Systems Programming

Virtual Memory: Concepts

Byoungyoung Lee

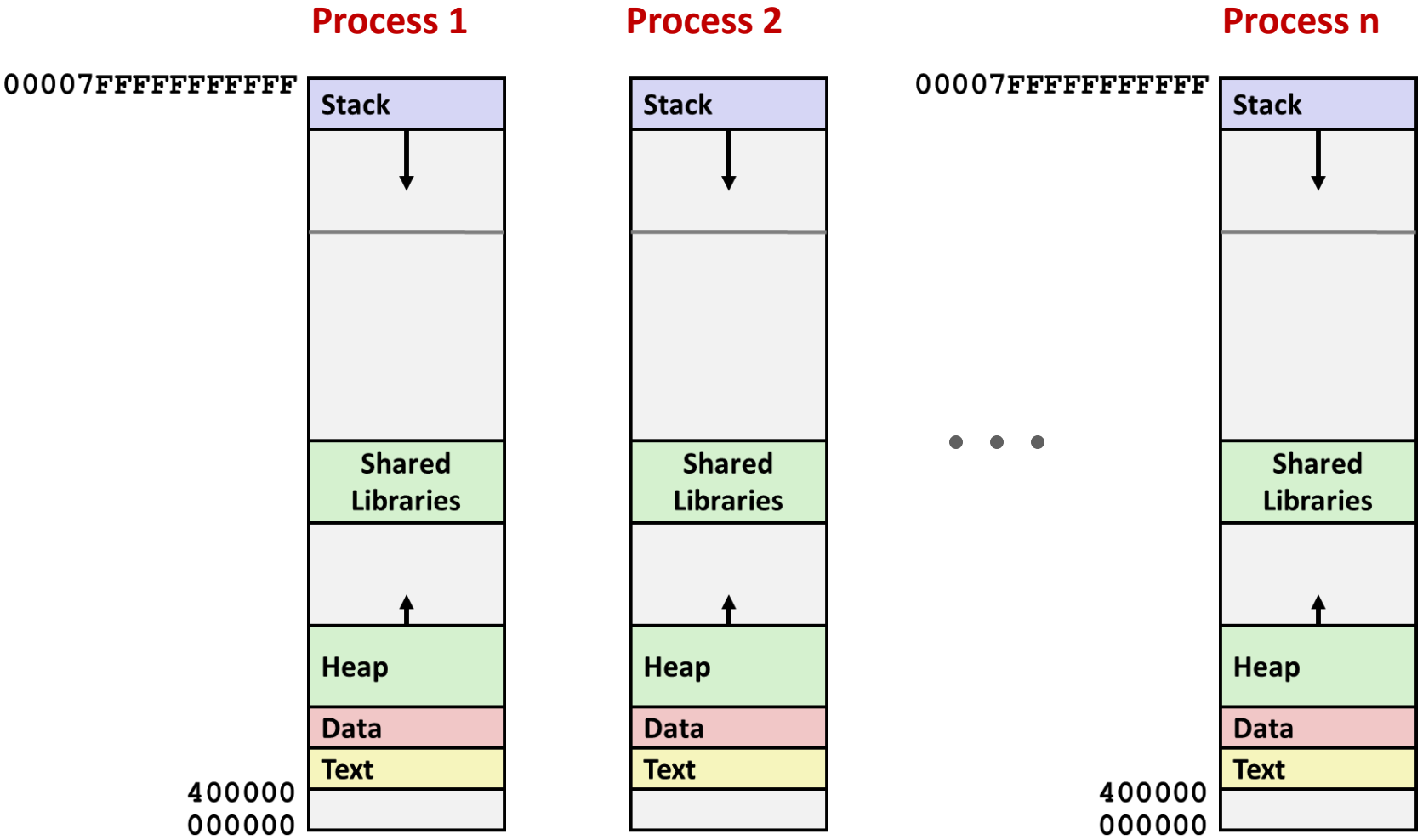
Seoul National University

byoungyoung@snu.ac.kr

https://lifeasageek.github.io

Lecture slides are prepared based on materials provided by CSAPP authors.

Hmmm, How Does This Work?!

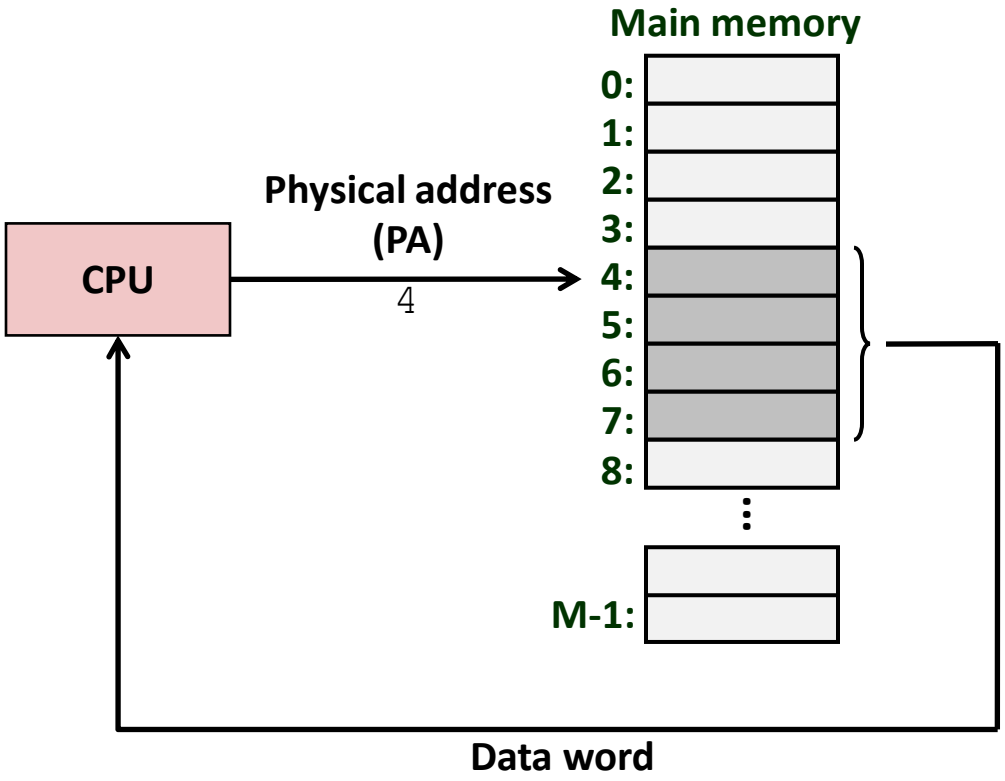


Solution: Virtual Memory

Today

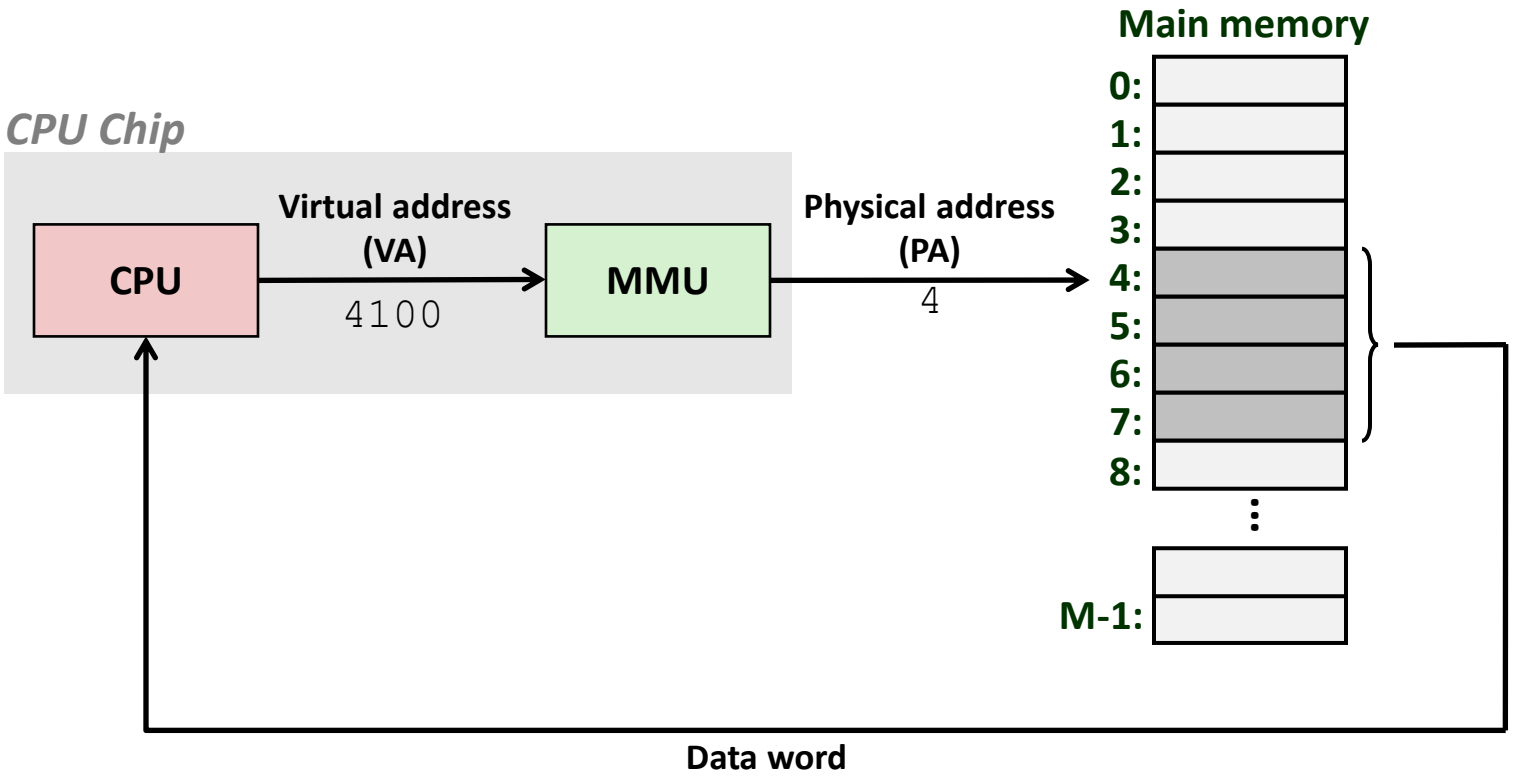
- **Address spaces** — CSAPP 9.1-9.2
- VM as a tool for caching — CSAPP 9.3
- VM as a tool for memory management — CSAPP 9.4
- VM as a tool for memory protection — CSAPP 9.5
- Address translation — CSAPP 9.6

Physical Addressing



- **Used in "simple" systems like embedded microcontrollers in devices**
  - ECUs in automobiles, elevators, sensors, and digital picture frames

Virtual Addressing



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

Address Spaces

- **Virtual address space:** Set of $N = 2^n$ virtual addresses
  {0, 1, 2, 3, …, N-1}

- **Physical address space:** Set of $M = 2^m$ physical addresses
  {0, 1, 2, 3, …, M-1}

- **In practice, N >> M**

Why Virtual Memory (VM)?

