

Jin-Soo Kim  
(jinsoo.kim@snu.ac.kr)

Systems Software &  
Architecture Lab.  
Seoul National University

Fall 2018

# Virtual Memory



# Virtualizing Memory

- Example

```
#include <stdio.h>

int n = 0;

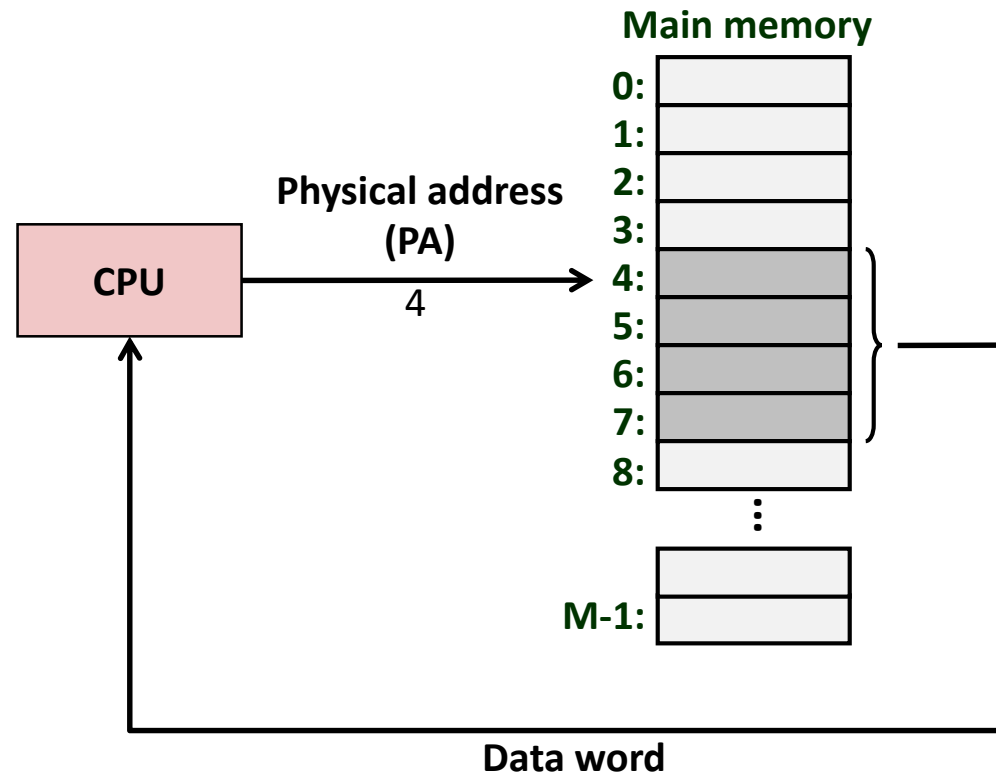
int main ()
{
    n++;
    printf (“&n = %p, n = %d\n”, &n, n);
}

% ./a.out
&n = 0x0804a024, n = 1
% ./a.out
&n = 0x0804a024, n = 1
```

- What happens if two users simultaneously run this program?

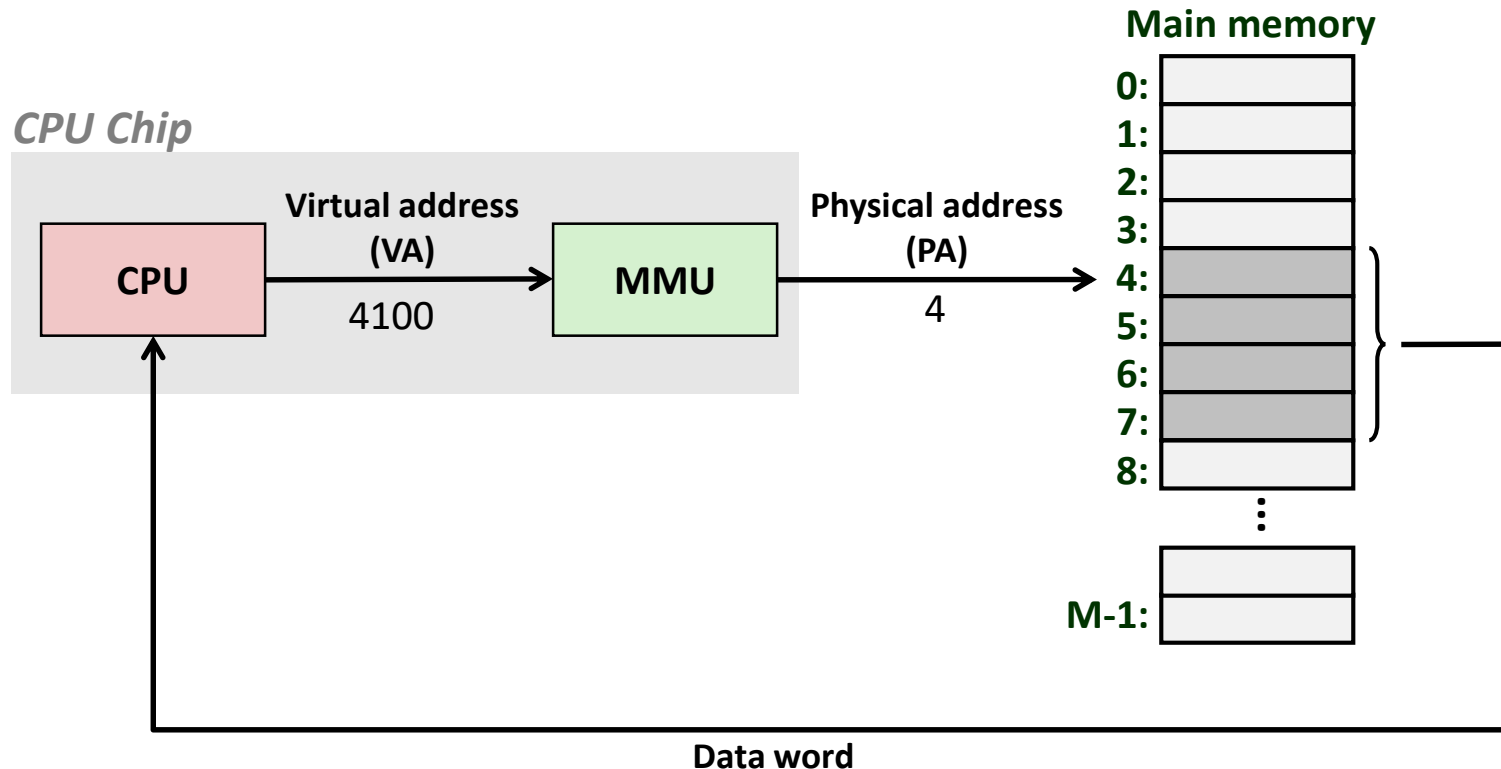
# Physical Addressing

- Used in “simple” systems like embedded microcontrollers
  - Cars, elevators, digital cameras, etc.



# Virtual Addressing

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



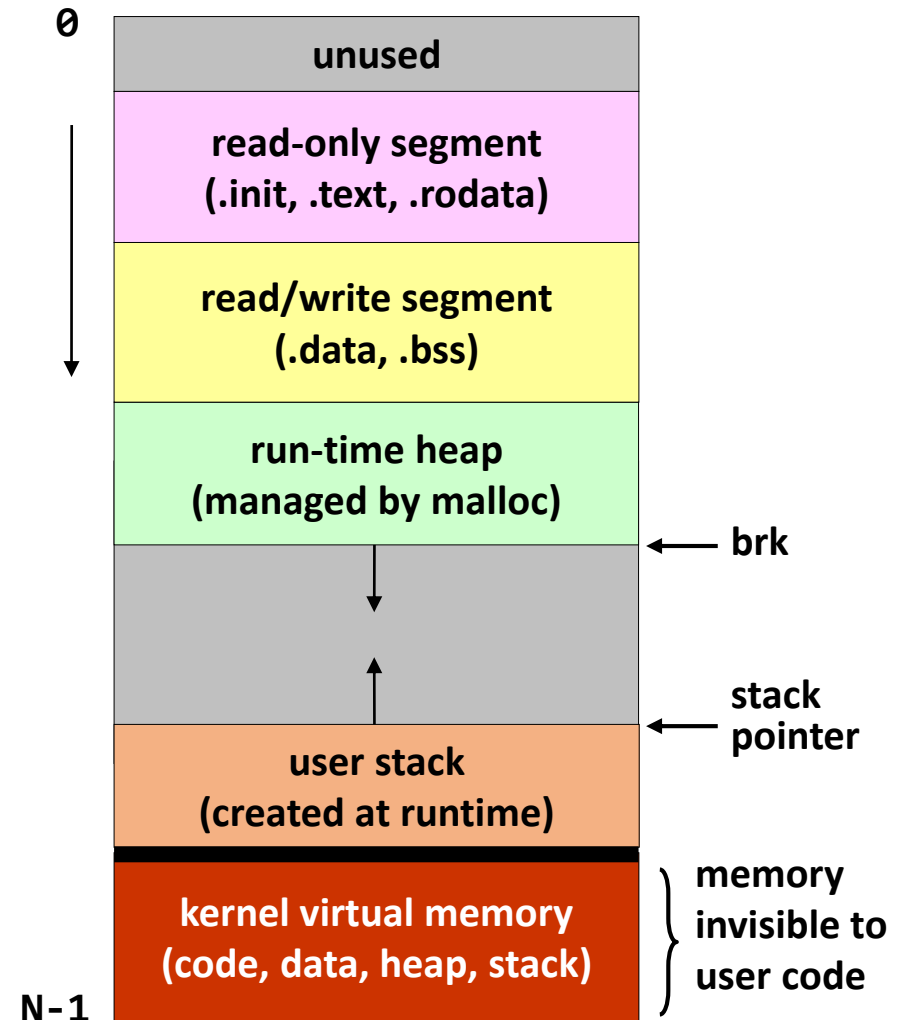
# Virtual Memory

- Each process has its own **virtual address space**
  - Large and contiguous
  - Use virtual addresses for memory references
  - Virtual addresses are private to each process
- **Address translation** is performed at run time
  - From a virtual address to the corresponding physical address
- Supports **lazy allocation**
  - Physical memory is dynamically allocated or released on demand
  - Programs execute without requiring their entire address space to be resident in physical memory

# (Virtual) Address space

## ■ Process' abstract view of memory

- OS provides illusion of private address space to each process
- Contains all of the memory state of the process
- Static area
  - Allocated on `exec()`
  - Code & Data
- Dynamic area
  - Allocated at runtime
  - Can grow or shrink
  - Heap & Stack

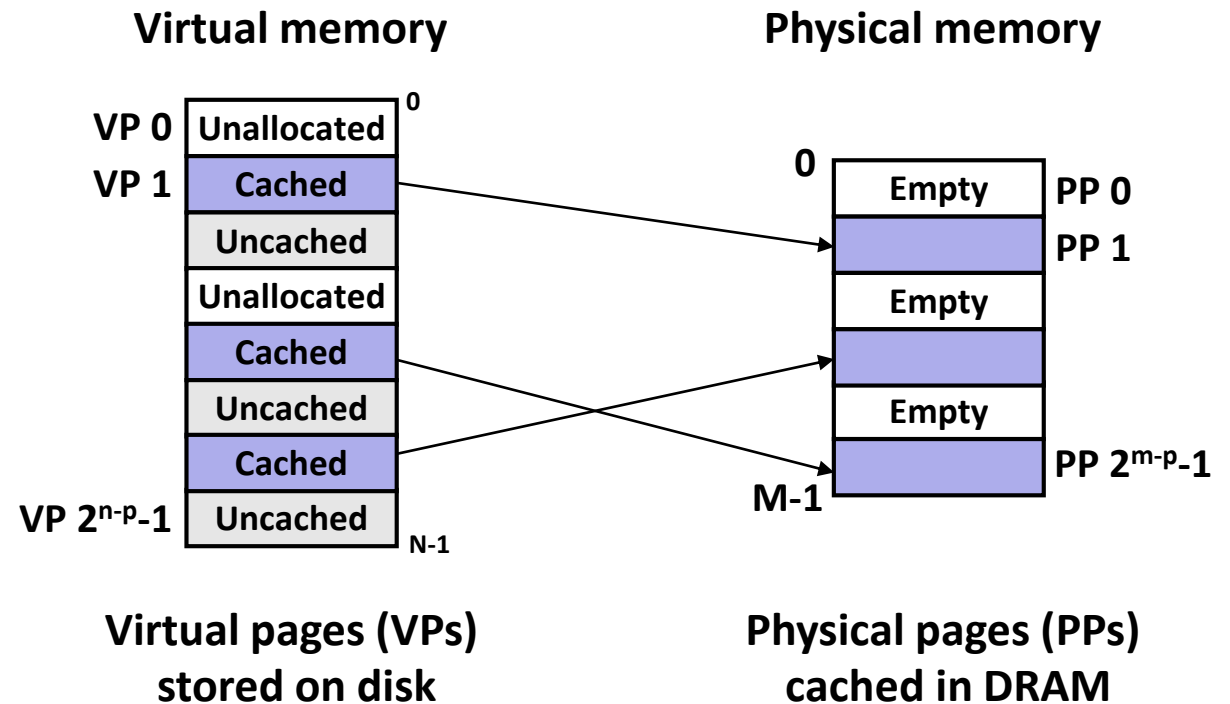


# Why Virtual Memory (VM)?

- VM as a tool for caching
  - Uses main memory efficiently
  - Use DRAM as a cache for parts of a virtual address space
- VM as a tool for memory management
  - Simplifies memory management
  - Each process gets the same uniform linear address space
- VM as a tool for memory protection
  - Isolates address spaces
  - One process can't interfere with another's memory
  - User program cannot access privileged kernel information and code

