COMP2310/COMP6310
# Systems, Networks, & Concurrency

Convener: Prof John Taylor

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

Acknowledgement of material: With changes suited to ANU needs, the slides are obtained from Carnegie Mellon University: https://www.cs.cmu.edu/~213/

A System Using Physical Addressing
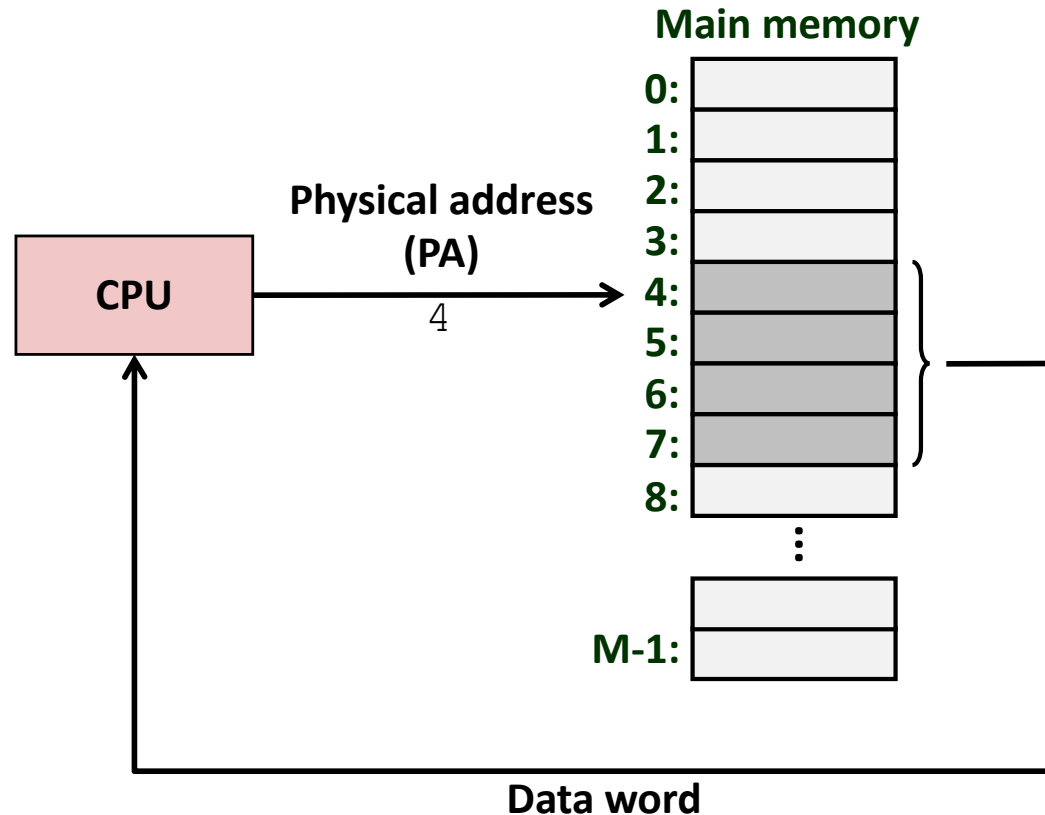
- **Used in "simple" systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames**

A System Using Virtual Addressing

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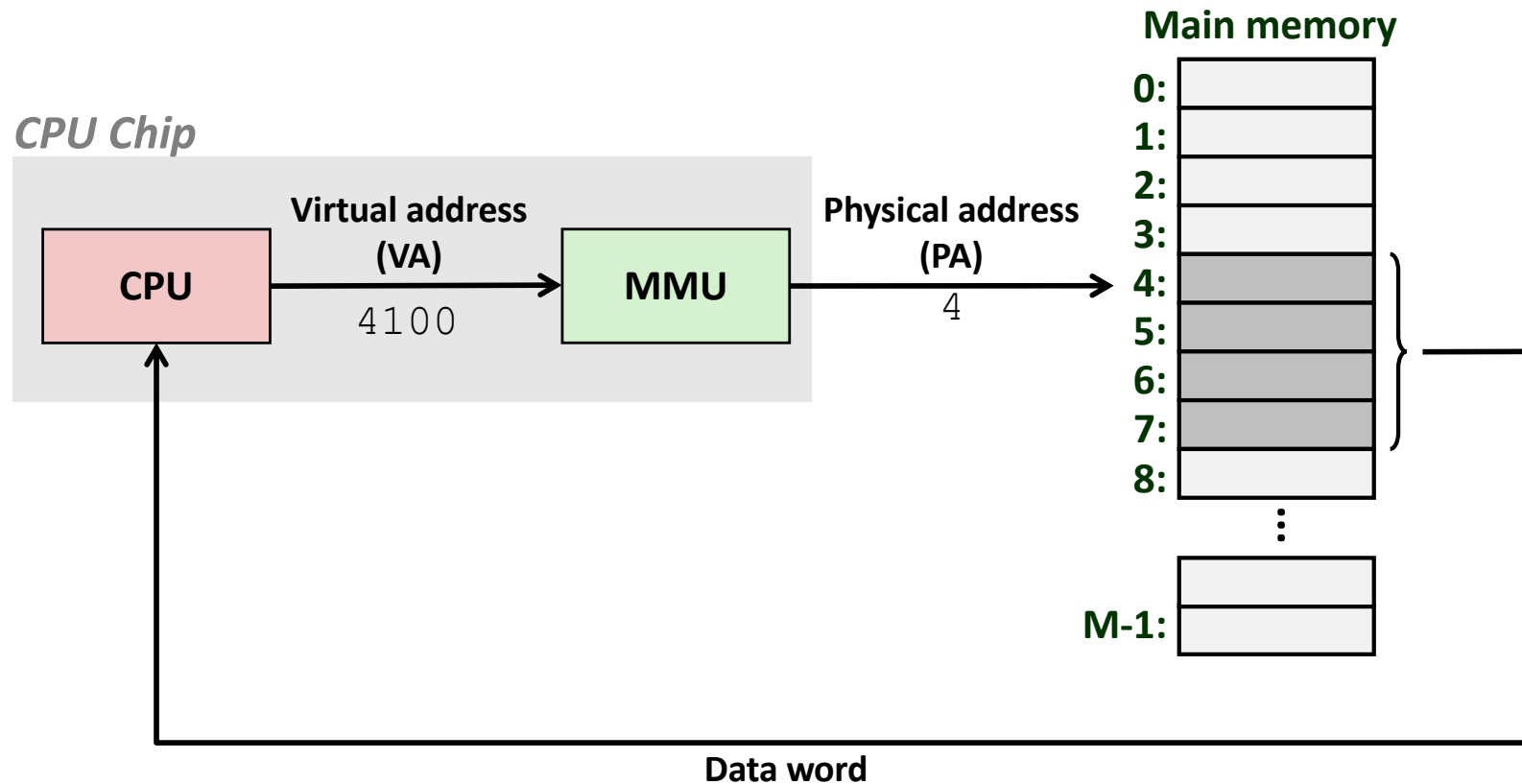
- **Used in all modern servers, laptops, and smart phones**
- **One of the great ideas in computer science**

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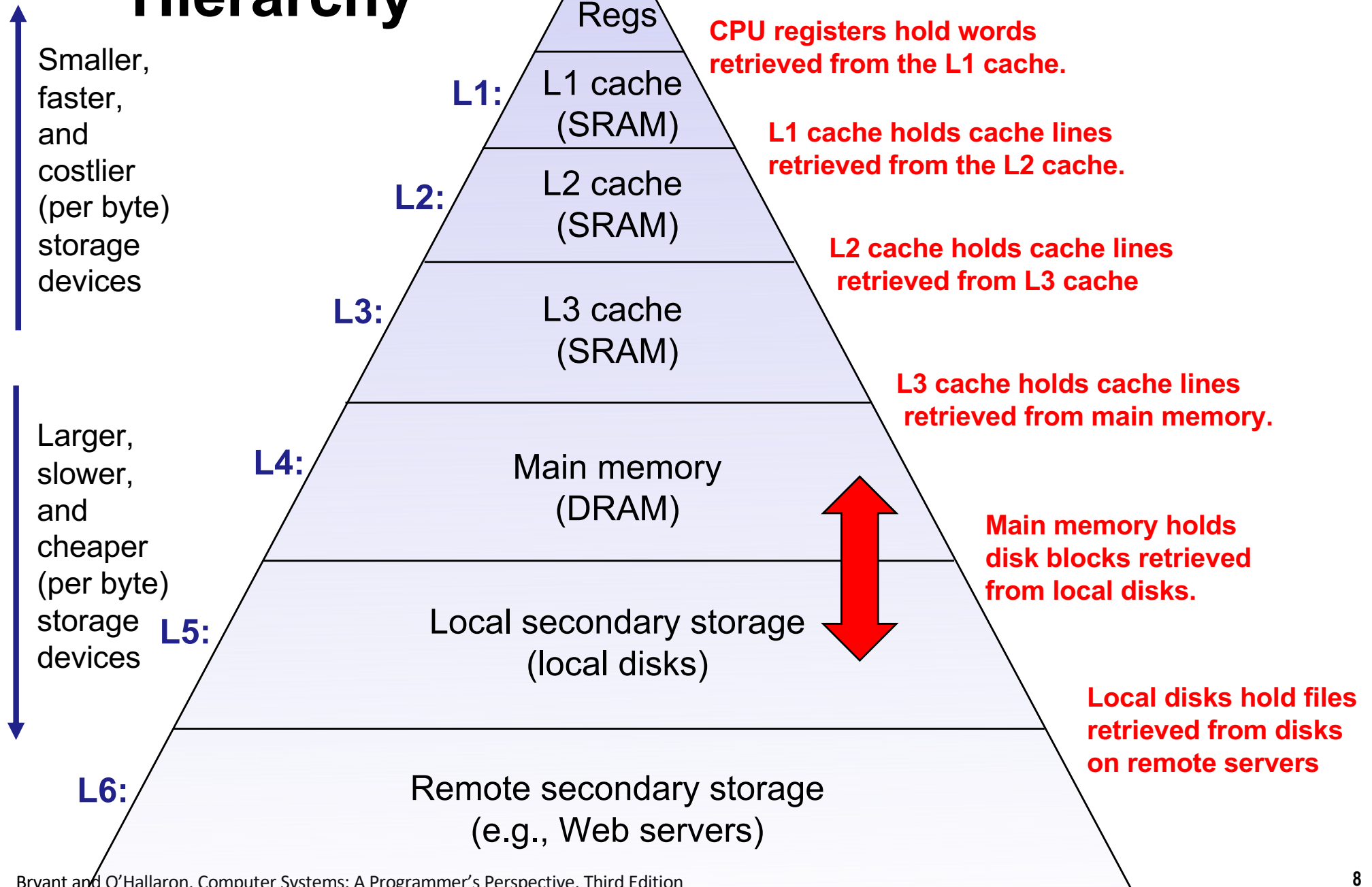
$$\{0, 1, 2, 3, \dots, M-1\}$$