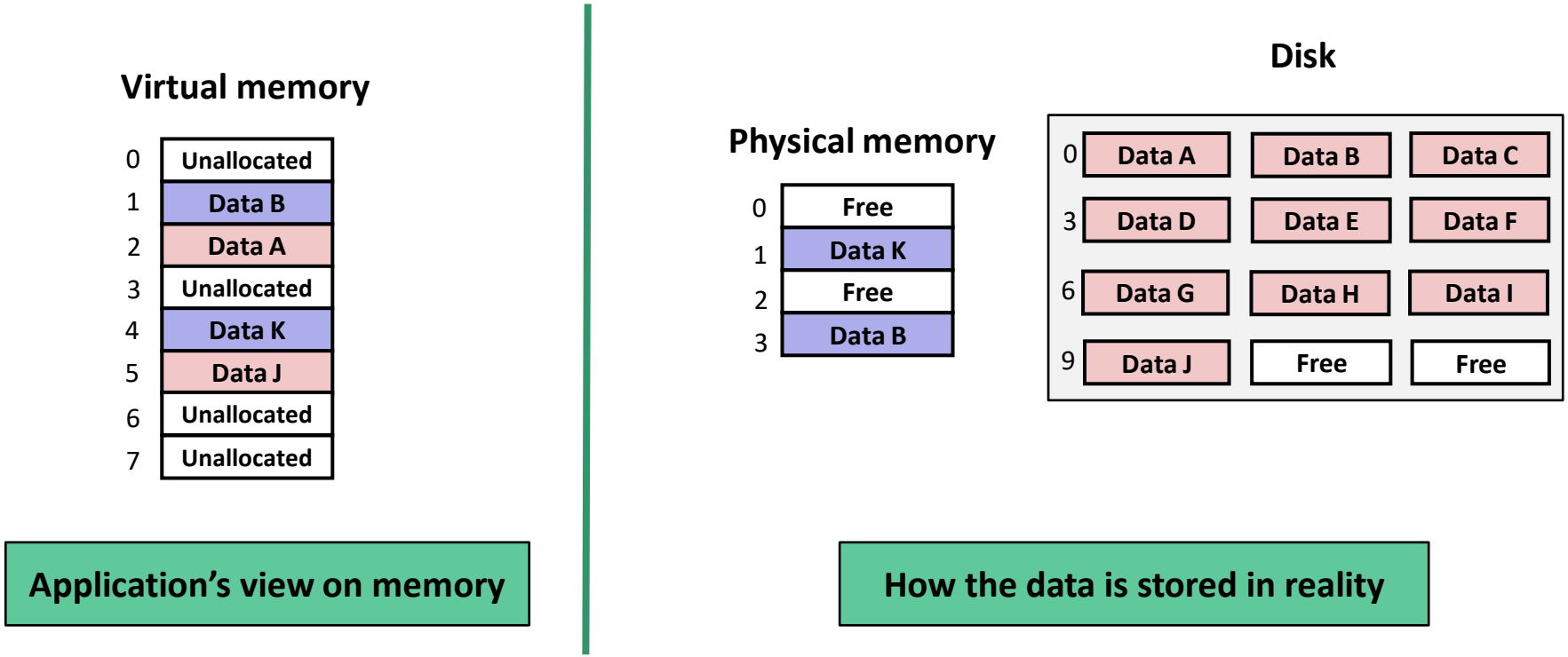
- **Uses main memory efficiently**
  - DRAM acts as a cache: programs can use more memory than DRAM has
- **Simplifies memory management**
  - Each process gets the same uniform address space
- **Isolates address spaces**
  - One process can't interfere with another's memory
  - User program cannot access 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

VM as a Tool for Caching

- **Data initially lives on disk (except zero-filled heap/stack pages)**
- **The data in disk is cached in *physical memory* (*DRAM*)**
  - These cache blocks are called *pages* (the size of each page is 2^p bytes)
  - This DRAM cache is not CPU cache!
- **Virtual memory provides imaginary/virtual memory view for applications**



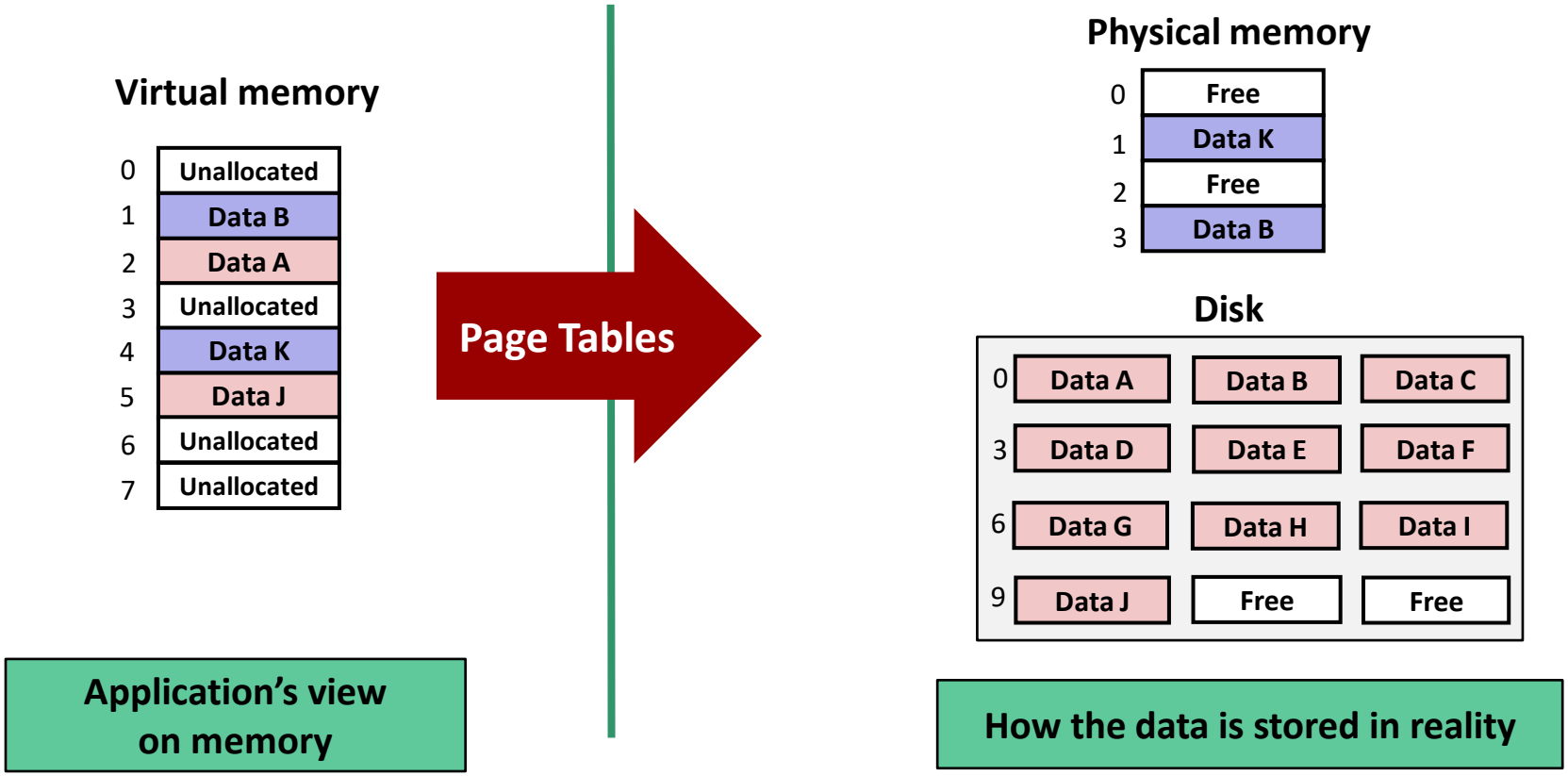
Design Consideration on Caching

- **DRAM is much faster than disk**
  - Disk is ***~1,000x (SSD) to ~100,000x (HDD)*** slower than DRAM

- **Consequences**
  - Fully associative
    - Any virtual page (VP) can be placed in any physical page (PP)
    - Requires a “large” mapping function
      - Quite different from CPU cache (Hardware)
  - Highly sophisticated, expensive software-only replacement algorithms
    - Too complicated and open-ended to be implemented in hardware

Enabling Data Structure: Page Table

- **A *page table* is an array of page table entries (PTEs)**
  - Per-process kernel data structure in the main memory
- **A page table maps virtual memory to physical memory.**
  - It maps from "imaginary memory view" to "real-world memory view"



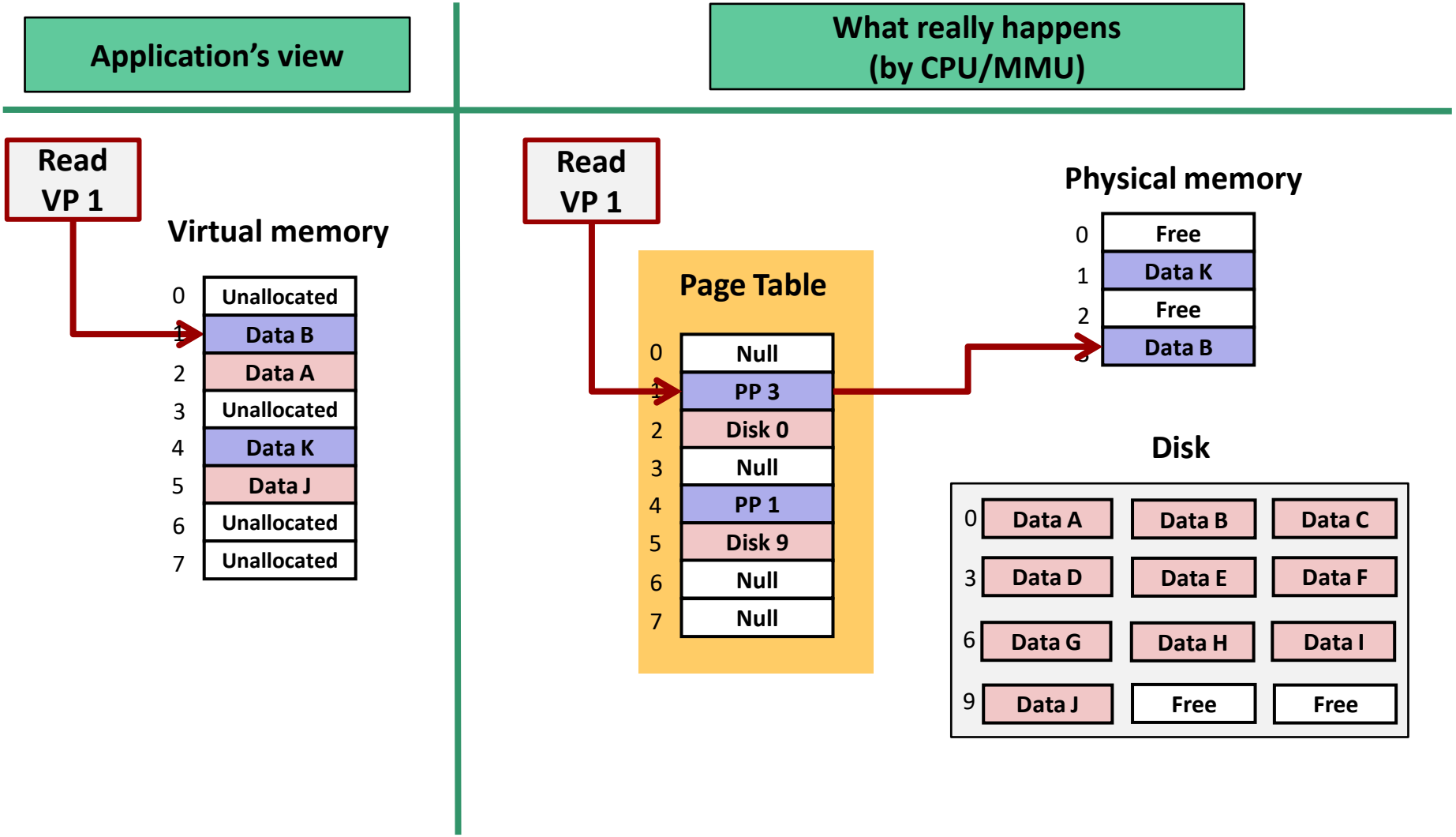
Page Table Entry (PTE)

- **Each PTE = valid bit + address**
  - Valid = 1: cached in DRAM; address = physical page (e.g., PP 3)
  - Valid = 0, disk address: not cached, stored on disk (e.g., Disk 0)
  - Valid = 0, null: unallocated
- **Real PTEs carry more bits**
  - Dirty bit: page was modified → write it back to disk on eviction
  - Permission bits (read/write/exec, kernel-only): later in this lecture