# 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)



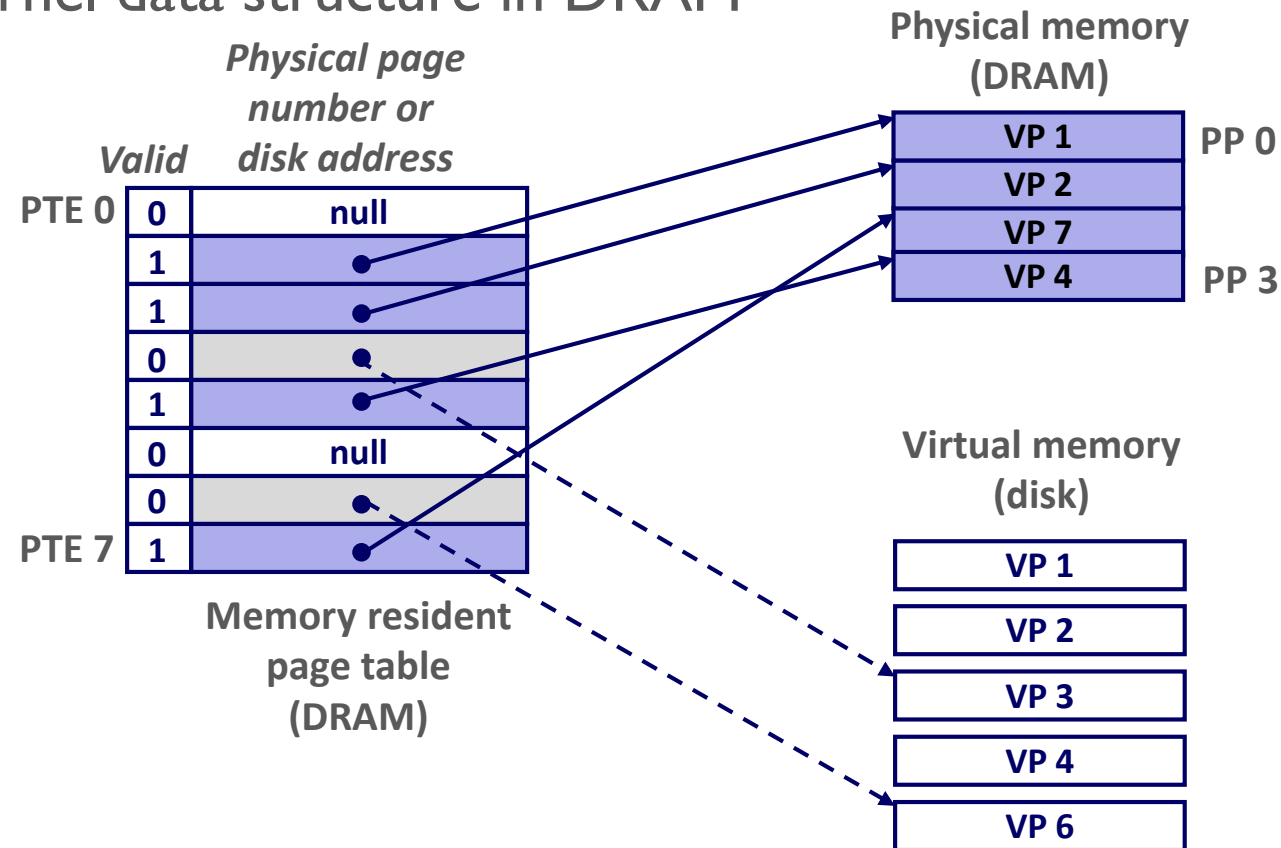


# DRAM Cache Organization

- 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 4KB, sometimes 4MB
  - 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

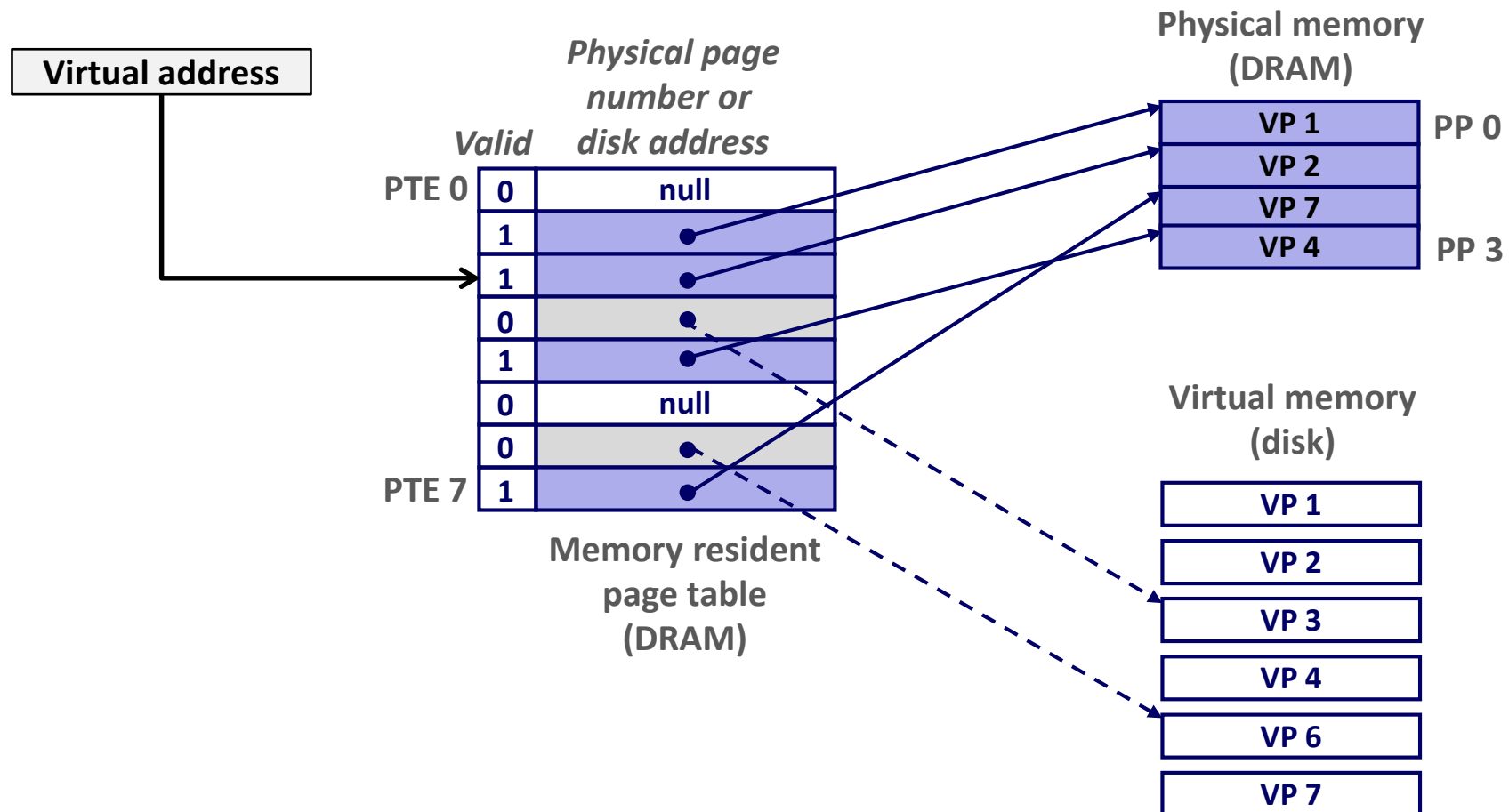
# Page Table

- 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



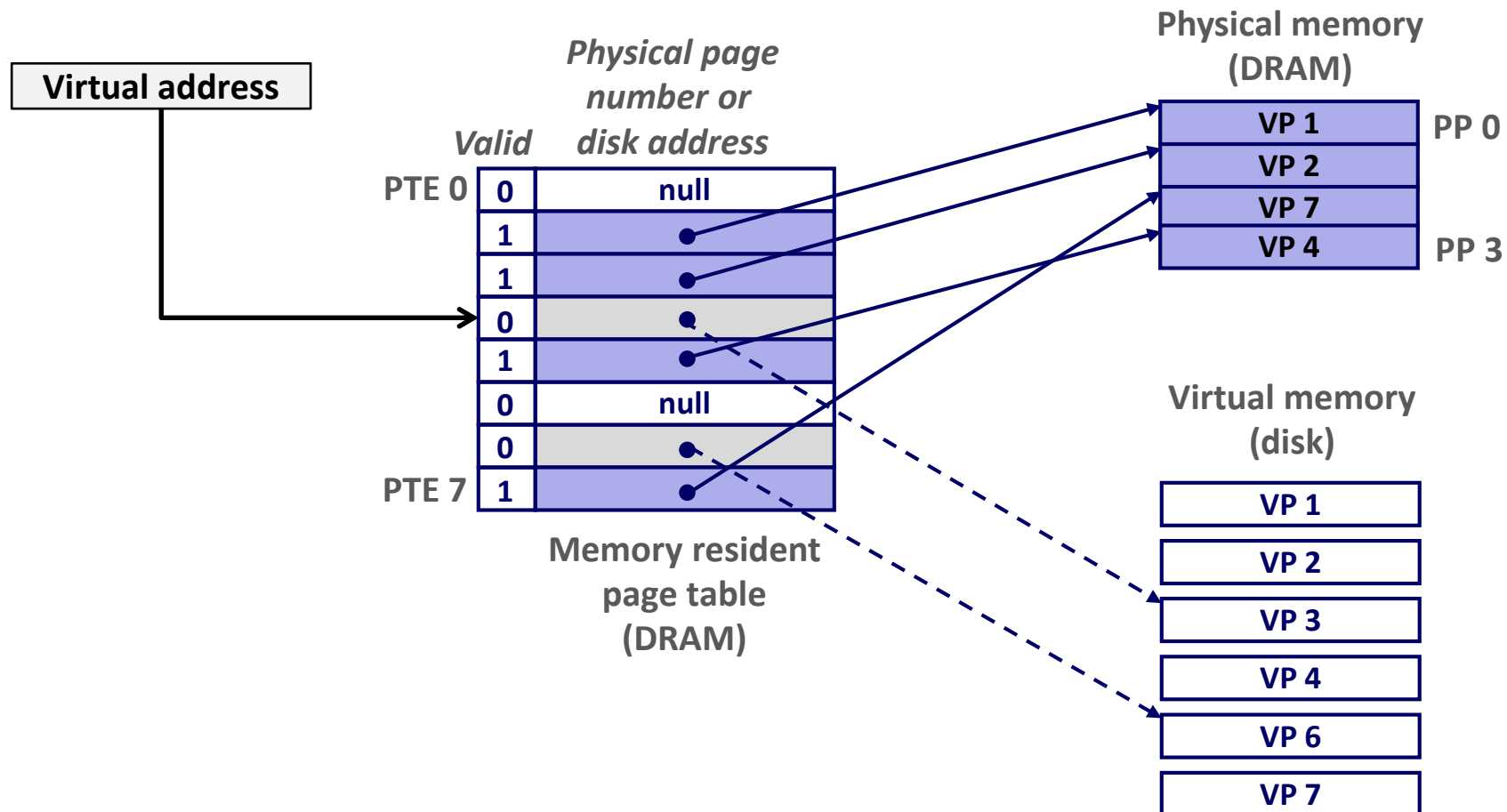
# Page Hit

- Reference to VM word that is in physical memory (DRAM cache hit)



