

Virtual Memory: Concepts

Introduction to Computer Systems
19th Lecture, Dec. 7, 2020

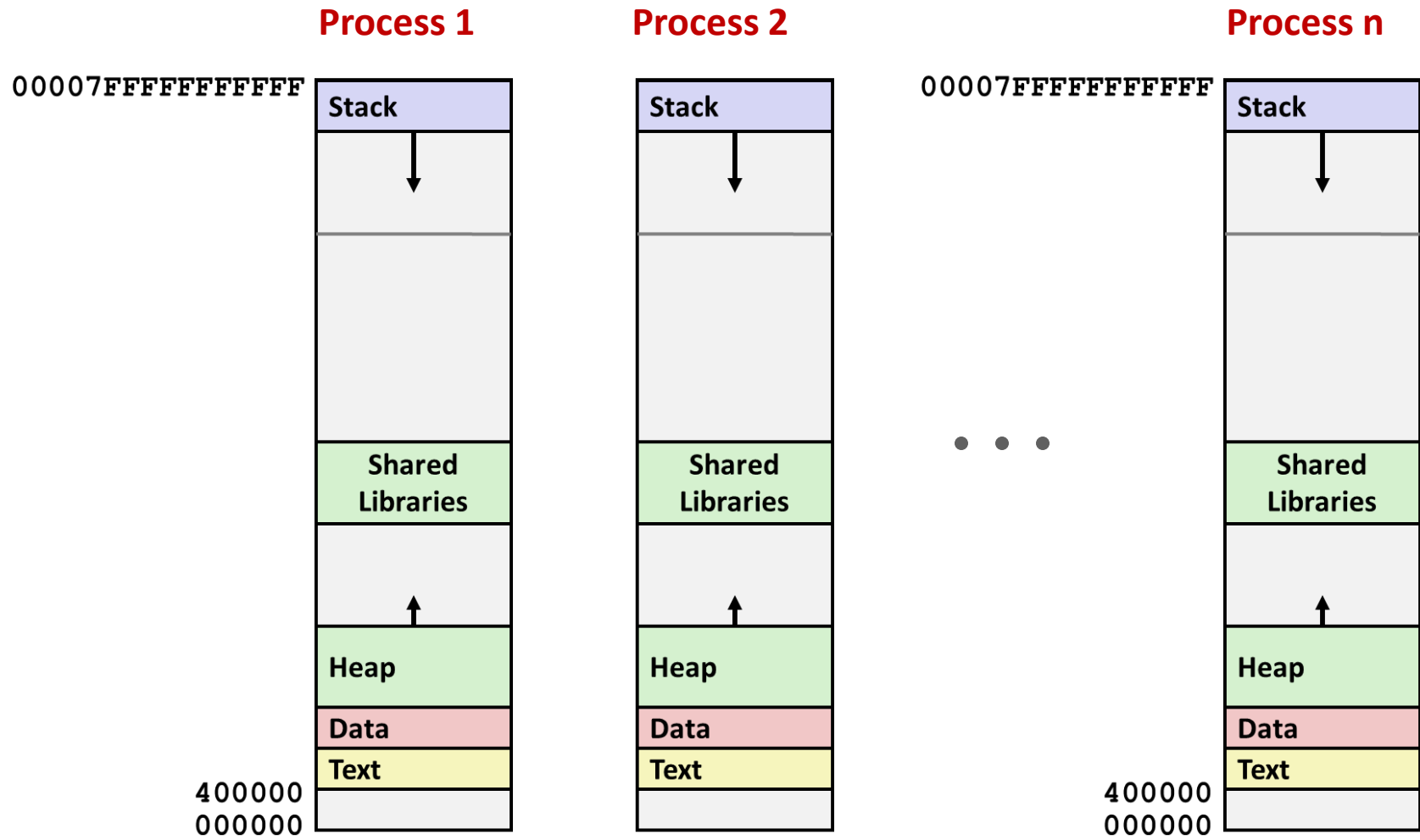
Instructors:

Class 1: Chen Xiangqun, Sun Guangyu, Liu Xianhua

Class 2: Guan Xuetao

Class 3: Lu Junlin

Hmmm, How Does This Work?!

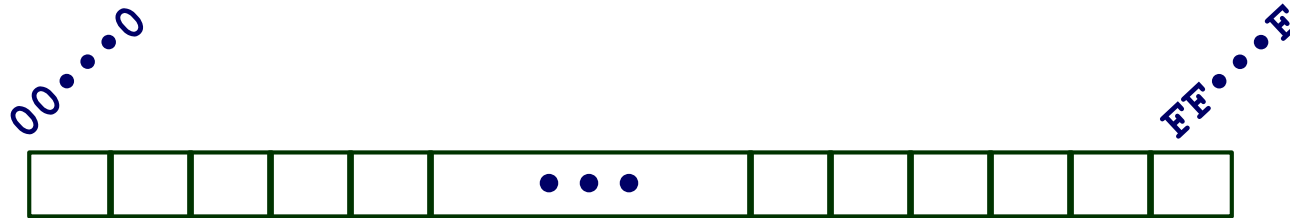


Solution: Virtual Memory (today and next lecture)