Valid	Address	
1	PP 3	← *cached in DRAM*
0	Disk 0	← *on disk, not cached*
0	Null	← *unallocated*

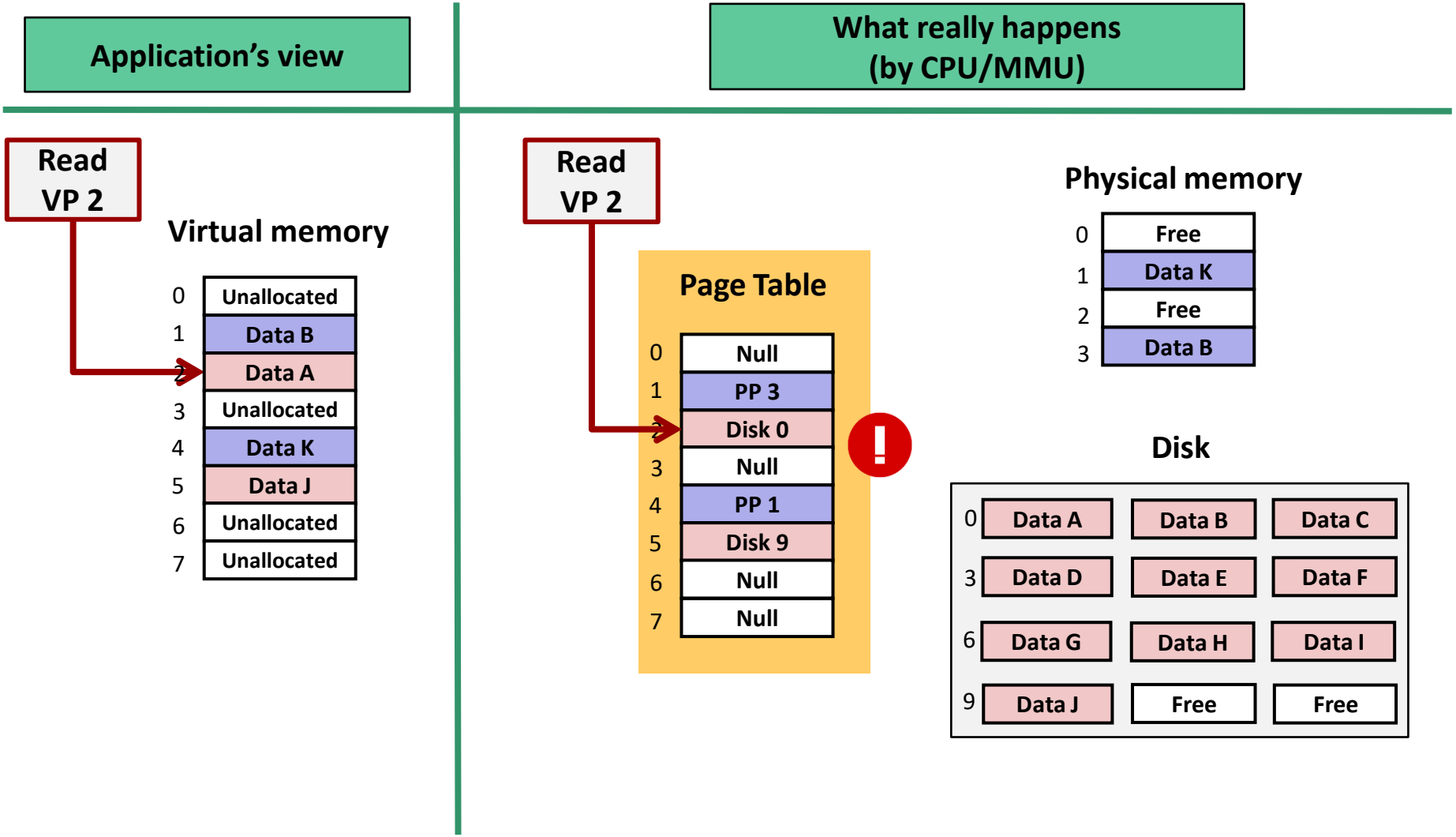
Page Hit

- *Page hit:* reference to VM word that is in physical memory



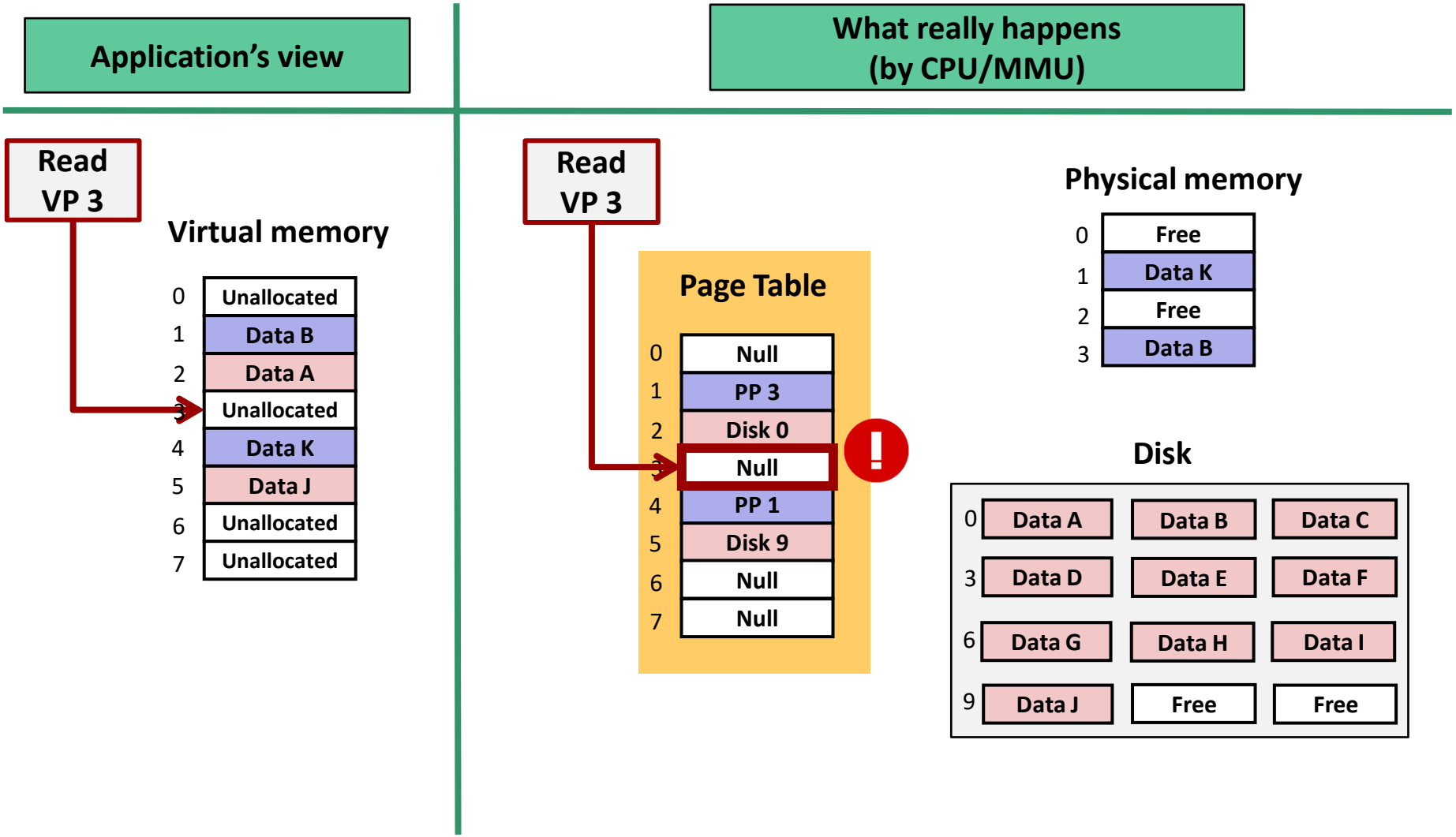
Page Fault (1)

- *Page fault:* reference to VM word not in physical memory (on disk)



Page Fault (2)

- *Page fault:* reference to VM word in an unallocated page

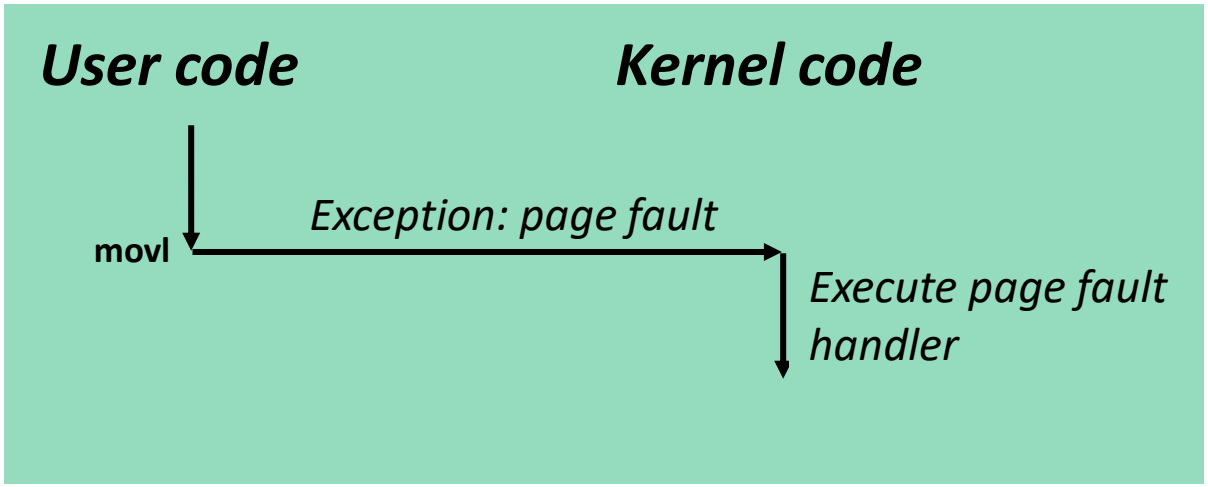


Triggering a Page Fault

- User accesses a memory location whose page is not in DRAM

```
40113a:  c7 04 25 10 9d 40 00 0d 00 00 00  movl $0xd,0x409d10
```

- The page may be on disk — or not allocated at all
- MMU triggers page fault exception
  - (More details in later lecture)
  - Raise privilege level to supervisor mode
  - Causes procedure call to software page fault handler



