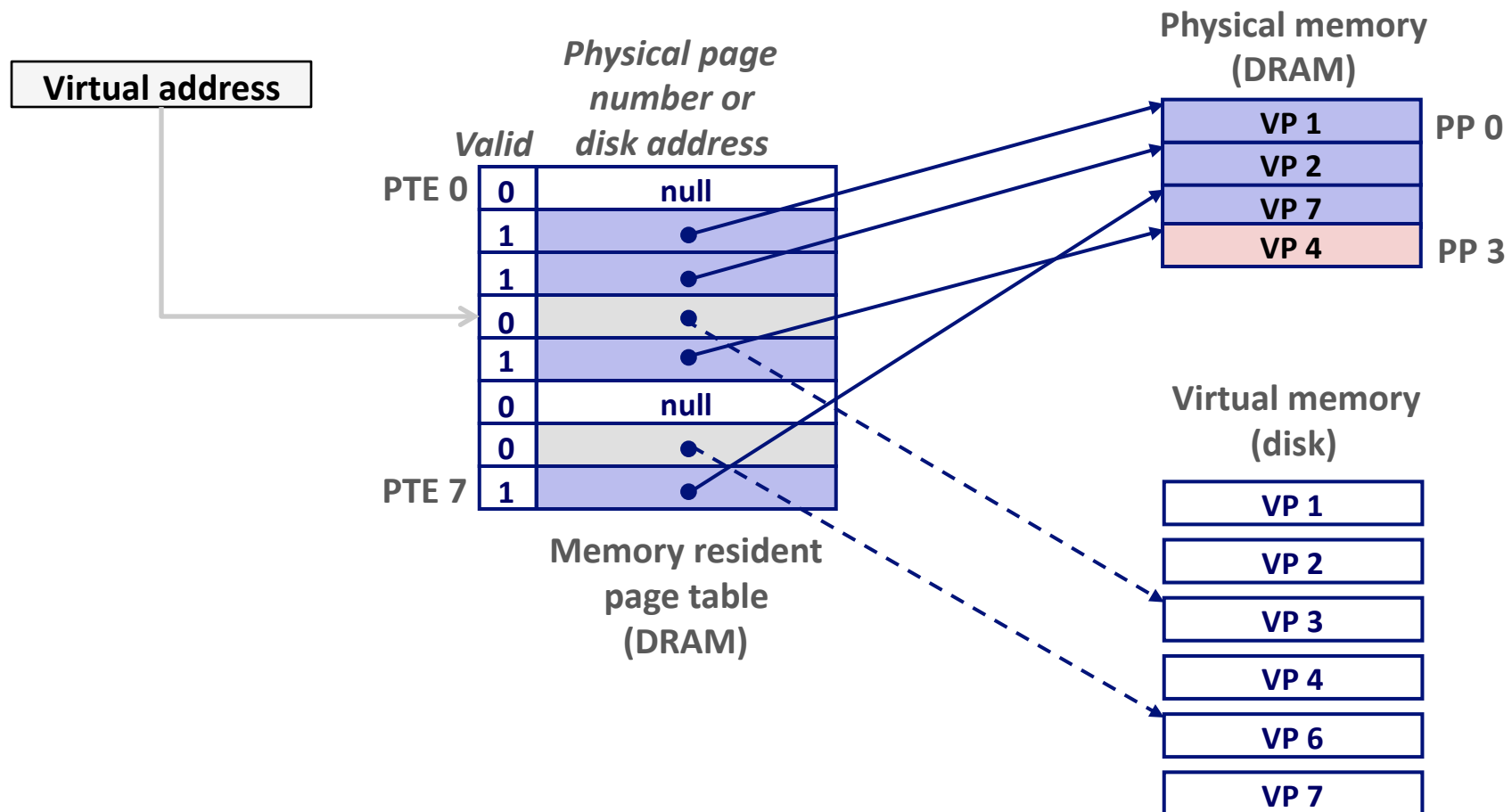
# Handling Page Fault

- Page miss causes page fault (an exception)