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

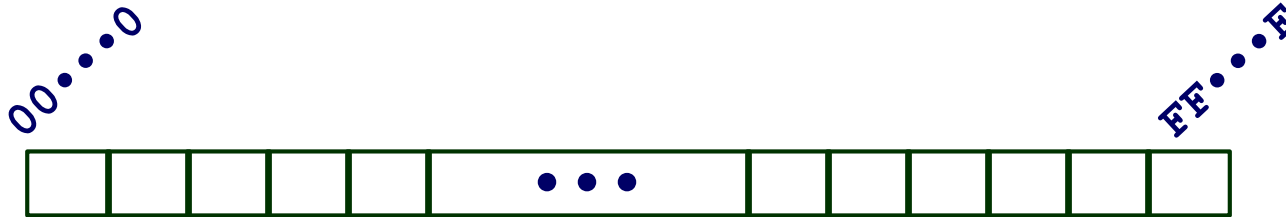
```
movl (%ecx), %eax
```

■ 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

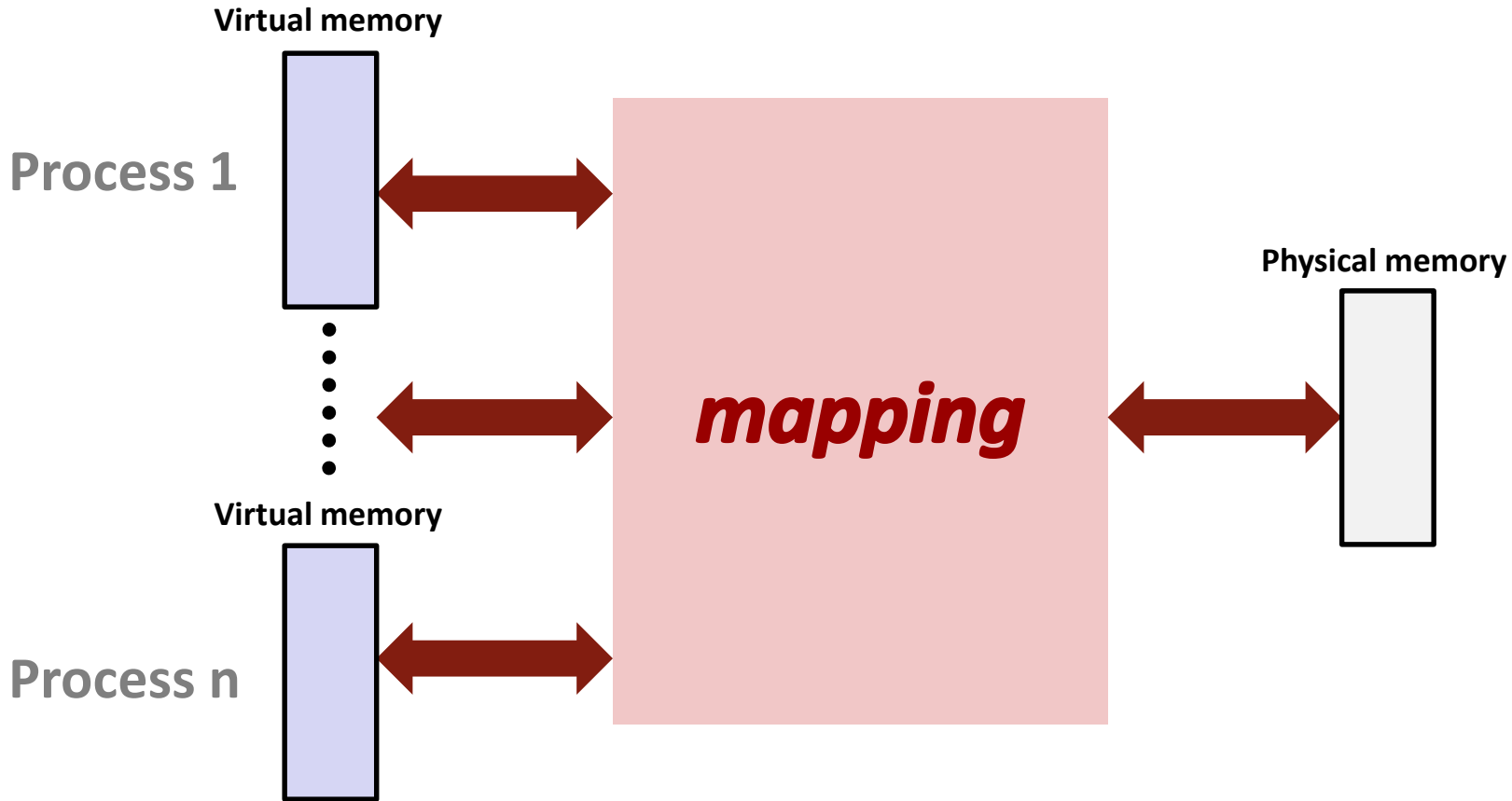
■ How to decide which memory to use in your program?

- 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

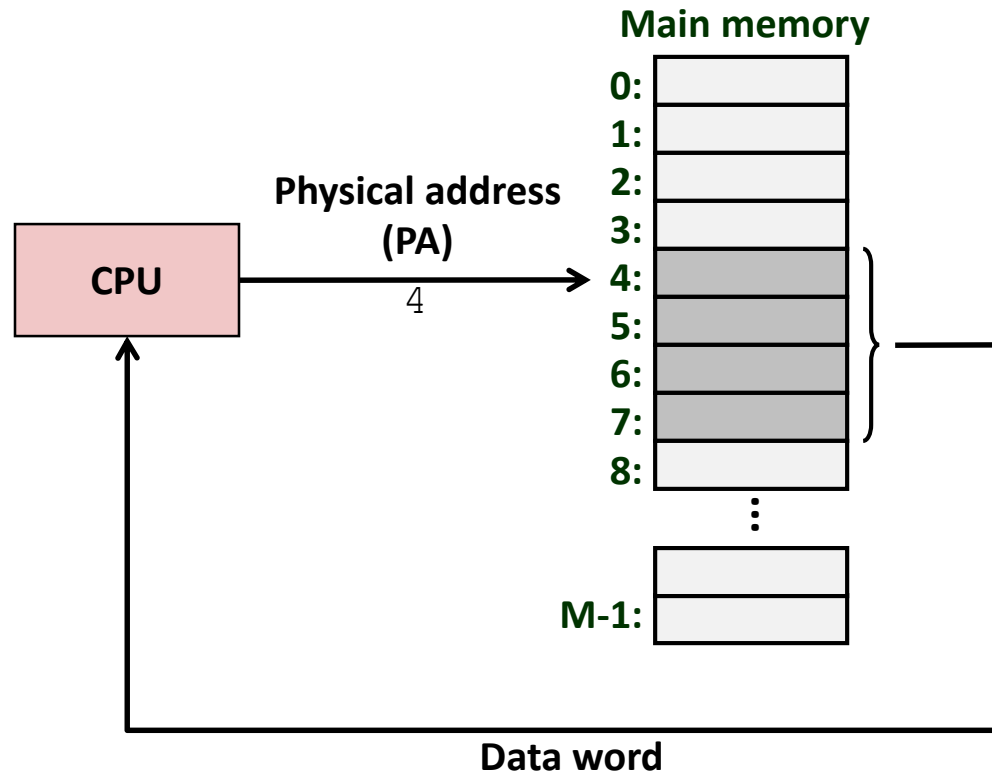


- Each process gets its own private memory space
- Addresses all of the previous problems