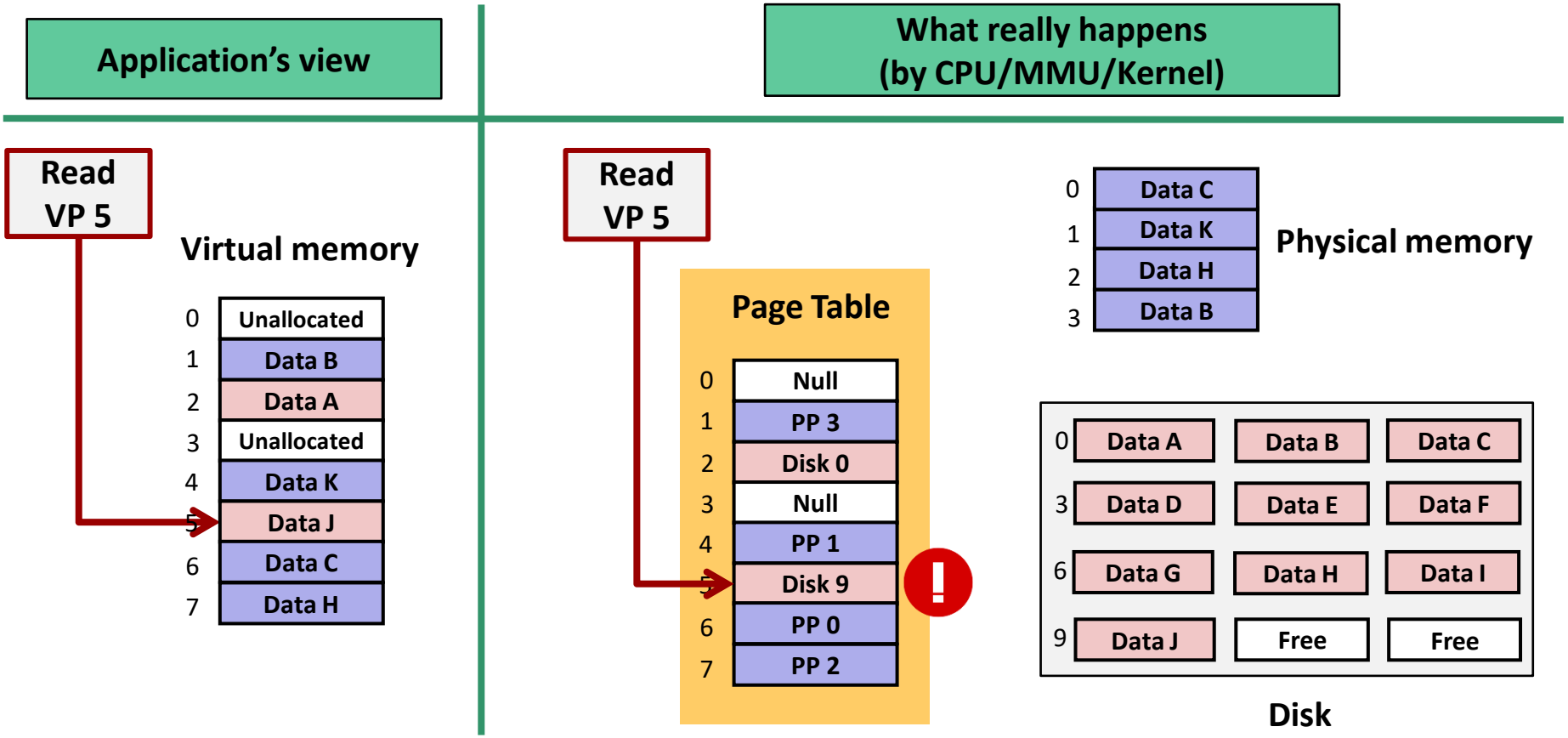
Page Fault: Recoverable or Not?

- **If the page is allocated but not in physical memory**
  - A page fault can be recovered
  - On disk: kernel loads the page from disk into physical memory
  - Never touched yet (e.g., new heap/stack page): kernel allocates a zero-filled page (more in VM 2)
  - Then re-executes the instruction that raised the page fault

- **If the page is not allocated (e.g., VP 3, via a wild pointer)**
  - A page fault cannot be recovered
  - Kernel terminates the process: segmentation fault (SIGSEGV)

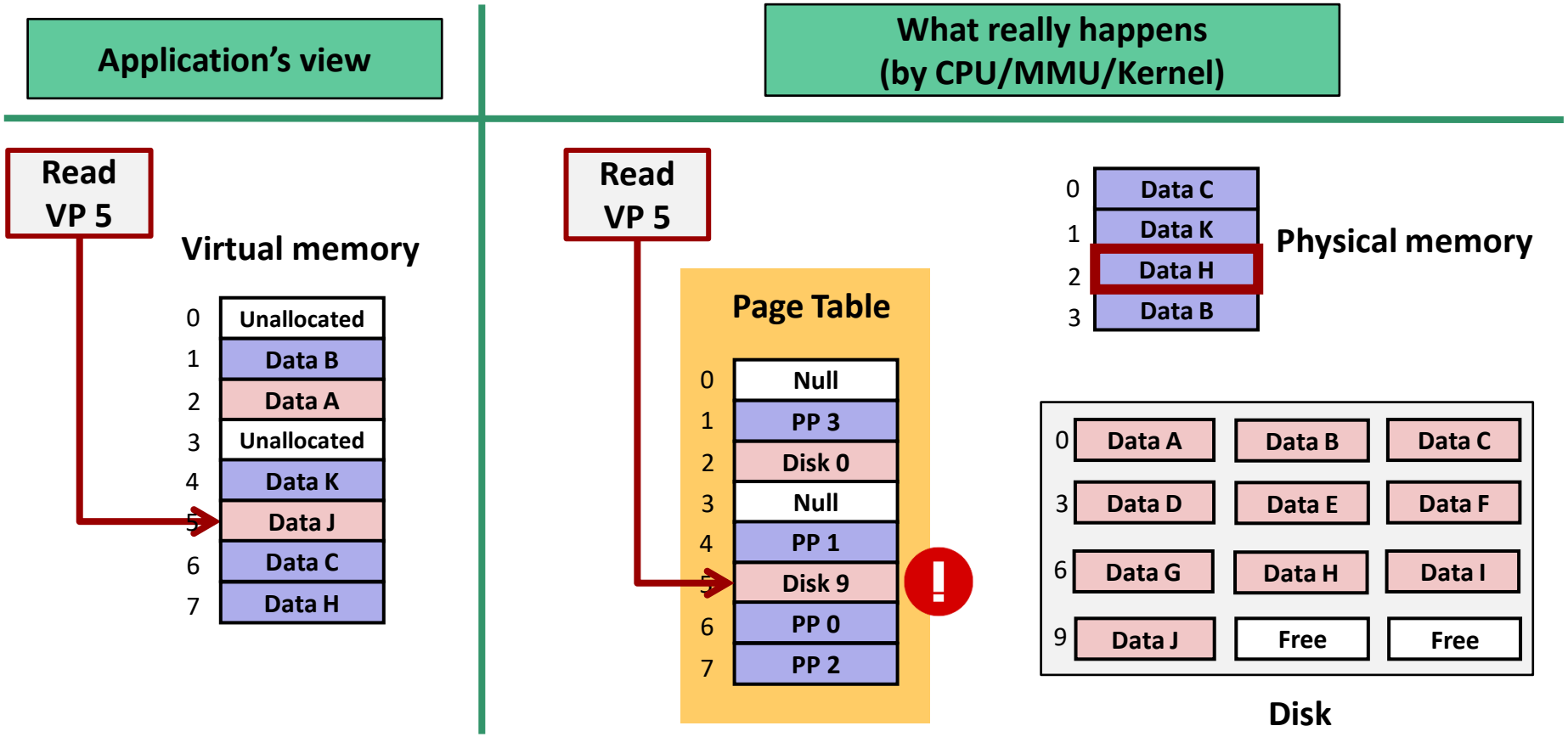
Handling Page Fault (1)

- Page miss (DRAM cache miss) causes page fault (an exception)



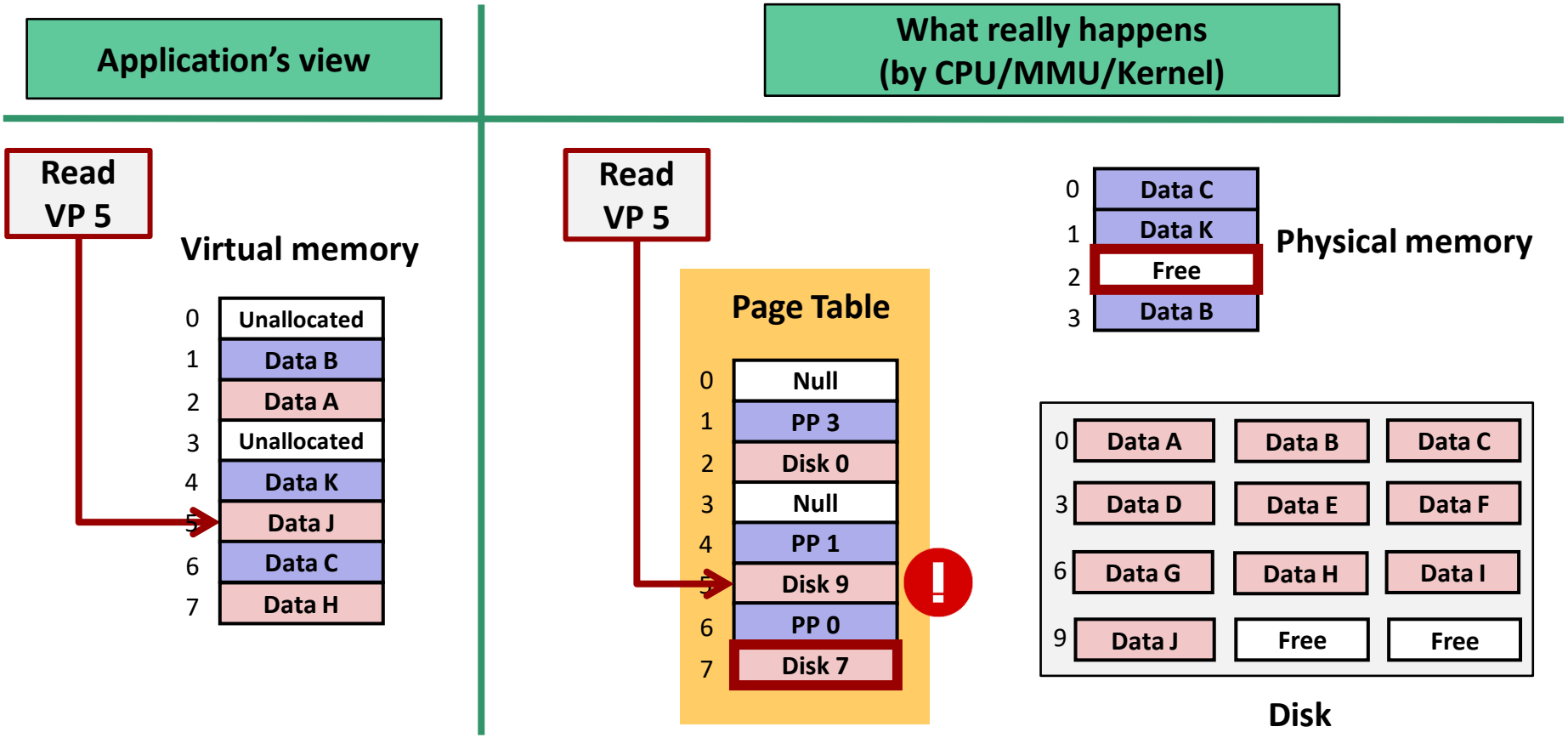
Handling Page Fault (2)

- Page miss (DRAM cache miss) causes page fault (an exception)
- **Page fault handler selects a victim to be evicted (here "Data H" at "VP 7" or "PP 2")**