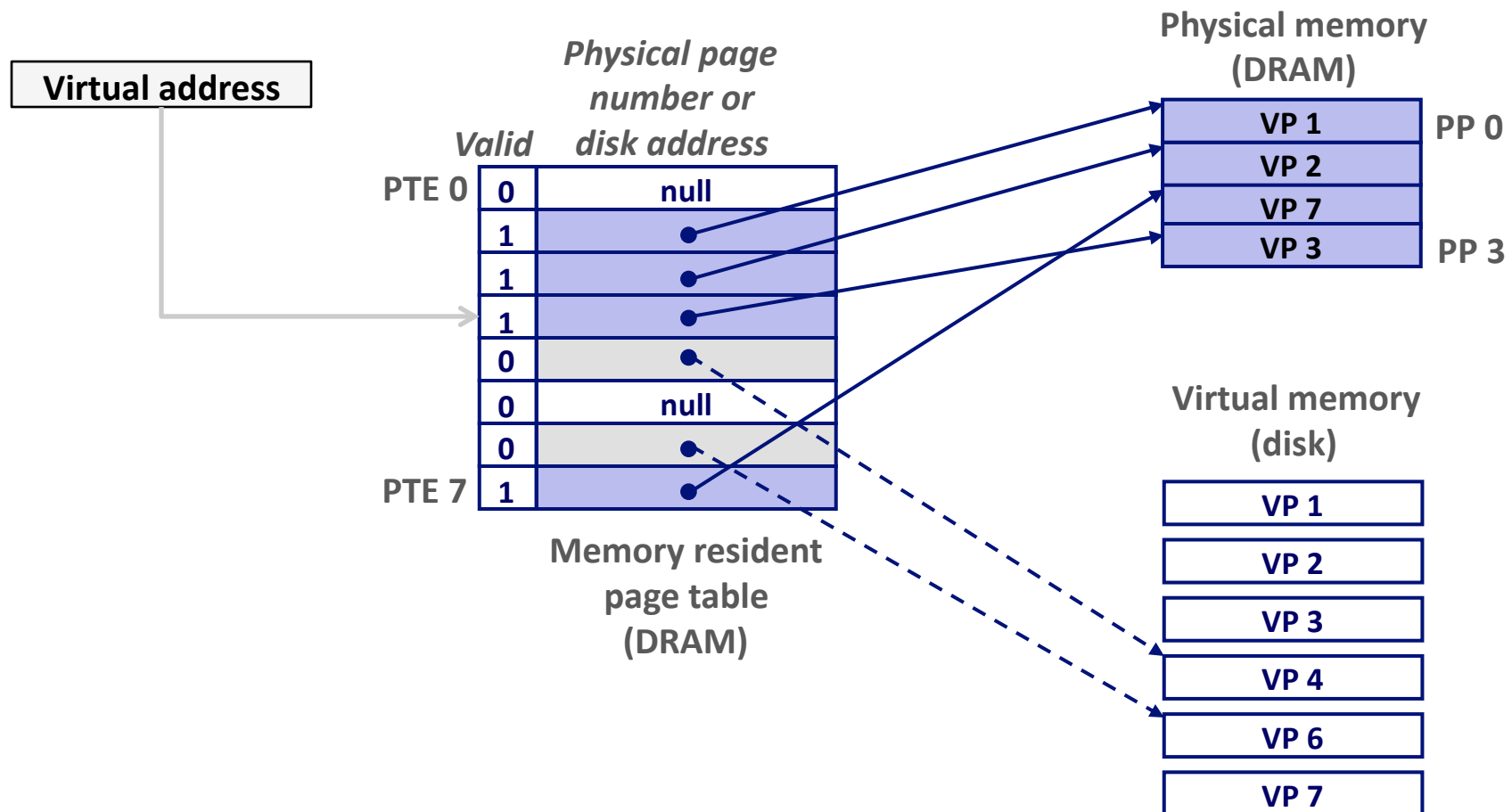
# Handling Page Fault

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