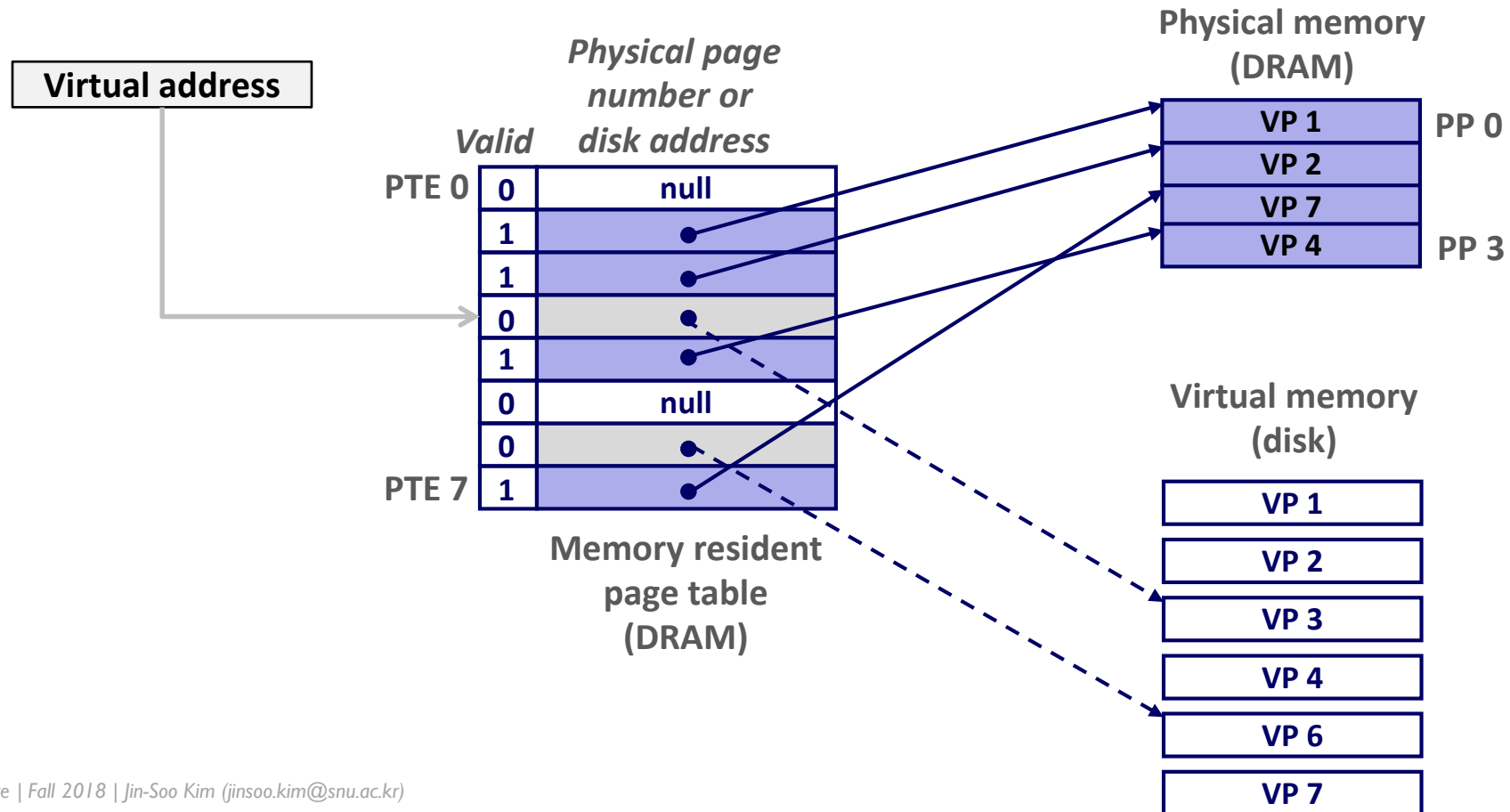
# Page Fault

- Reference to VM word that is not in physical memory (DRAM cache miss)



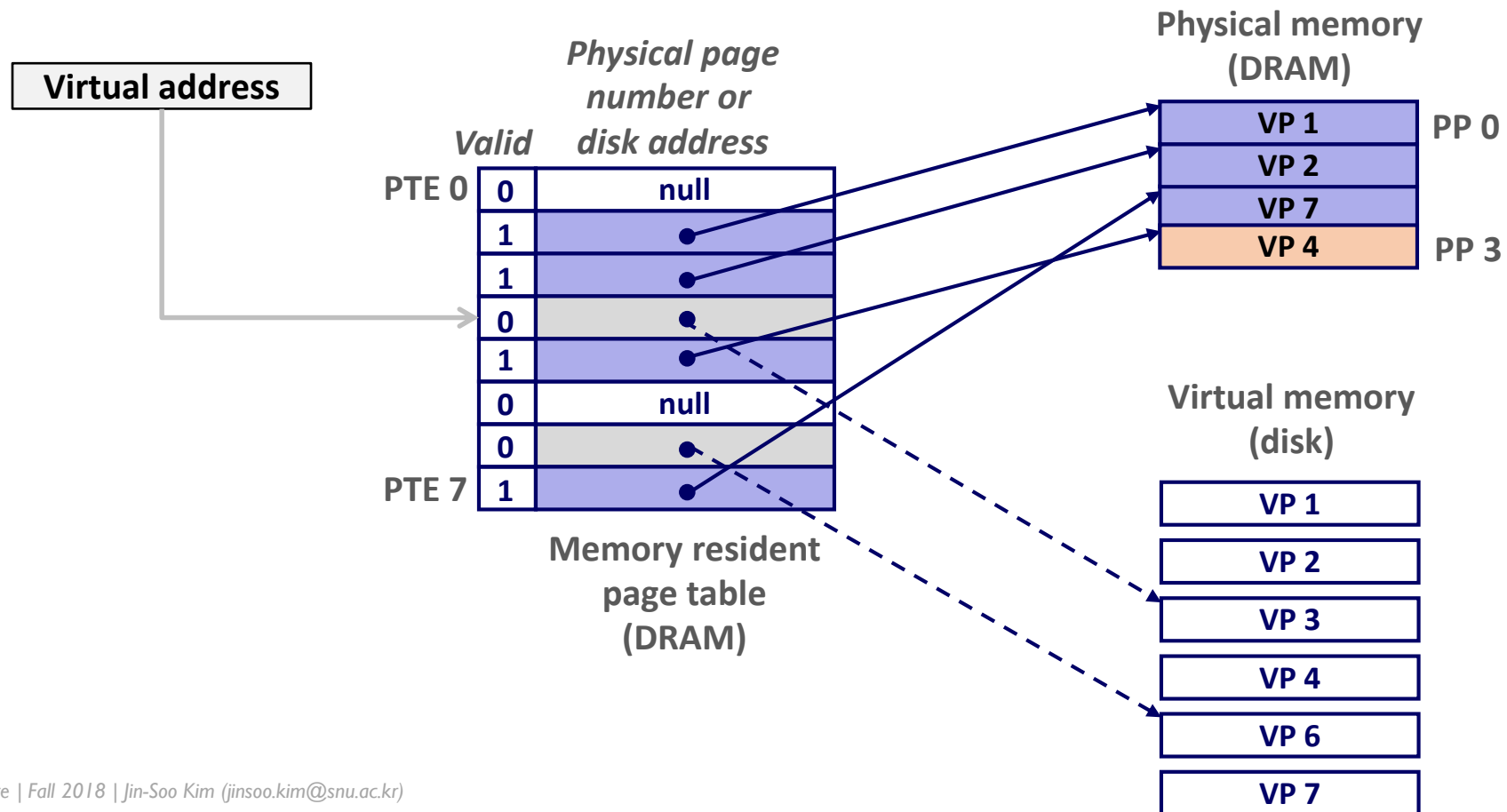
# Handling Page Fault

- Page miss causes page fault (an exception)



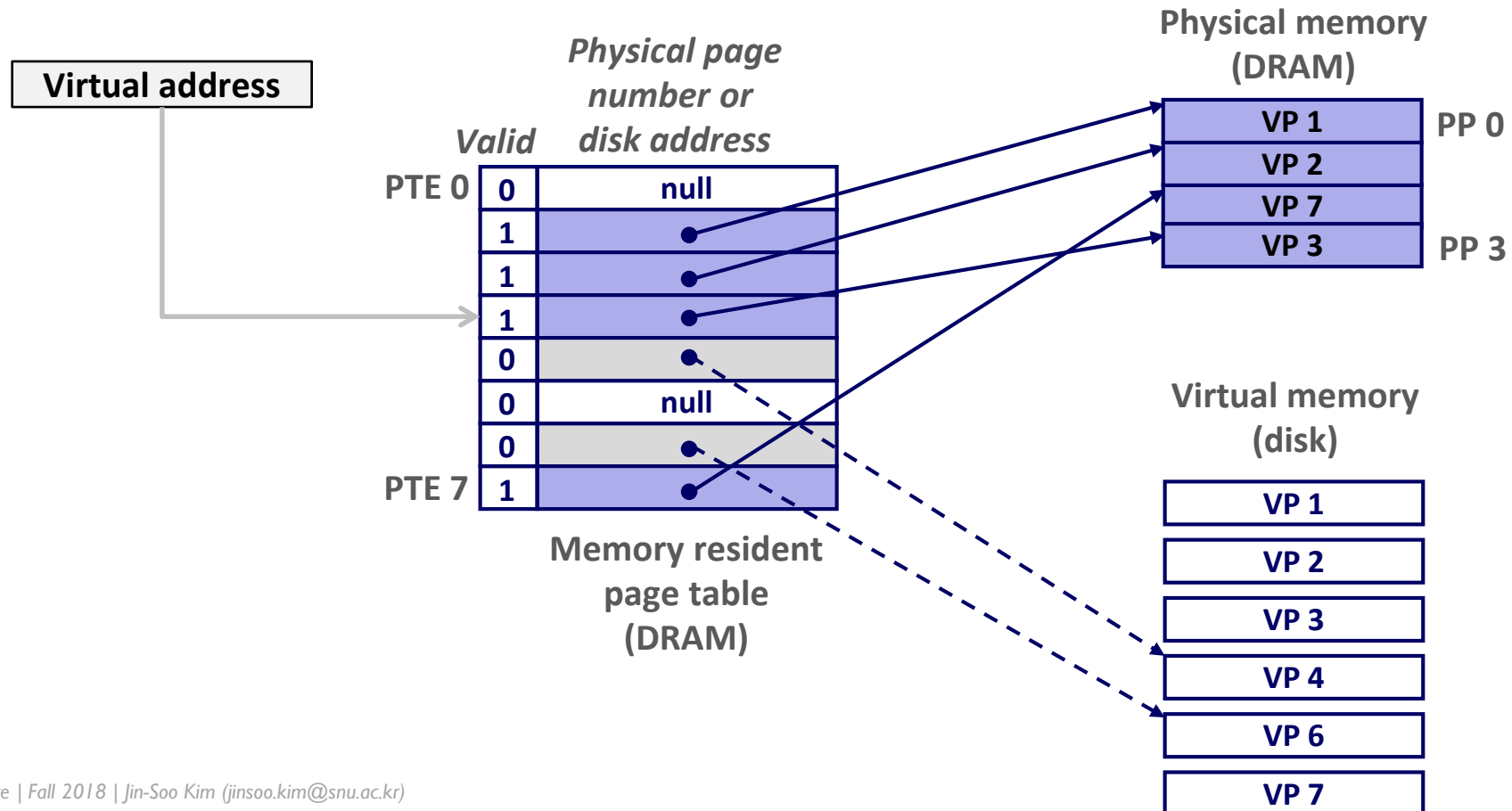
# Handling Page Fault

- Page fault handler selects a victim to be evicted (here VP 4)