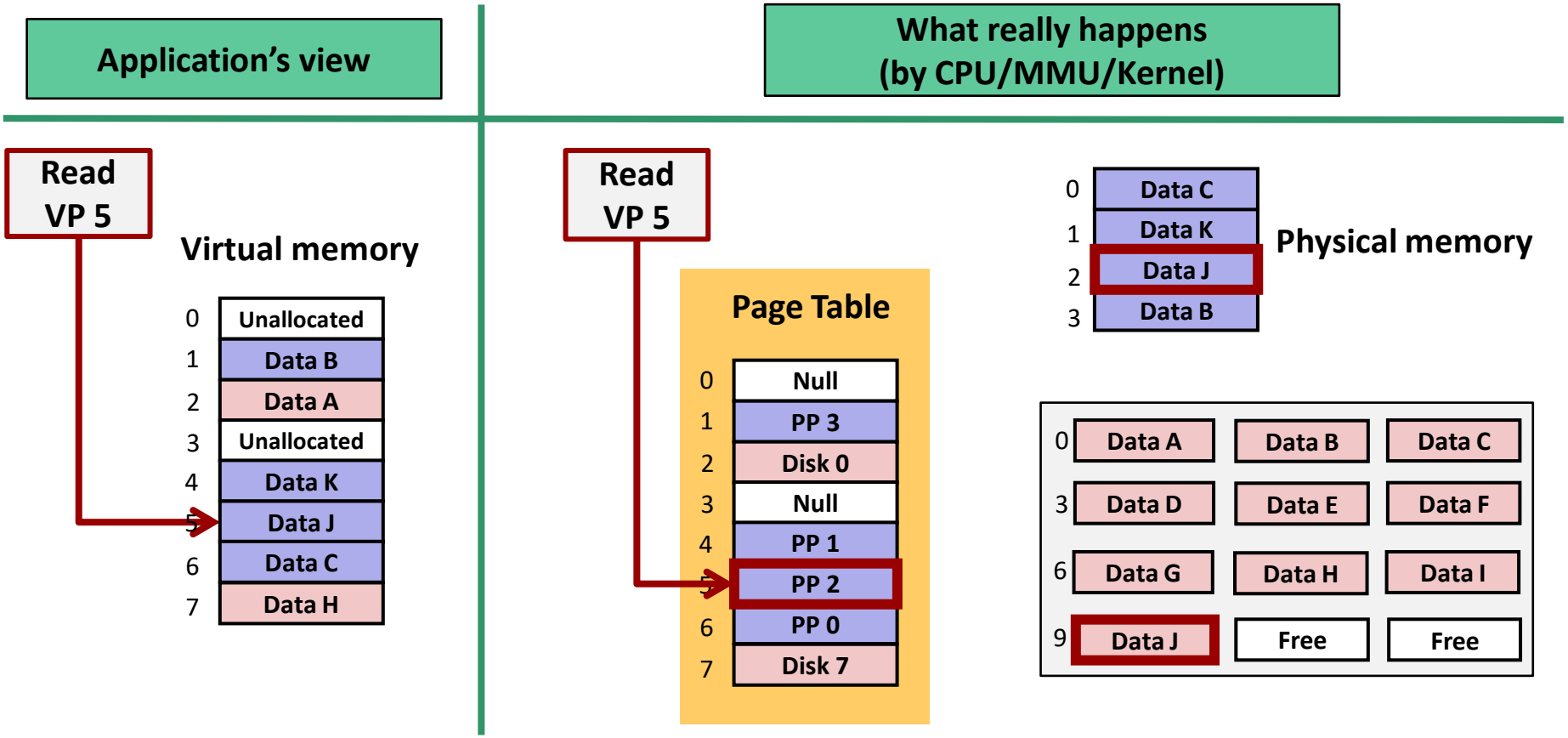
Handling Page Fault (3)

- Page miss (DRAM cache miss) causes page fault (an exception)
- **Page fault handler selects a victim to be evicted (here "Data H" at "VP 7" or "PP 2")**



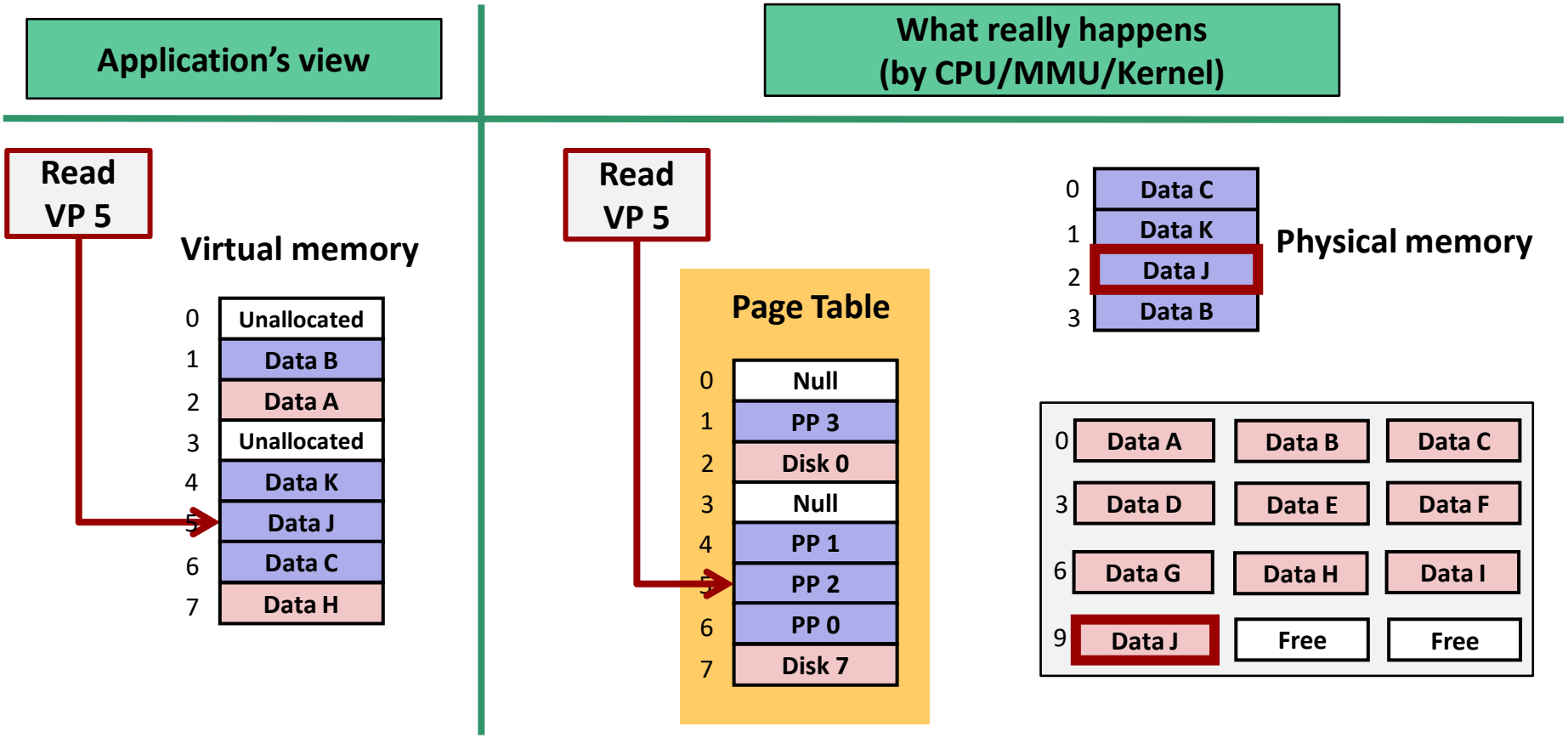
Handling Page Fault (4)

- Page miss (DRAM cache miss) causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here “Data H” at “VP 7” or “PP 2”)
- **Page fault handler loads the target page from disk (here “Data J” at “Disk 9”)**



Handling Page Fault (5)

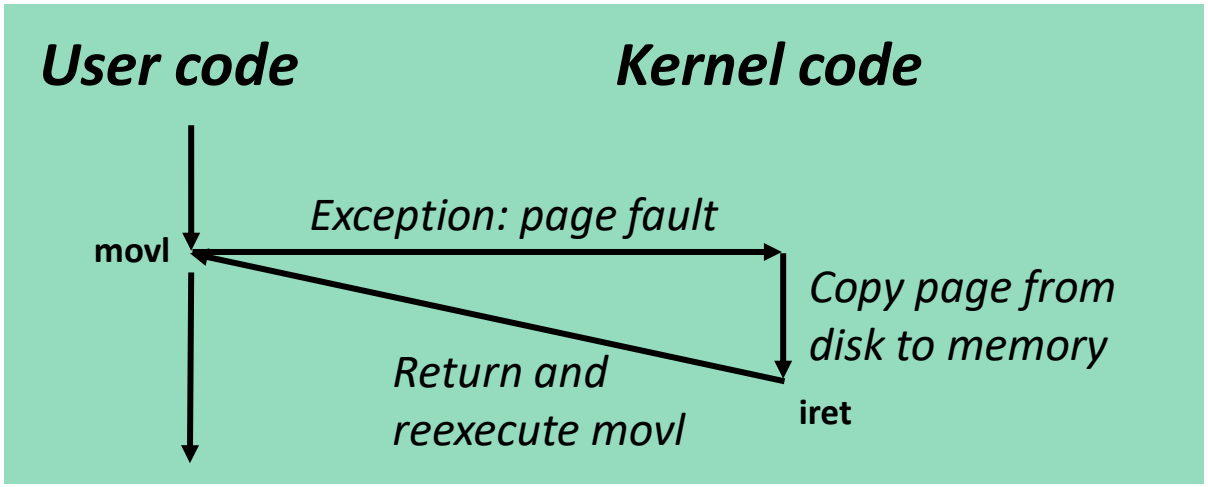
- Page miss (DRAM cache miss) causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here “Data H” at “VP 7” or “PP 2”)
- Page fault handler loads the target page from disk (here “Data J” at “Disk 9”)
- **Offending instruction is restarted: page hit!**



Completing page fault

- Page fault handler executes “return from interrupt” (**`iret`**; **`iretq`** on x86-64)
  - Like **`ret`** instruction, but also restores privilege level
  - Return to instruction that caused fault
  - But, this time there is no page fault

```
40113a:  c7 04 25 10 9d 40 00 0d 00 00 00  movl $0xd,0x409d10
```



Locality to the Rescue Again!

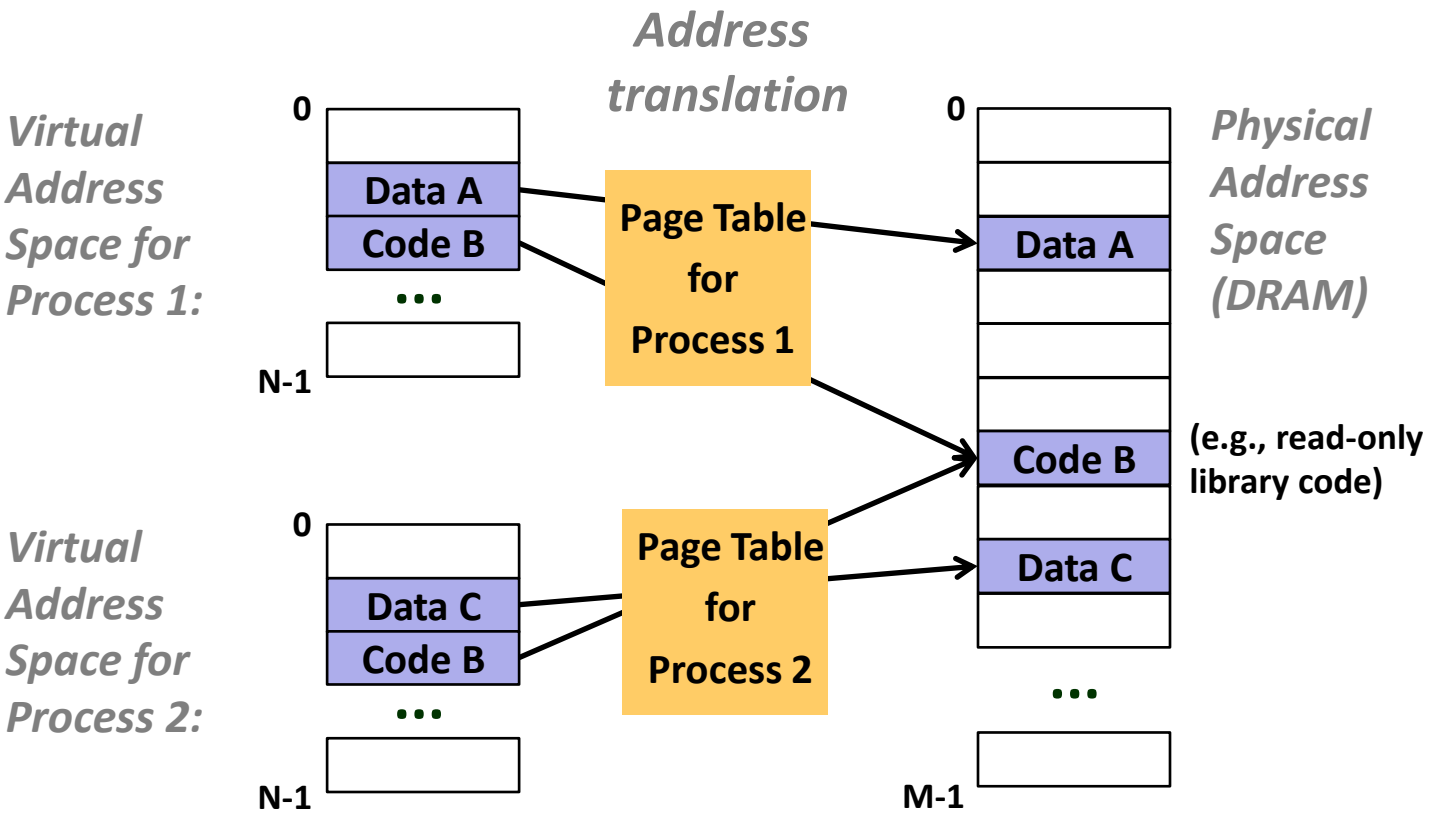
- **In practice, VM with paging surprisingly efficient**
  - because of **locality**.
- **At any point in time, programs tend to access a set of active virtual pages called the *working set***
  - Programs with better temporal locality will have smaller working sets
- **If (working set size < main memory size)**
  - Good performance for one process
- **If (working set size > main memory size)**
  - *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously
  - If multiple processes run at the same time, thrashing occurs if their total working set size > main memory size