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
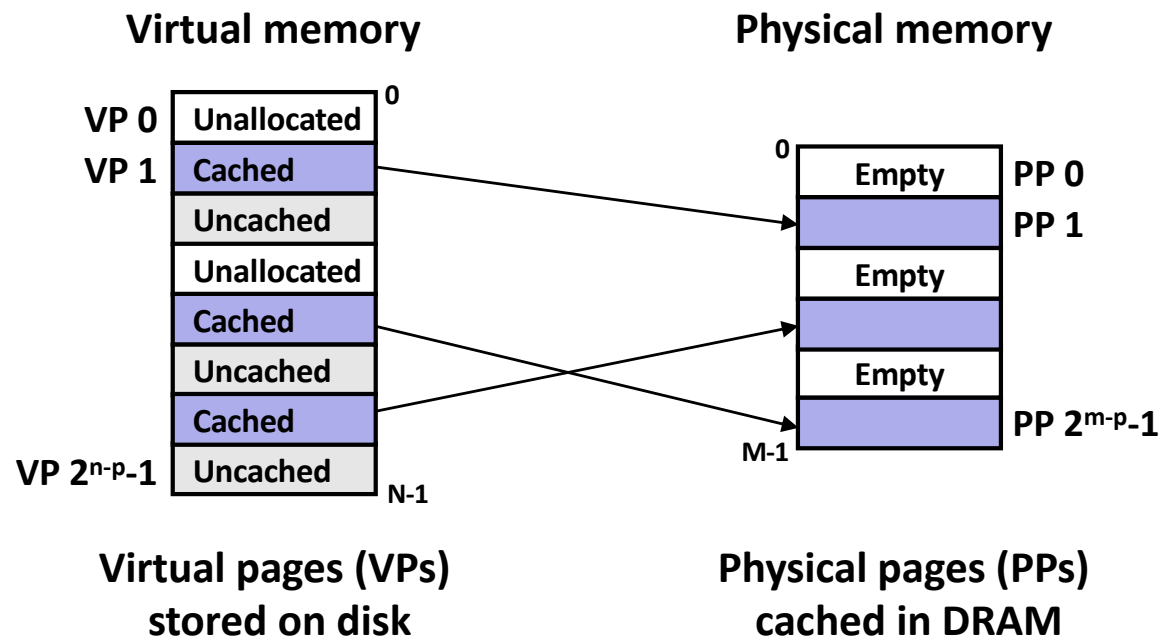 - User program cannot access privileged kernel information and code

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**

Example Memory Hierarchy

Smaller, faster, and costlier (per byte) storage devices

Larger, slower, and cheaper (per byte) storage devices

L0: Regs

L1: L1 cache (SRAM)

L2: L2 cache (SRAM)

L3: L3 cache (SRAM)

L4: Main memory (DRAM)

L5: Local secondary storage (local disks)

L6: Remote secondary storage (e.g., Web servers)

CPU registers hold words retrieved from the L1 cache.

L1 cache holds cache lines retrieved from the L2 cache.

L2 cache holds cache lines retrieved from L3 cache

L3 cache holds cache lines retrieved from main memory.

Main memory holds disk blocks retrieved from local disks.

Local disks hold files retrieved from disks on remote servers

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)

Virtual memory   **Physical memory**

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

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

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

- **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
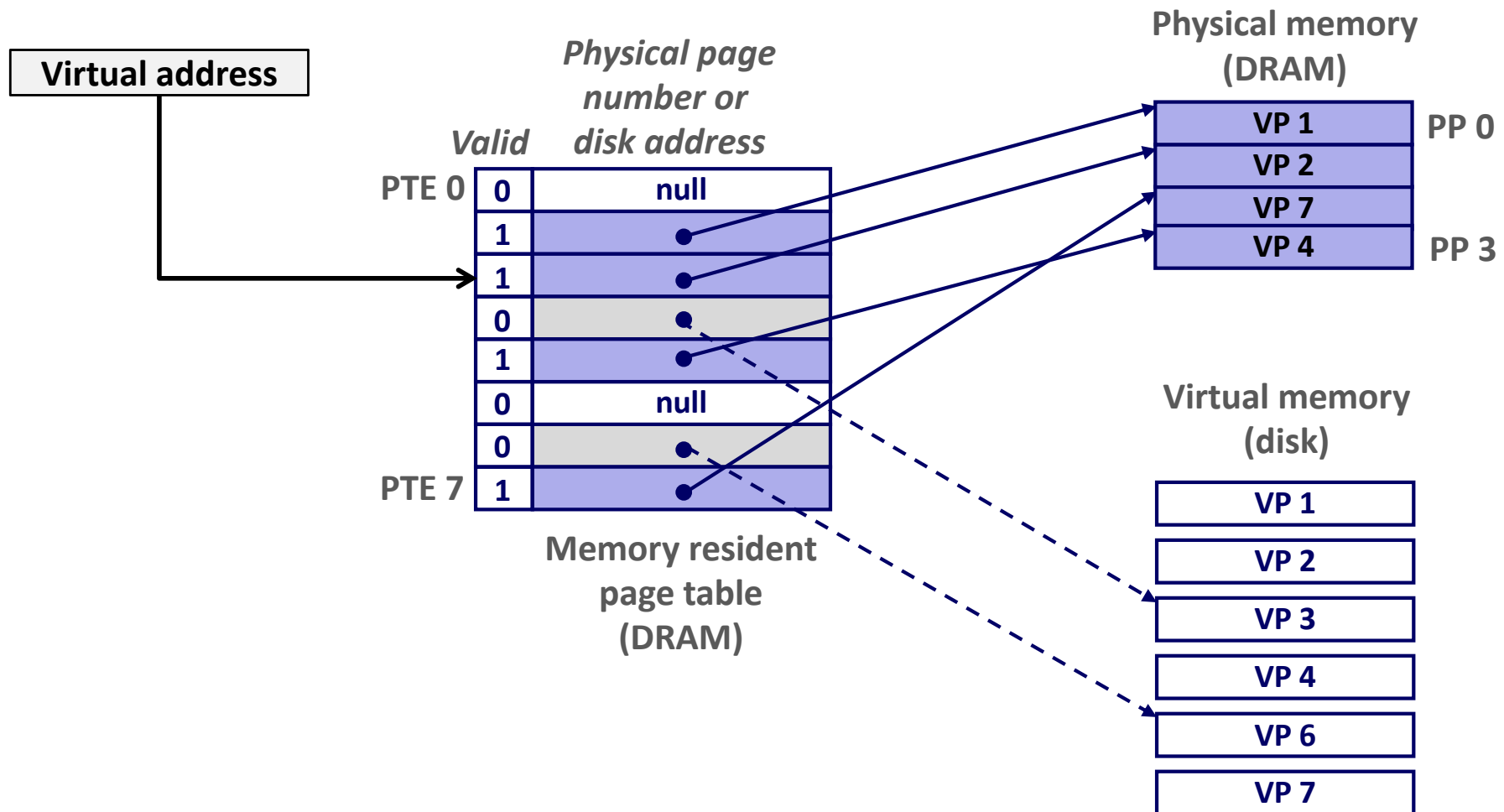   - Write-back rather than write-through

Enabling Data Structure: Page Table

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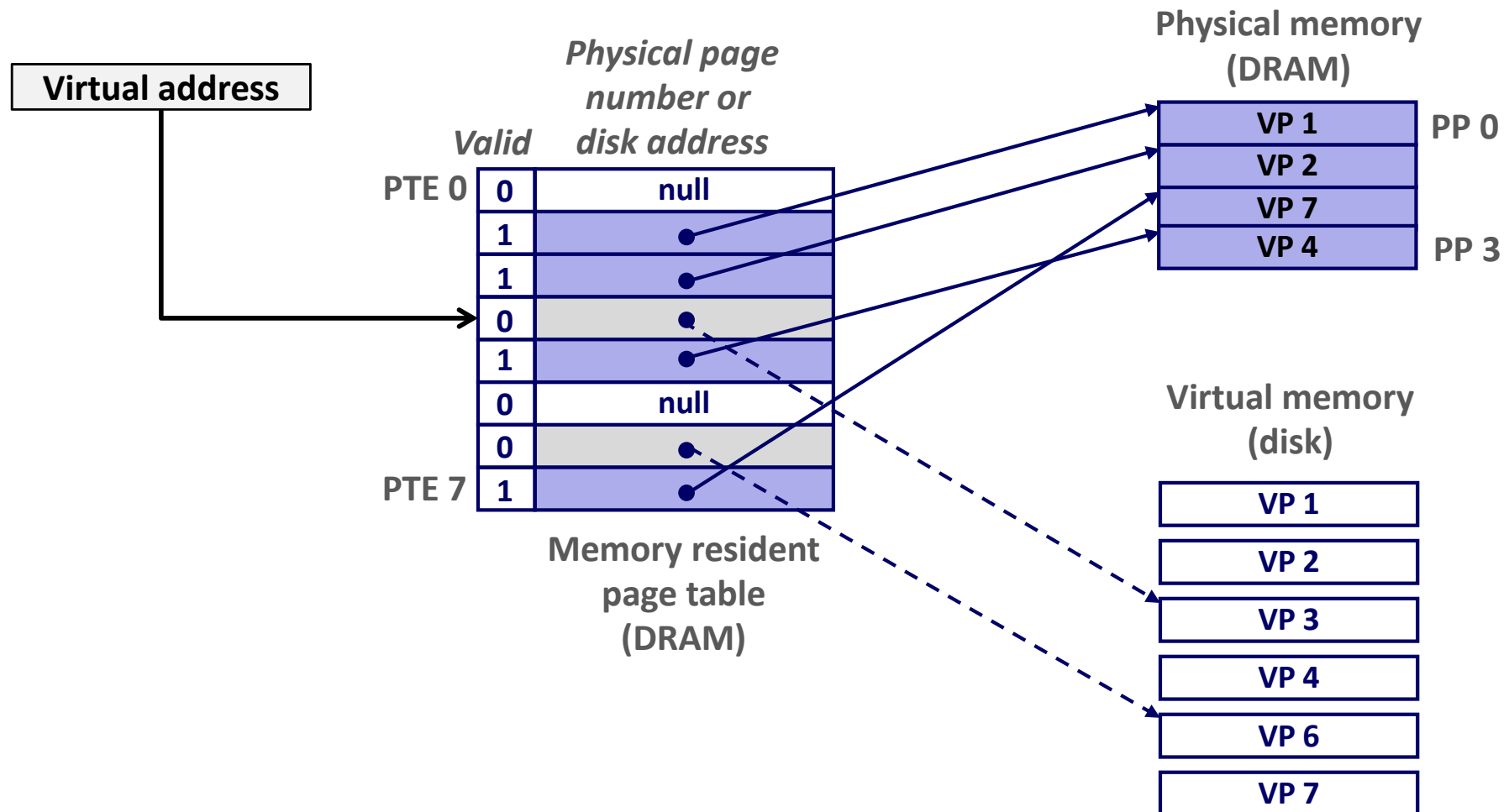 - Per-process kernel data structure in DRAM
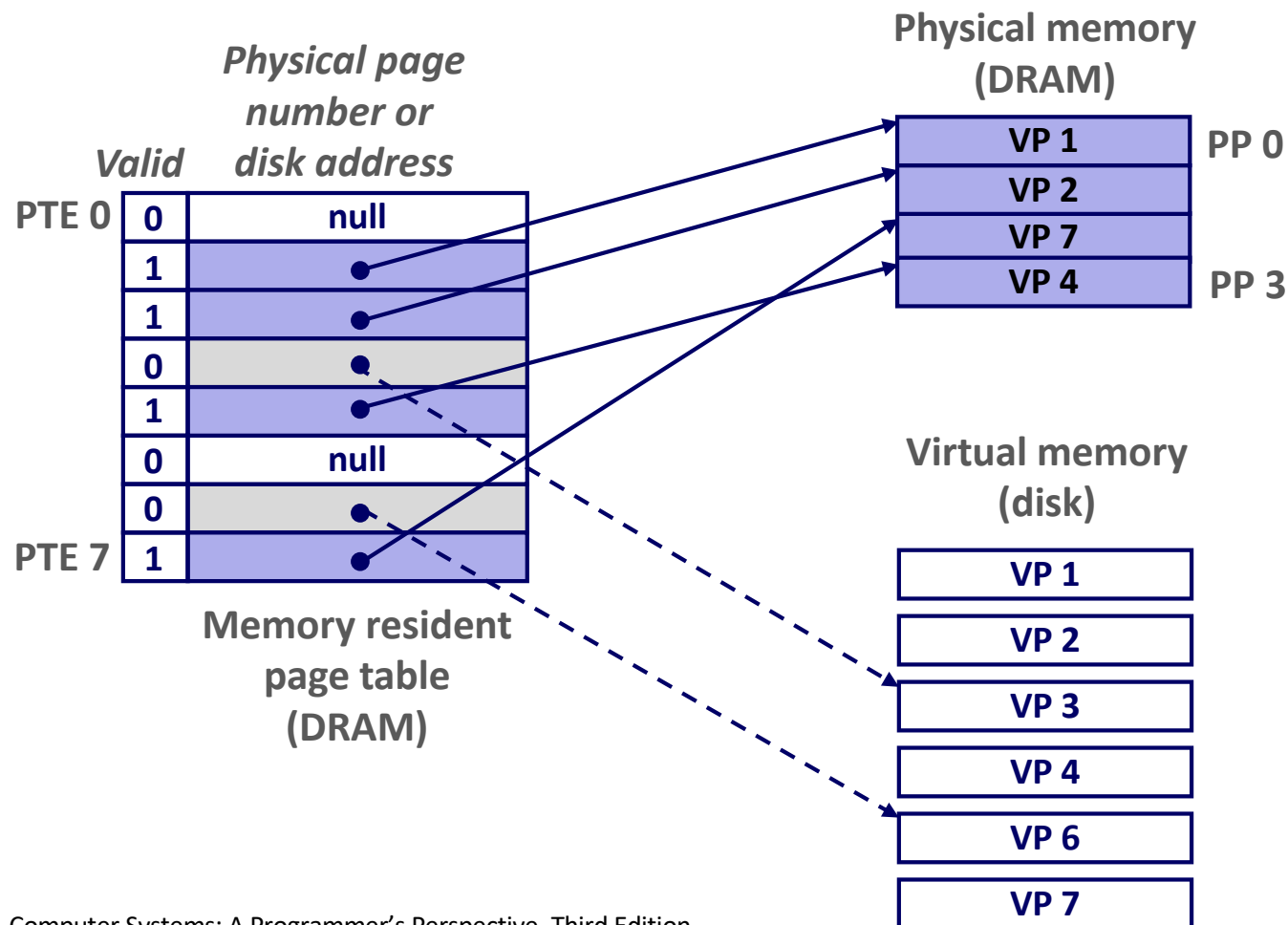
Page Hit