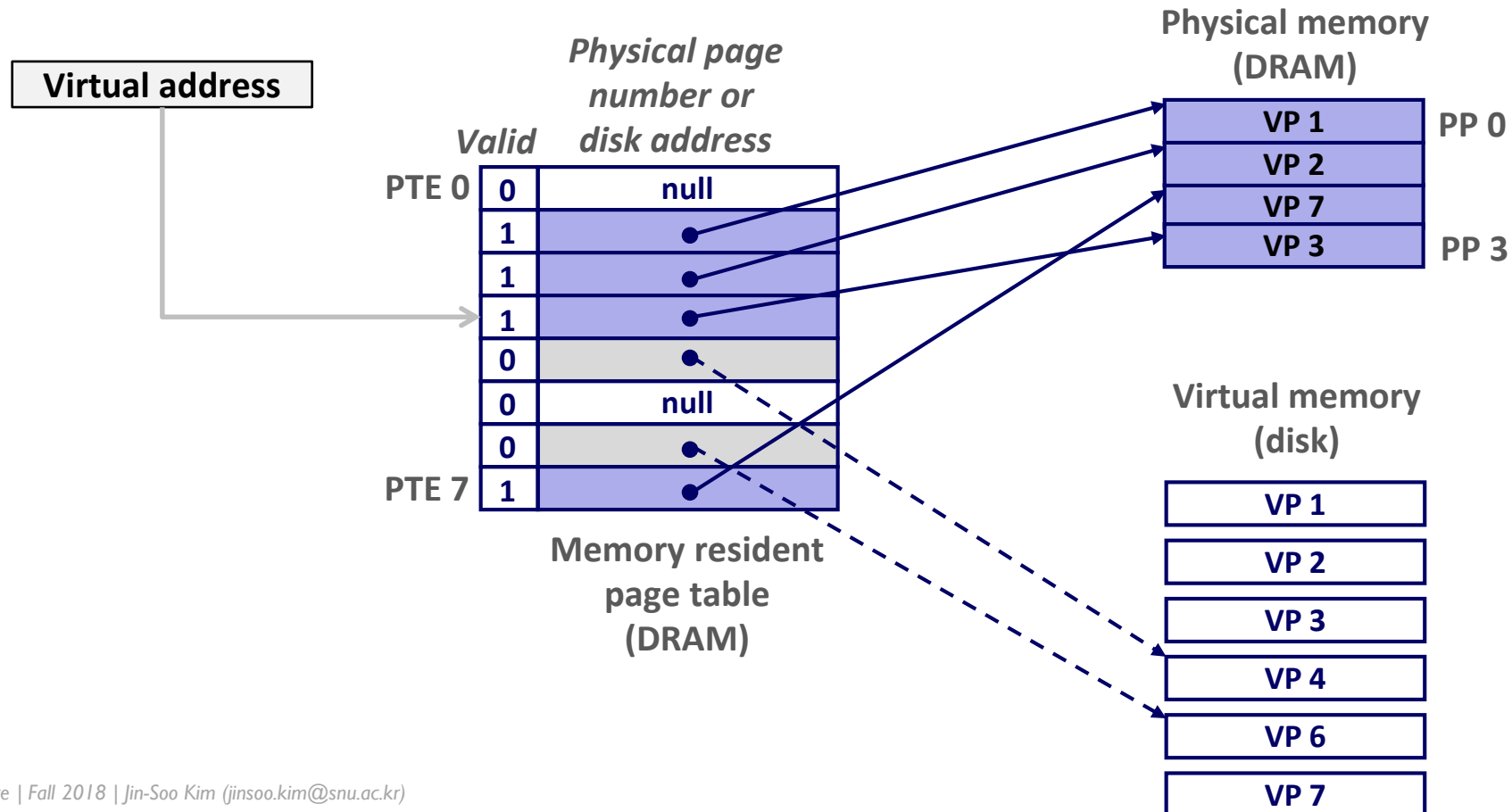
# Handling Page Fault

- Page fault handler loads the faulting page into physical memory (here VP 3)



# Handling Page Fault

- Offending instruction is restarted: page hit!
- **Demand paging:** wait until the miss to copy the page to DRAM



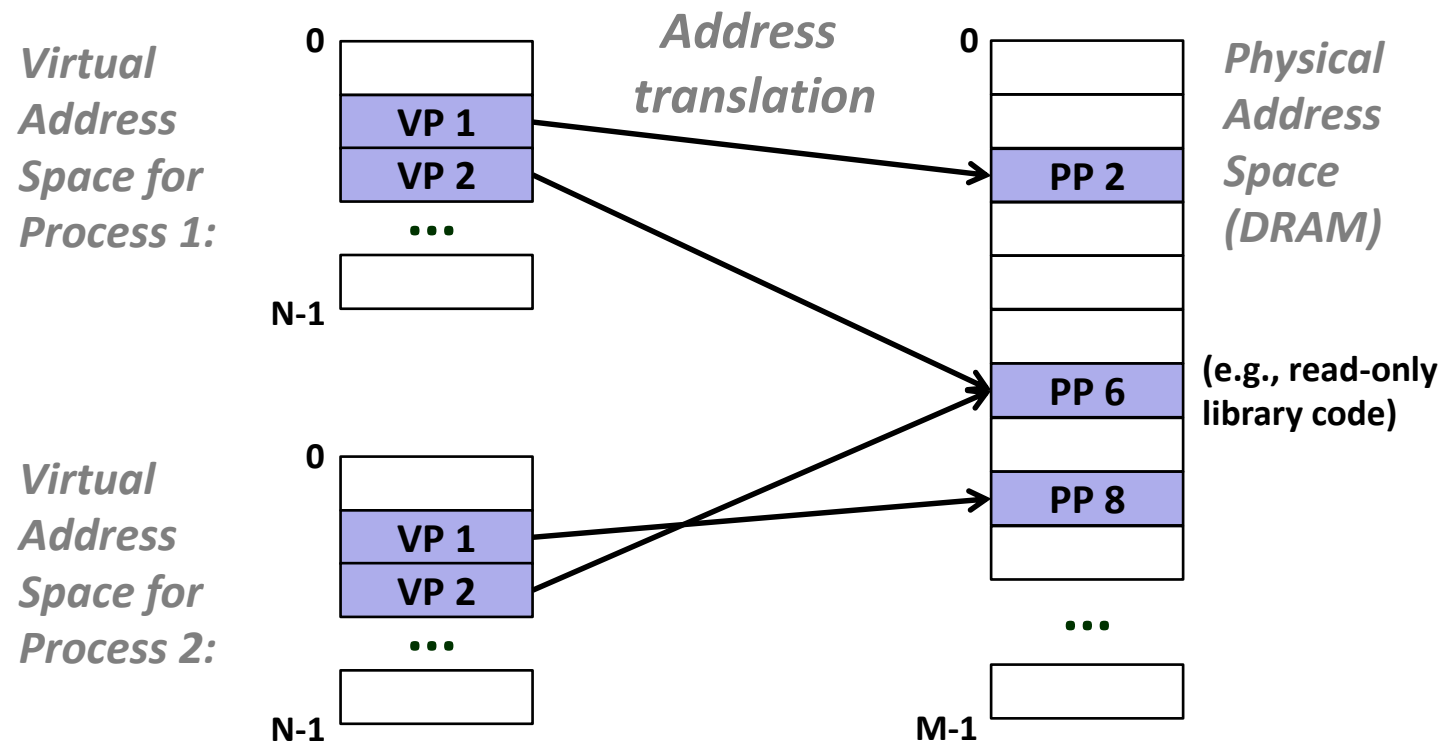


# Locality to the Rescue Again!

- 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

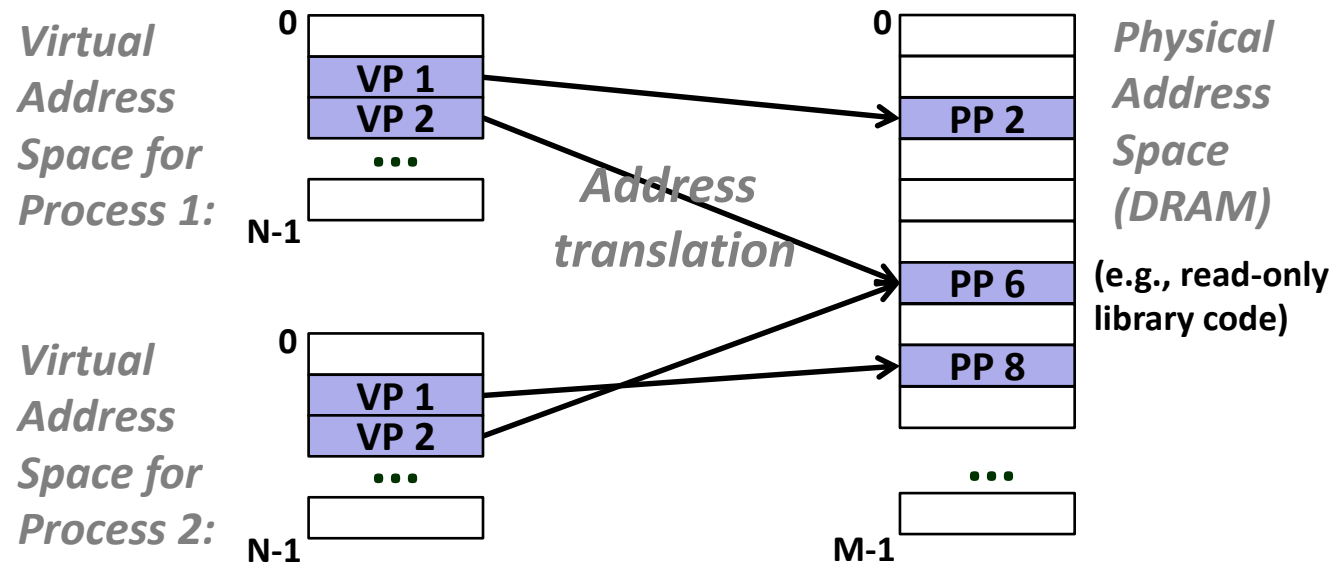
# VM as a Tool for Memory Management

- Each process has its own virtual address space
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory



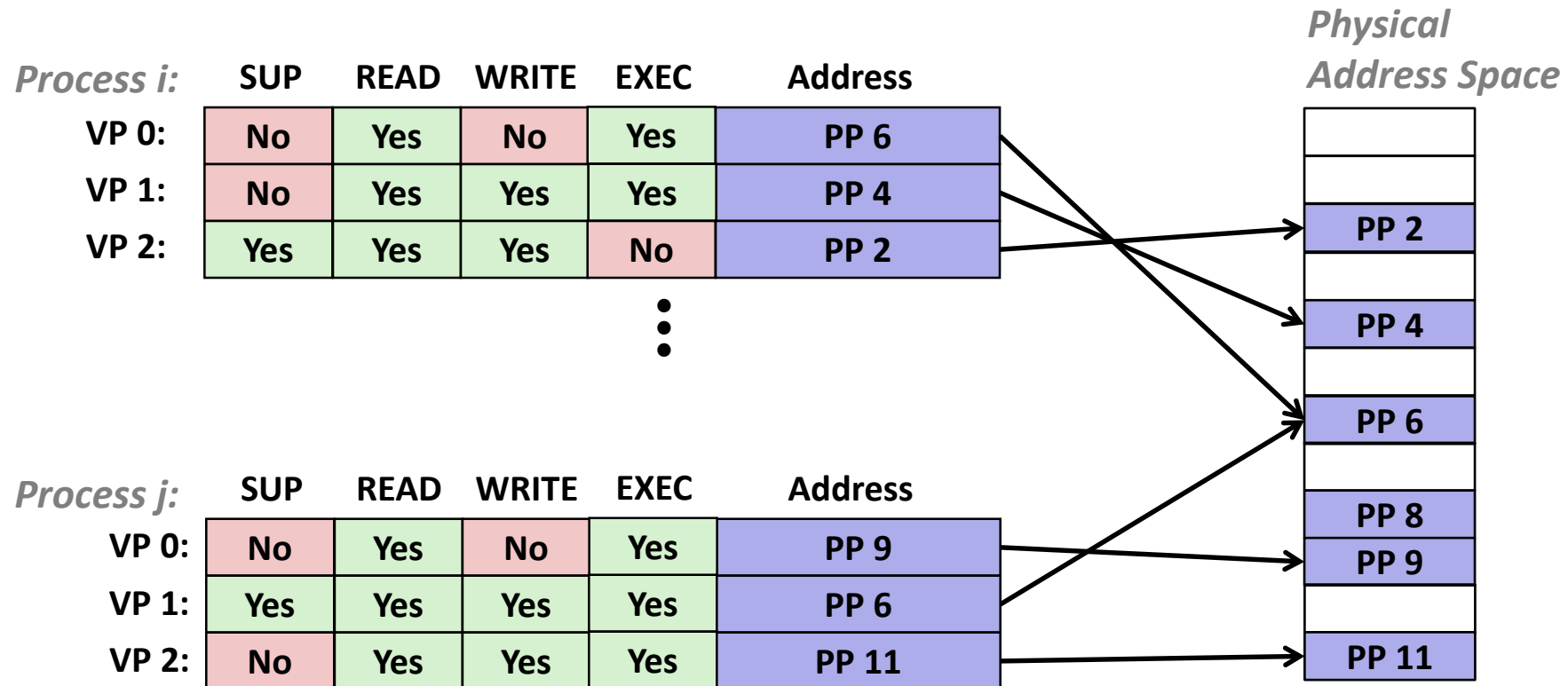
# VM as a Tool for Memory Management

- 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)