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

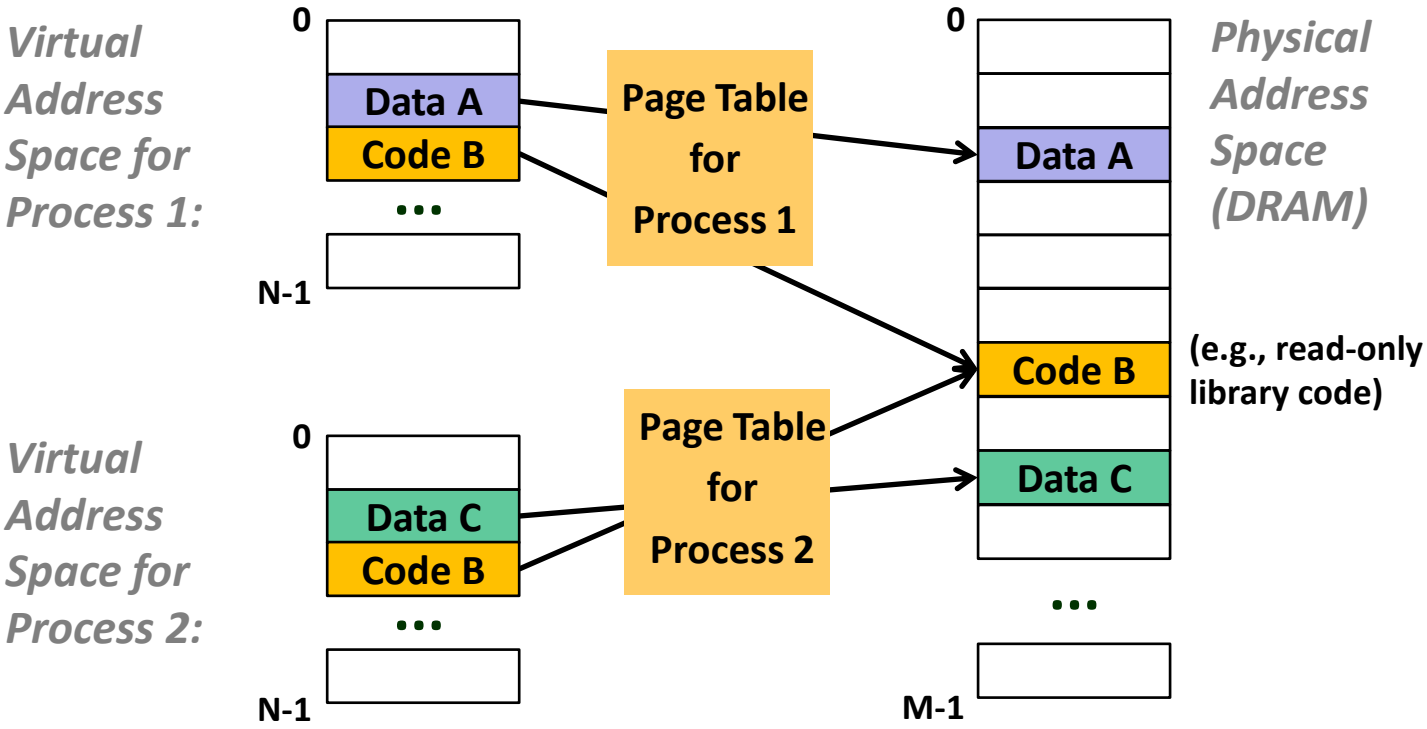
VM as a Tool for Memory Management

- **Key idea: each process has its own virtual address space**
  - It can view memory as a simple linear array
  - Every process can use the same addresses (e.g., code at 0x400000)



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 migrated to a different physical page
- **Sharing code and data among processes**
  - Map multiple virtual pages to the same physical page (here: Code B)

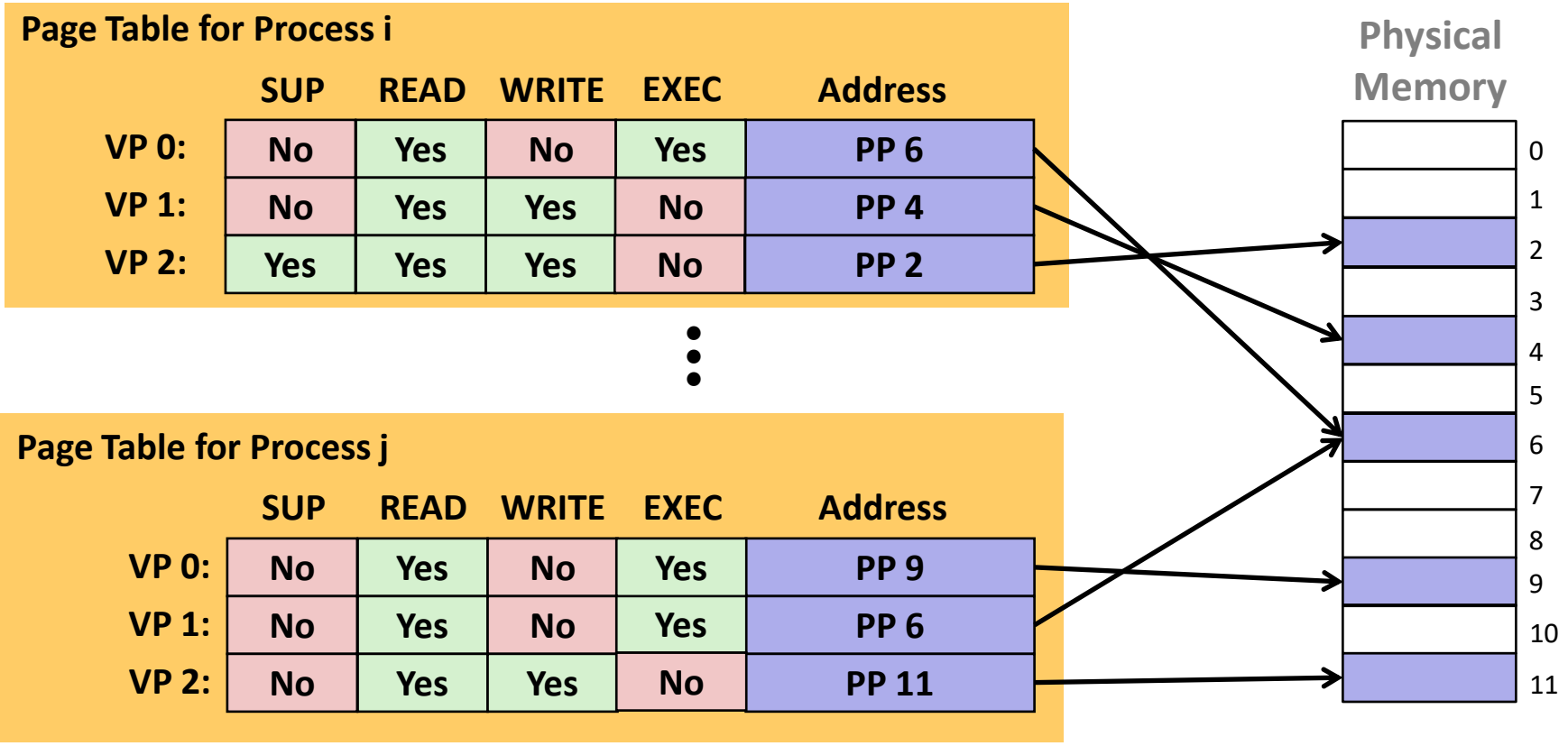


Today

- 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**
  - Violation → page fault → kernel sends SIGSEGV (segmentation fault)



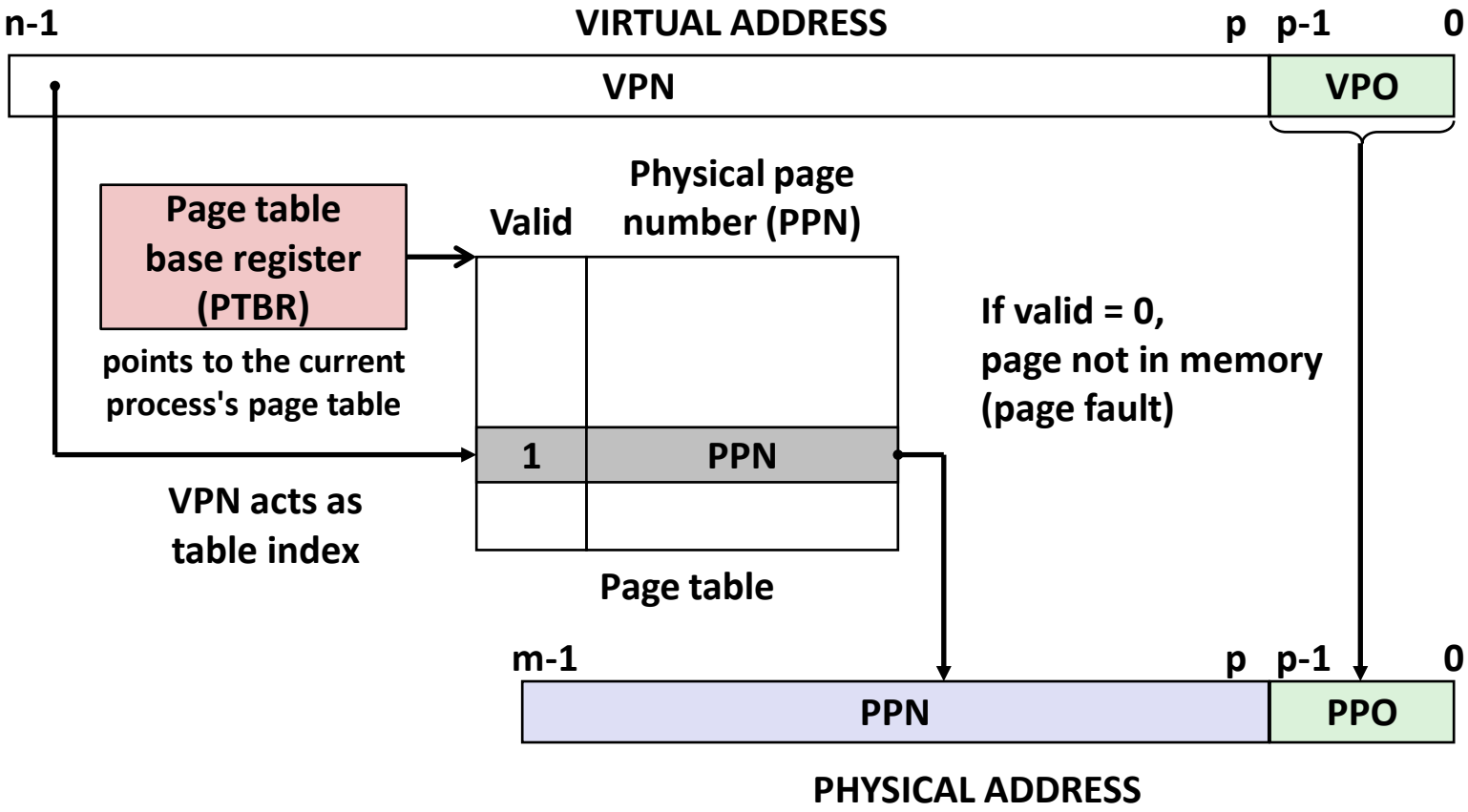
W^X: a page is never both writable and executable (NX bit)