One simple trick solves all three problems

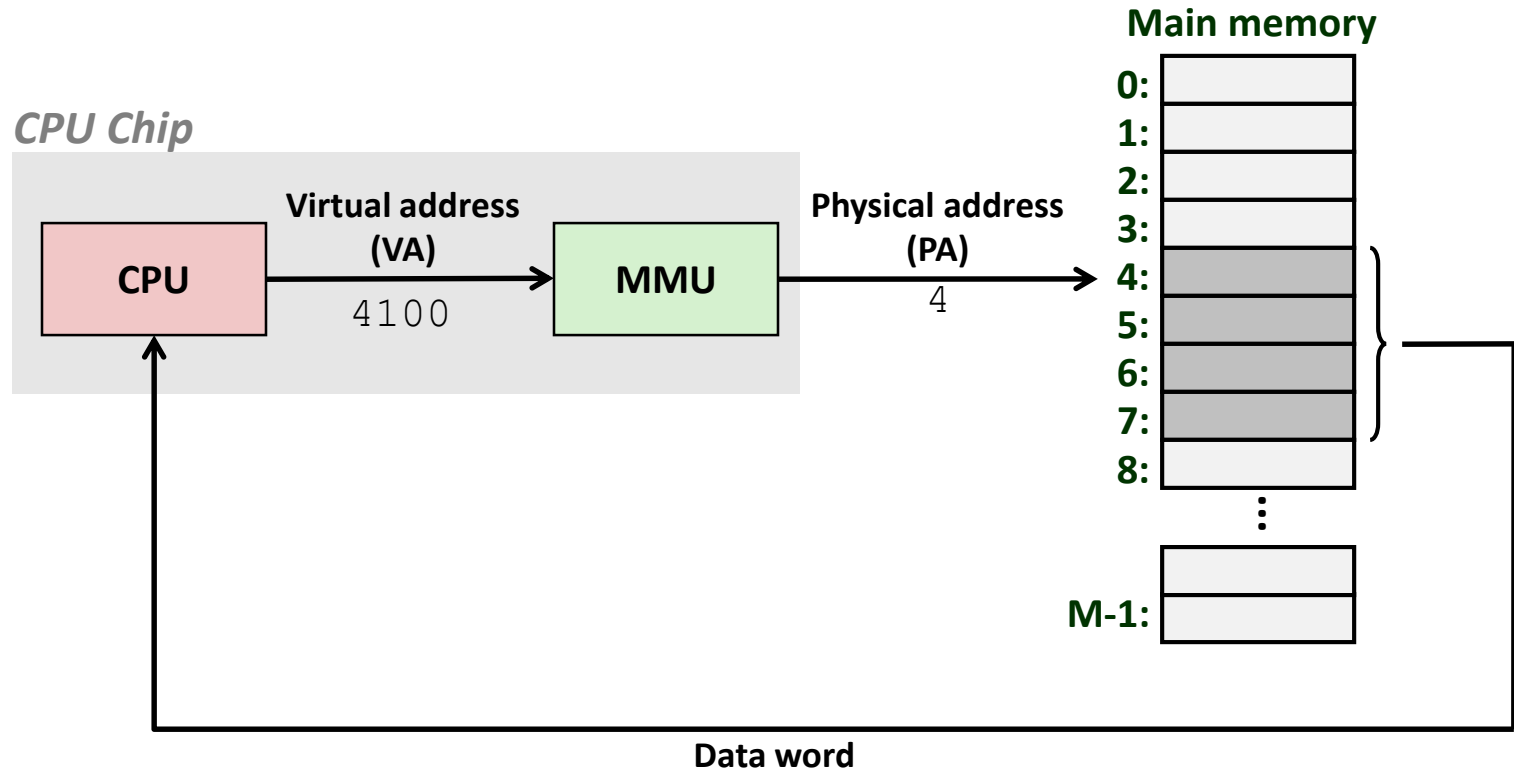
- One simple trick solves all three problems
- Each process gets its own private image of memory
 - appears to be a full-sized private memory range
- This fixes “how to choose” and “others shouldn’t mess w/yours”
 - surprisingly, it also fixes “making everything fit”
- **Implementation: translate addresses 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*

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



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

- 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

Why Virtual Memory (VM)?

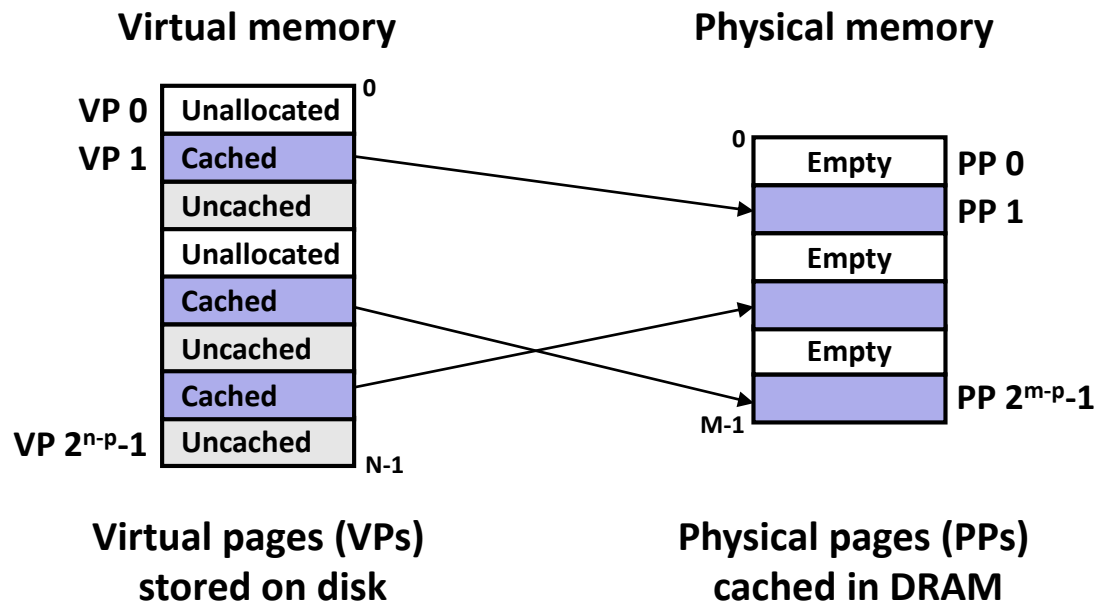
- **Uses main memory efficiently**
 - Use DRAM as a cache for parts of a virtual address space
- **Simplifies memory management**
 - Each process gets the same uniform linear address space
- **Isolates address spaces**
 - One process can't interfere with another's memory
 - User program cannot access 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