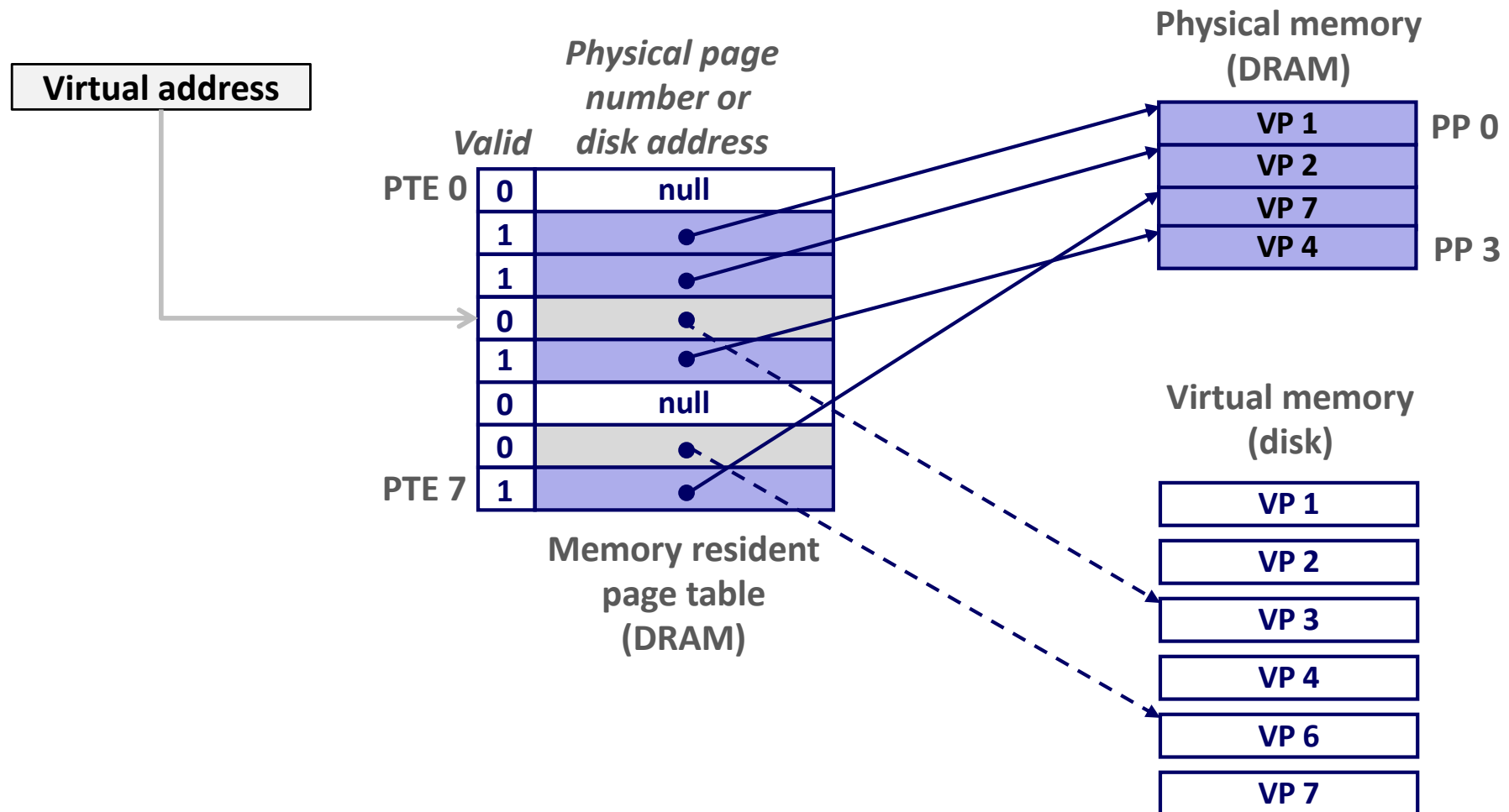- *Page hit:* reference to VM word that is in physical memory (DRAM cache hit)

Page Fault

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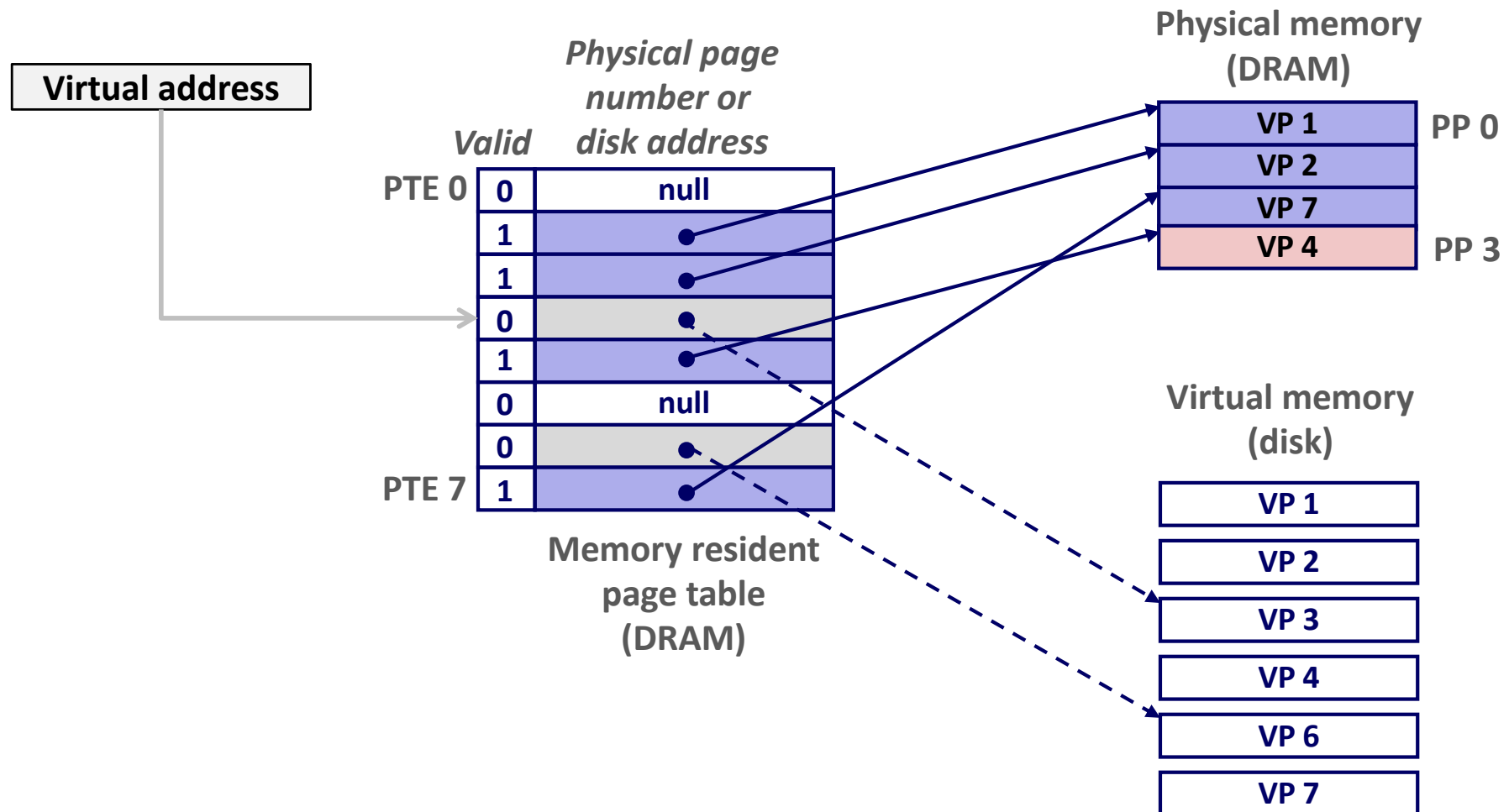
Handling Page Fault

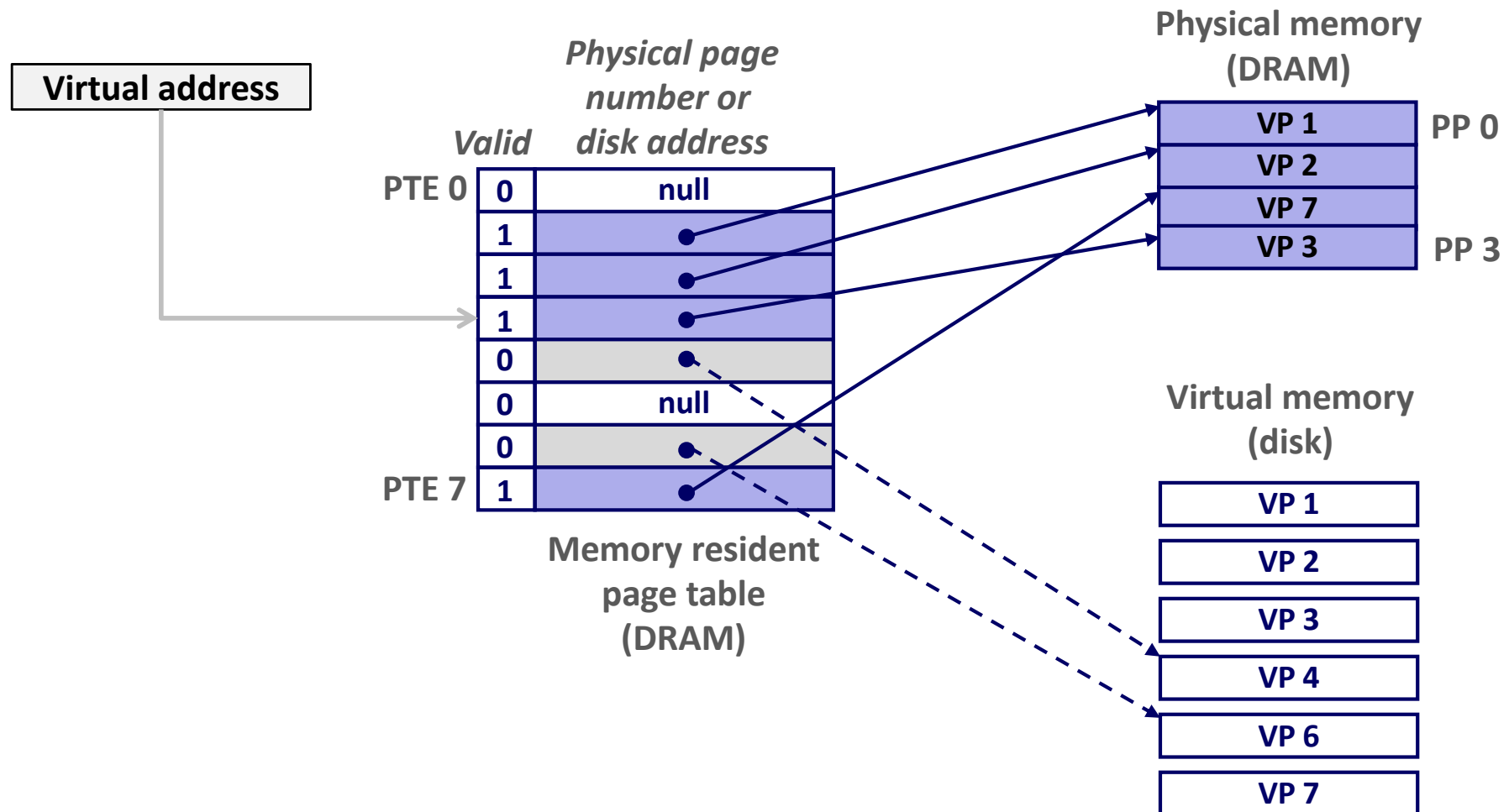■ Page miss causes page fault (an exception)