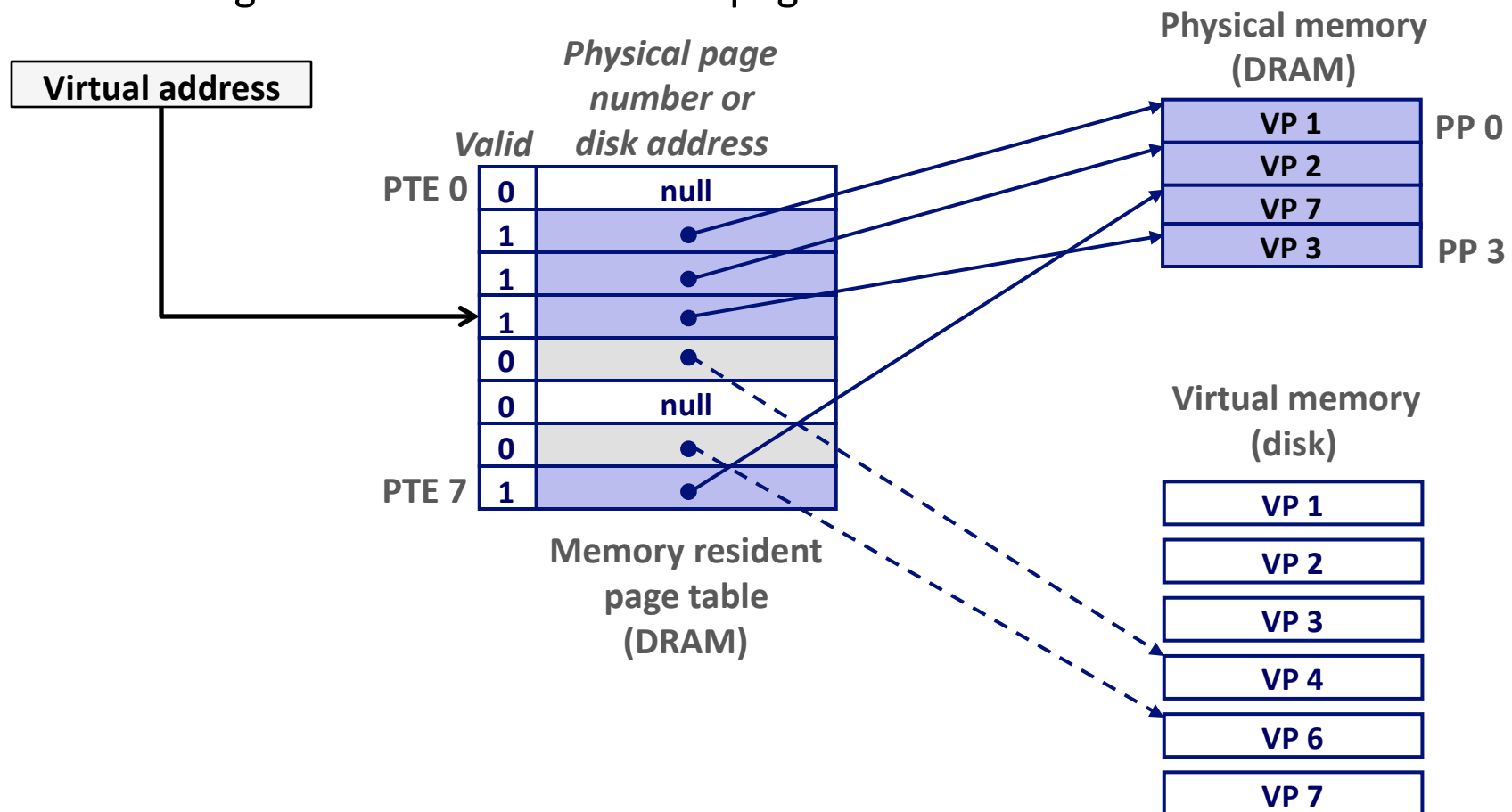
# Handling Page Fault

- 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: page hit!



