Virtual Memory: Concepts

15-213 / 18-213: Introduction to Computer Systems
19th Lecture, Jul 1, 2015
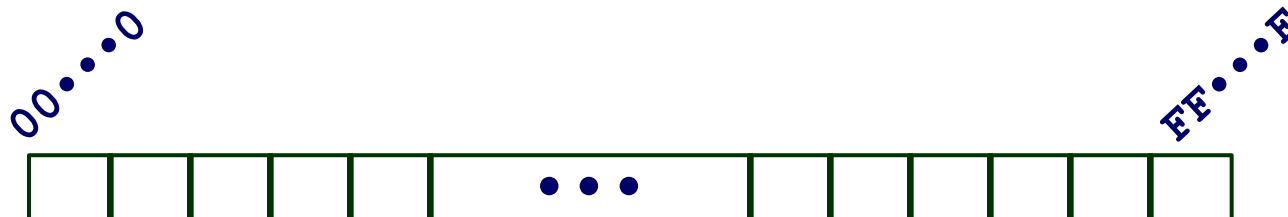
Instructors:

nwf 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

Recall: Simple Addressing Modes

- **Normal   (R)   Mem[Reg[R]]**
  - Register R specifies memory address
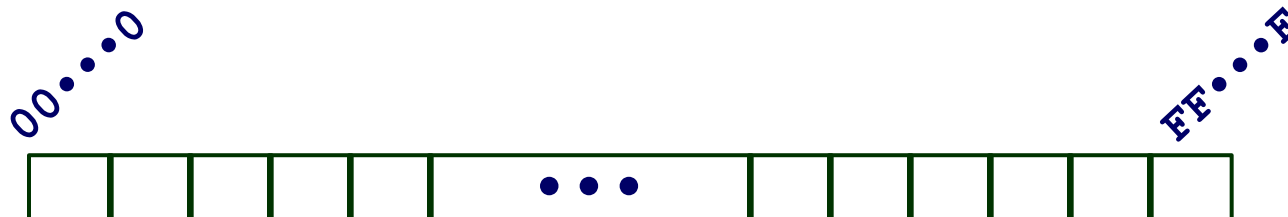
  ```
  movl (%ecx),%eax
  ```

- **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?

Solution: Add a level of indirection

Virtual memory

Process 1

Physical memory

mapping

Virtual memory
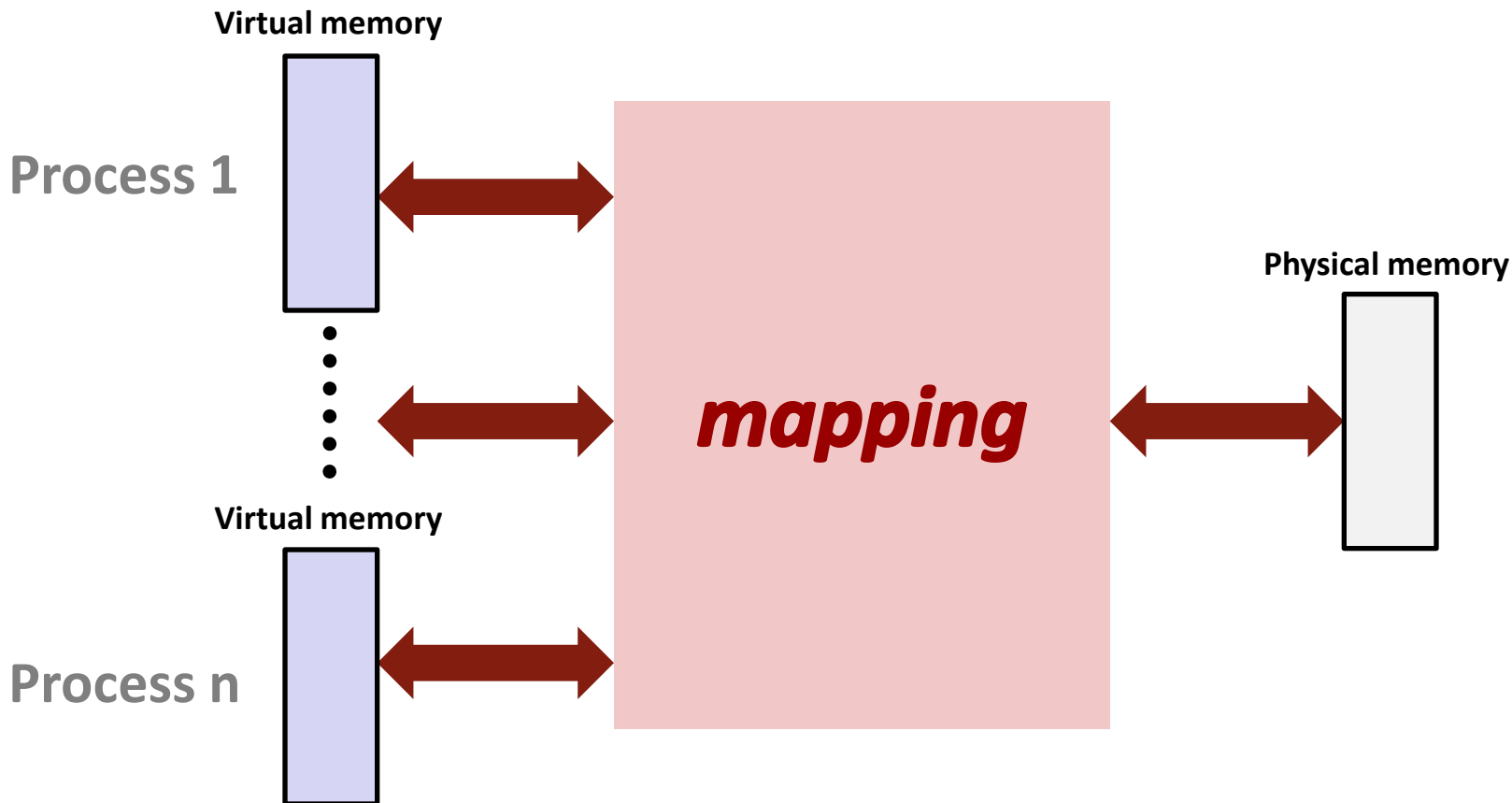
Process n

- ■ **Each process gets its own private memory space**
- ■ **Addresses all of the previous problems**

One simple trick solves all three problems

- **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:

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

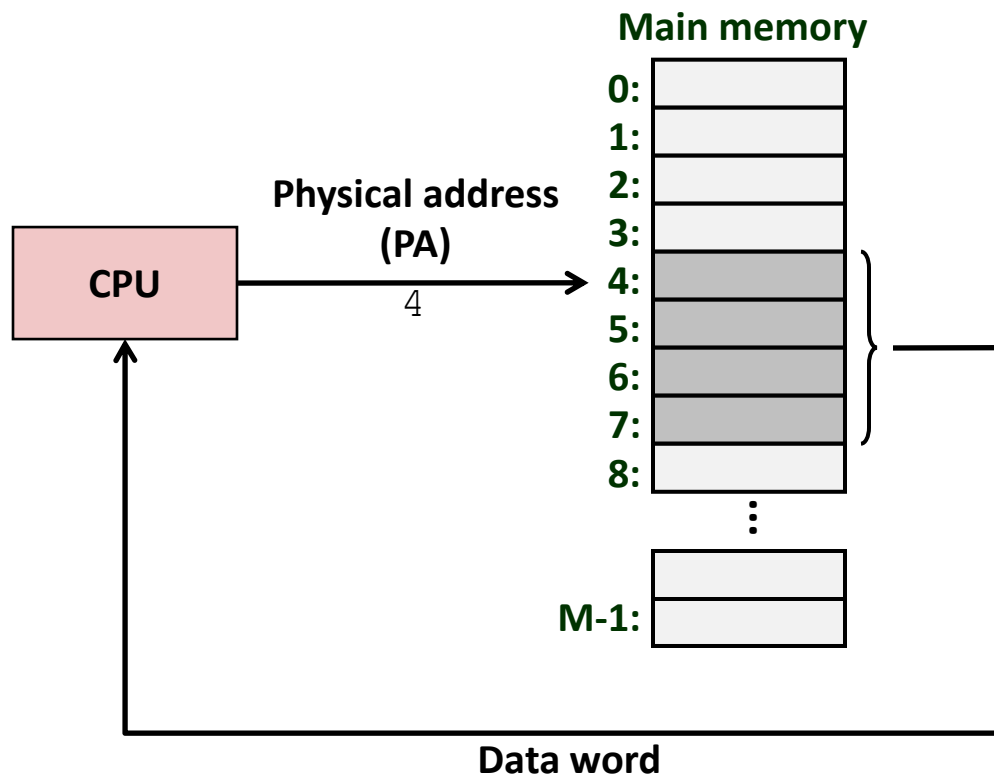- **Virtual address space:** Set of $N = 2^n$ virtual addresses

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

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

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

- **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**
- embedded microcontrollers

A System Using Virtual Addressing

Main memory

CPU Chip

| CPU | Virtual address (VA) 4100 | MMU | Physical address (PA) 4 |

0:
1:
2:
3:
4:
5:
6:
7:
8:
⋮
M-1:

Data word
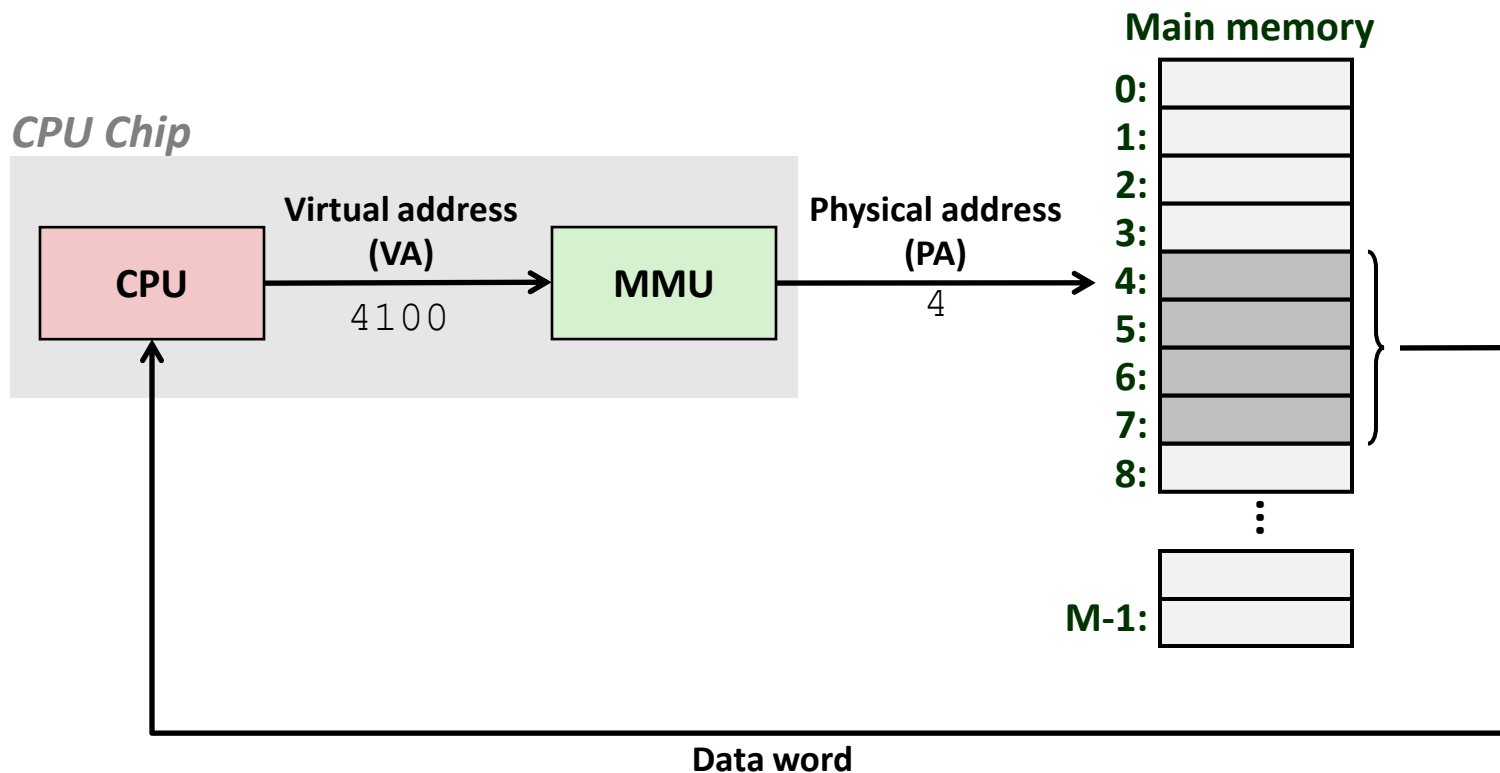
- **Used in all modern servers, desktops, and laptops**
  - Increasingly, in the "embedded" space, too!
- **One of the great ideas in computer science**

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

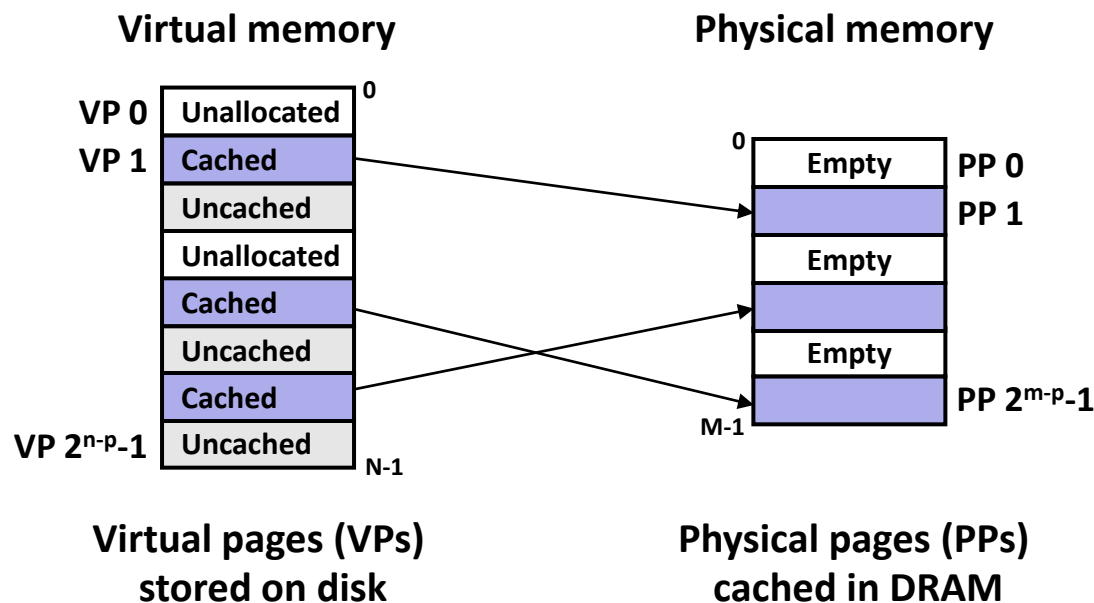- One process can't interfere with another's memory
  - because they operate in different address spaces
- User process cannot access privileged information
  - different sections of address spaces have different permissions

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 that may be stored on disk (or network, or tape, or paper…)
- The contents of the array are cached in *physical memory* (*DRAM cache*)
    - These cache blocks are called *pages* (size is P = 2^p bytes)

Virtual memory

VP 0	Unallocated
VP 1	Cached
	Uncached
	Unallocated
	Cached
	Uncached
	Cached
VP 2^{n-p}-1	Uncached

0

N-1

Physical memory

0

Empty	PP 0
	PP 1
Empty	
Empty	
	PP 2^{m-p}-1

M-1

Virtual pages (VPs)
stored on disk

Physical pages (PPs)
cached in DRAM

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
  - Network even slower, tape even slower still, paper way slower…

- **Consequences**
  - Large page (block) size: typically 4-8 KB, sometimes 4 MB
  - Fully associative
    - Any virtual page (VP) can be placed in any physical page (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
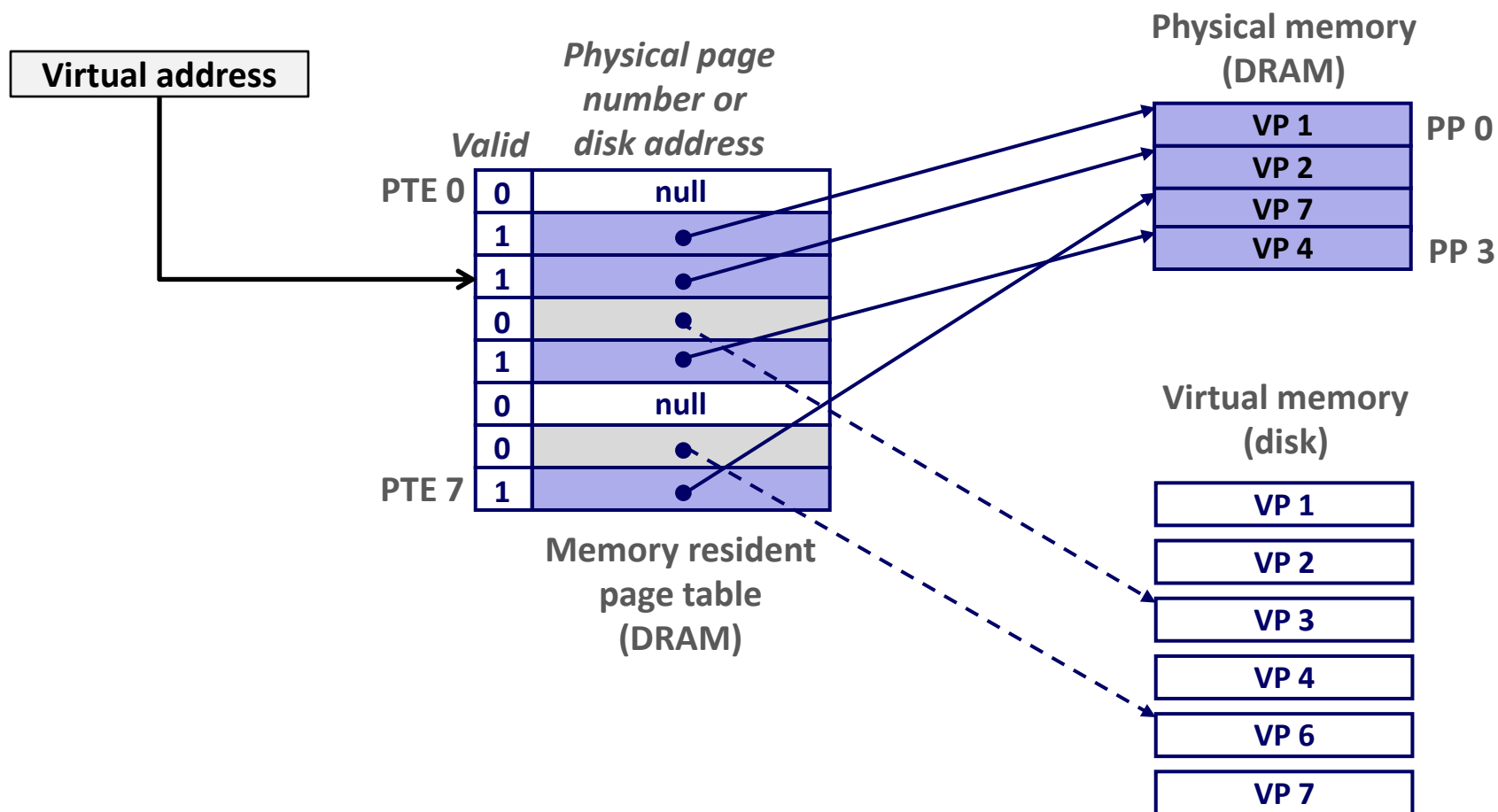 - Write-back rather than write-through