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

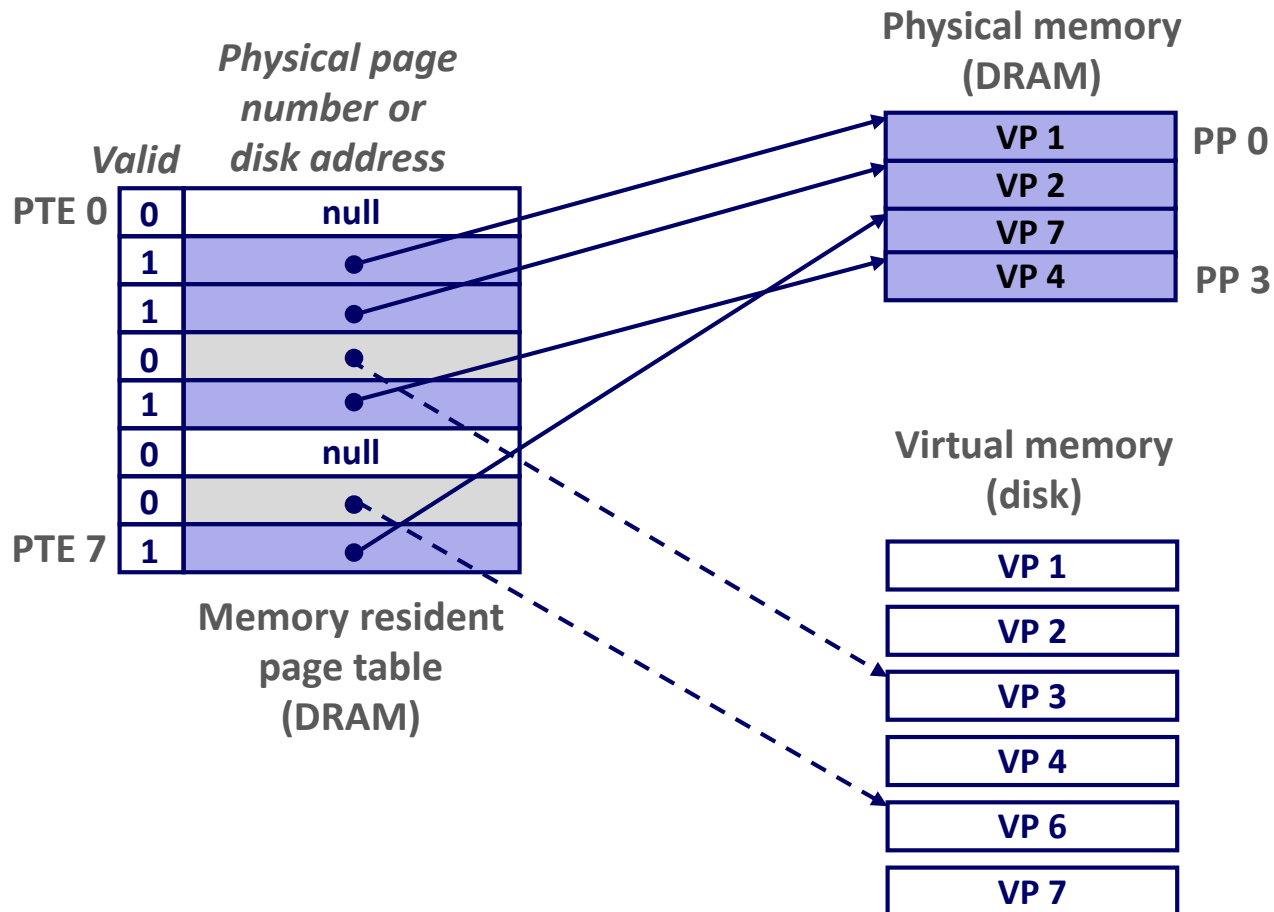
- DRAM is about **10x** slower than SRAM
- Disk is about **10,000x** slower than DRAM
- Time to load block from disk > 1ms (> 1 million clock cycles)
 - CPU can do a lot of computation during that time

■ Consequences

- Large page (block) size: typically 4 KB
 - Linux “huge pages” are 2 MB (default) to 1 GB
- 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