# Locality to the Rescue Again!

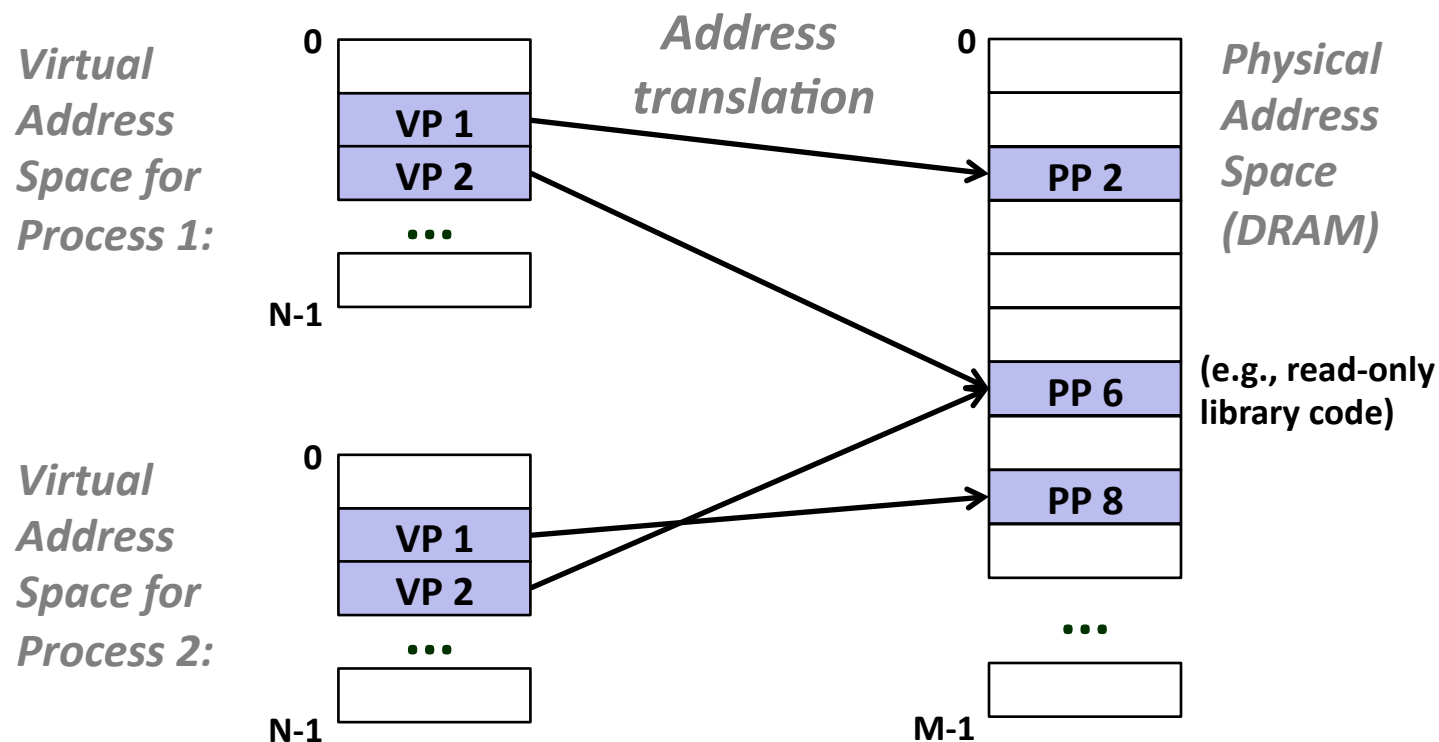
- Virtual memory works because of locality
- At any point in time, programs tend to access a set of active virtual pages called the *working set*
  - Programs with better temporal locality will have smaller working sets
- If (working set size < main memory size)
  - Good performance for one process after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

# Today

- Address spaces
- (1) VM as a tool for caching
- **(2) VM as a tool for memory management**
- (3) VM as a tool for memory protection
- Address translation