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
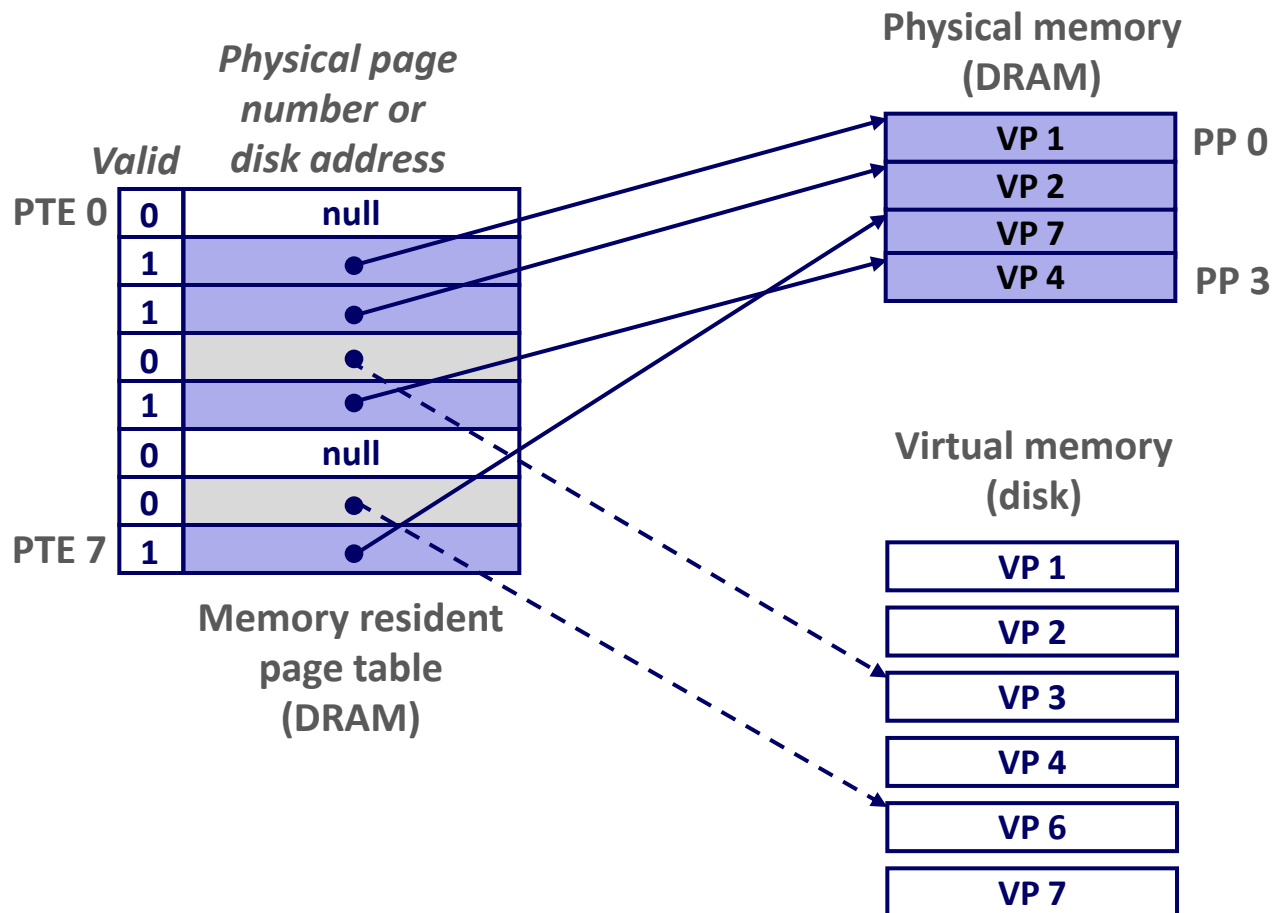
Page Hit

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

Physical page number or disk address

Virtual address

Physical memory (DRAM)

	Valid	
PTE 0	0	null
	1	
	1	
	0	
	1	
	0	null
	0	
PTE 7	1	

VP 1	PP 0
VP 2	
VP 7	
VP 4	PP 3

Memory resident page table (DRAM)

Virtual memory (disk)

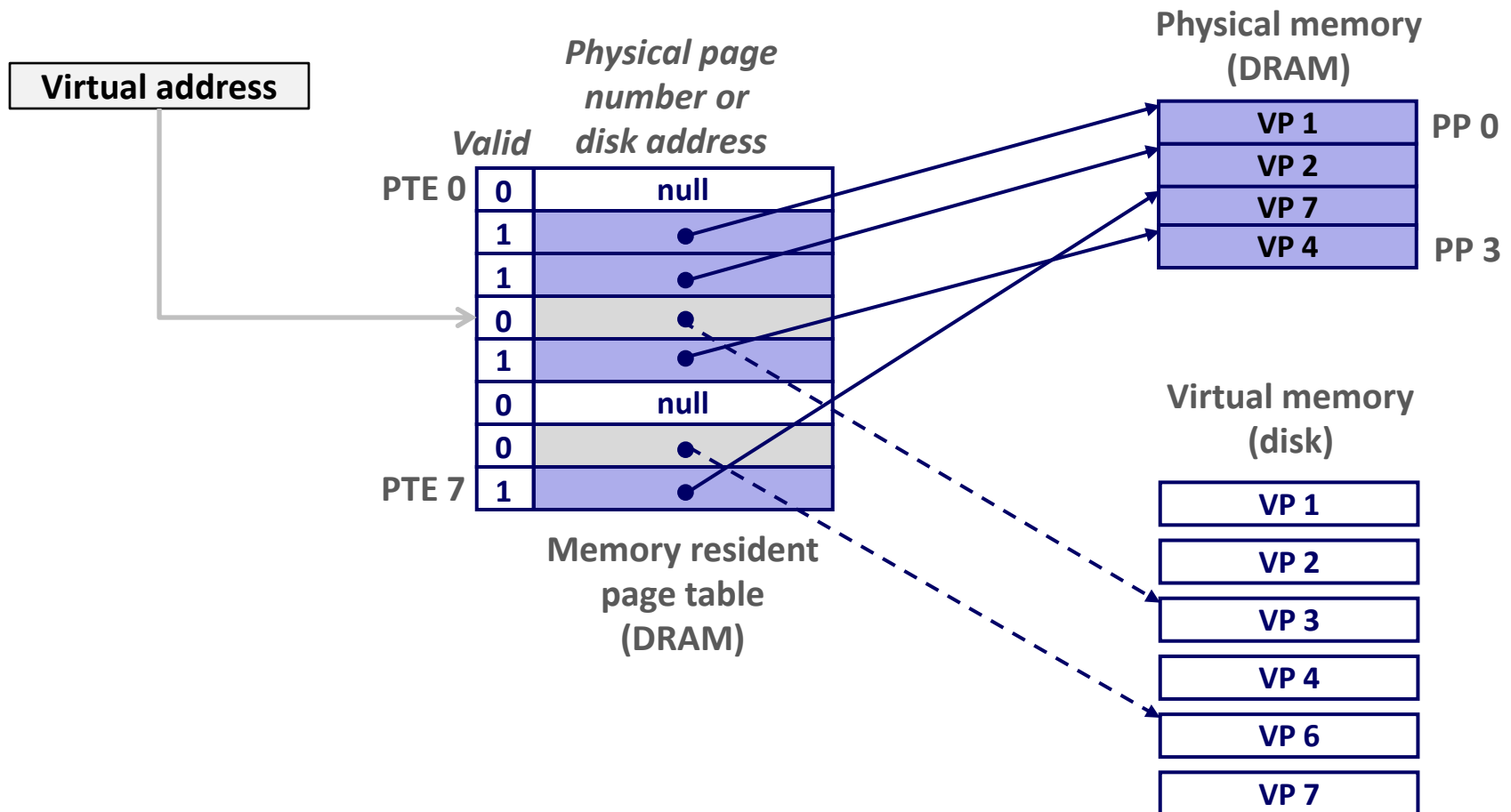| VP 1 |
|---|
| VP 2 |
| VP 3 |
| VP 4 |
| VP 6 |
| VP 7 |