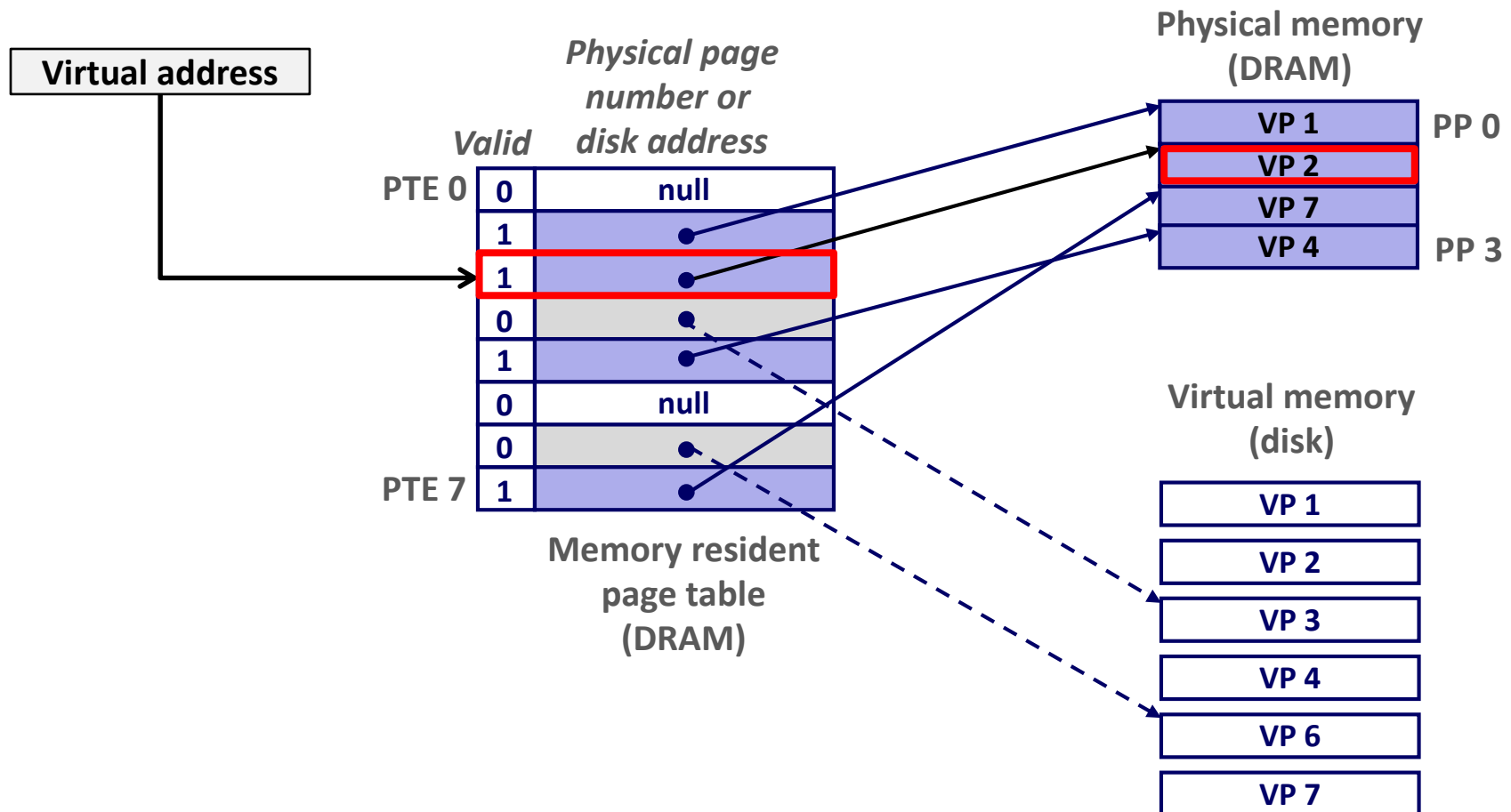
Enabling Data Structure: Page Table

- 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



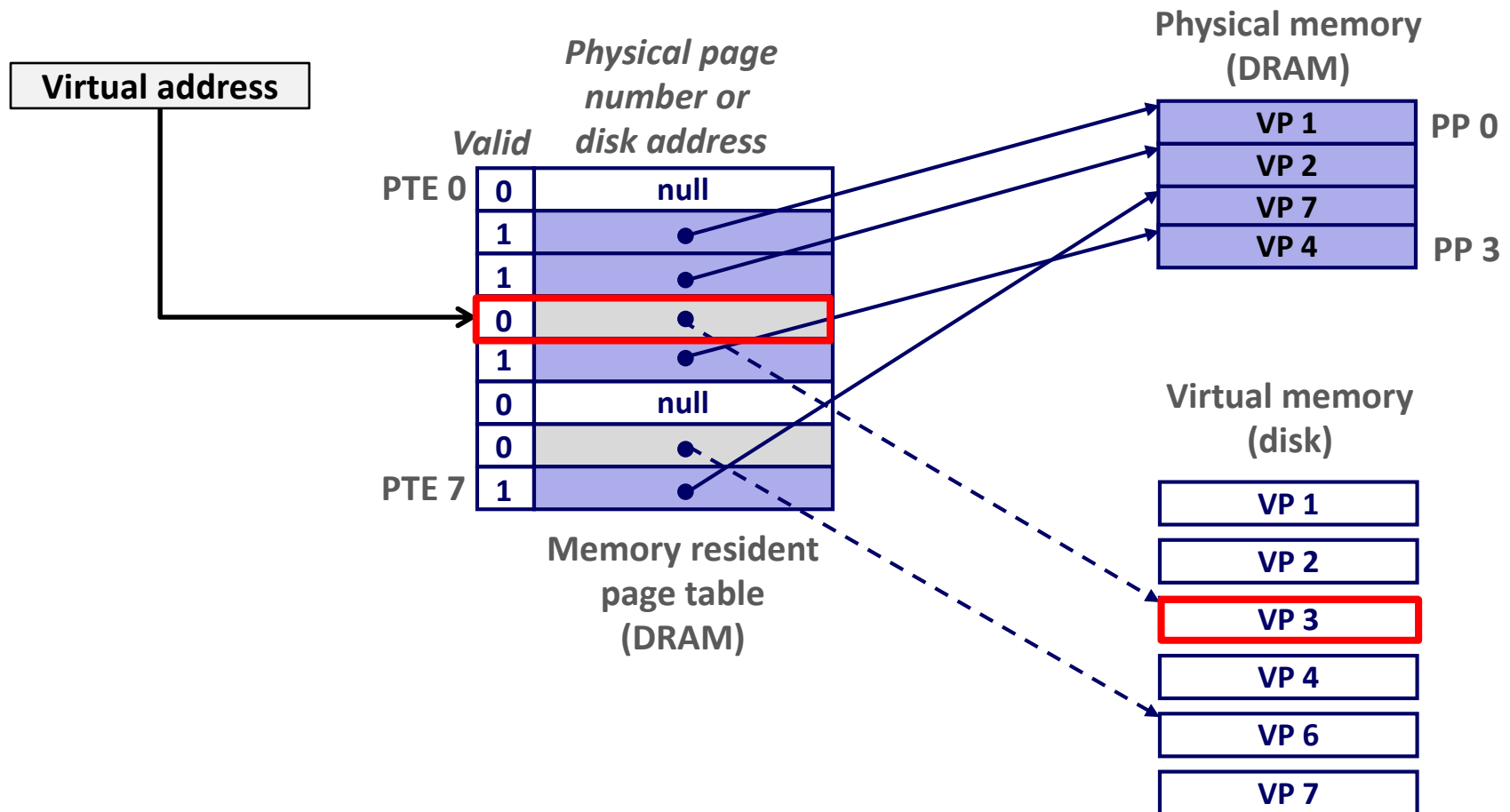
Page Hit

- **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)