## (2) 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 simplify memory allocation and management



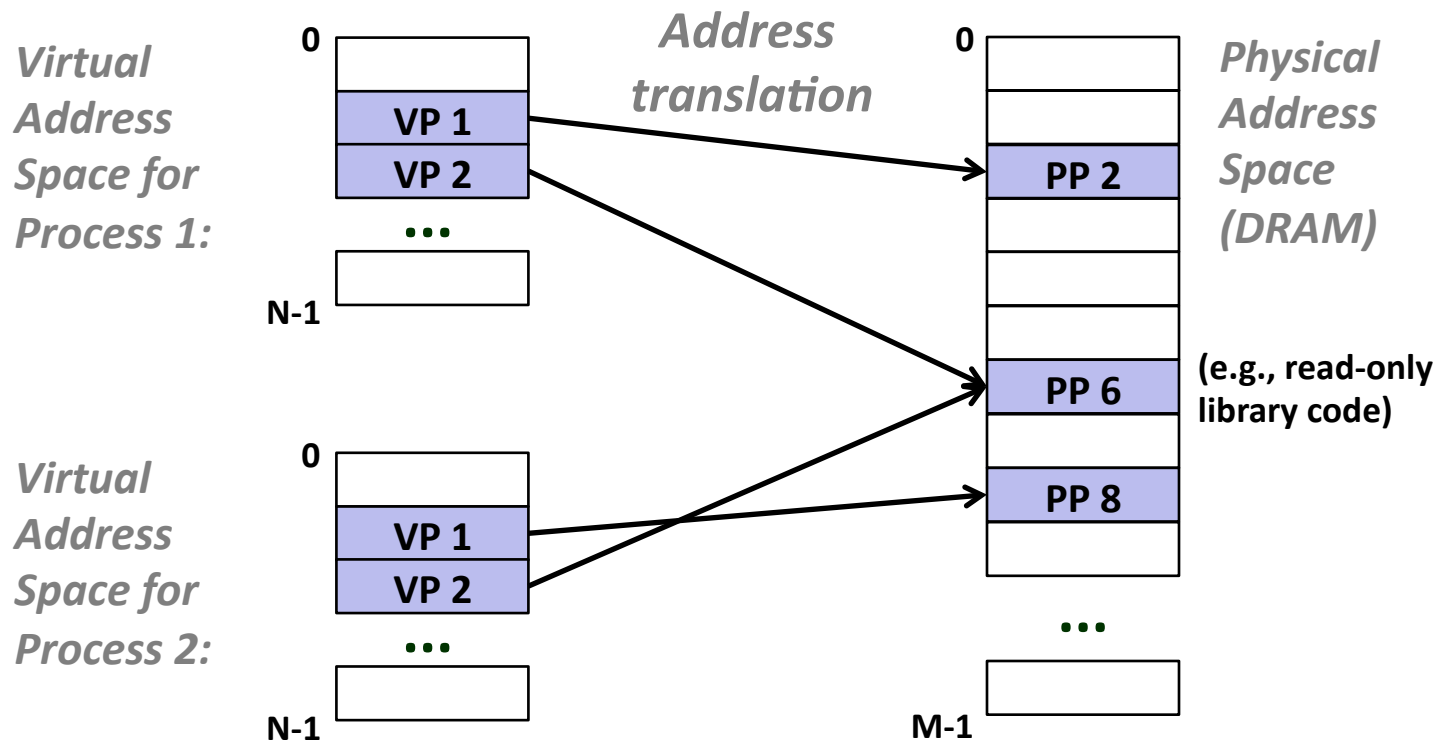
# Simplifying allocation and sharing

## ■ 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 multiple virtual pages to the same physical page (here: PP 6)





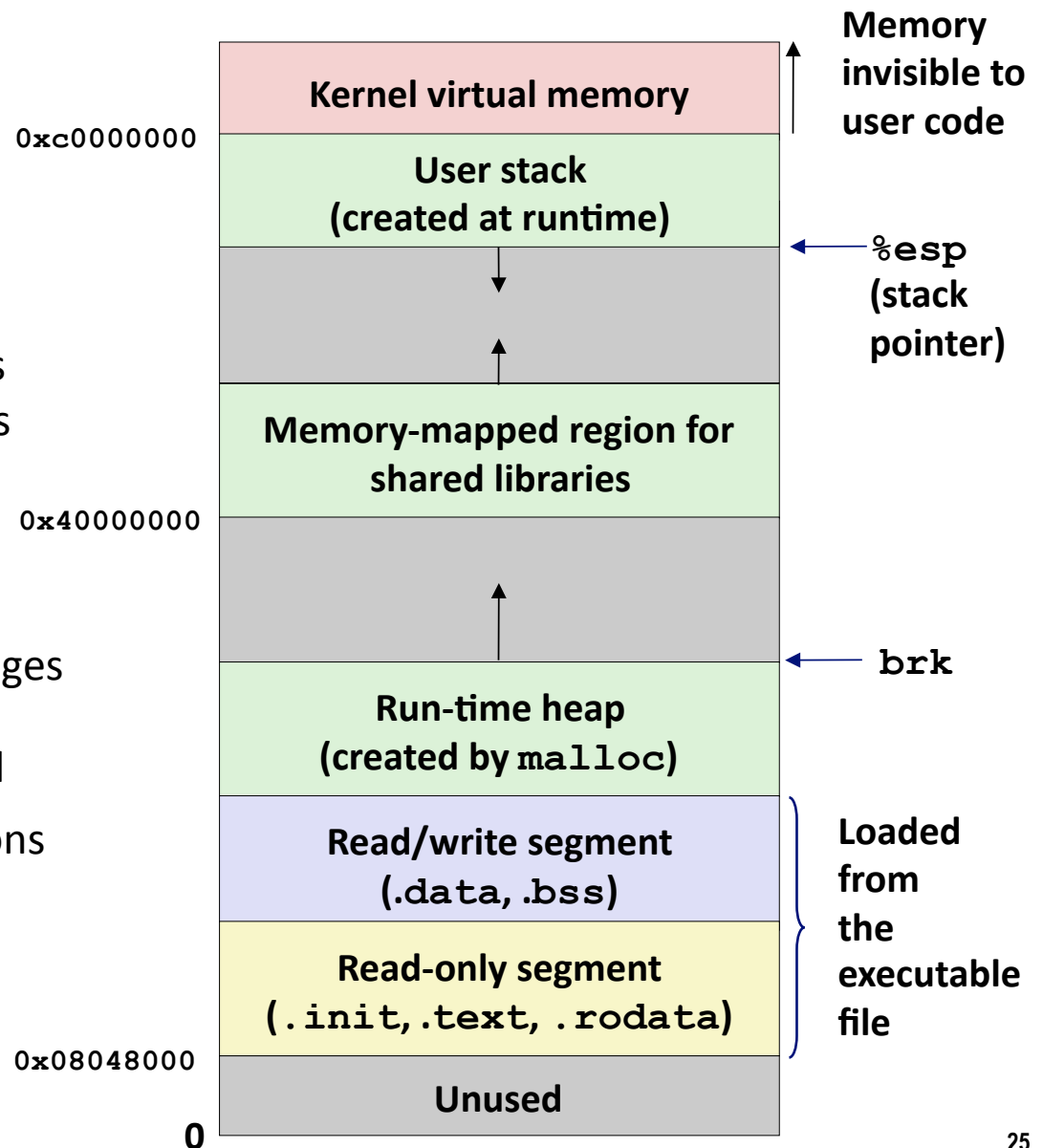
# Simplifying Linking and Loading

## ■ Linking

- Each program has similar virtual address space
- Code, stack, and shared libraries always start at the same address