Page Fault

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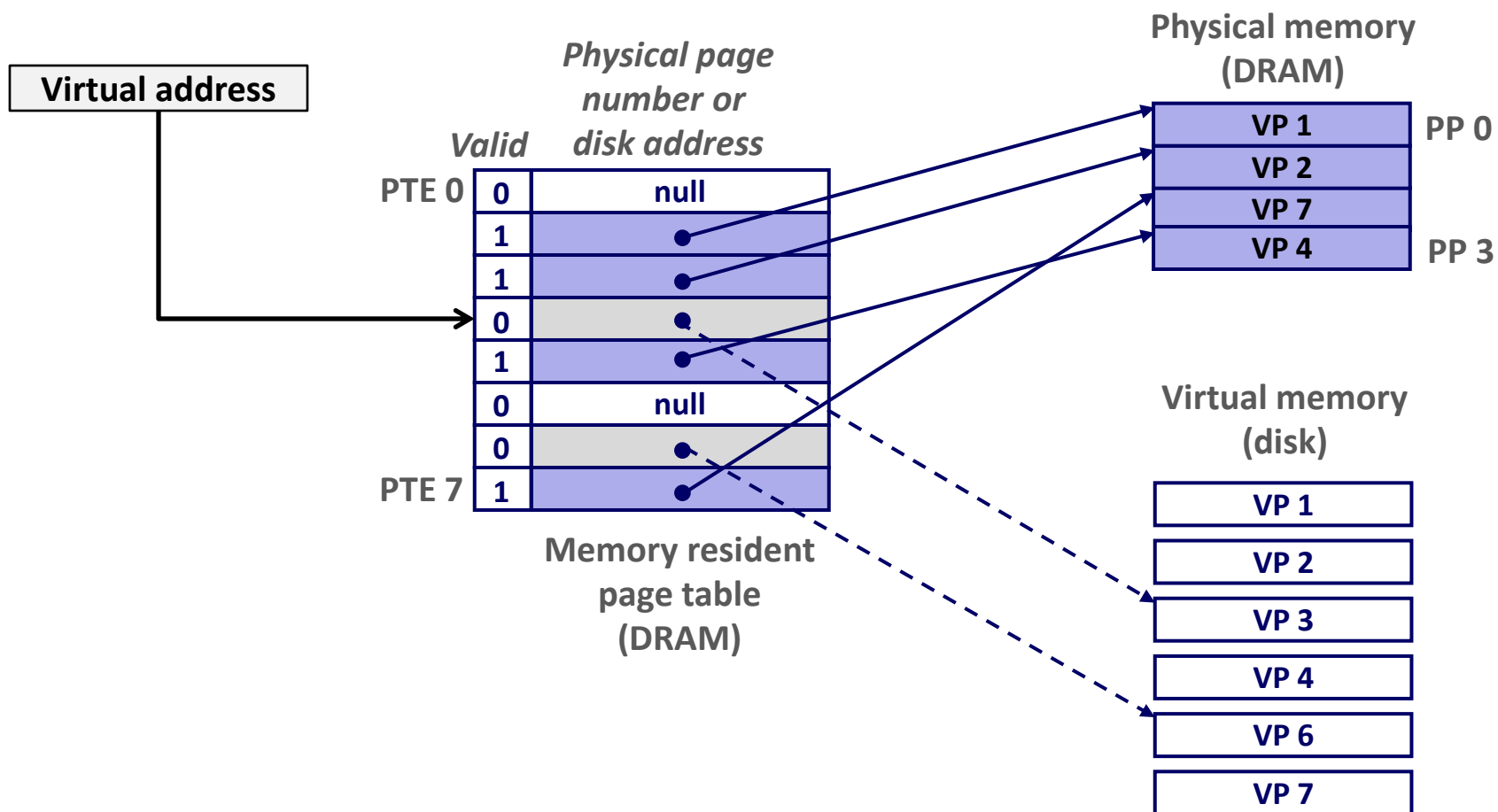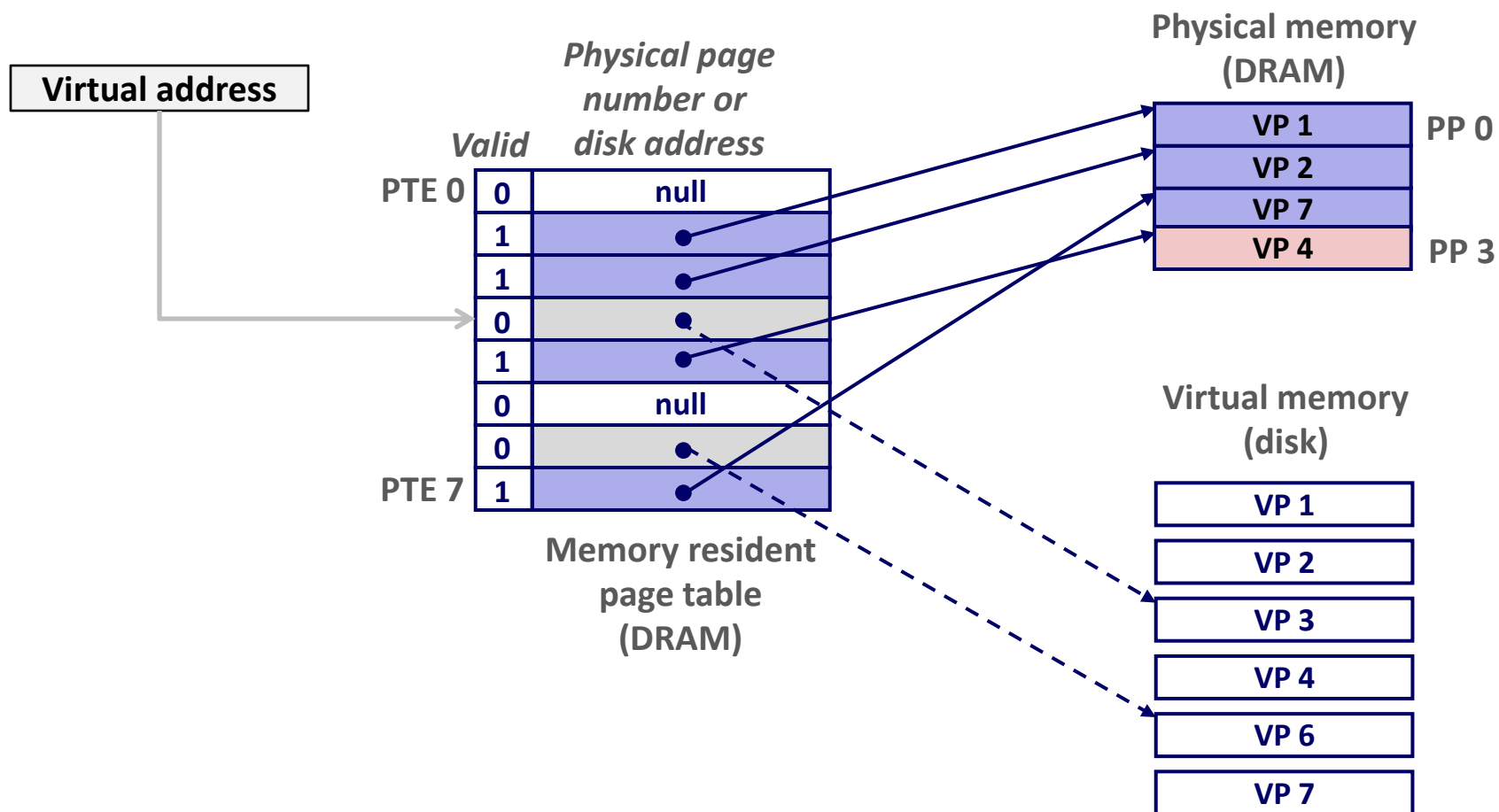
Handling Page Fault

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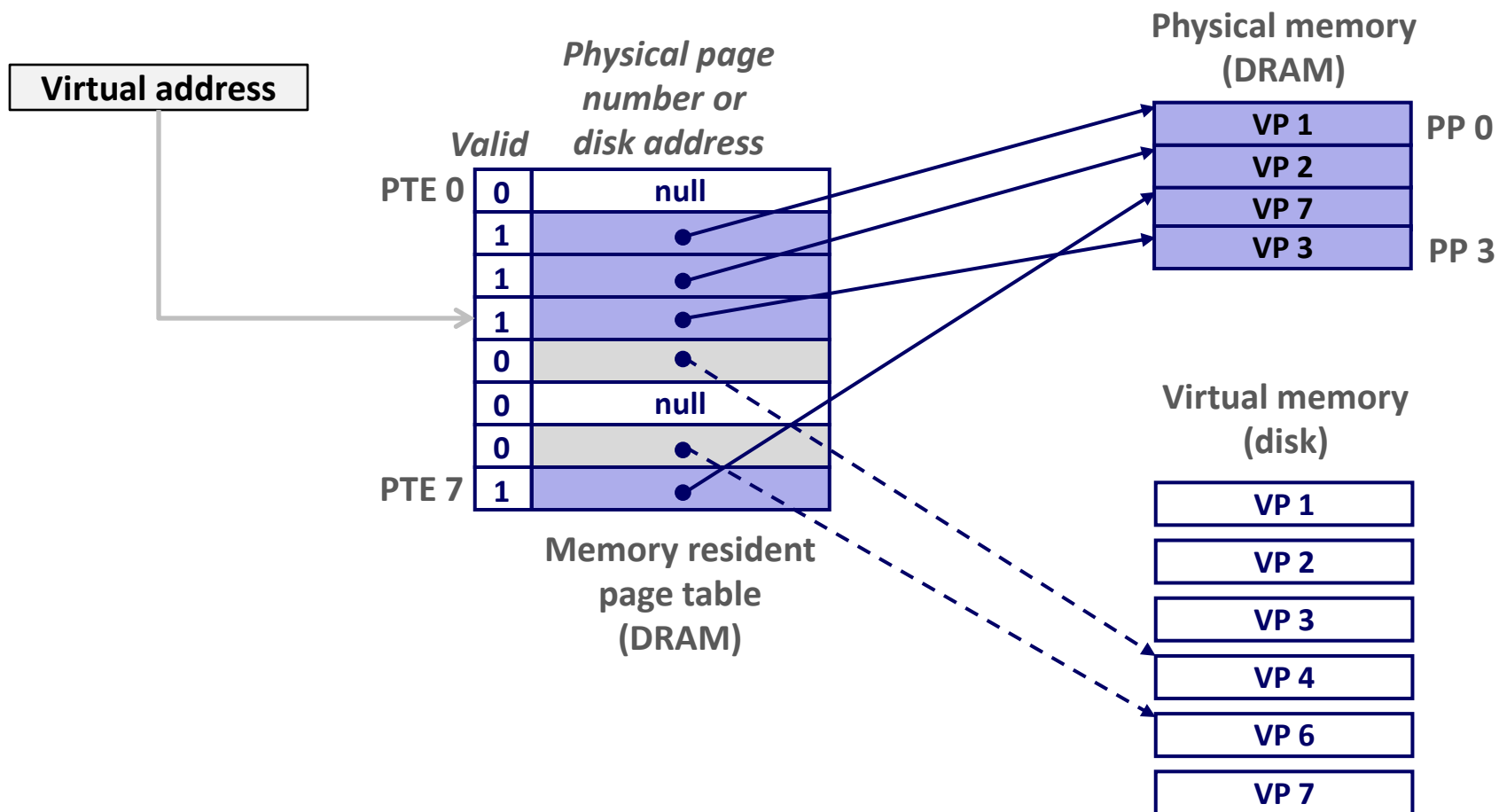- **Page fault handler selects a victim to be evicted (here VP 4)**

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!**

Physical memory (DRAM)