# VM as a Tool for Memory Protection

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



# Optimizing Virtual Memory

# VM Address Translation

- Virtual Address Space

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

- Physical Address Space

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

- Address Translation

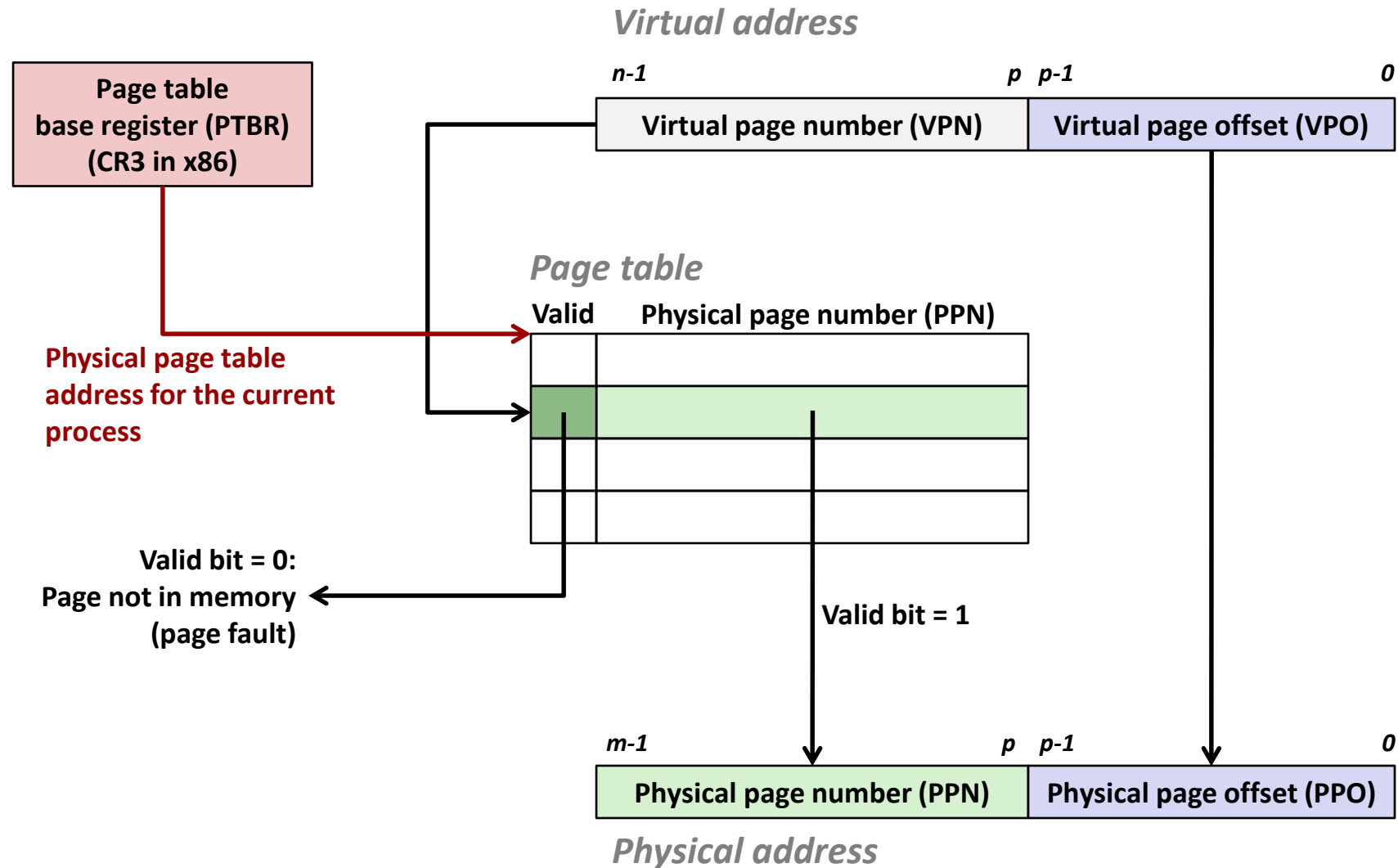
- $MAP: V \rightarrow P \cup \{\emptyset\}$

- For virtual address  $v$

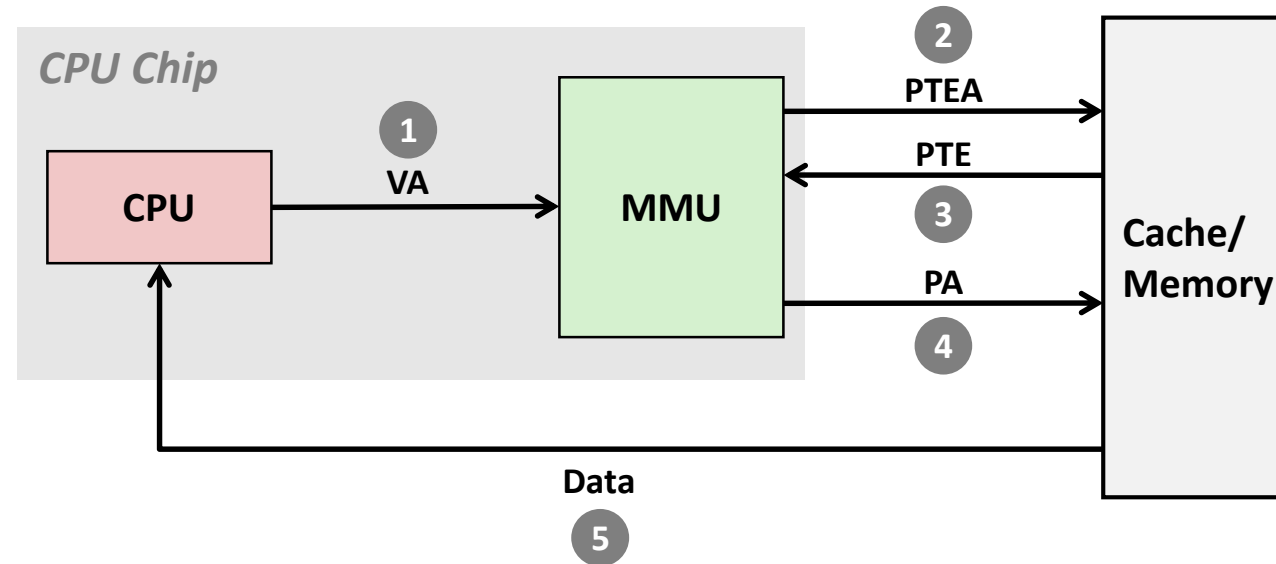
- $MAP(v) = p$  if data at virtual address  $v$  is at physical address  $p$  in  $P$

- $MAP(v) = \emptyset$  if data at virtual address  $v$  is not in physical memory (either invalid or stored on disk)