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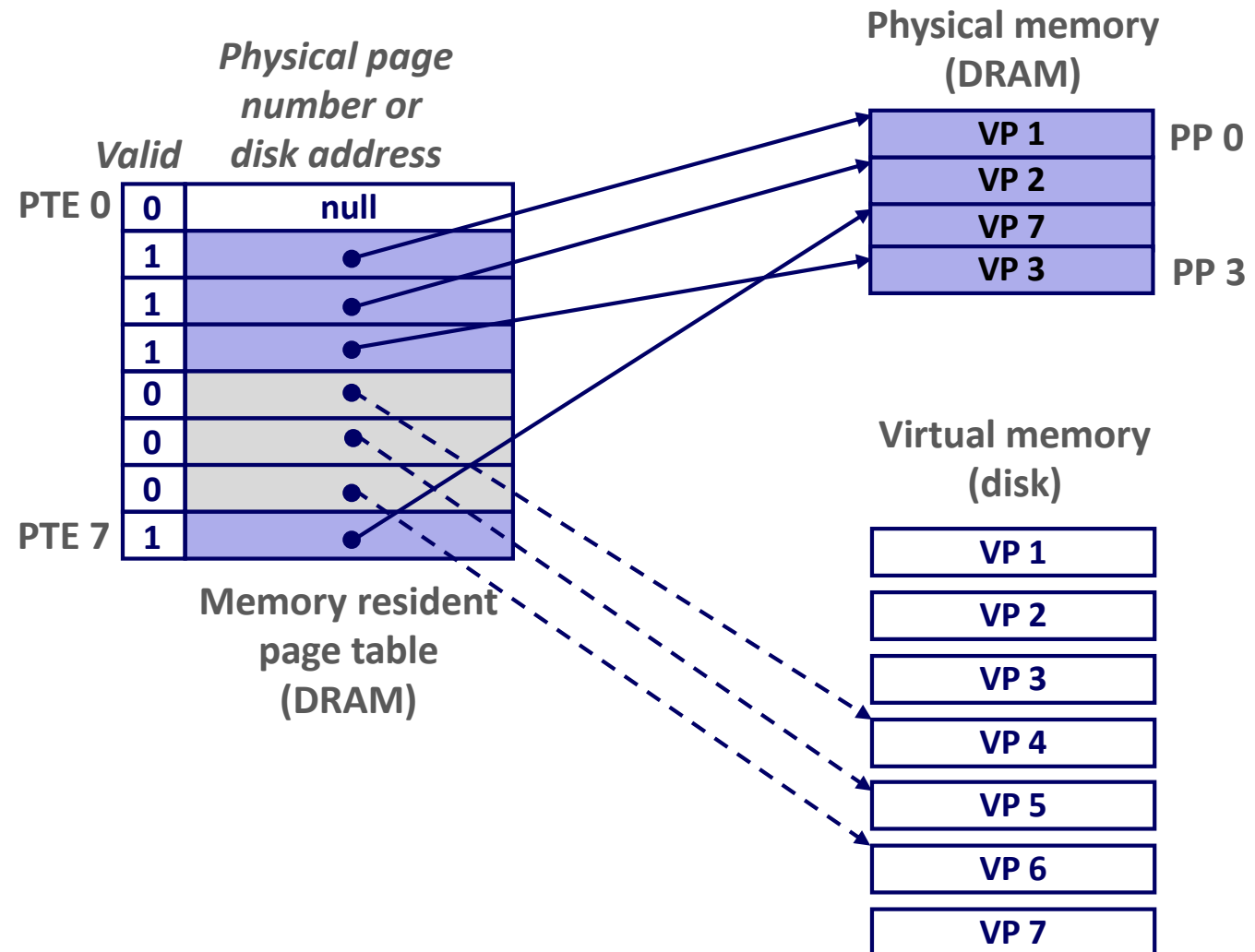- Page fault handler selects a victim to be evicted (here VP 4)

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!

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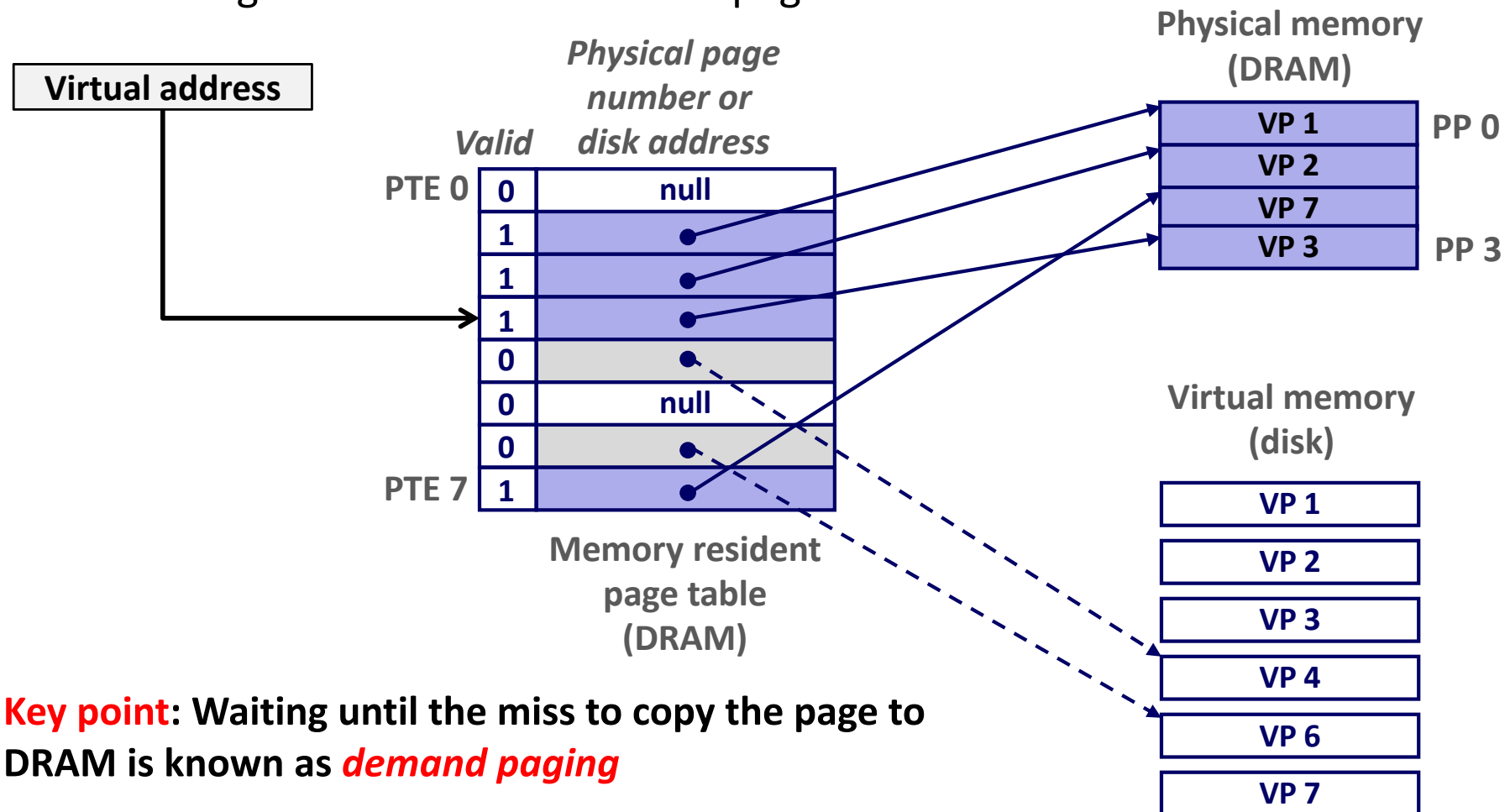
Allocating Pages

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

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

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**

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

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

Address translation

0 ... M-1 *Physical Address Space (DRAM)* — PP 2, PP 6, PP 8

(e.g., read-only library code)