VP 1	PP 0
VP 2	
VP 7	
VP 3	PP 3

Virtual address

Physical page number or disk address

Valid

	Valid	
PTE 0	0	null
	1	●
	1	●
	1	●
	0	●
	0	null
	0	●
PTE 7	1	●

Memory resident page table (DRAM)

Virtual memory (disk)

VP 1
VP 2
VP 3
VP 4
VP 6
VP 7
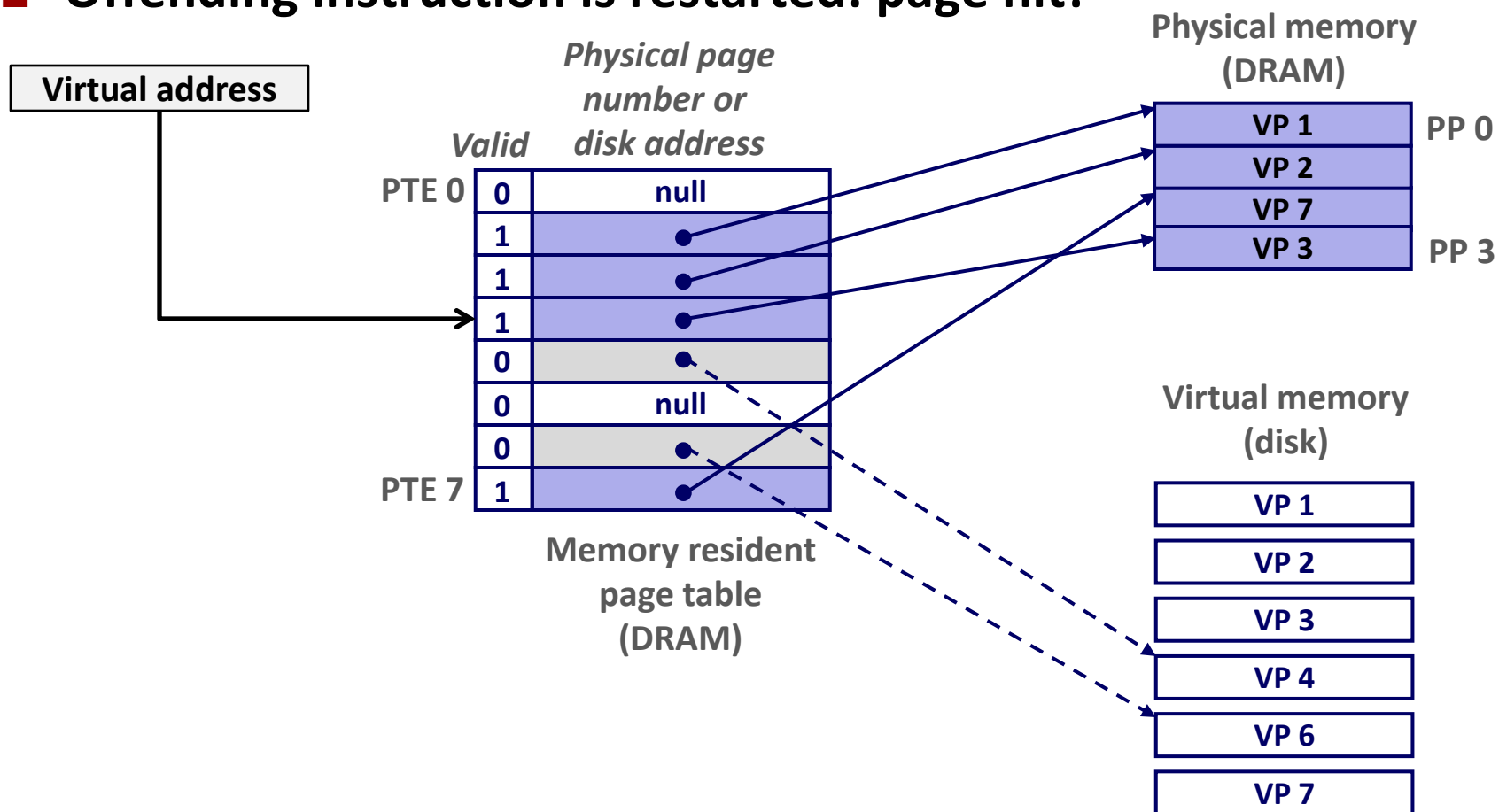
Locality to the Rescue Again!

- **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
    - Kernel might try to keep this "less than completely arbitrary"

Virtual Address Space for Process 1:

0

VP 1
VP 2
...

N-1

Address translation

Virtual Address Space for Process 2:

0

VP 1
VP 2
...

N-1

0

PP 2

PP 6

PP 8

...

M-1

Physical Address Space (DRAM)
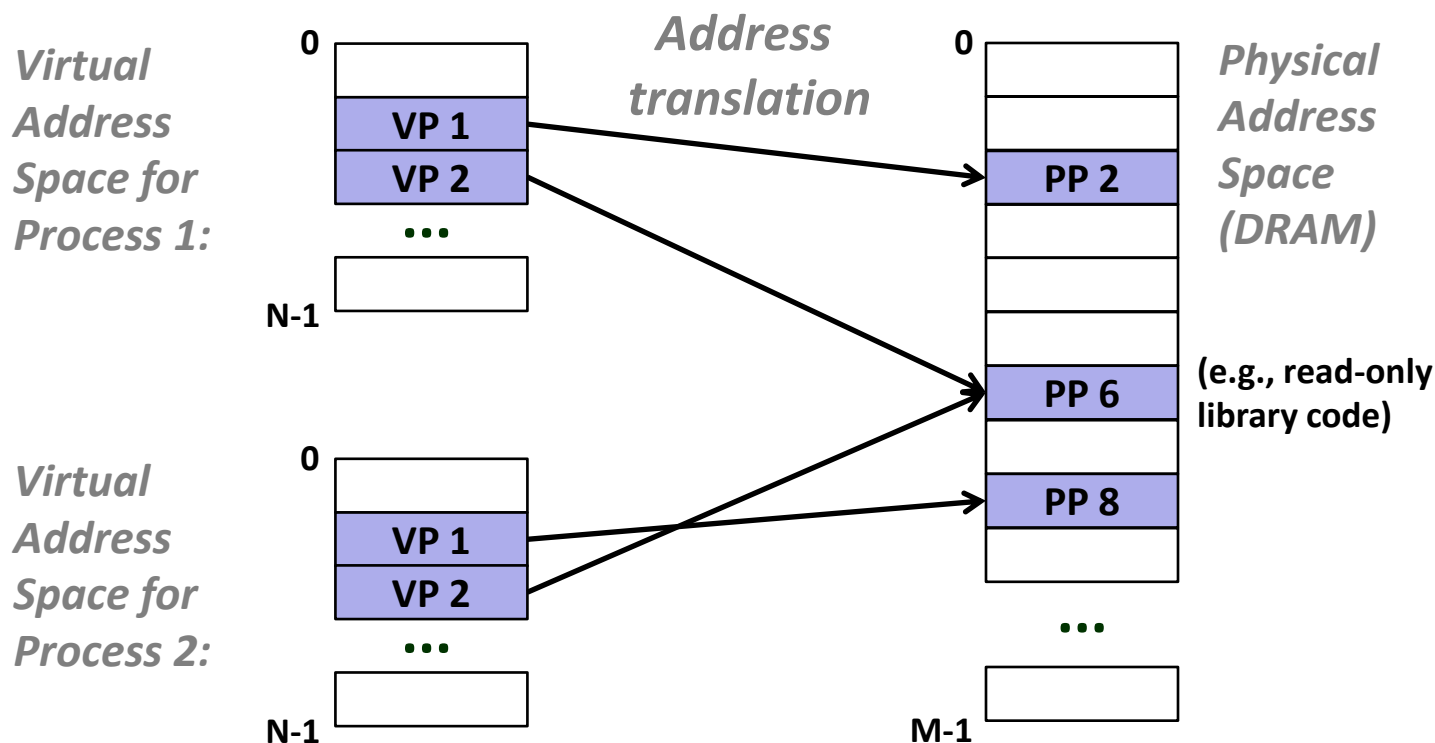
(e.g., read-only library code)
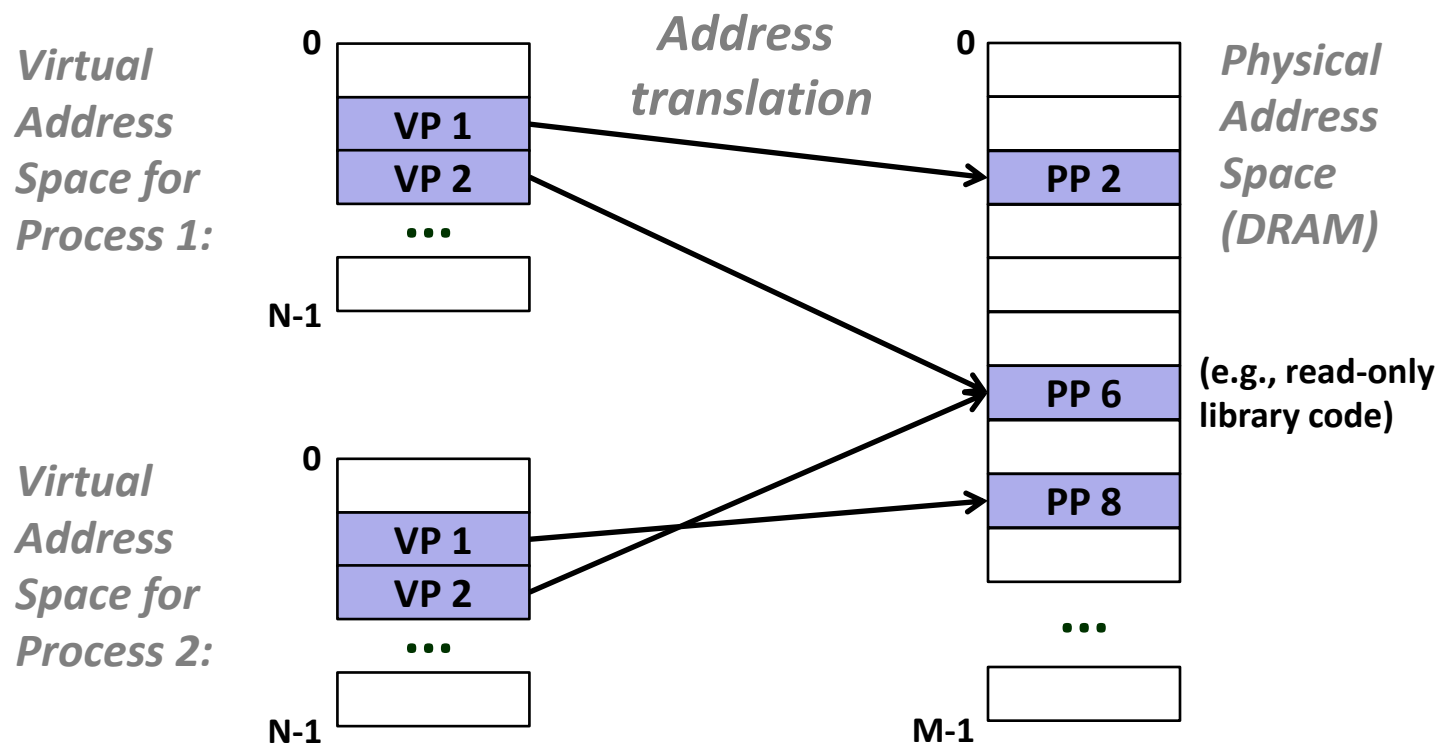
Simplifying allocation and sharing