# Address Translation with a Page Table



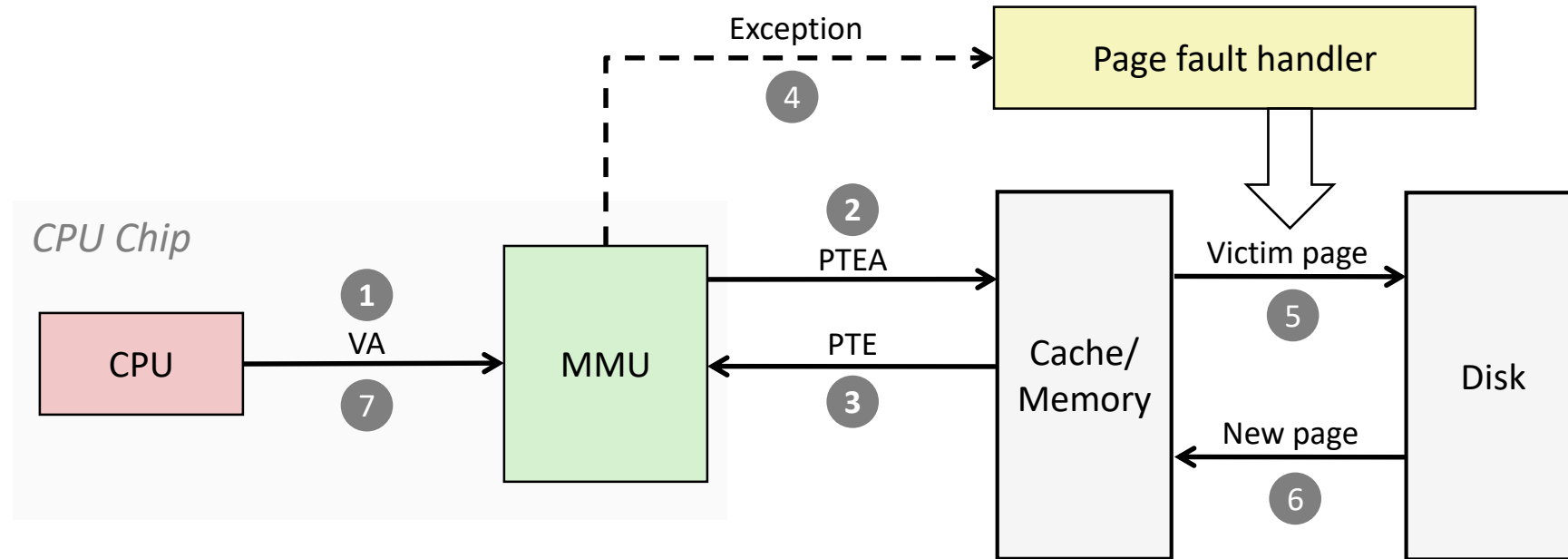
# Address Translation: Page Hit



- 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

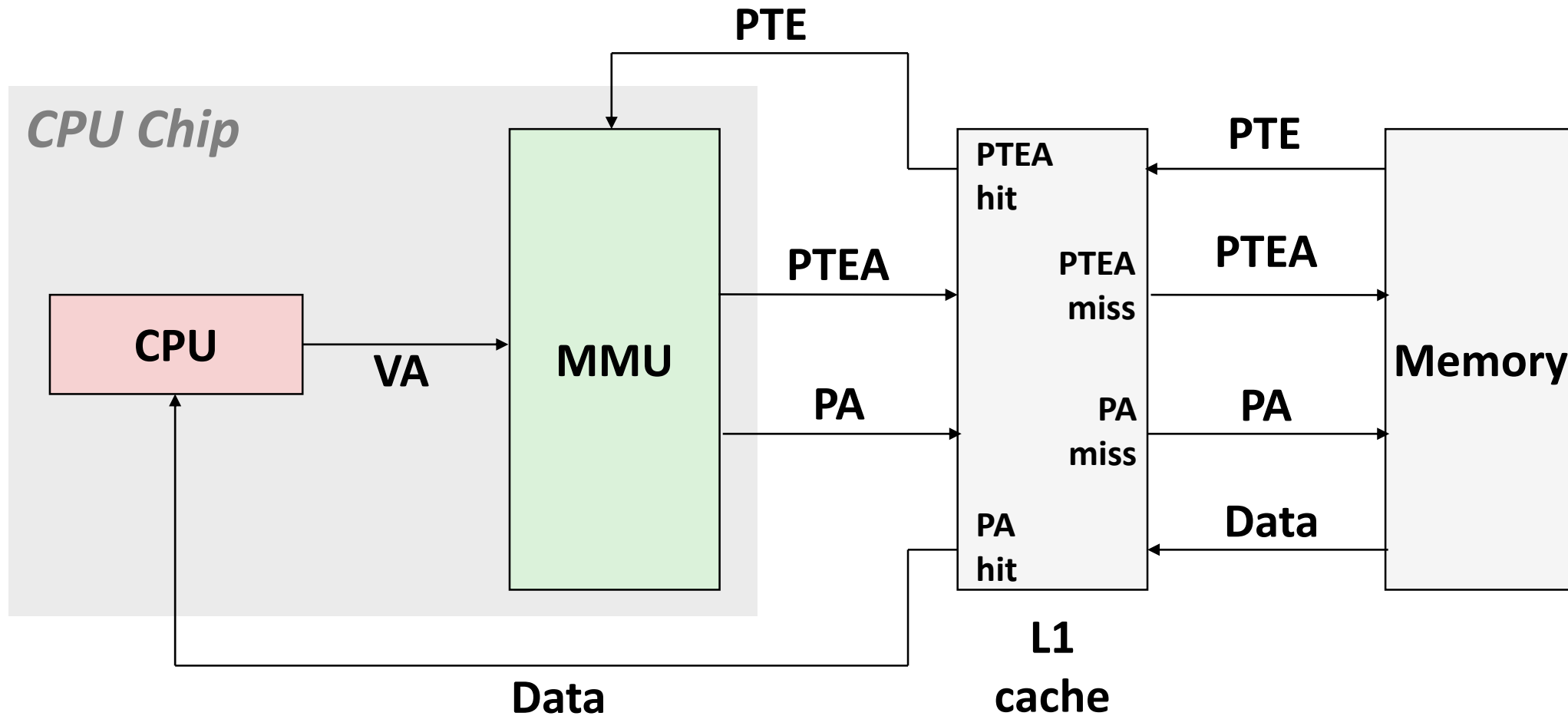


# Address Translation: Page Fault



- 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

# Integrating VM and Cache



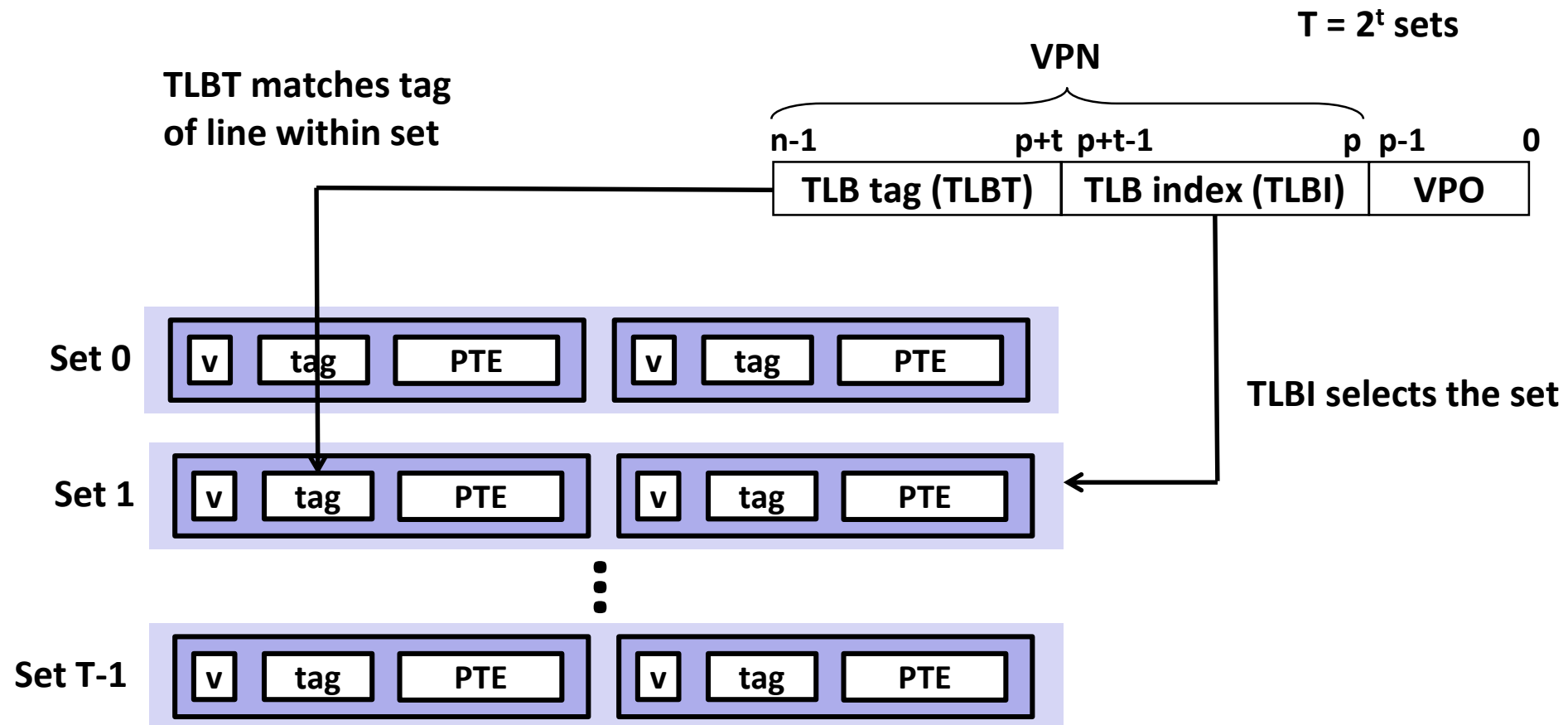
**VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address**

# Speeding up Translation with TLB

- Page table entries (PTEs) are cached in LI like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small LI 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