Virtual Address Space for Process 2: 0 ... N-1 — VP 1, VP 2
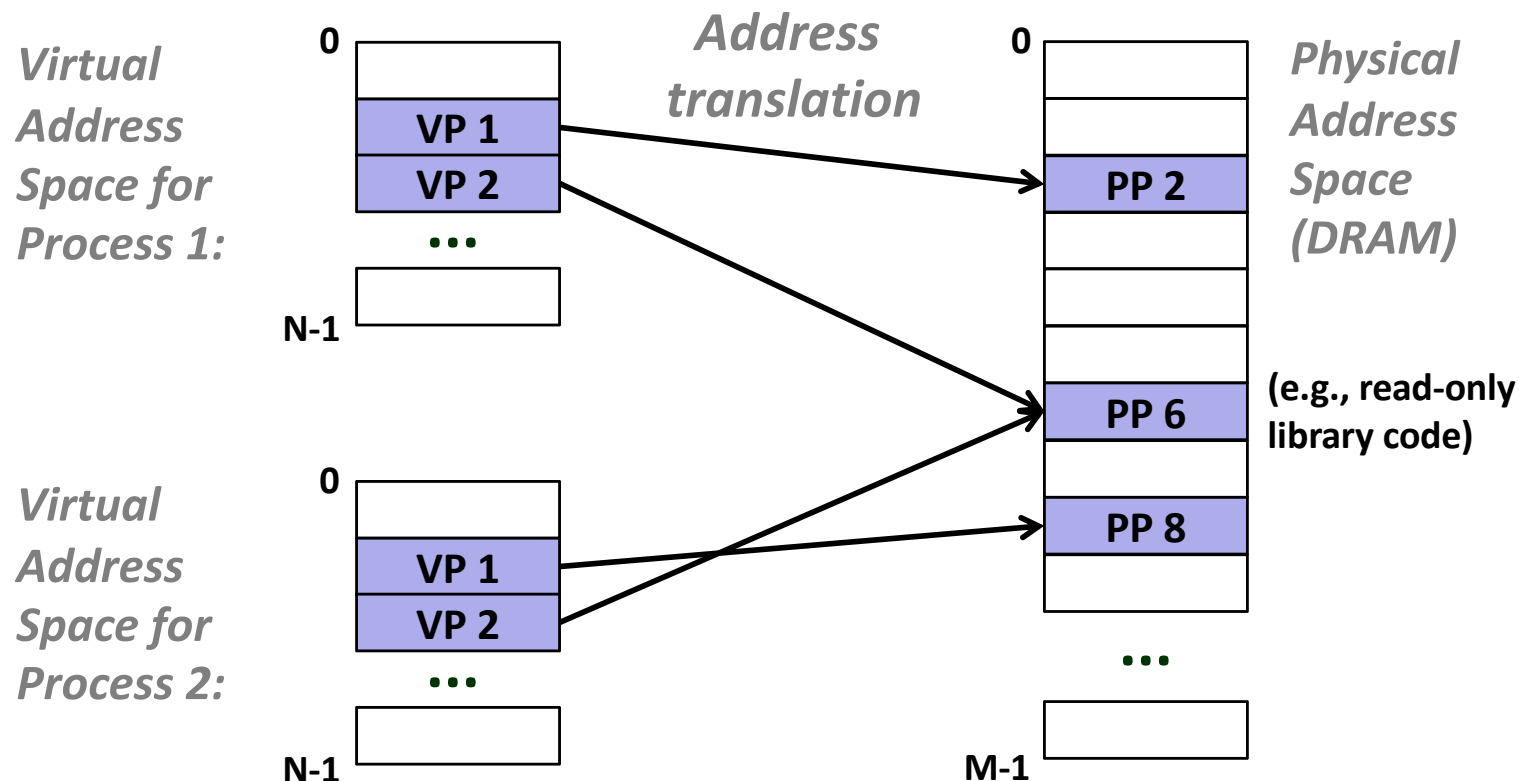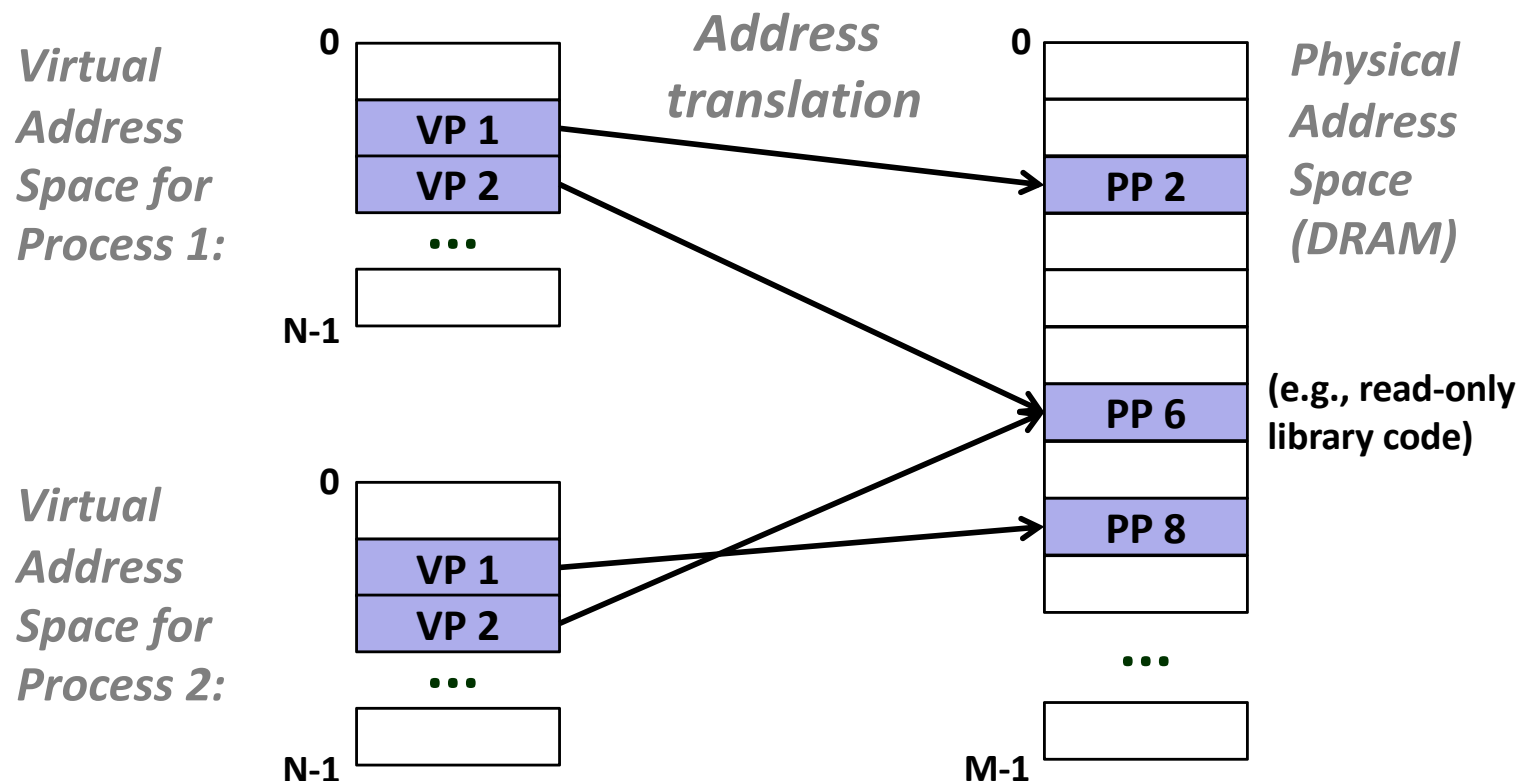
VM as a Tool for Memory Management

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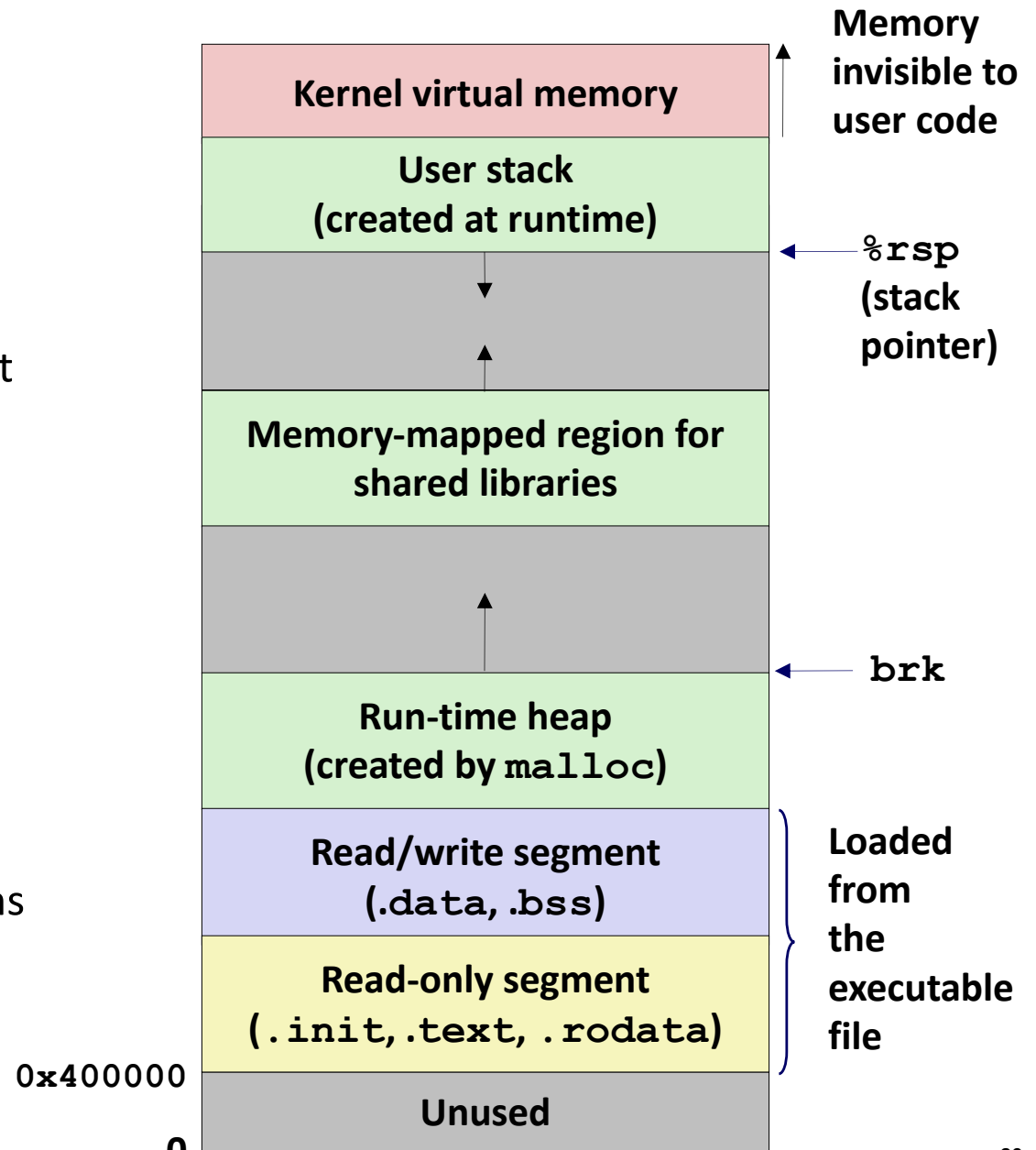 - Map virtual pages to the same physical page (here: PP 6)

Simplifying Linking and Loading

■ Linking

- Each program has similar virtual address space

- Code, data, and heap always start at the same addresses – they are set by the compiler.

■ 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

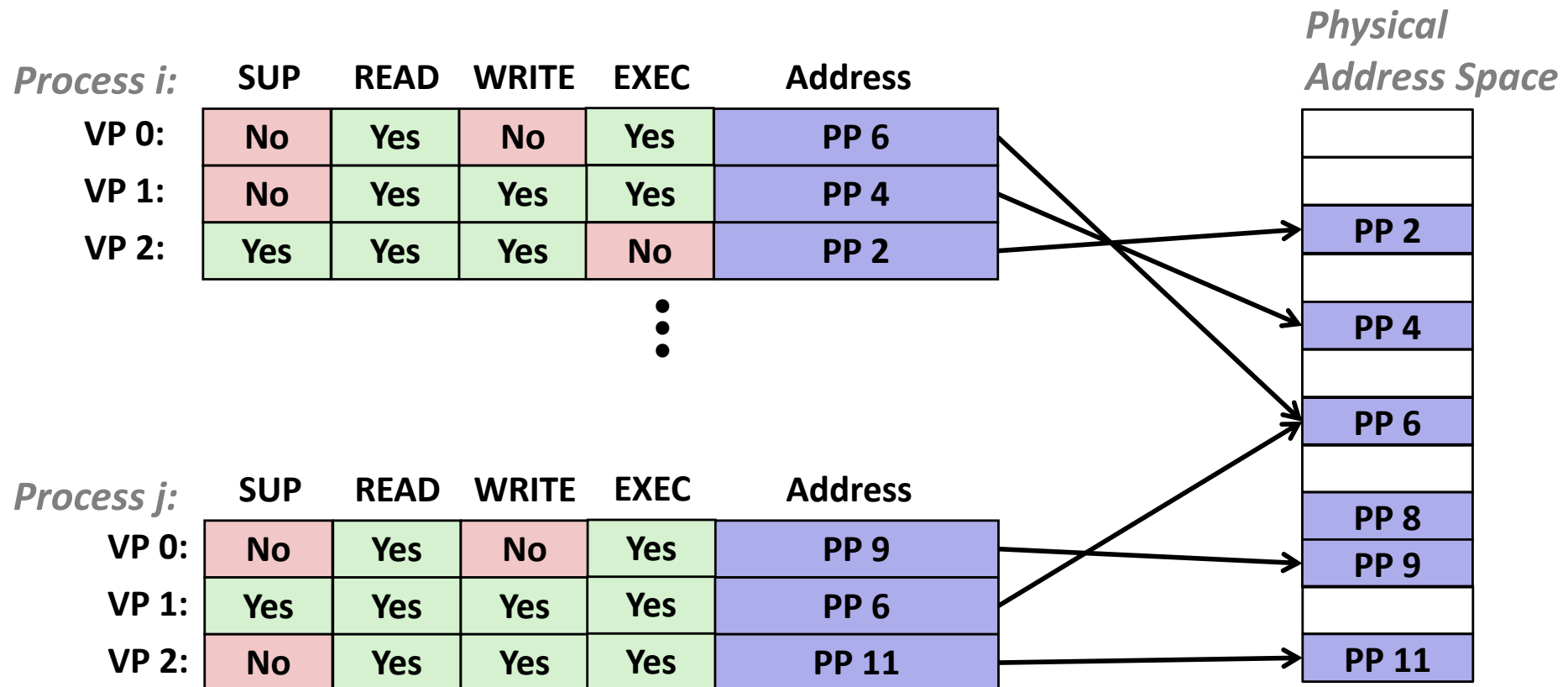| Kernel virtual memory | ← Memory invisible to user code |
User stack (created at runtime)	
↓ ↑	← `%rsp` (stack pointer)
Memory-mapped region for shared libraries	
↑	← `brk`
Run-time heap (created by `malloc`)	
Read/write segment (`.data, .bss`)	Loaded from the executable file
Read-only segment (`.init,.text,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