- **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)

Virtual Address Space for Process 1: 0 VP 1 VP 2 ... N-1

Address translation

Virtual Address Space for Process 2: 0 VP 1 VP 2 ... N-1

0 PP 2 PP 6 PP 8 ... M-1 *Physical Address Space (DRAM)*
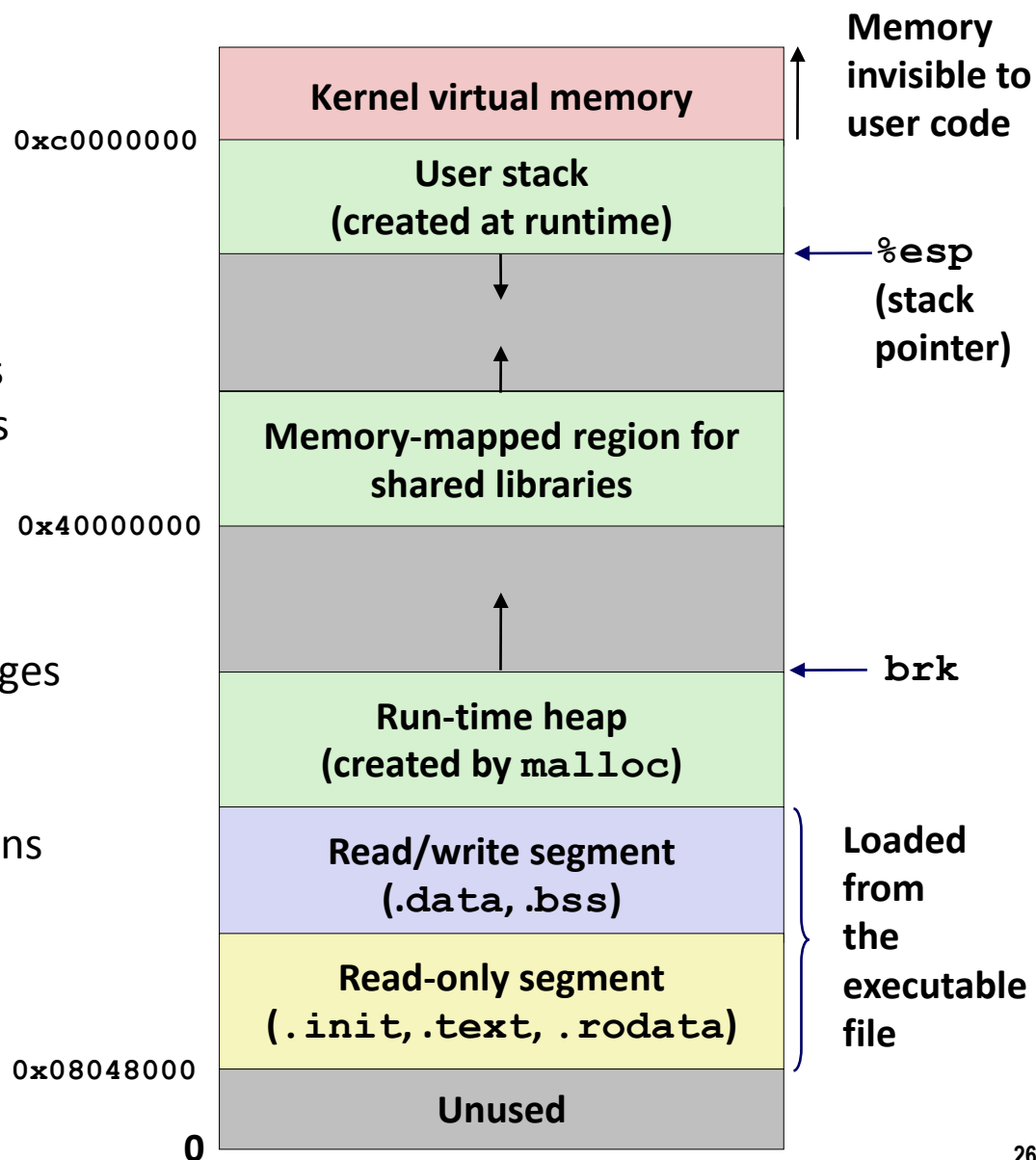
(e.g., read-only library code)

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

Kernel virtual memory	Memory invisible to user code
User stack (created at runtime)	
	%esp (stack pointer)
Memory-mapped region for shared libraries	
Run-time heap (created by malloc)	brk
Read/write segment (.data, .bss)	Loaded from the executable file
Read-only segment (.init, .text, .rodata)	
Unused	

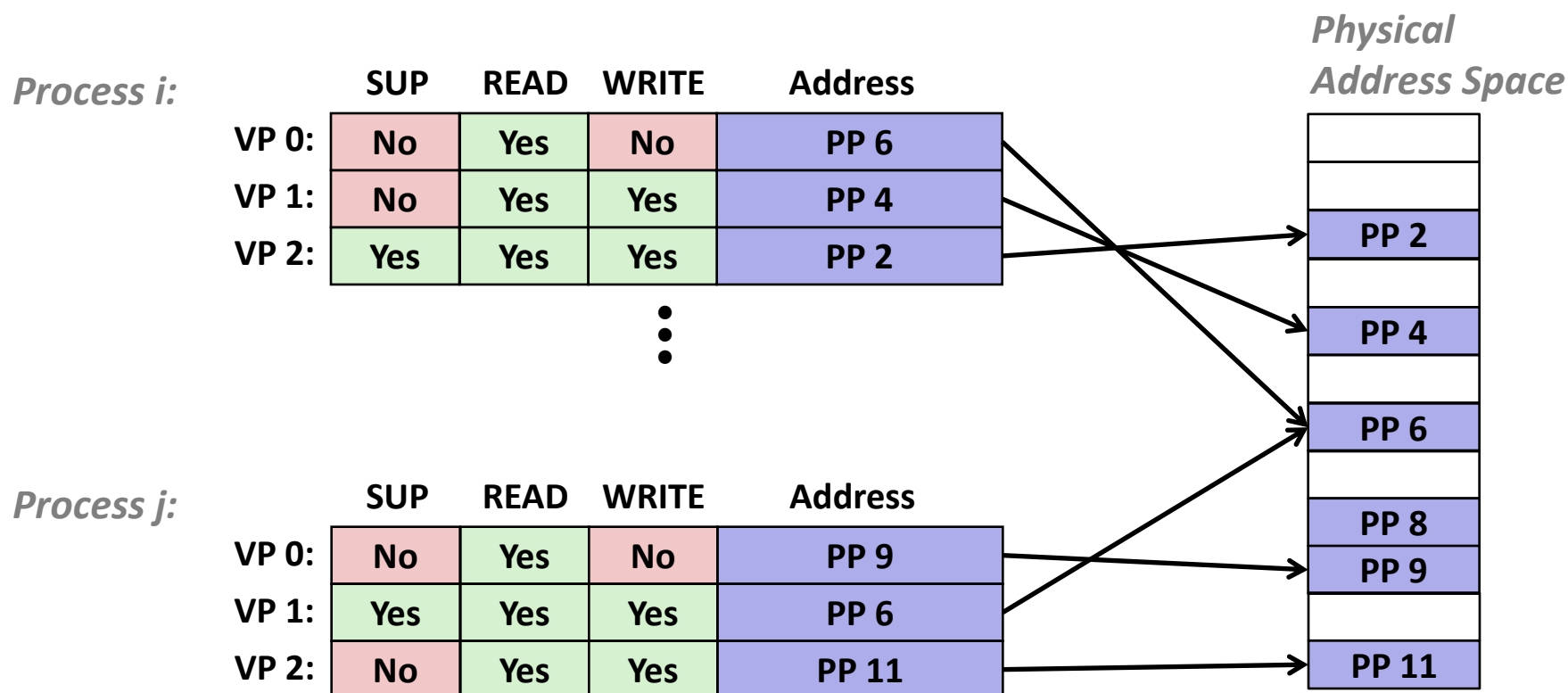0xc0000000

0x40000000

0x08048000

0

26

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)

Physical Address Space

Process i:

	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 6
VP 1:	No	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	PP 2

⋮

Process j:

	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 9
VP 1:	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	PP 11