## ■ 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

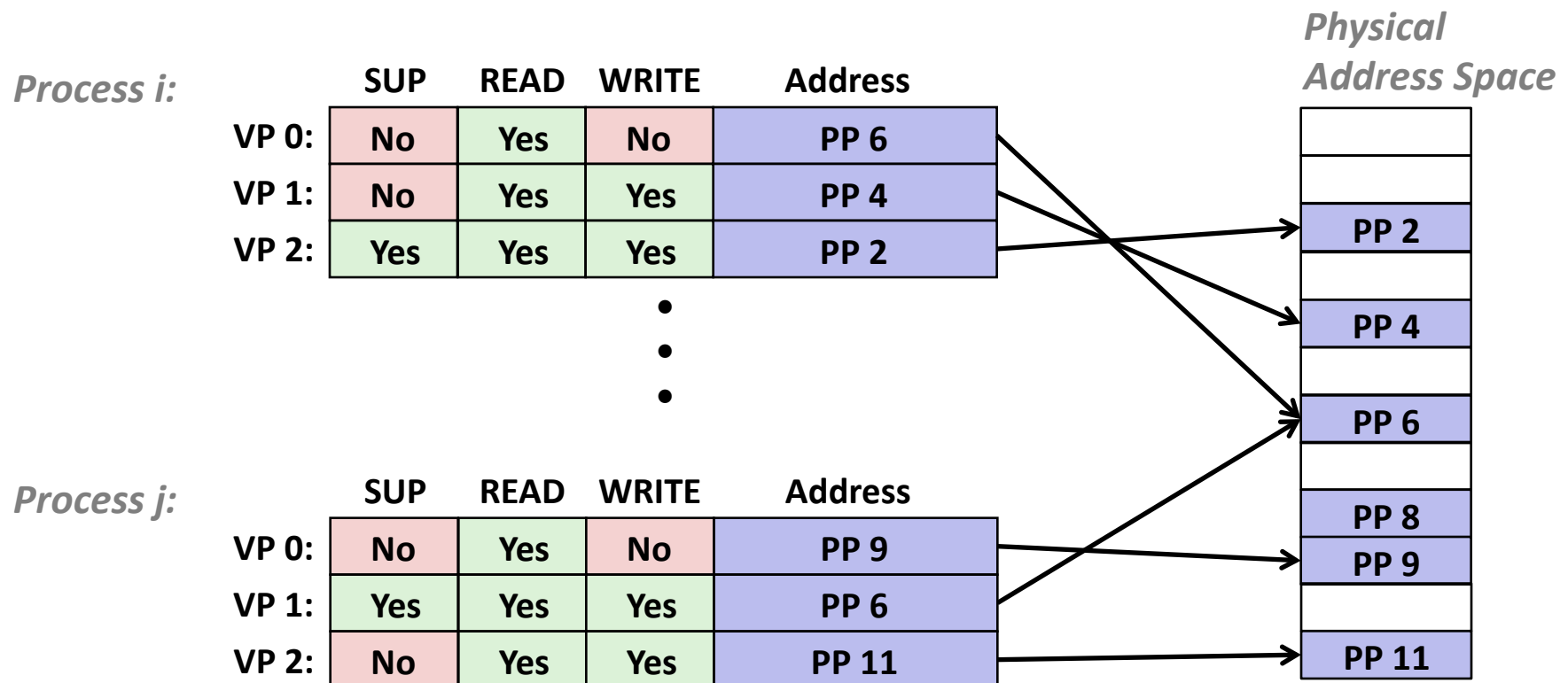


# Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- **(3) VM as a tool for memory protection**
- Address translation

# VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- Page fault handler checks these before remapping
  - If violated, send process SIGSEGV (segmentation fault)



# Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- **Address translation**

# 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

# 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)

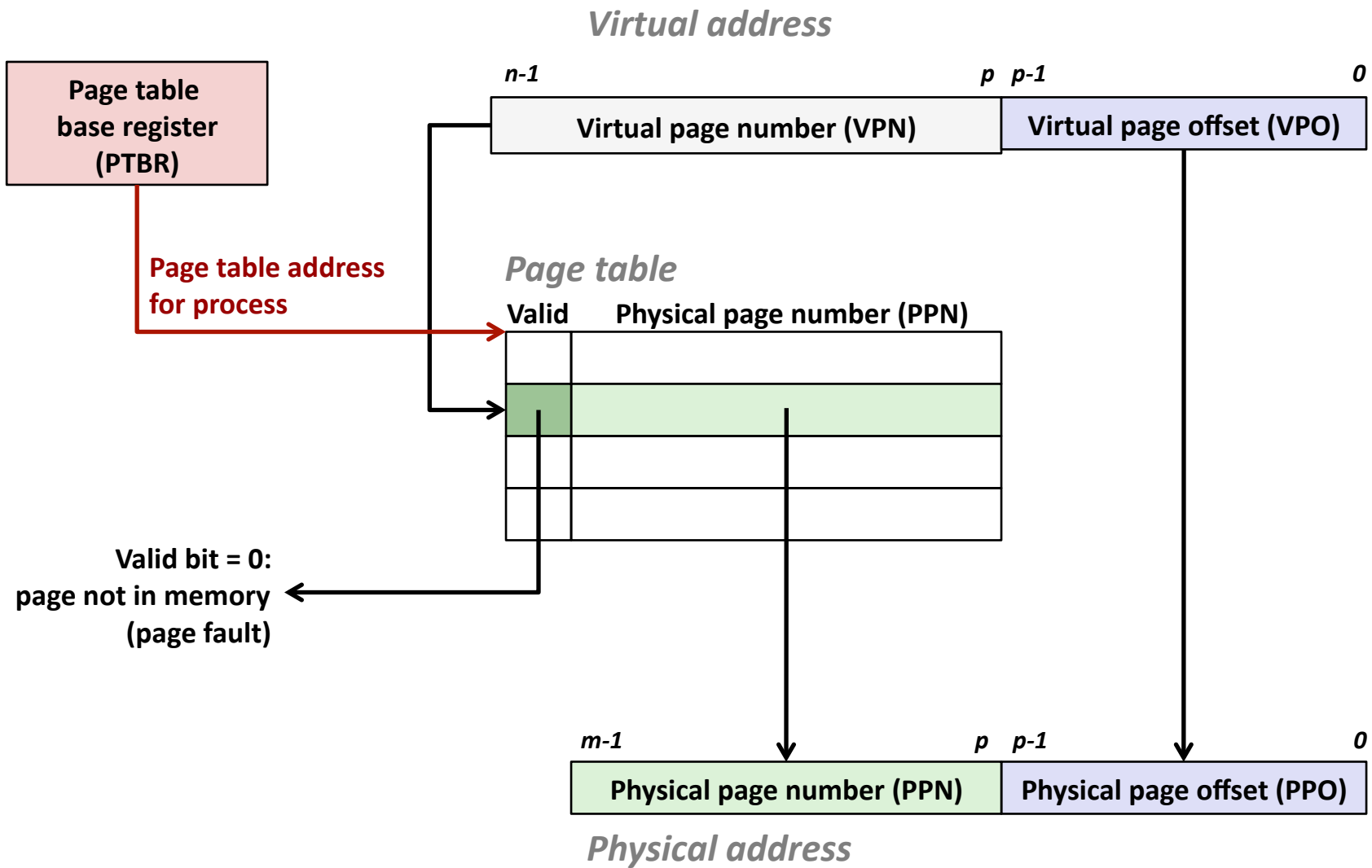
## ■ Components of the virtual address (VA)