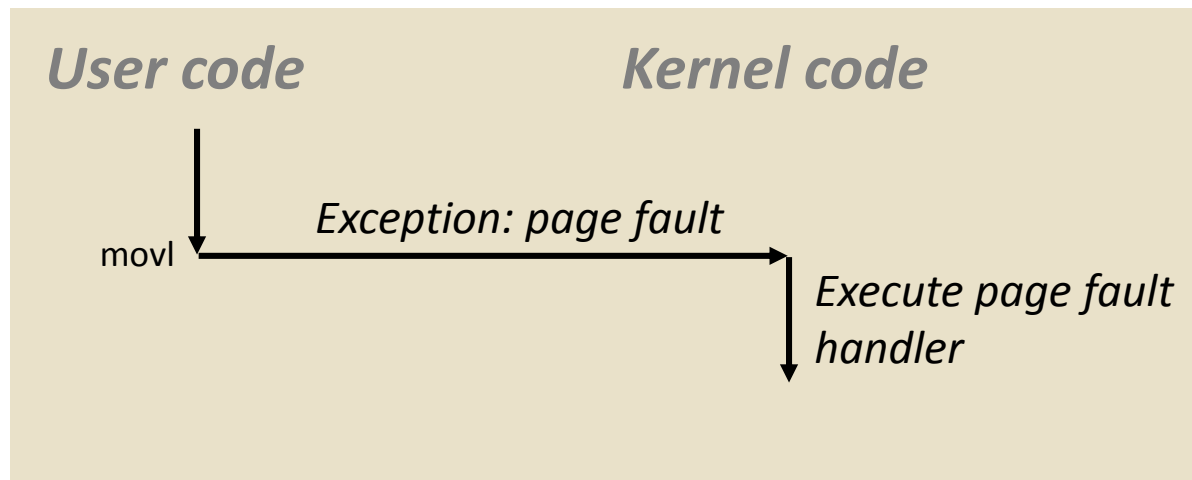
Triggering a Page Fault

- User writes to memory location

80483b7:	c7 05 10 9d 04 08 0d	movl	\$0xd,0x8049d10
----------	----------------------	------	-----------------

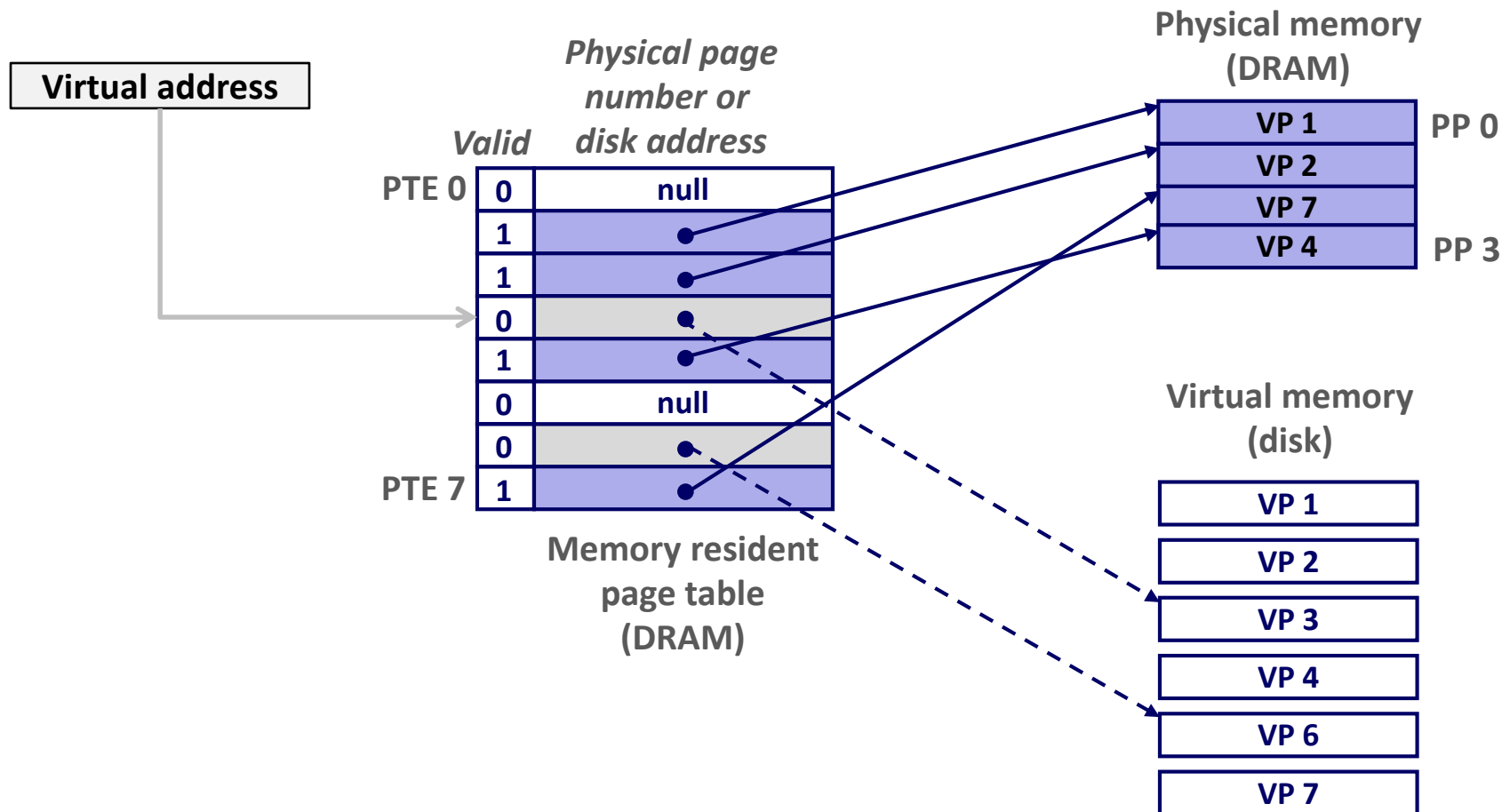
- That portion (page) of user's memory is currently on disk
- MMU triggers page fault exception
 - (More details in later lecture)
 - Raise privilege level to supervisor mode
 - Causes procedure call to software page fault handler

```
int a[1000];
main ()
{
    a[500] = 13;
}
```