PP 2
PP 4
PP 6
PP 8
PP 9
PP 11

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, …, N–1}*

- **Physical Address Space**
  - *P = {0, 1, …, M–1}*

- **Address Translation**
  - ***MAP: V $\rightarrow$ P U {$\varnothing$}***
  - For virtual address ***a***:
    - ***MAP(a) = a'*** if data at virtual address ***a*** is at physical address ***a'*** in ***P***
    - ***MAP(a) =*** $\varnothing$ 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
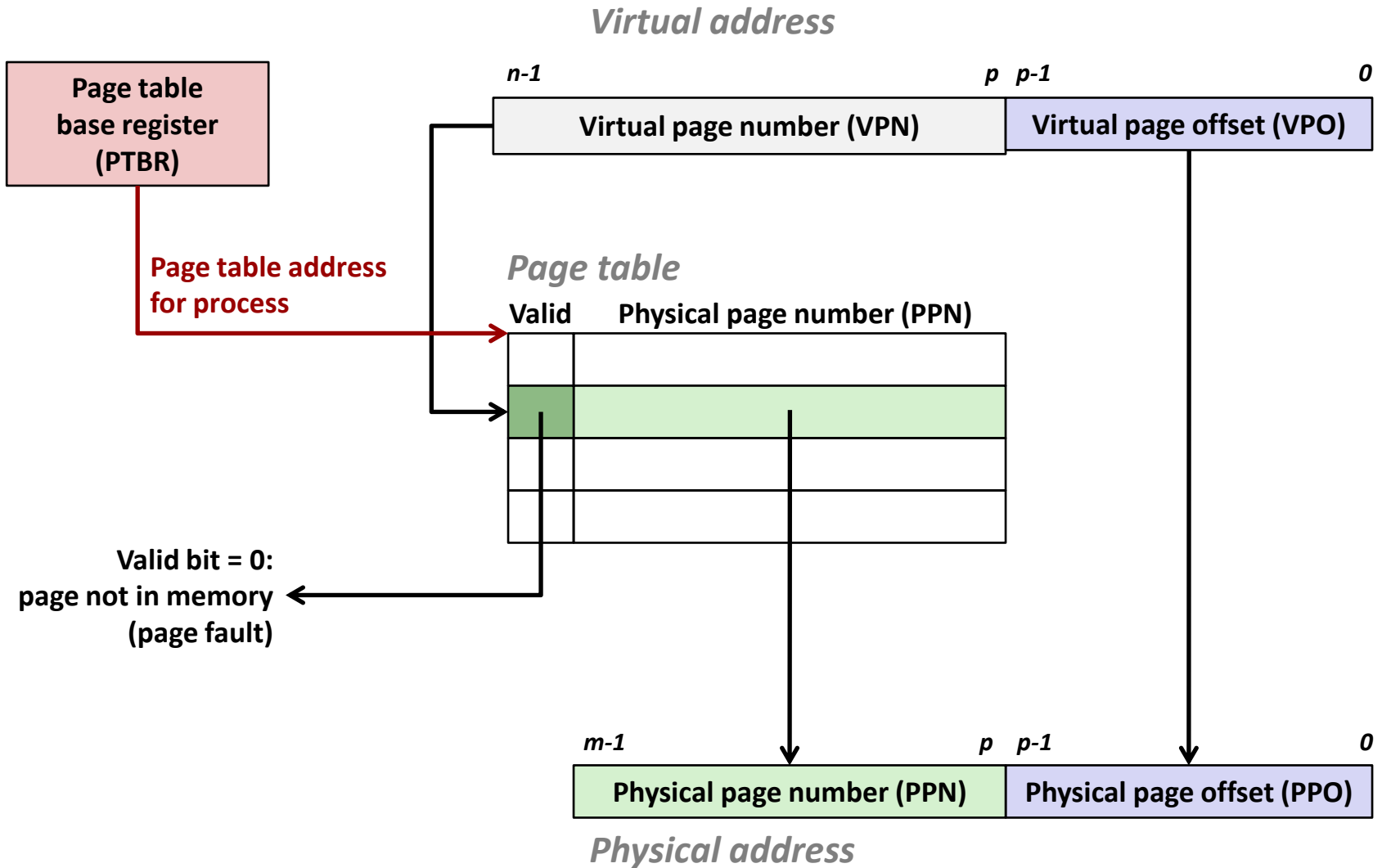 - **P = 2^p** : Page size (bytes)
- **Components of the virtual address (VA)**
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
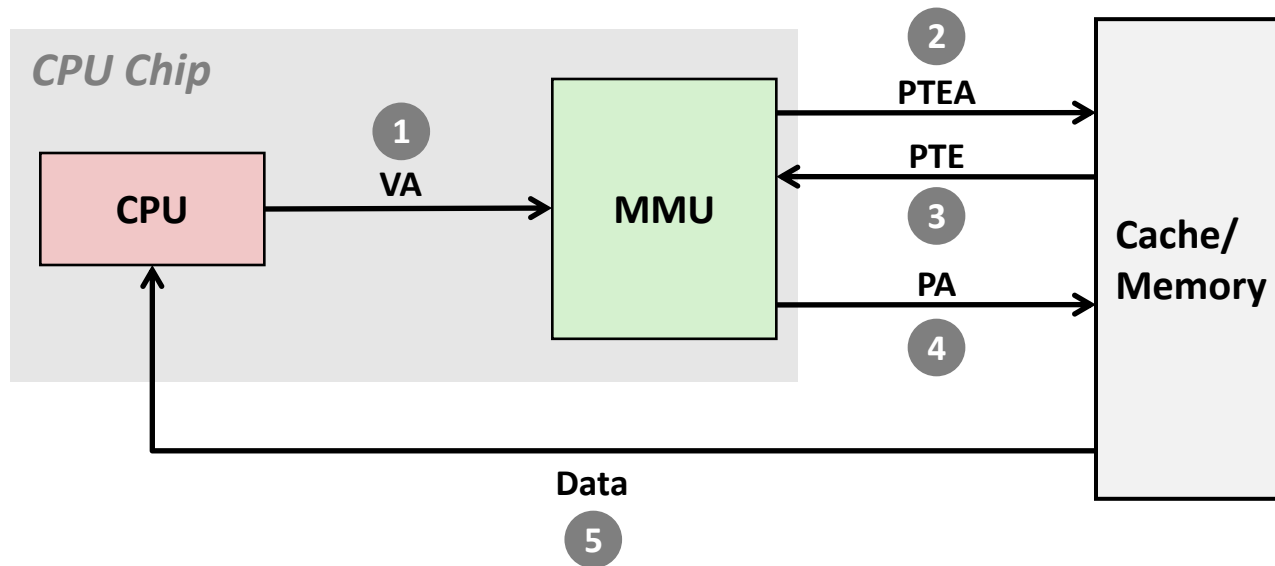 - **TLBI**: TLB index
  - **TLBT**: TLB tag
- **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number
  - **CO**: Byte offset within cache line
  - **CI:** Cache index
 - **CT**: Cache tag

Address Translation With a Page Table

Virtual address

n-1 .... *p* *p-1* *0*

Virtual page number (VPN)	Virtual page offset (VPO)

Page table base register (PTBR)

Page table address for process

Page table

Valid   **Physical page number (PPN)**

Valid bit = 0: page not in memory (page fault)

m-1 .... *p* *p-1* *0*

Physical page number (PPN)	Physical page offset (PPO)

Physical address

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
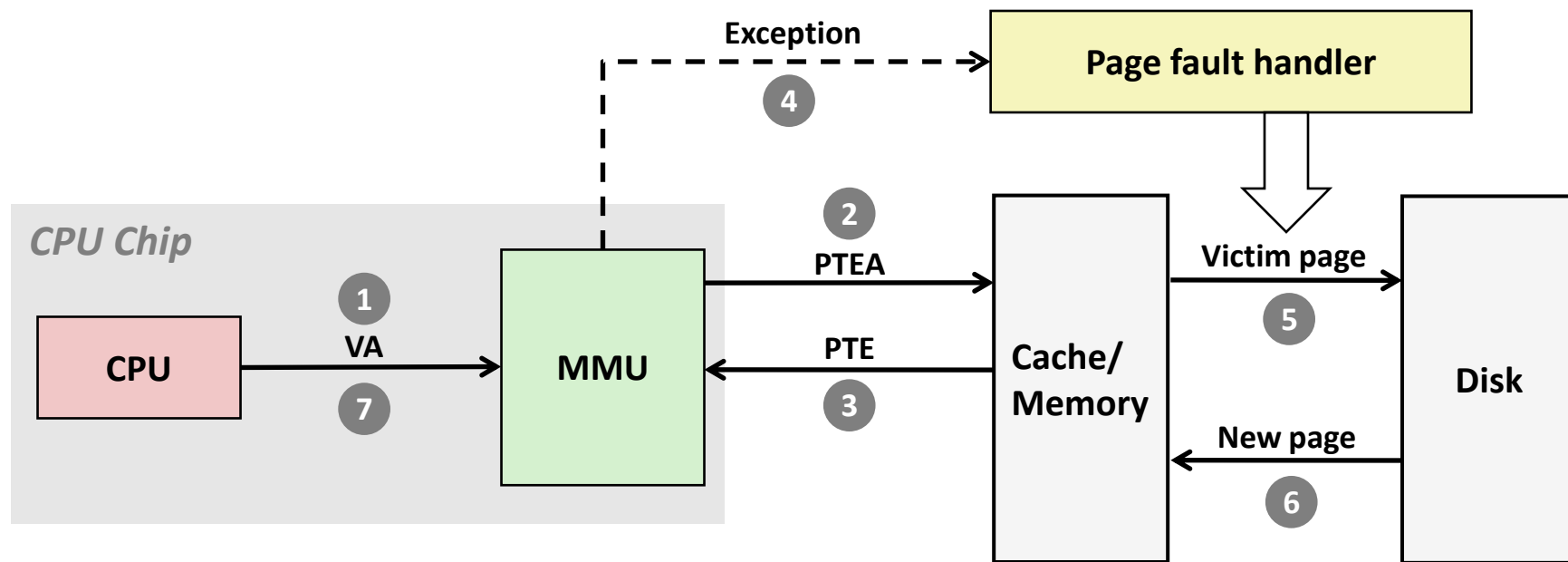
1) Processor sends virtual address to MMU