0x400000

0

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**

VM as a Tool for Memory Protection

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

Physical Address Space

Process i:

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

Process j:

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

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

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**

VM Address Translation

- **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* 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
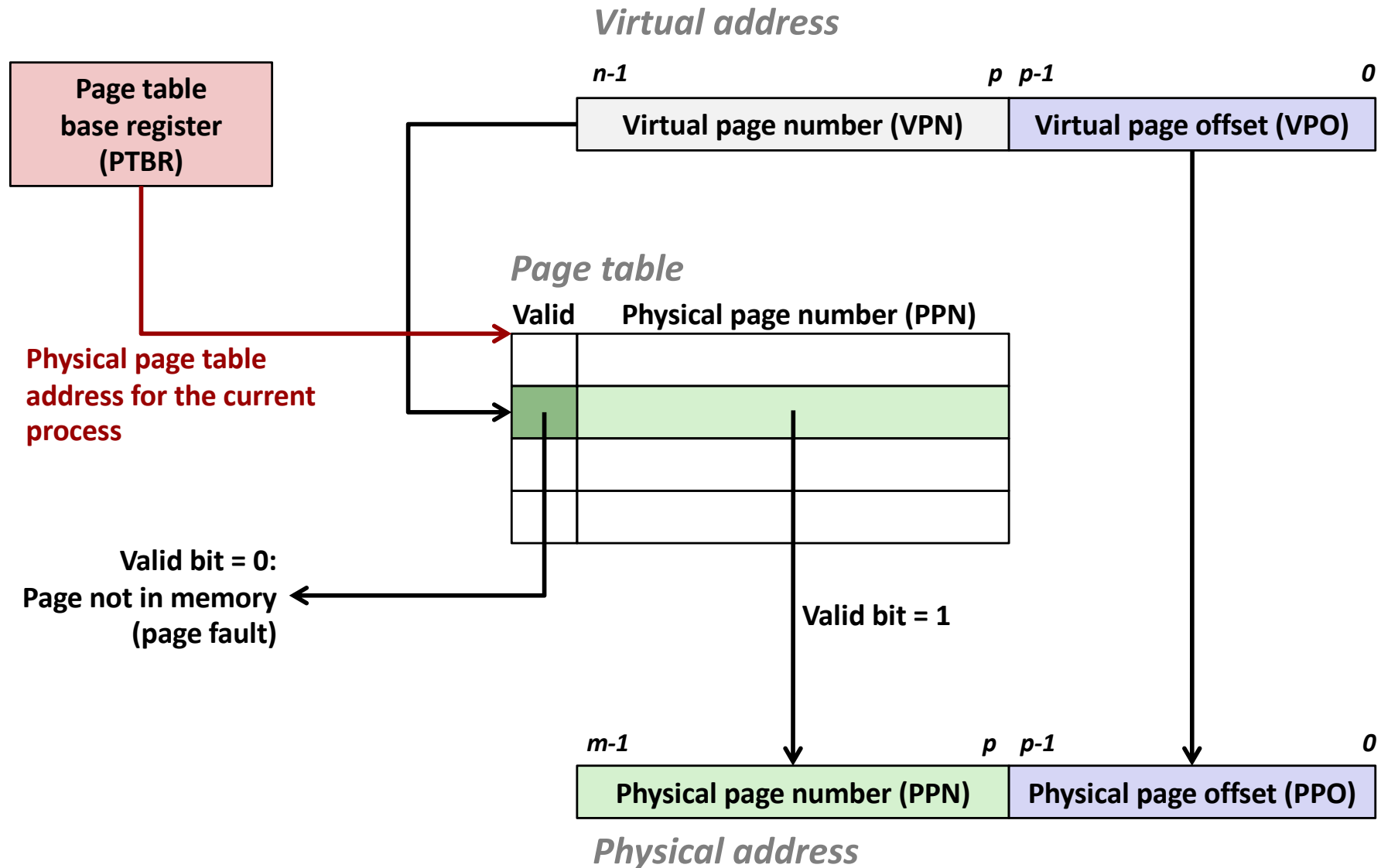 - **P = 2^p** : Page size (bytes)

- **Components of the virtual address (VA)**
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
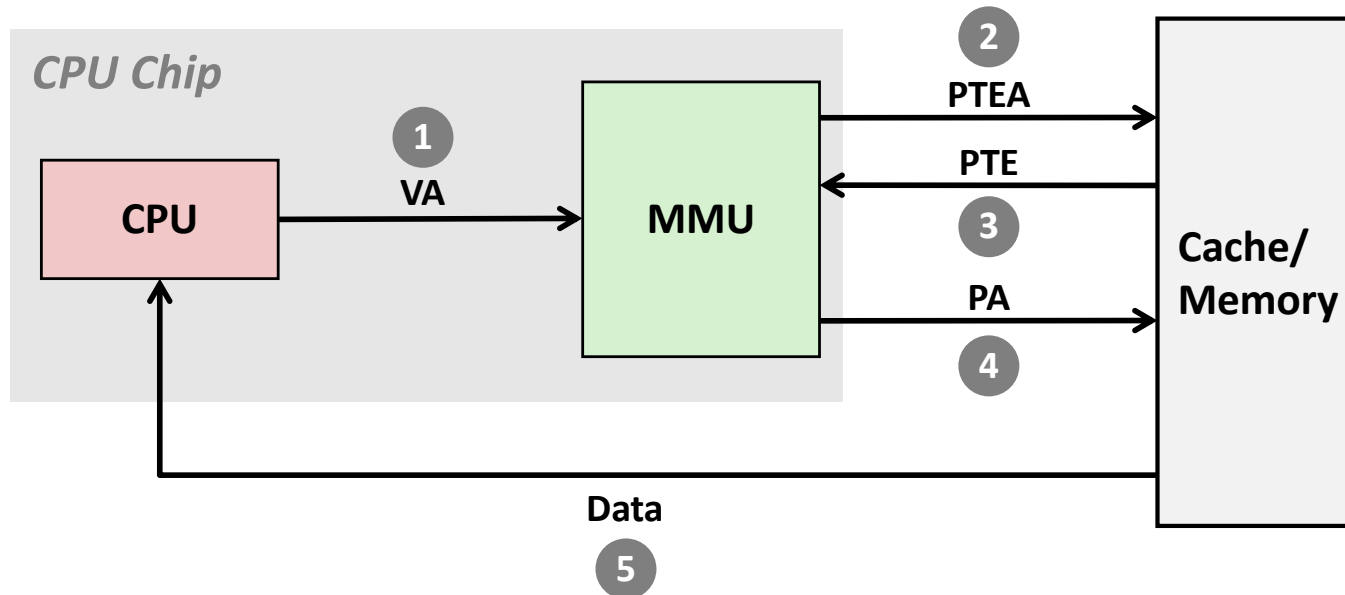 - **VPN**: Virtual page number

- **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number

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)

Physical page table address for the current process

Page table

Valid	Physical page number (PPN)

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

Valid bit = 1

Physical address

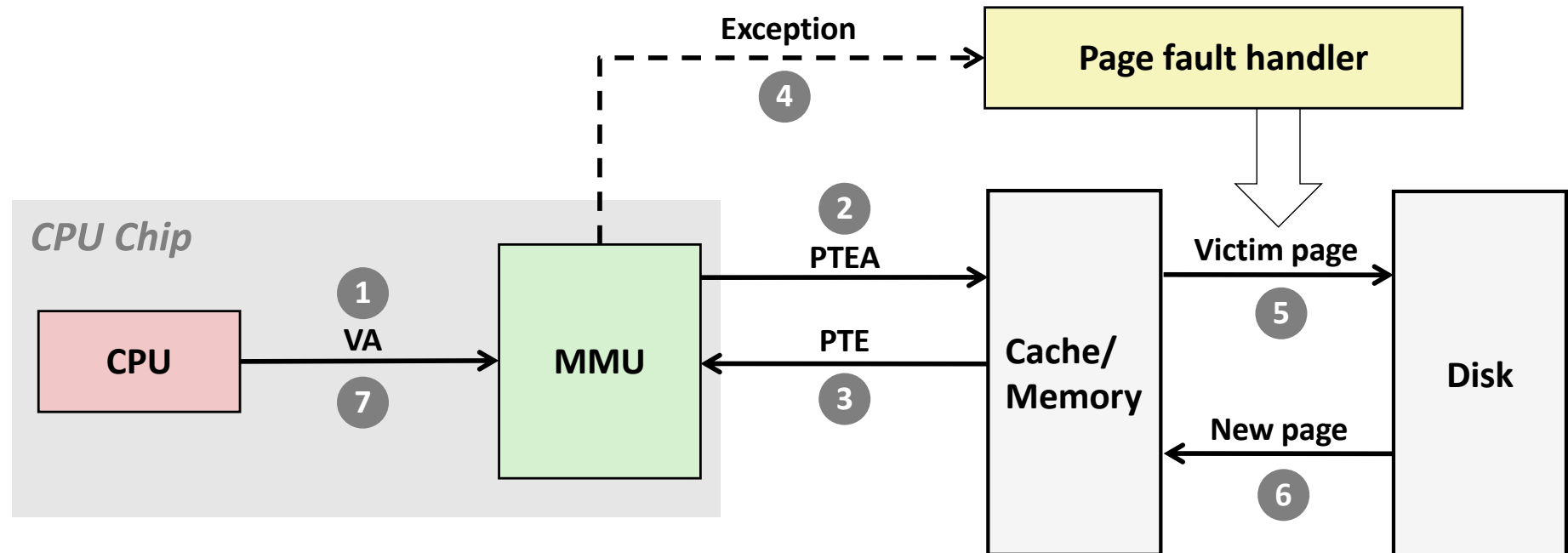| m-1	p p-1	0
Physical page number (PPN)	Physical page offset (PPO)	