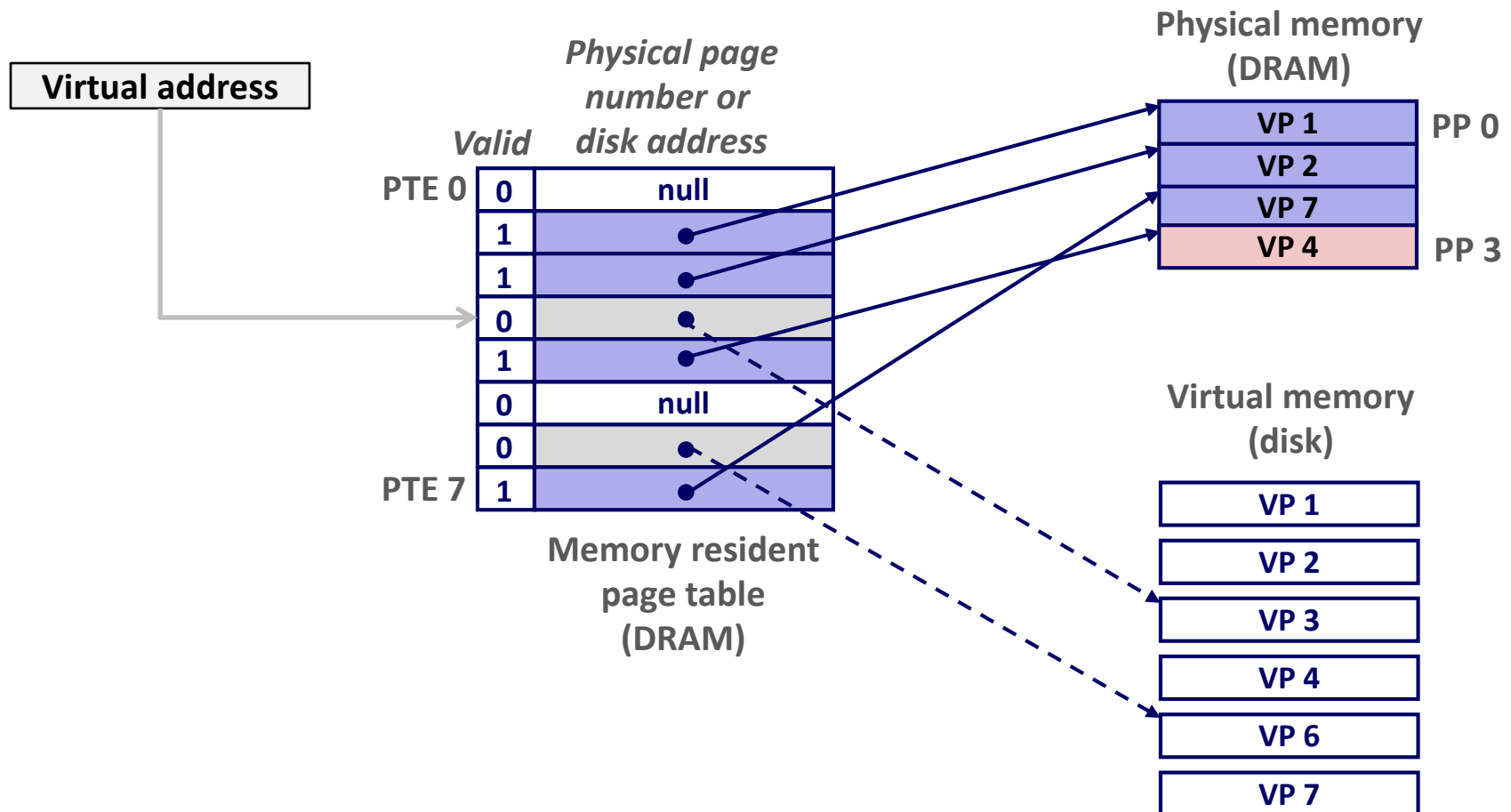
Handling Page Fault

- Page miss causes page fault (an exception)