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

Integrating VM and Cache
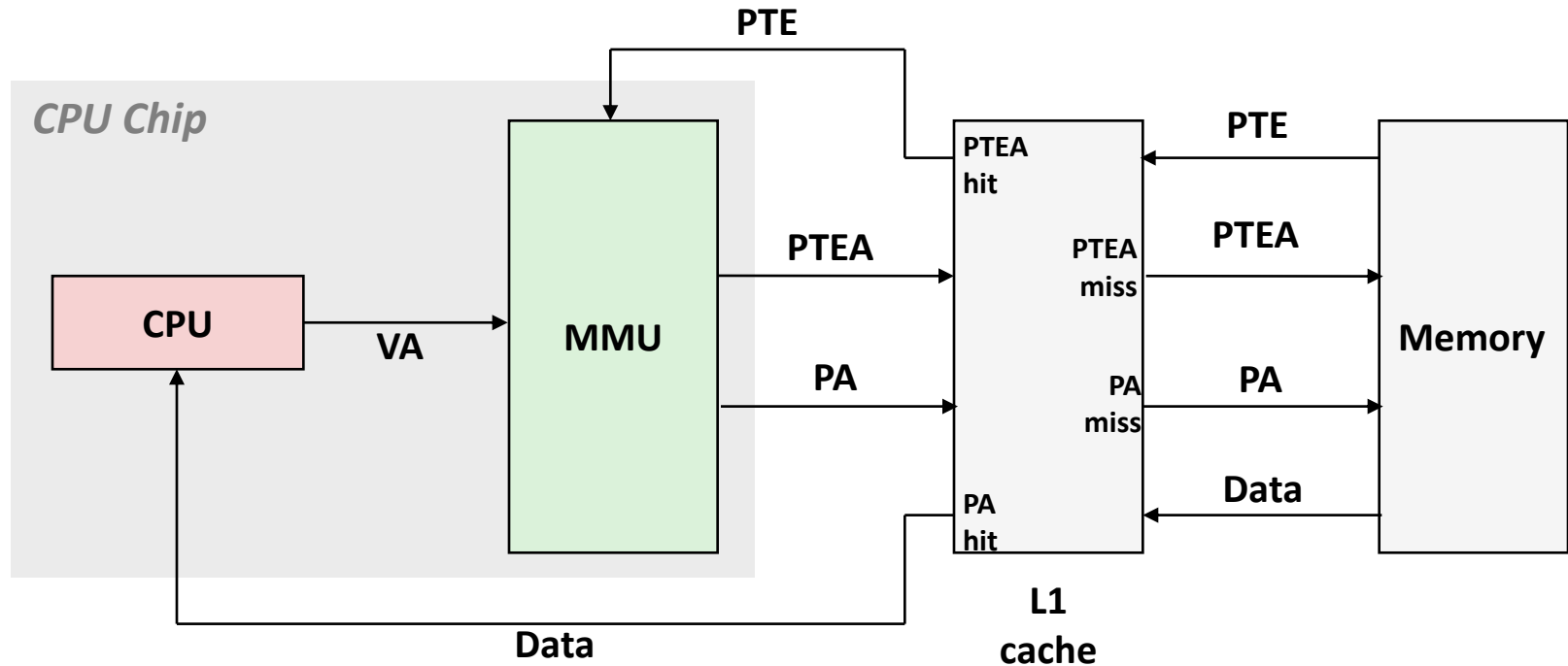
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

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 hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

TLB Hit
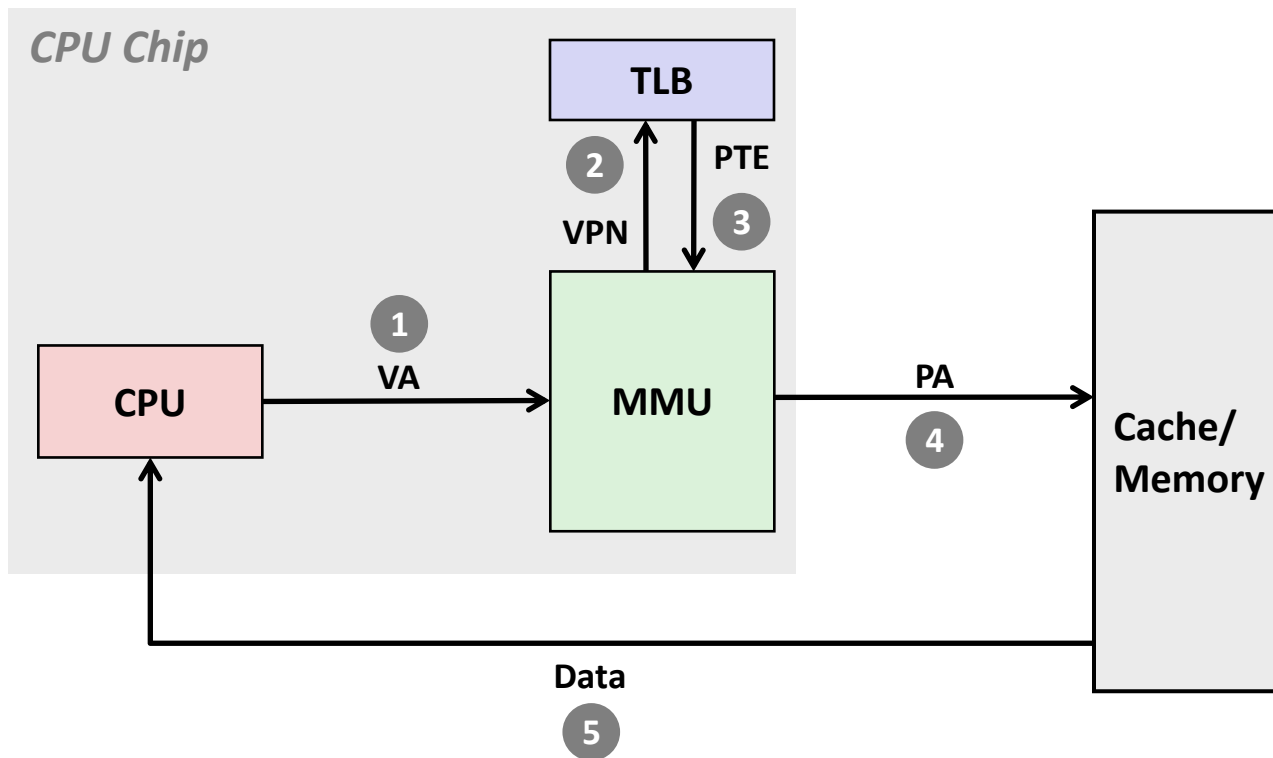
A TLB hit eliminates a memory access

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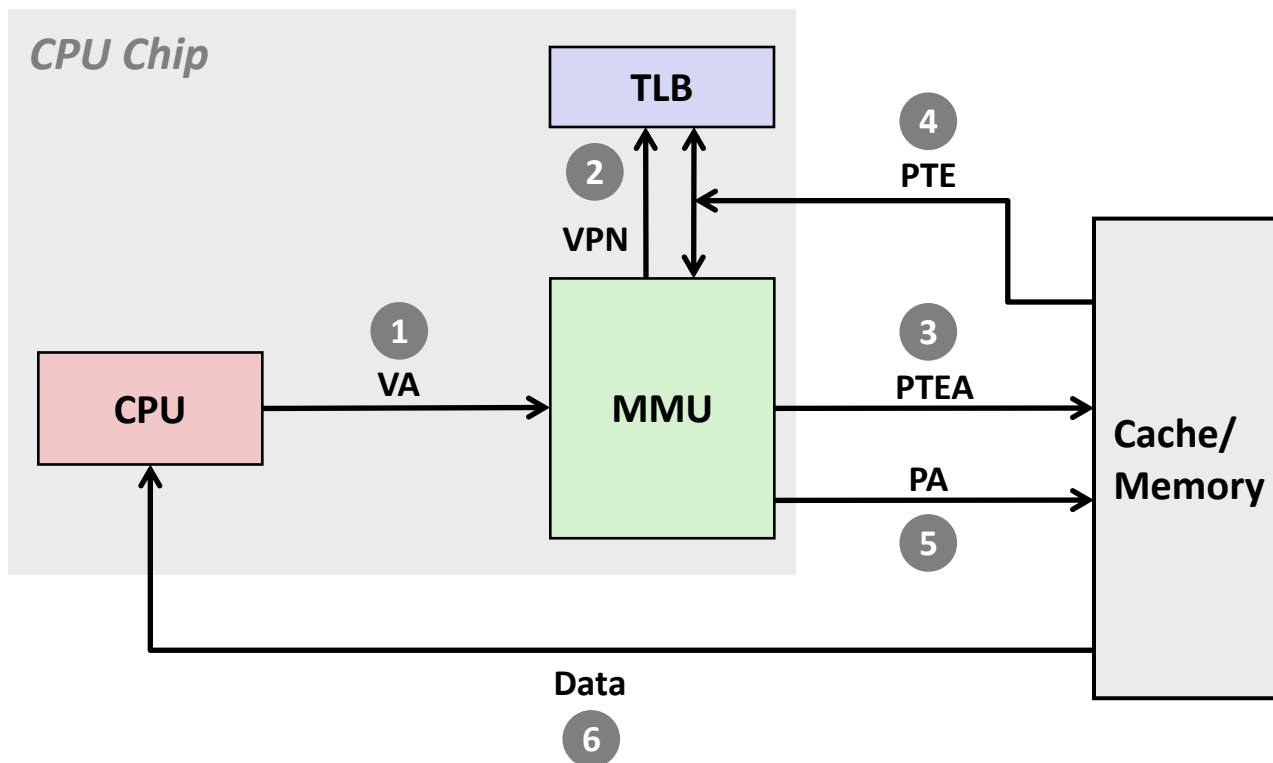
A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?