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
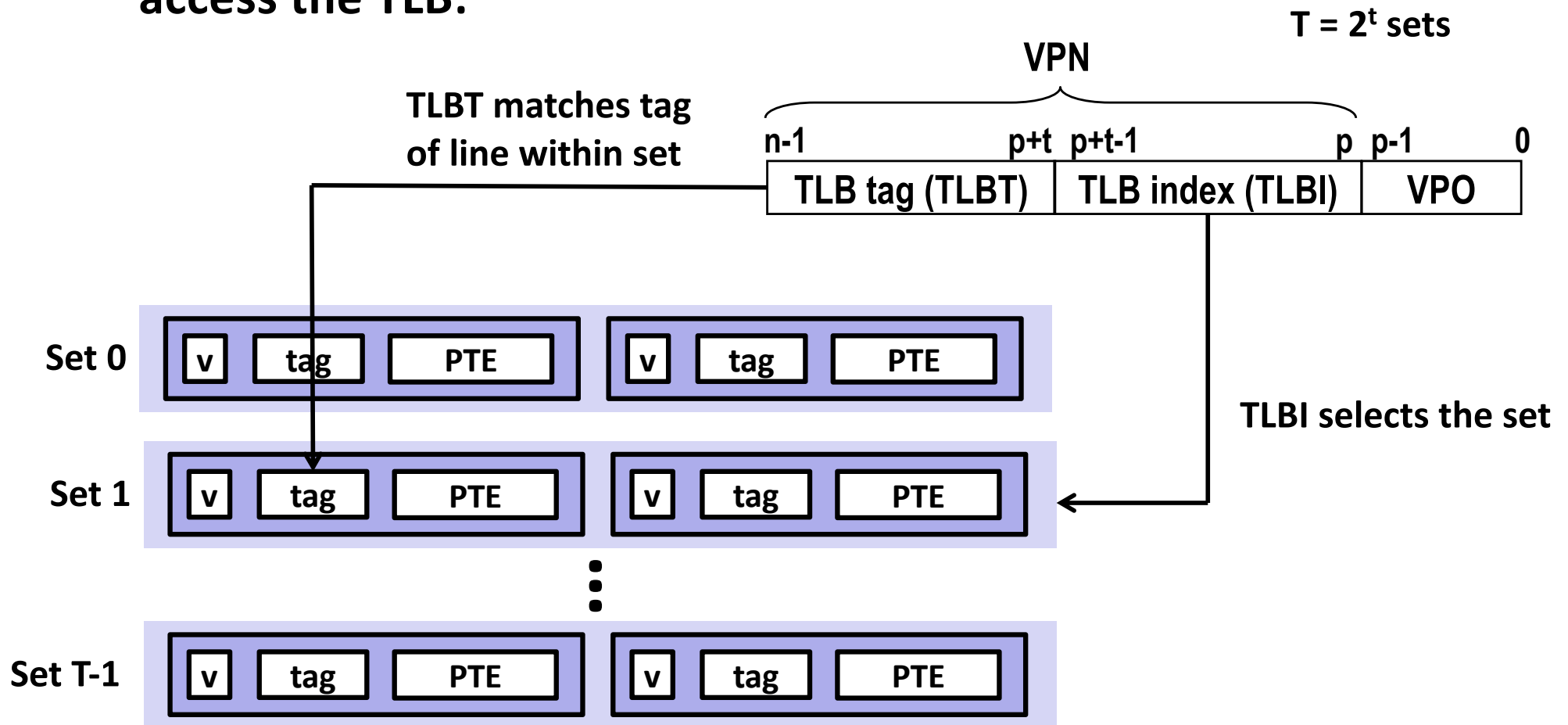
Integrating VM and Cache
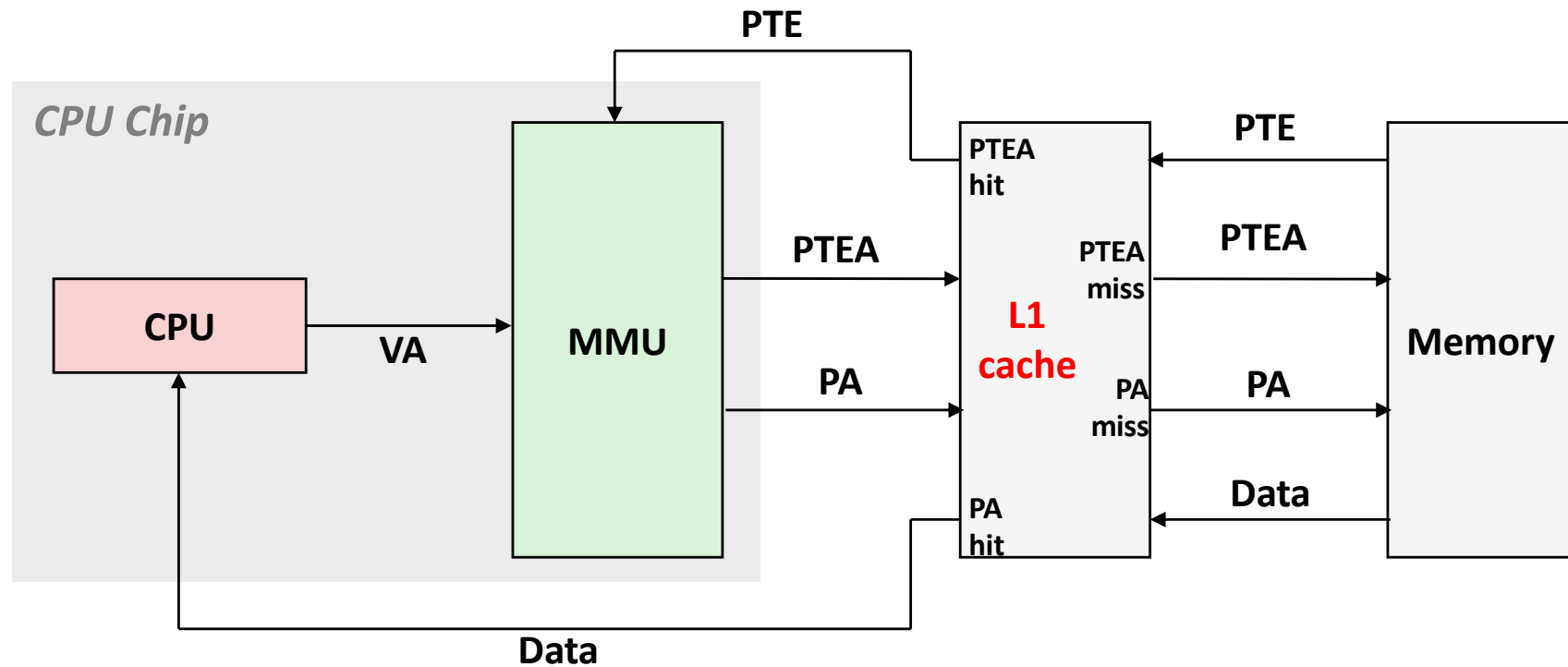
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
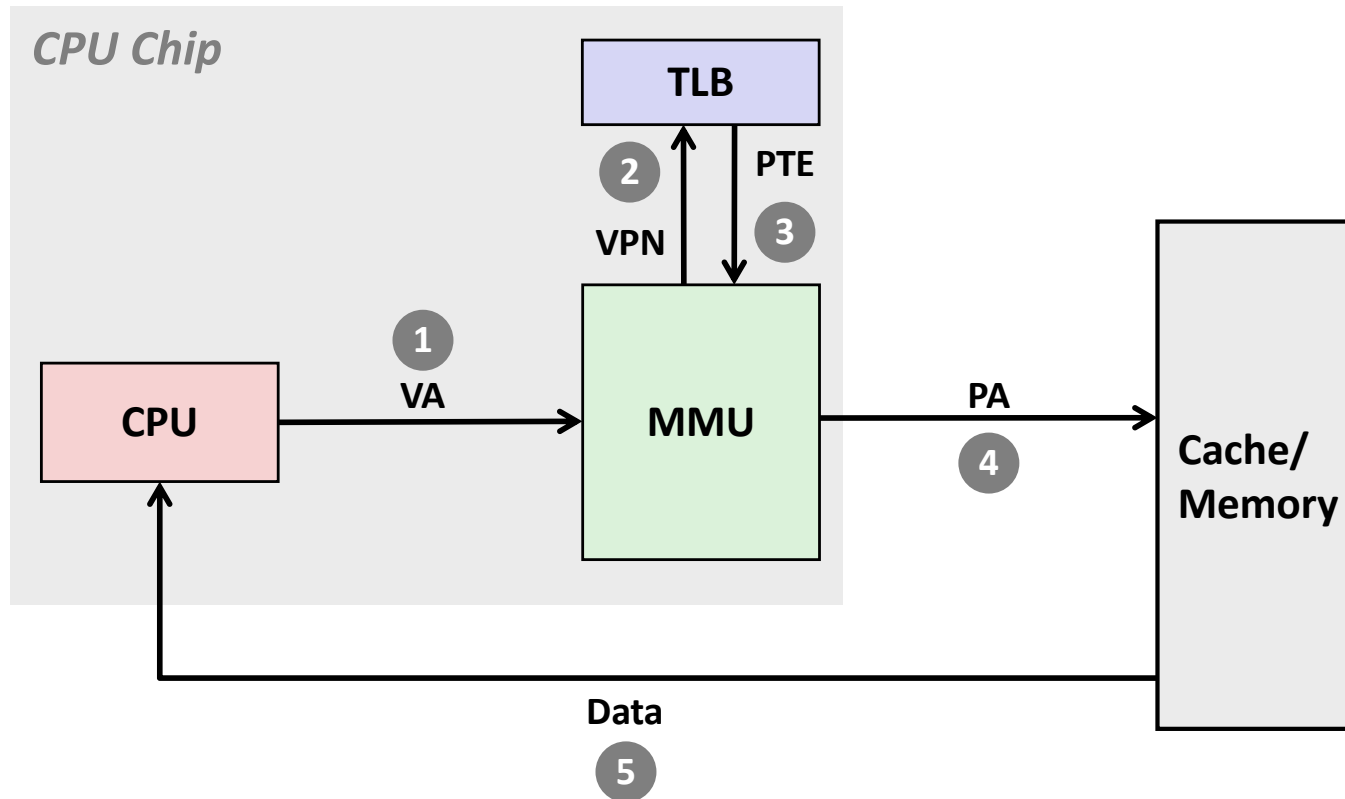
Speeding up Translation with a TLB

- **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:**

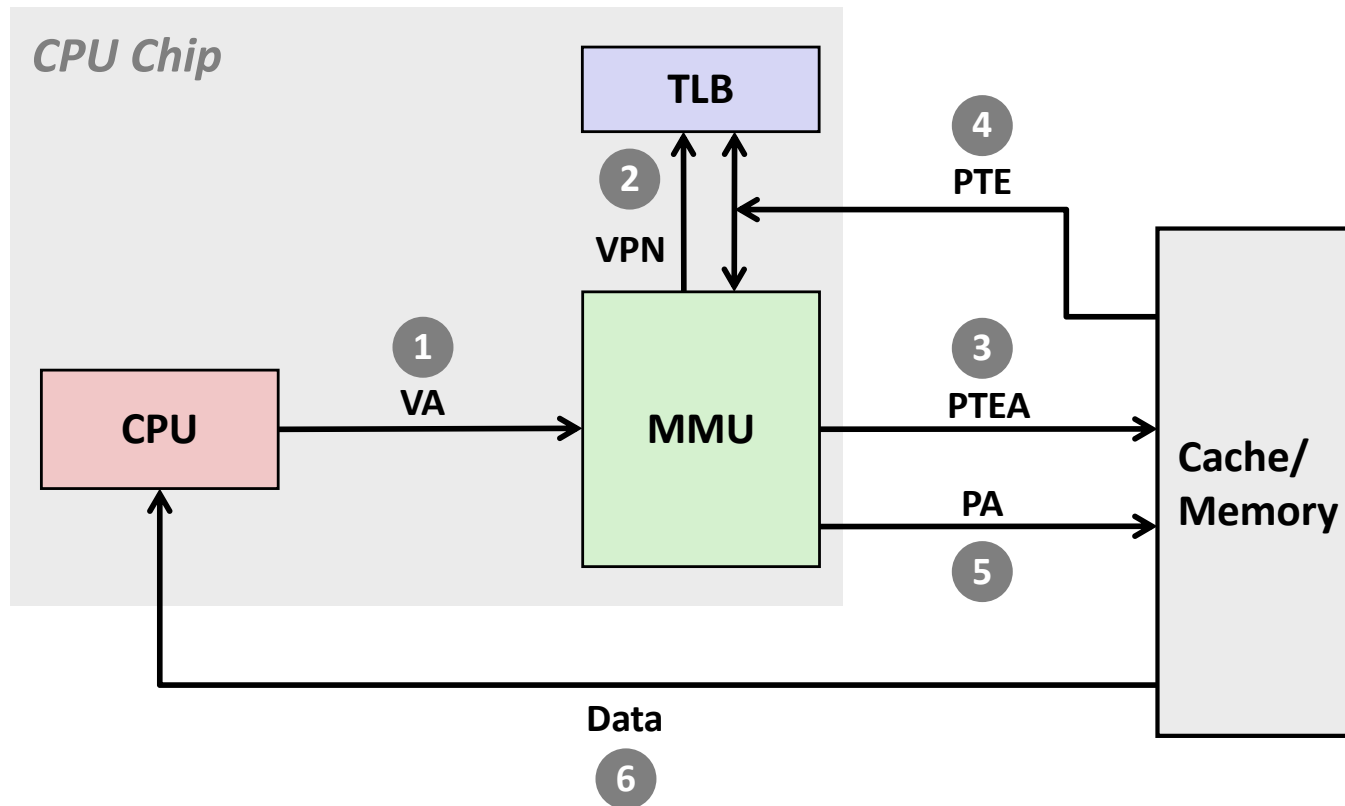$T = 2^t$ sets

TLBT matches tag of line within set

TLBI selects the set

VPN: bits $n-1$ down to $p+t$ (TLB tag, TLBT); bits $p+t-1$ down to p (TLB index, TLBI); bits $p-1$ down to 0 (VPO)