SUP: requires kernel mode

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**

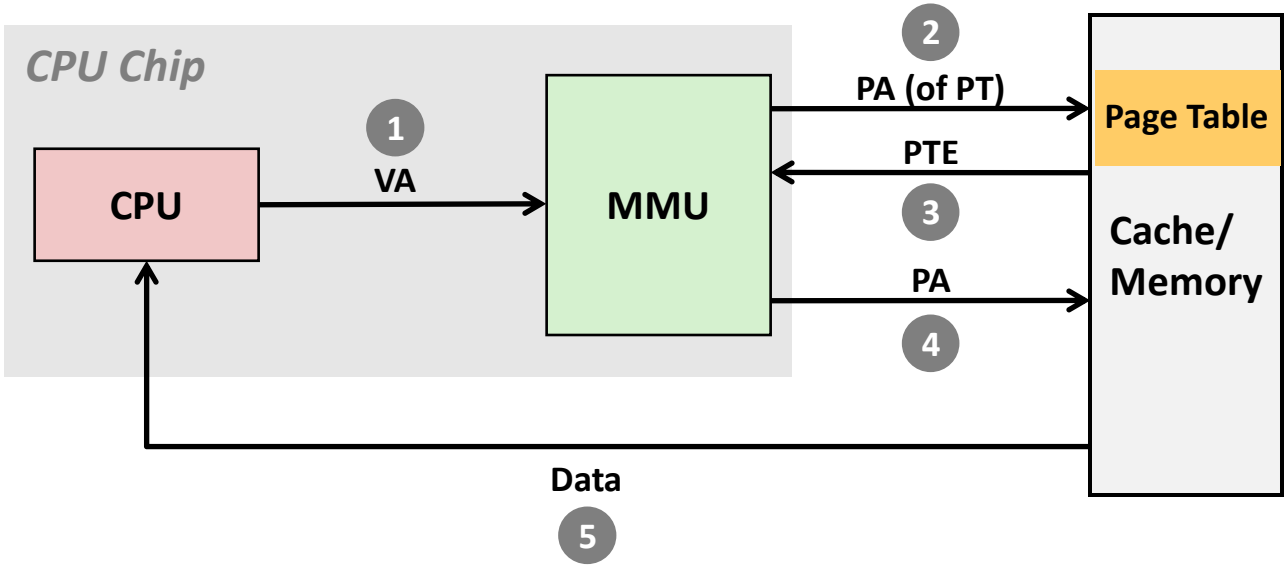
Address Translation With a Page Table



Components of the virtual address (VA)
- 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: Page Hit



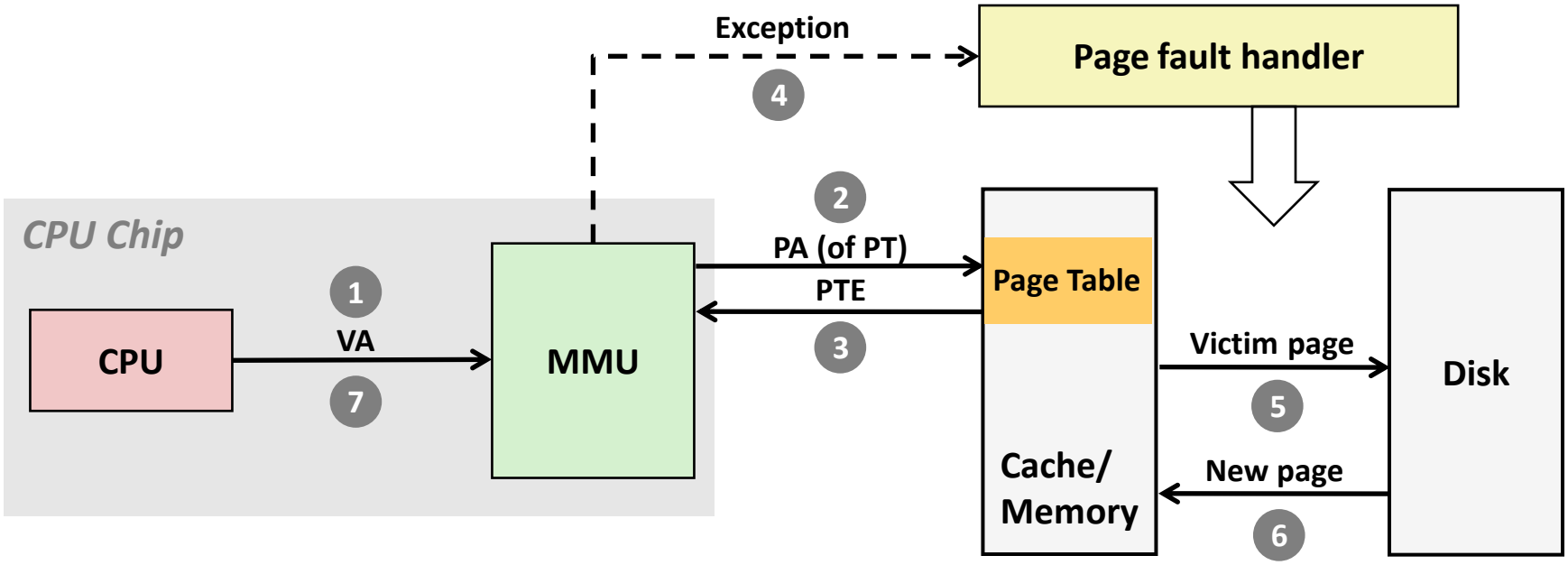
1) Processor sends virtual address to MMU

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 (modified), pages it out to disk)

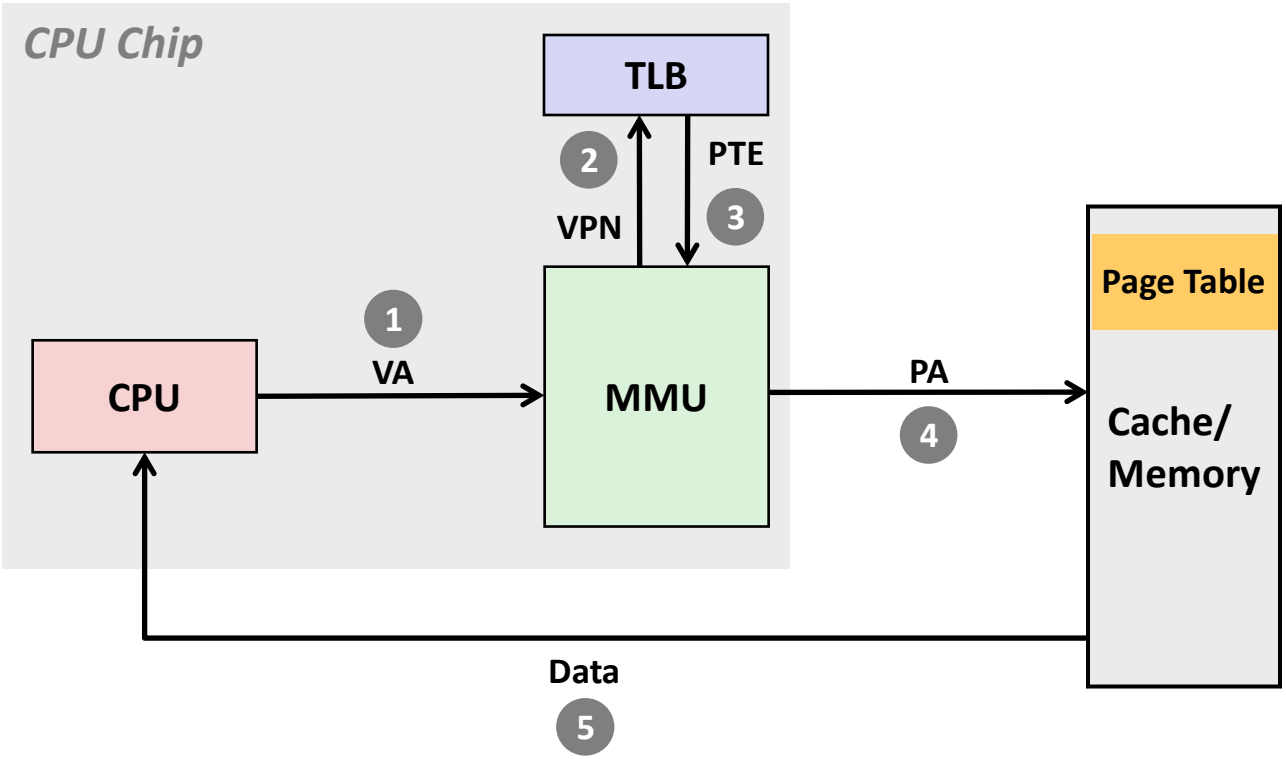
6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

Speeding up Translation with a TLB

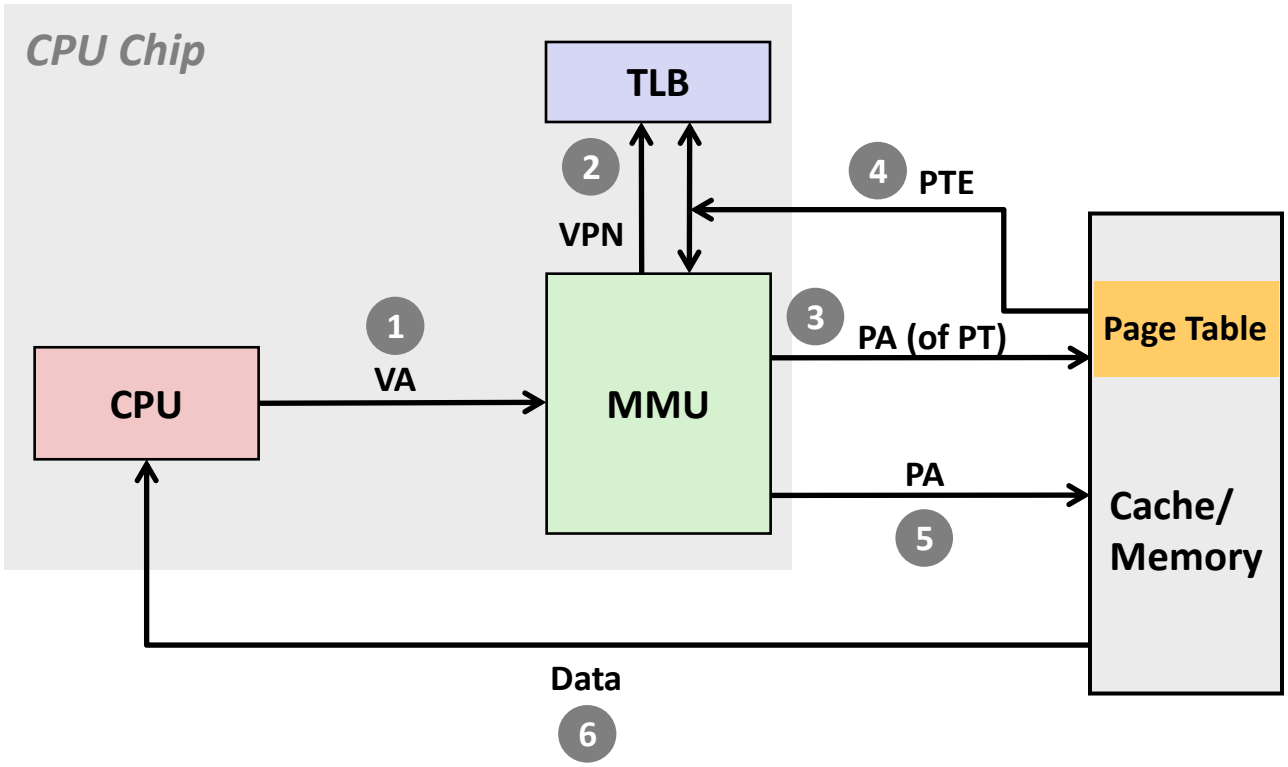
- **Address translation can be slow**
  - Page table entries (PTEs) are stored in memory
  - Every memory access through VA would result in two memory accesses
    - #1. To translate VA to PA, accessing PTE
    - #2. Fetch the data using PA
  - Two memory accesses? Can we make it faster?
- **Idea: *Translation Lookaside Buffer* (TLB)**
  - Small set-associative hardware cache in MMU
  - Cache a subset of page table entries
    - virtual page numbers to physical page numbers