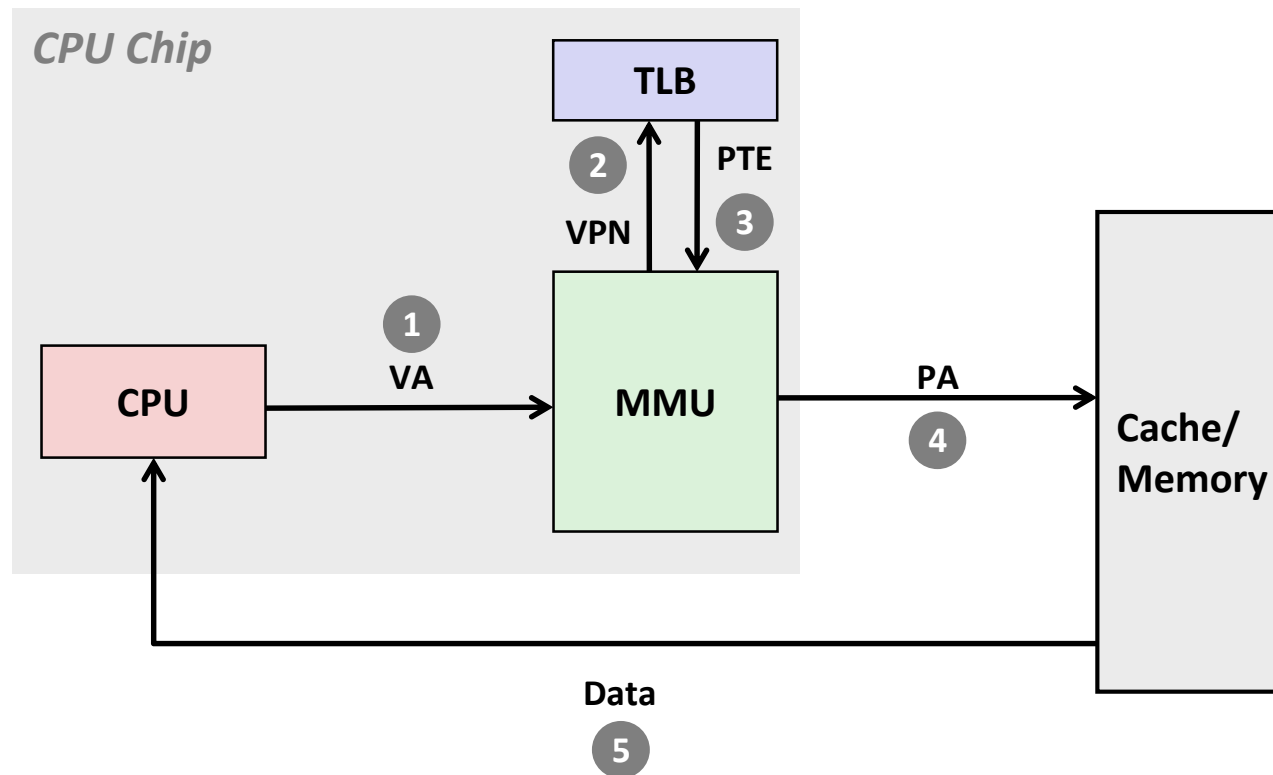
# Accessing the TLB

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



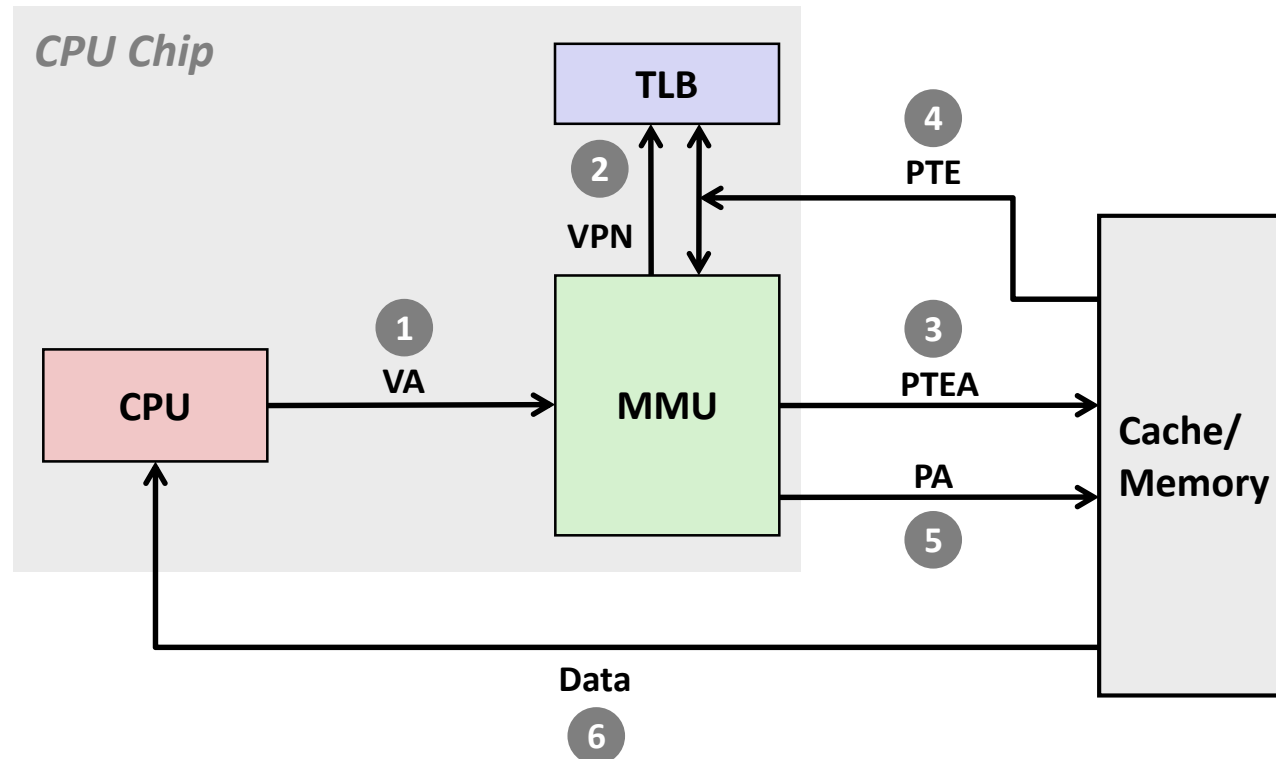
# TLB Hit

- A TLB hit eliminates a memory access



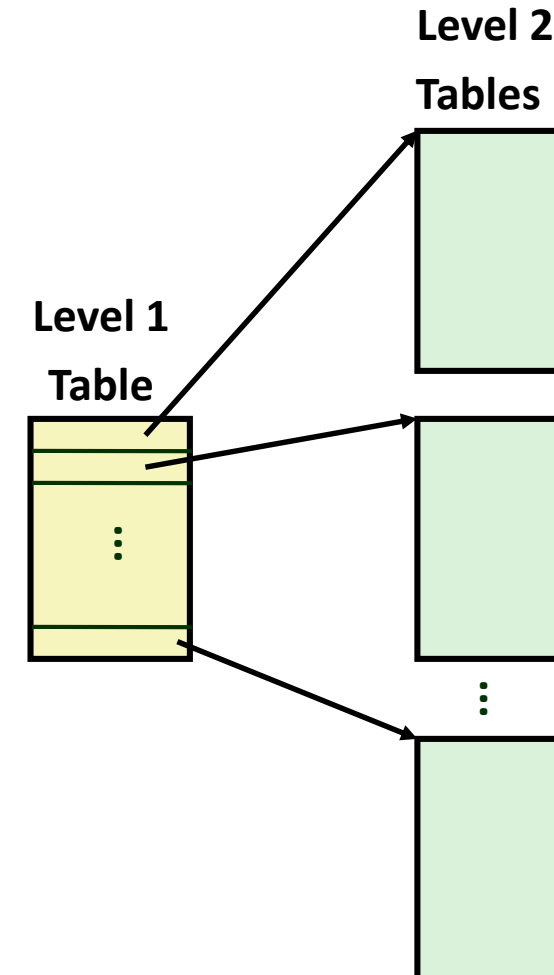
# TLB Miss

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

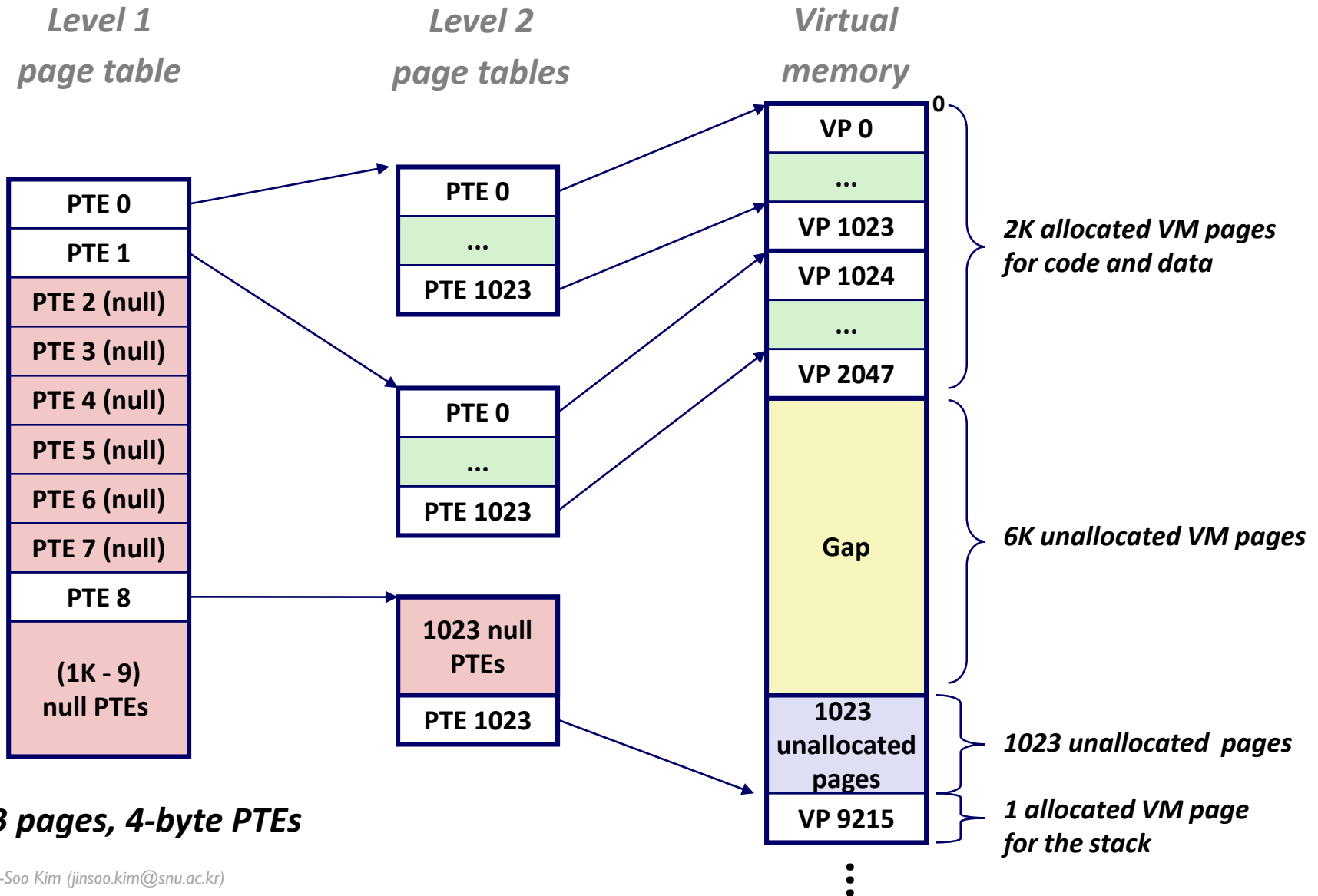


# Multi-Level Page Tables

- **Suppose:**
  - 4KB ( $2^{12}$ ) page size, 48-bit address space, 8-byte PTE
- **Problem:**
  - Would need a 512GB 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)

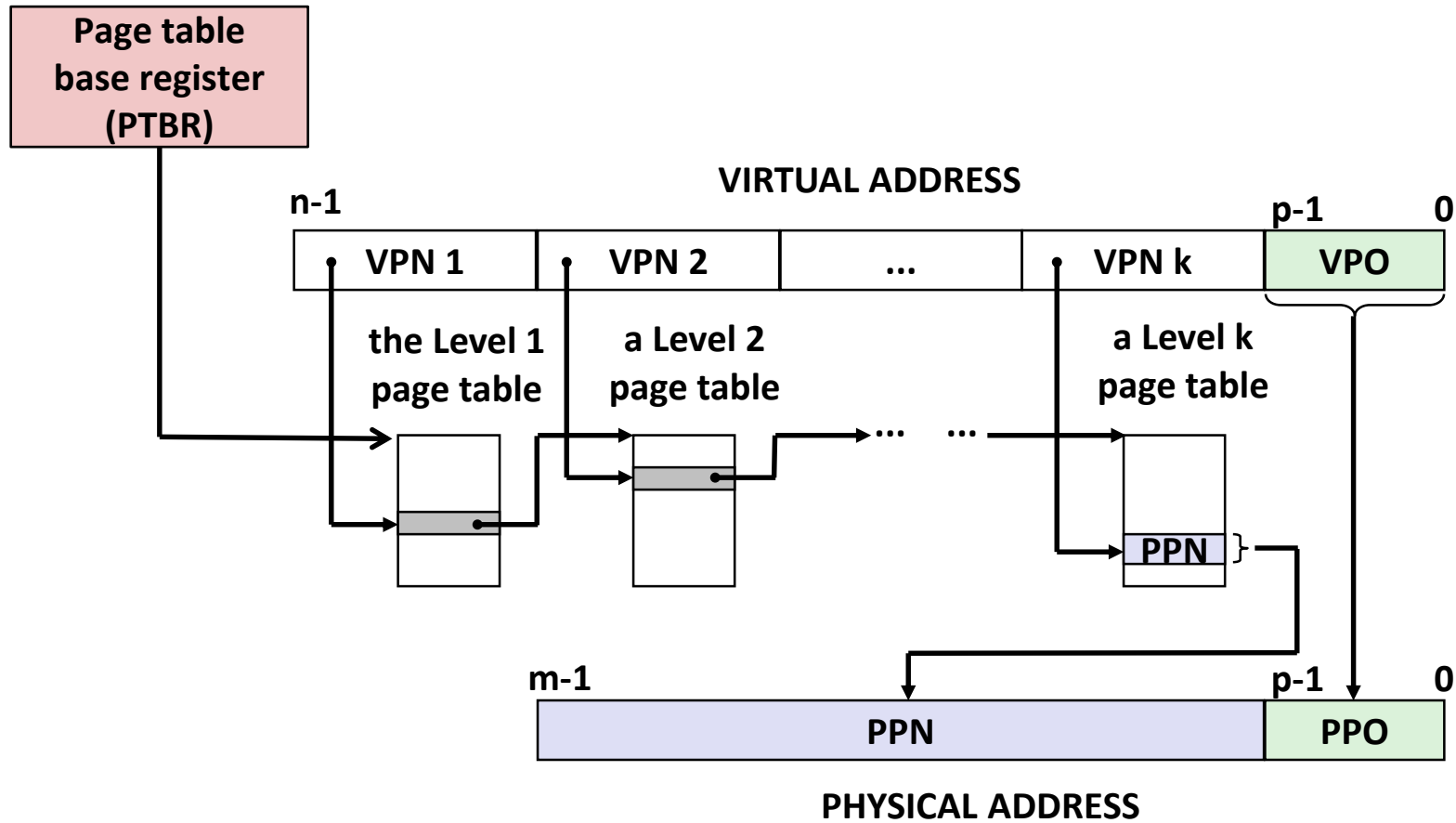


# A Two-Level Page Table Hierarchy



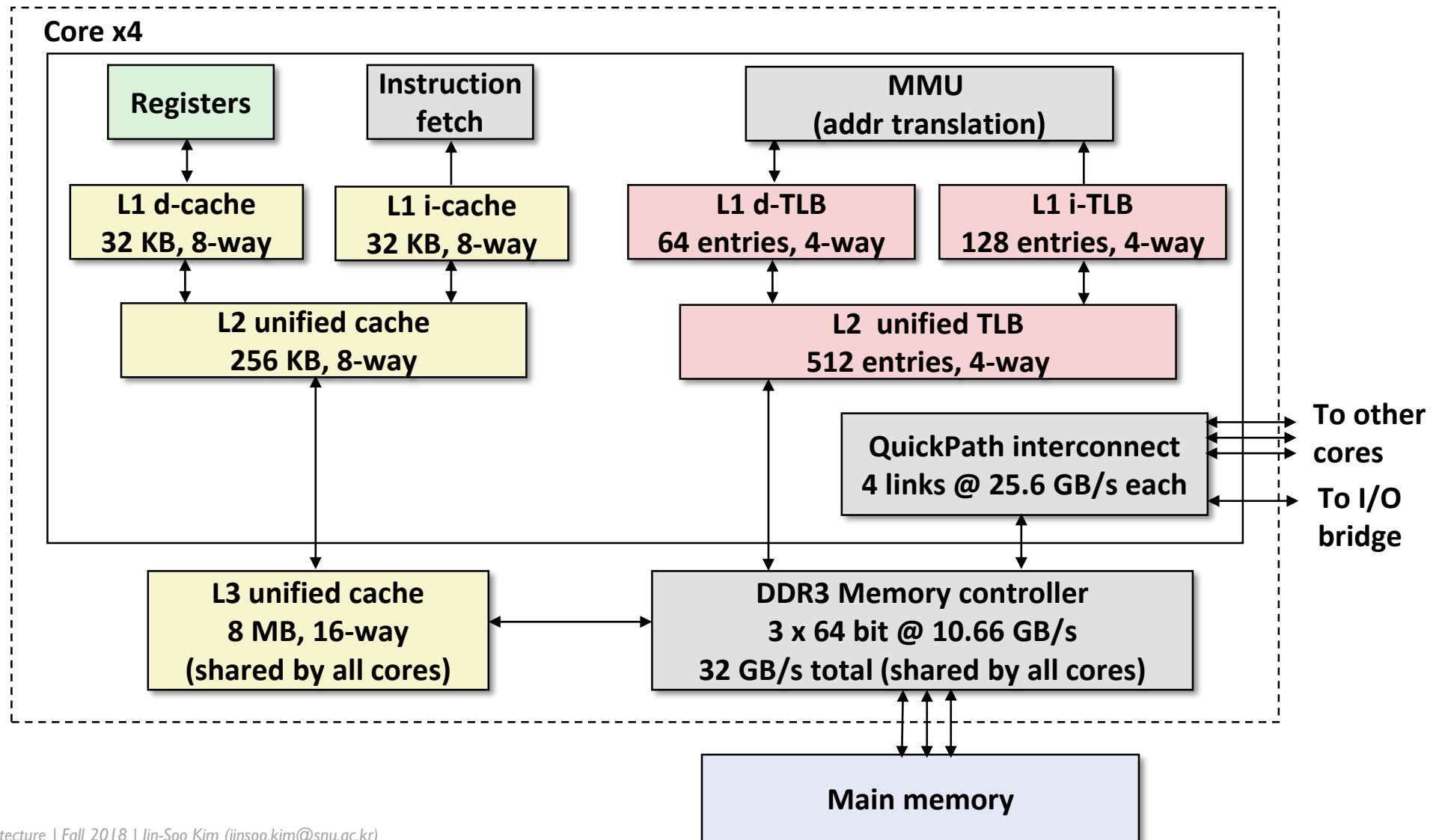


# Translating with a $k$ -level Page Table



# Intel Core i7 Memory System

Processor package



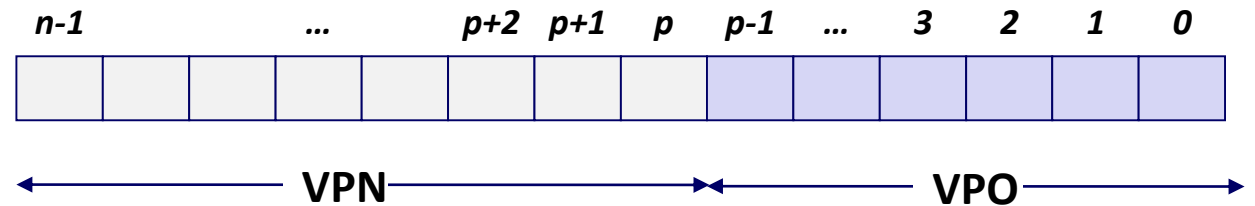
# Symbols

## 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)

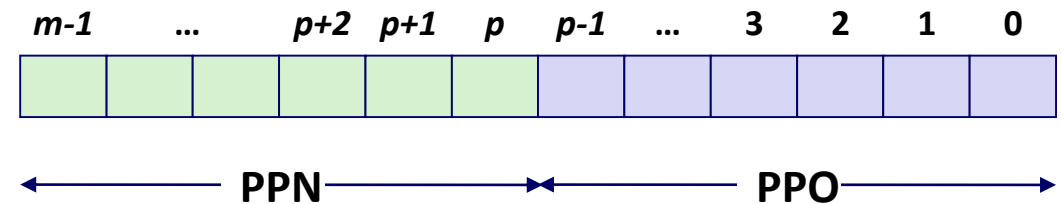
## Components of the virtual address (VA)