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

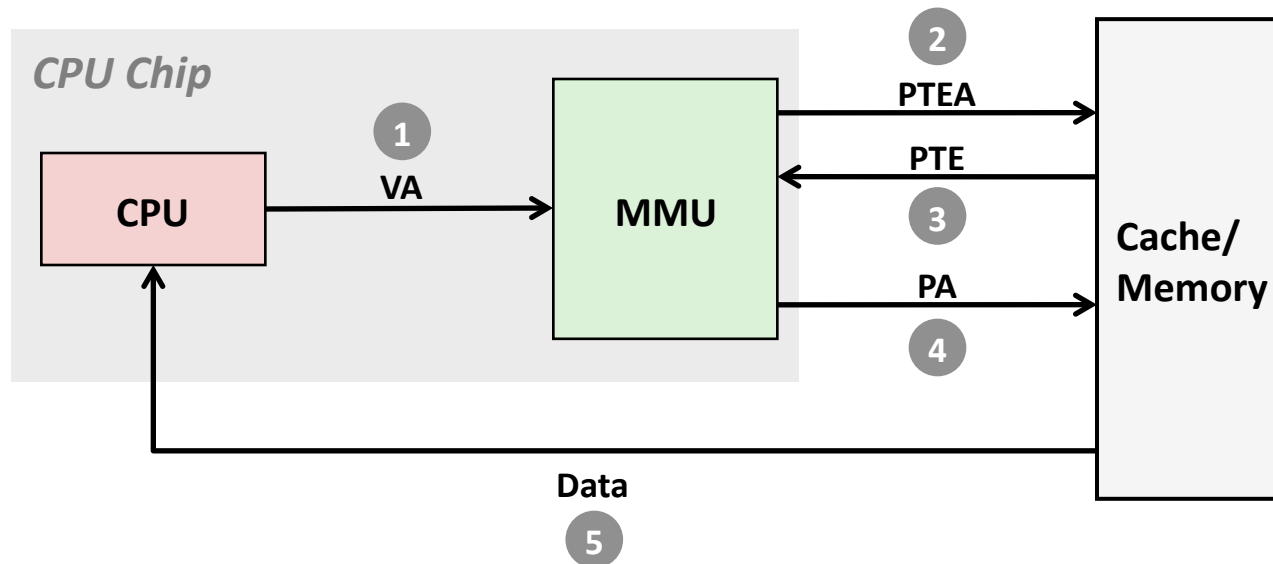
## ■ Components of the physical address (PA)

- **PPO**: Physical page offset (same as VPO)
- **PPN**: Physical page number
- **CO**: Byte offset within cache line
- **CI**: Cache index
- **CT**: Cache tag

# Address Translation With a Page Table



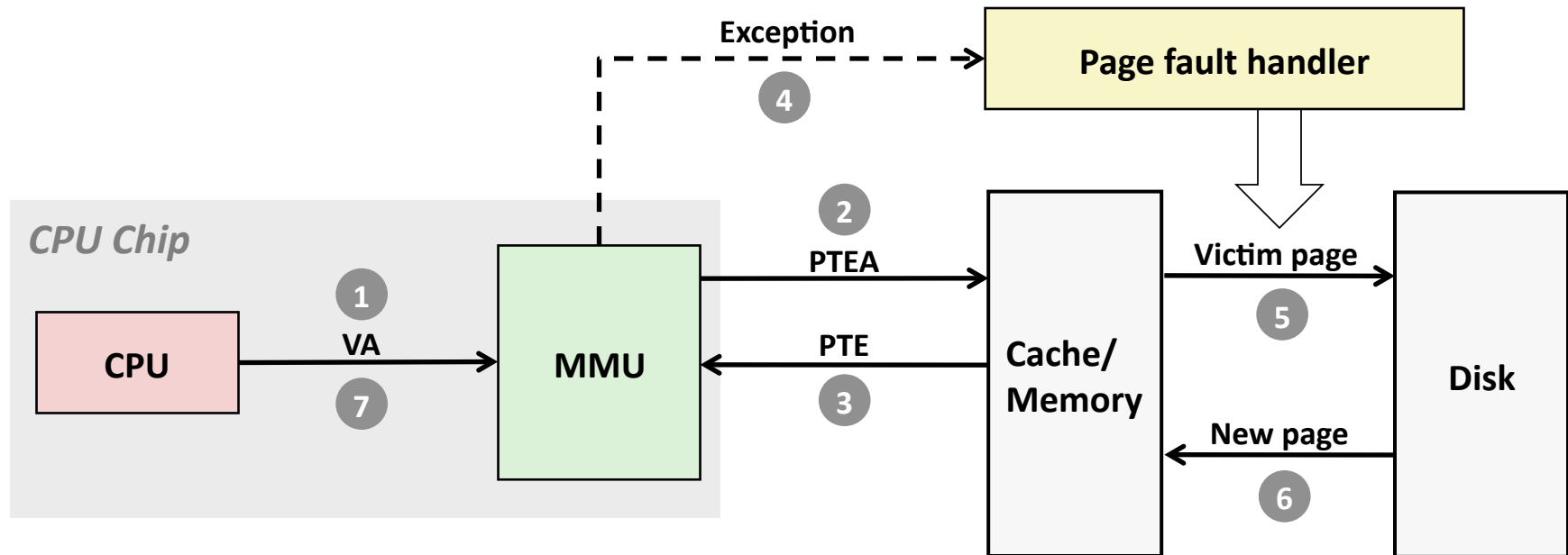
# Address Translation: Page Hit



- 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



# 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

# Views of virtual memory

## ■ 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