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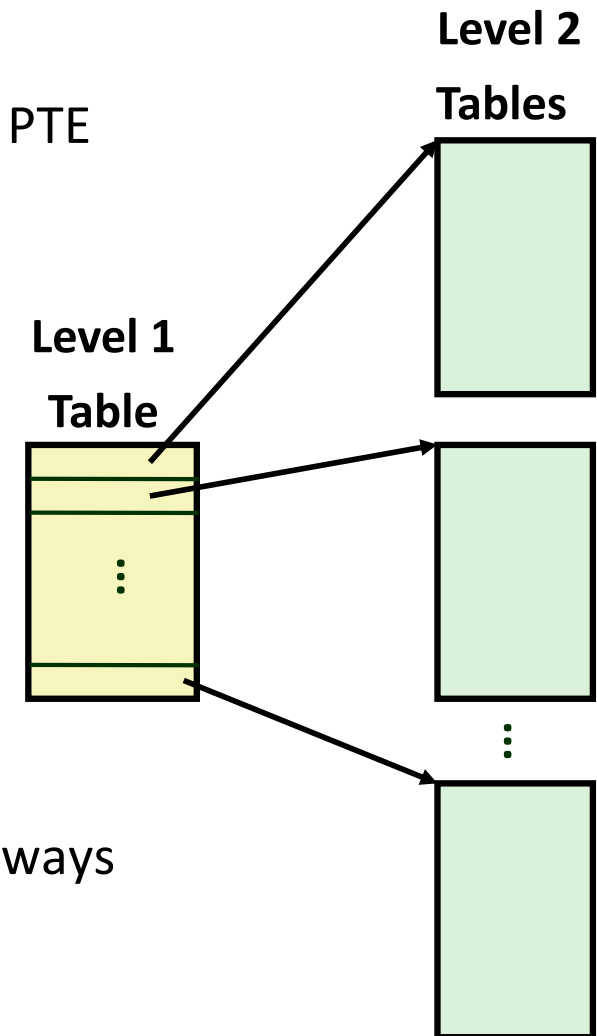
- **Problem:**
  - Would need a 512 GB page table!
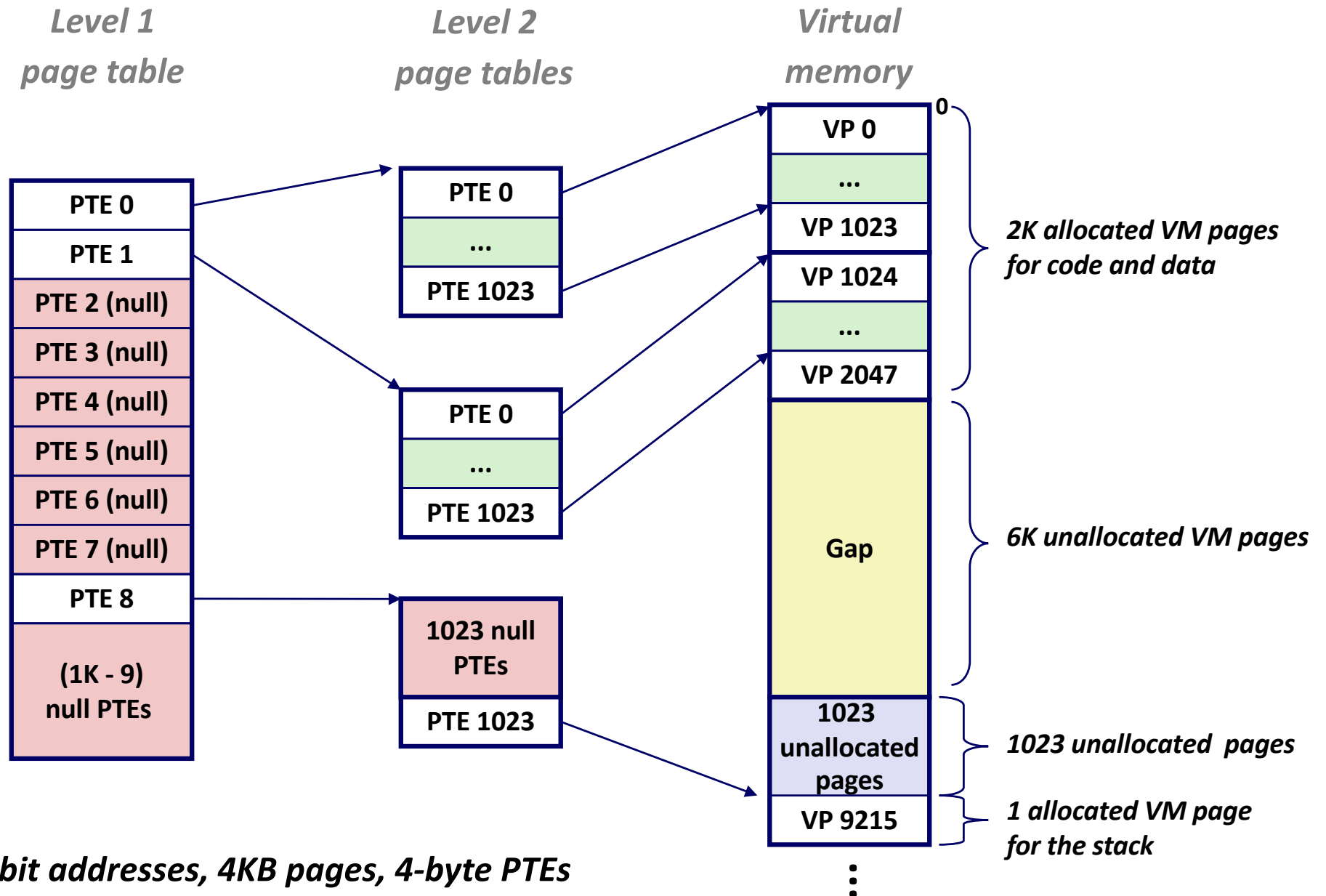   - $2^{48} * 2^{-12} * 2^3 = 2^{39}$ bytes

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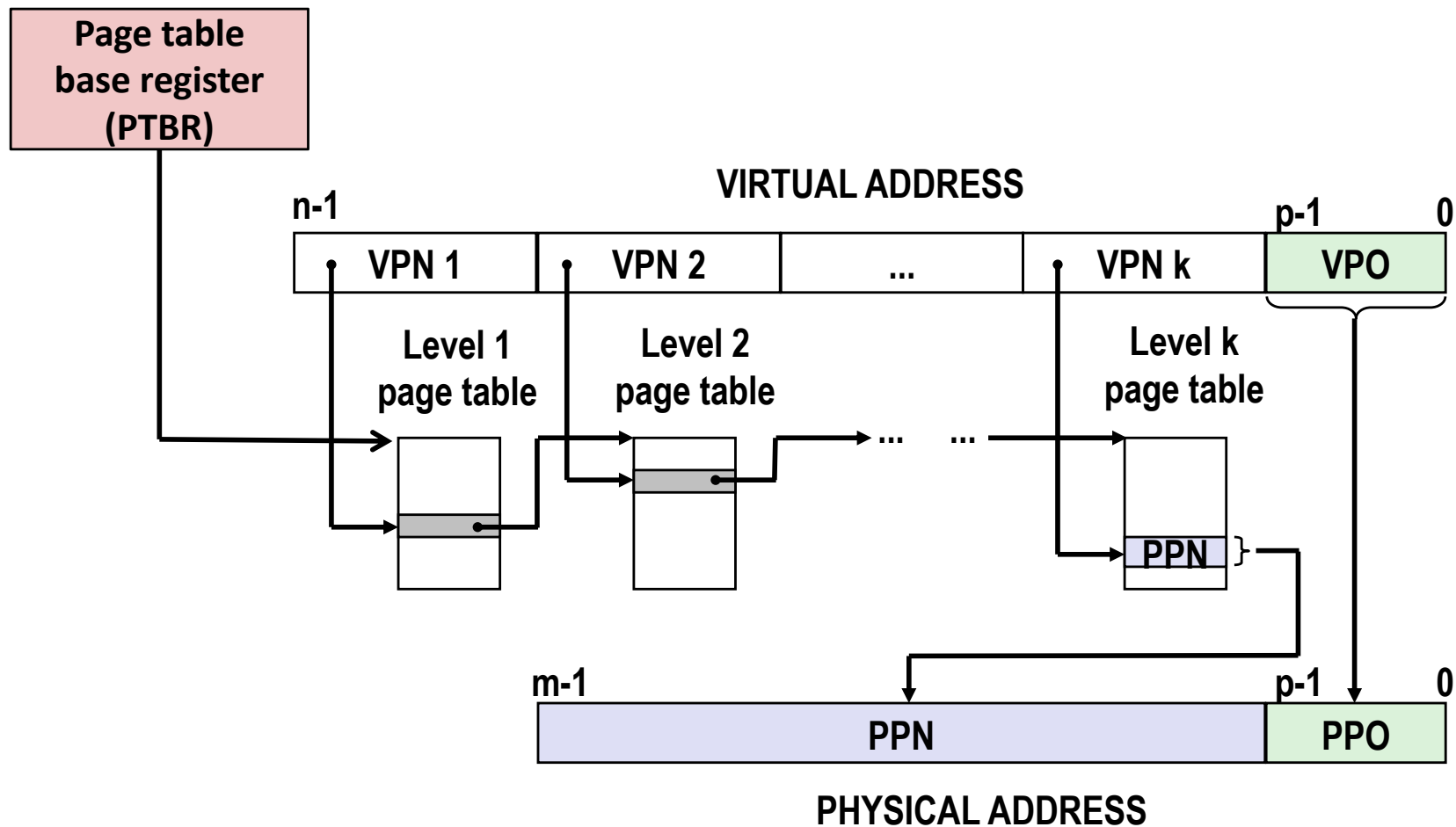
- **Example: 2-level page table**
  - Level 1 table: each PTE points to a page table (always memory resident)
  - Level 2 table: each PTE points to a page (paged in and out like any other data)

Level 2 Tables

Level 1 Table

A Two-Level Page Table Hierarchy

Level 1 page table

Level 2 page tables

Virtual memory

| PTE 0 |
| PTE 1 |
| PTE 2 (null) |
| PTE 3 (null) |
| PTE 4 (null) |
| PTE 5 (null) |
| PTE 6 (null) |
| PTE 7 (null) |
| PTE 8 |
| (1K - 9) null PTEs |

| PTE 0 |
| ... |
| PTE 1023 |

| PTE 0 |
| ... |
| PTE 1023 |

| 1023 null PTEs |
| PTE 1023 |

0

| VP 0 |
| ... |
| VP 1023 |
| VP 1024 |
| ... |
| VP 2047 |

2K allocated VM pages for code and data

| Gap |

6K unallocated VM pages

| 1023 unallocated pages |

1023 unallocated pages

| VP 9215 |

1 allocated VM page for the stack

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

Translating with a k-level Page Table

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