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

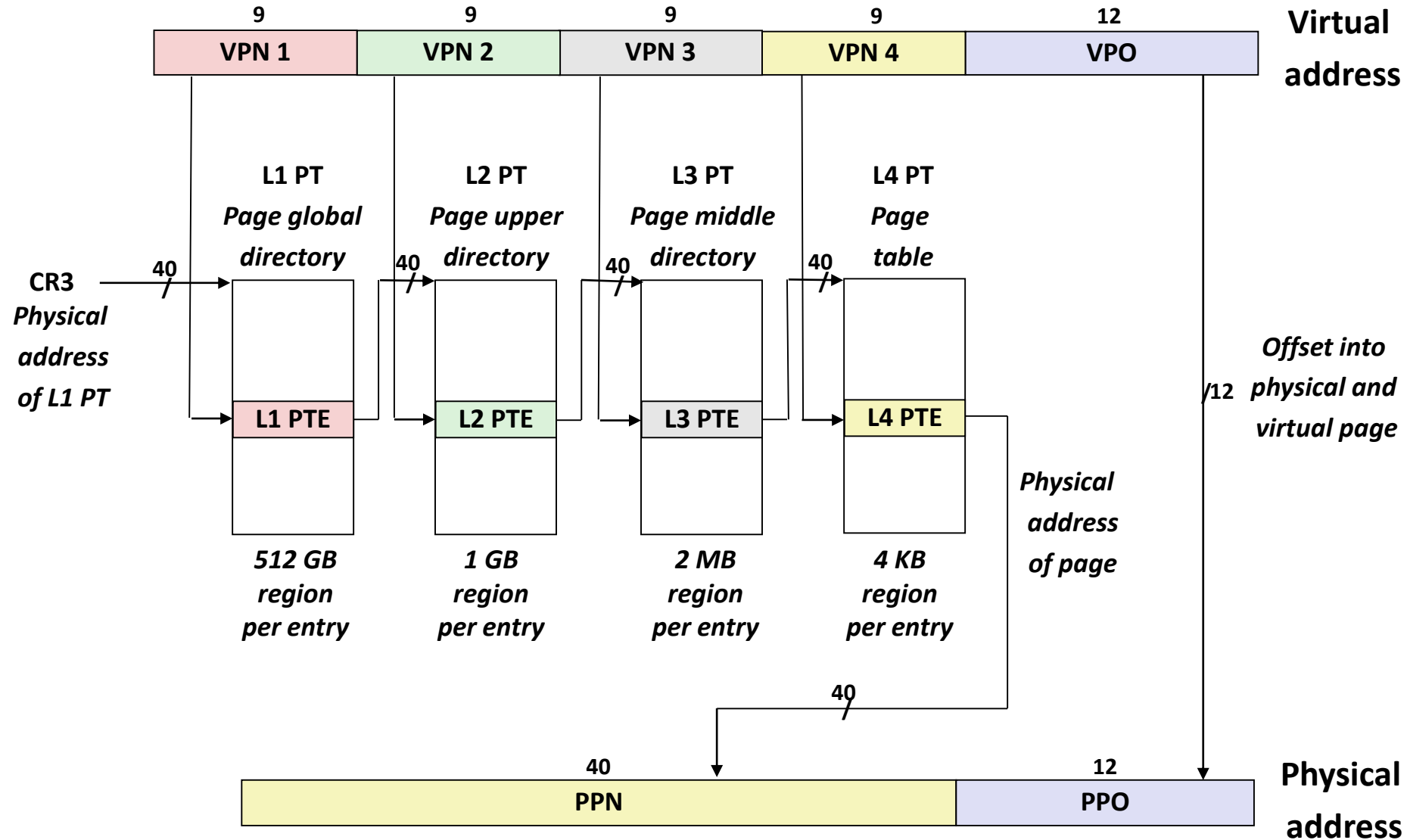


## Components of the physical address (PA)

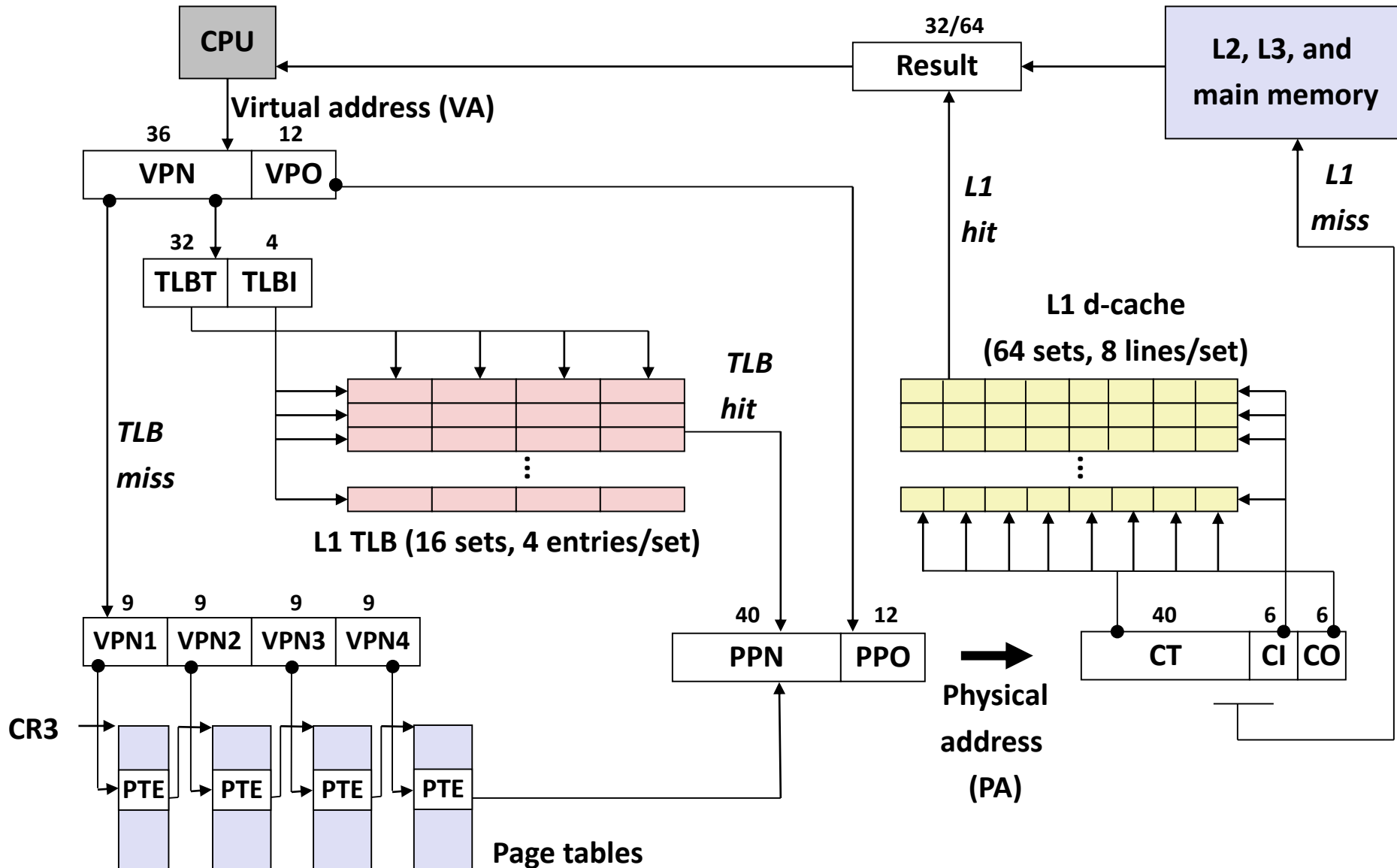
- PPO: Physical page offset (same as VPO)
- PPN: Physical page number
- CO: Byte offset within cache line
- CI: Cache index
- CT: Cache tag



# Core i7 Page Table Translation



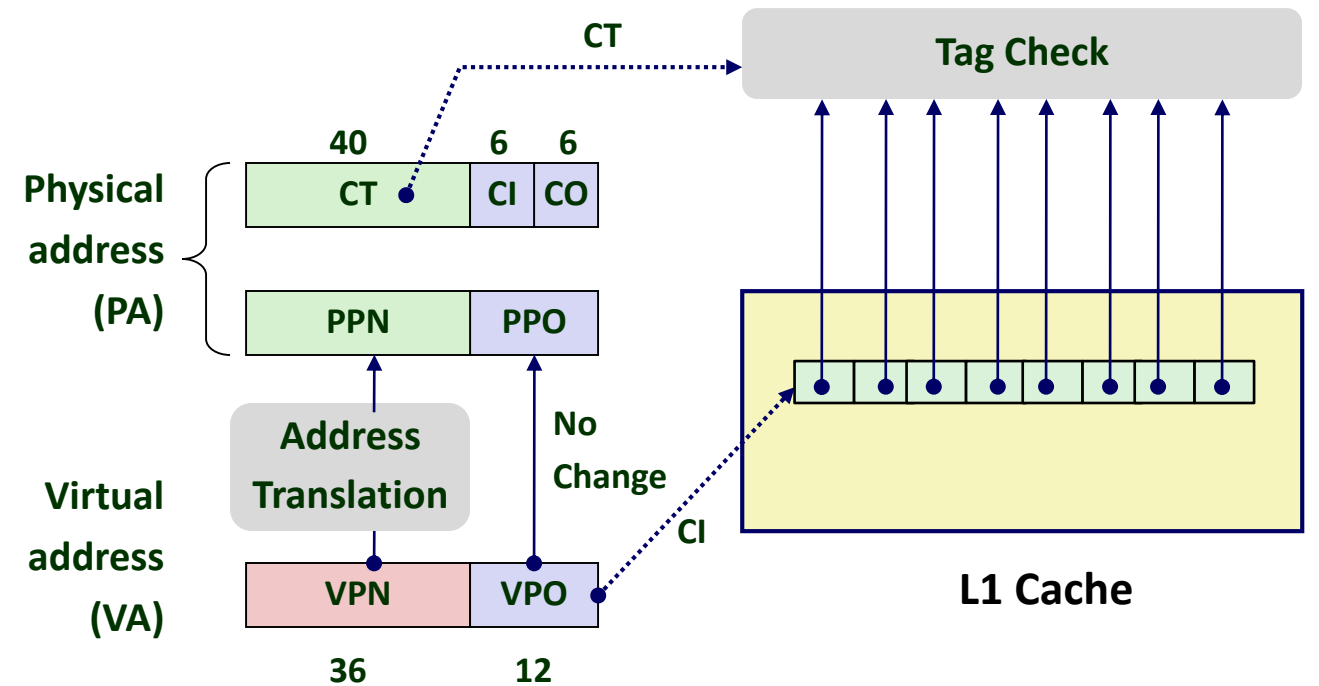
# End-to-end Core i7 Address Translation



# Cute Trick for Speeding up LI Access

## ■ Observation

- Bits that determine CI identical in virtual and physical address
- Can index into cache while address translation taking place
- Generally we hit in TLB, so PPN bits (CT bits) available next
- Cache carefully sized to make this possible
- “**Virtually indexed, physically tagged**”



CT: Cache tag  
CI: Cache index  
CO: Byte offset within cache line

# 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