TLB Hit



A TLB hit eliminates a slow memory/cache access

TLB Miss



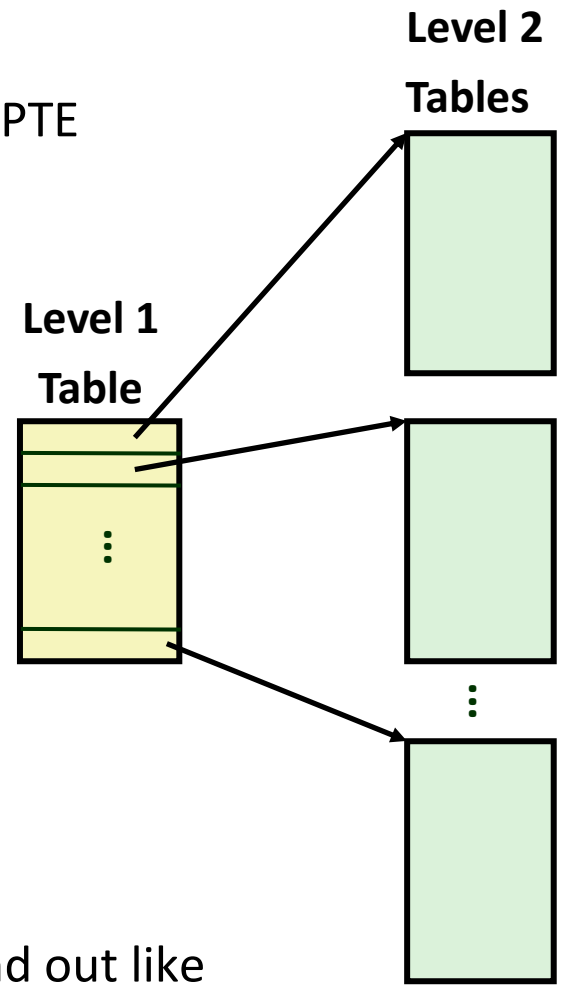
A TLB miss incurs an additional cache/memory access (the PTE)

Fortunately, TLB misses are rare. Why?

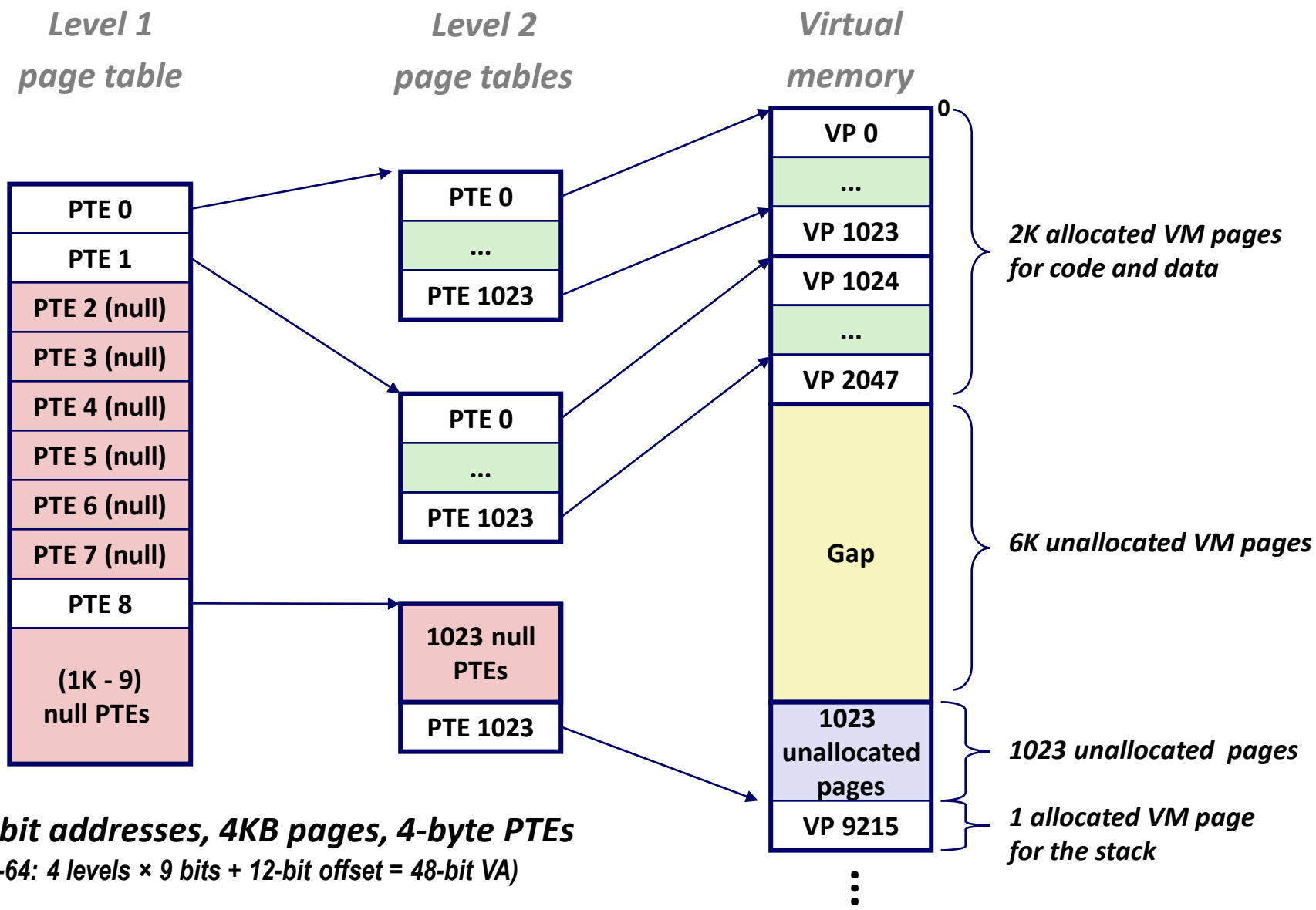
Multi-Level Page Tables

- **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**
  - L1 table (always memory resident):

    each PTE points to an L2 page table
  - L2 tables: each PTE points to a page (paged in and out like any other data)



A Two-Level Page Table Hierarchy



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

(x86-64: 4 levels × 9 bits + 12-bit offset = 48-bit VA)

Translating with a k-level Page Table

Components of the virtual address (VA)

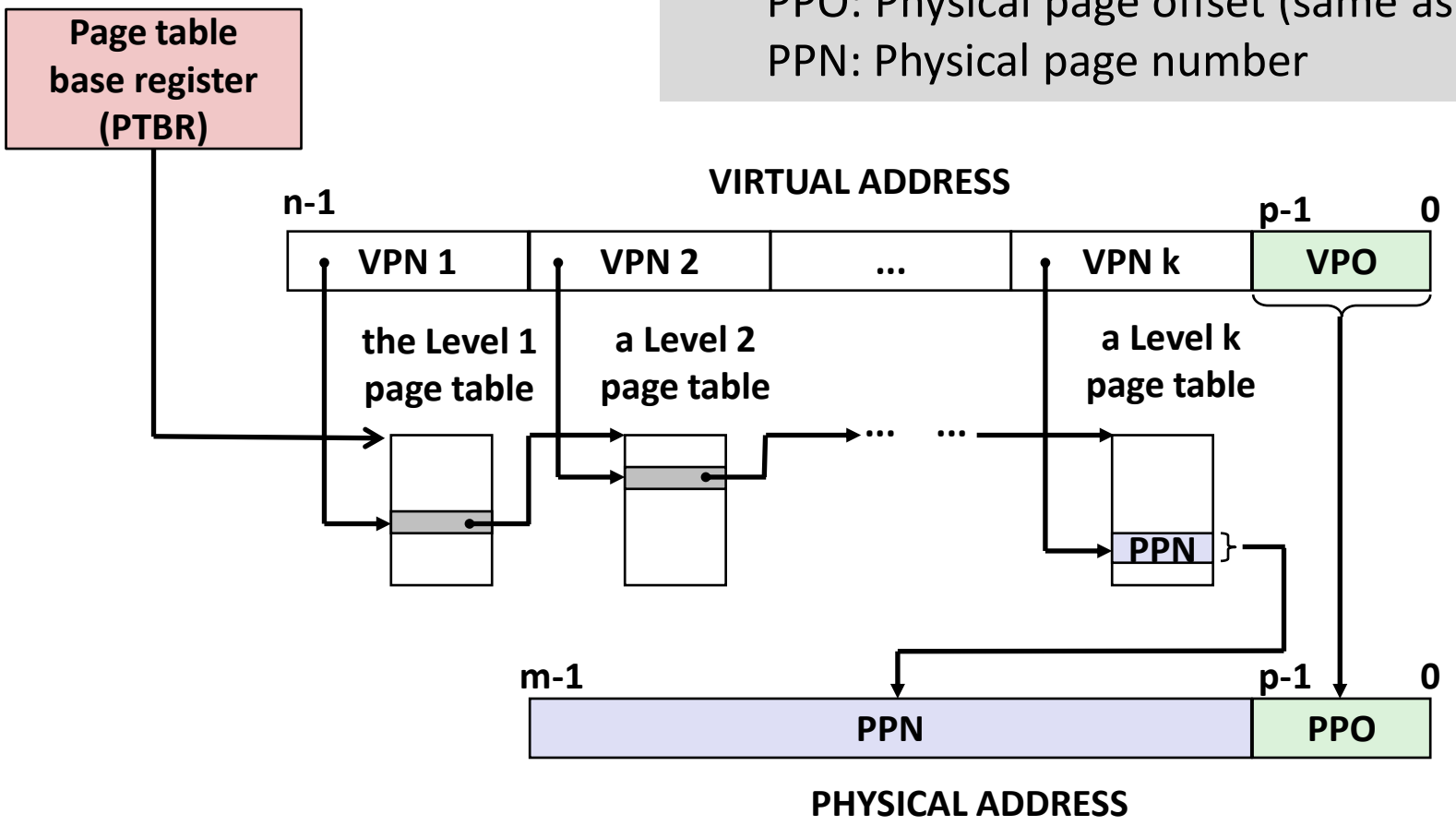
VPO: Virtual page offset

VPN: Virtual page number

Components of the physical address (PA)

PPO: Physical page offset (same as VPO)

PPN: Physical page number



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 because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions
- **Implemented via combination of hardware & software**
  - MMU, TLB, exception handling mechanisms part of hardware
  - Page fault handlers, page tables, TLB flushes managed in software