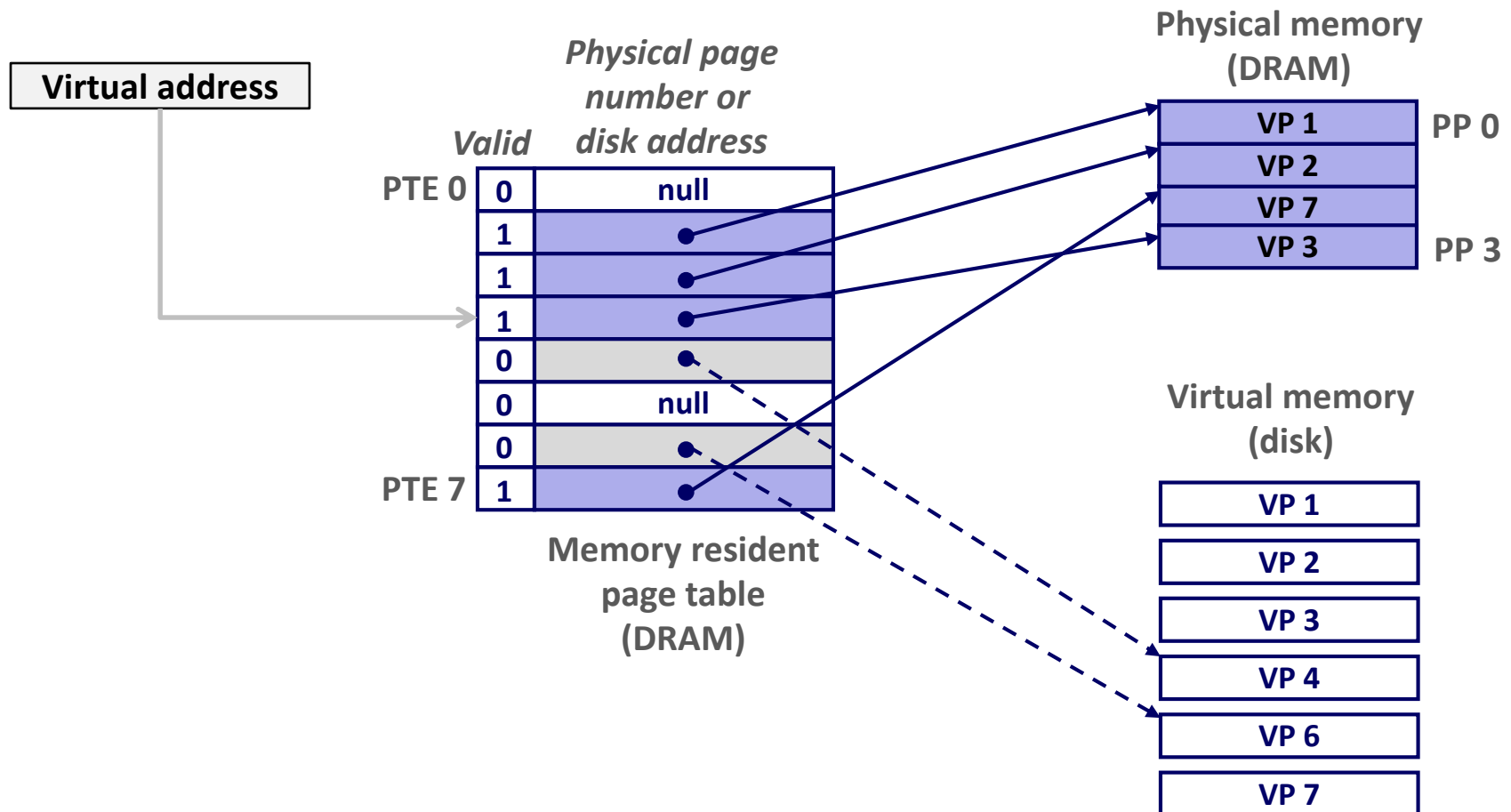
Handling Page Fault

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



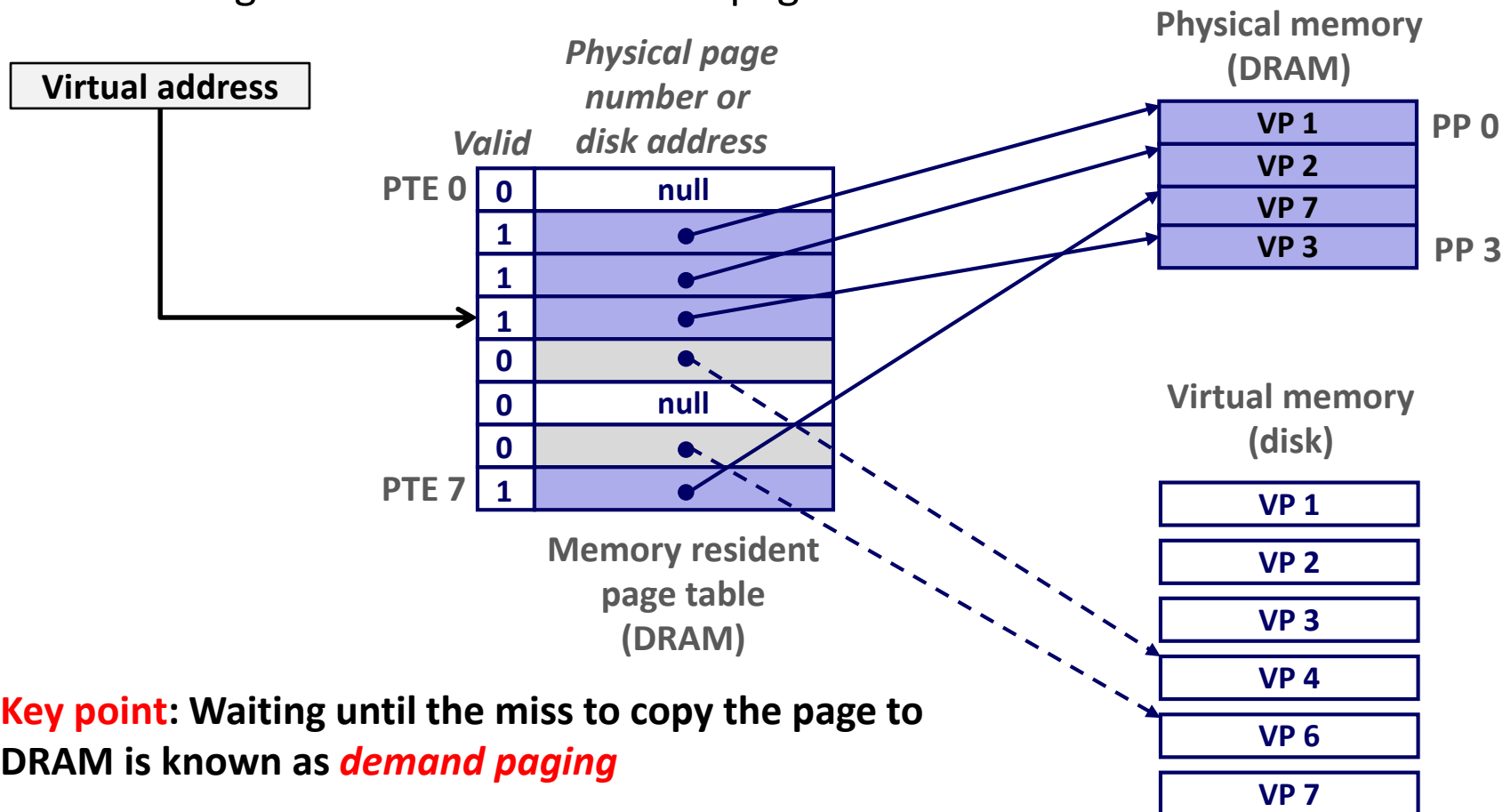
Handling Page Fault

- 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!

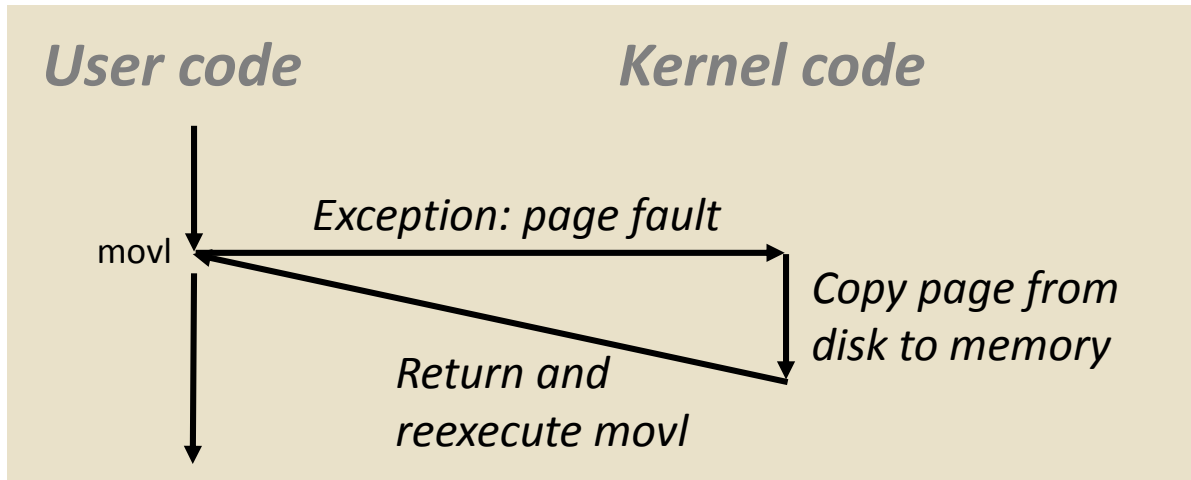


Completing page fault

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

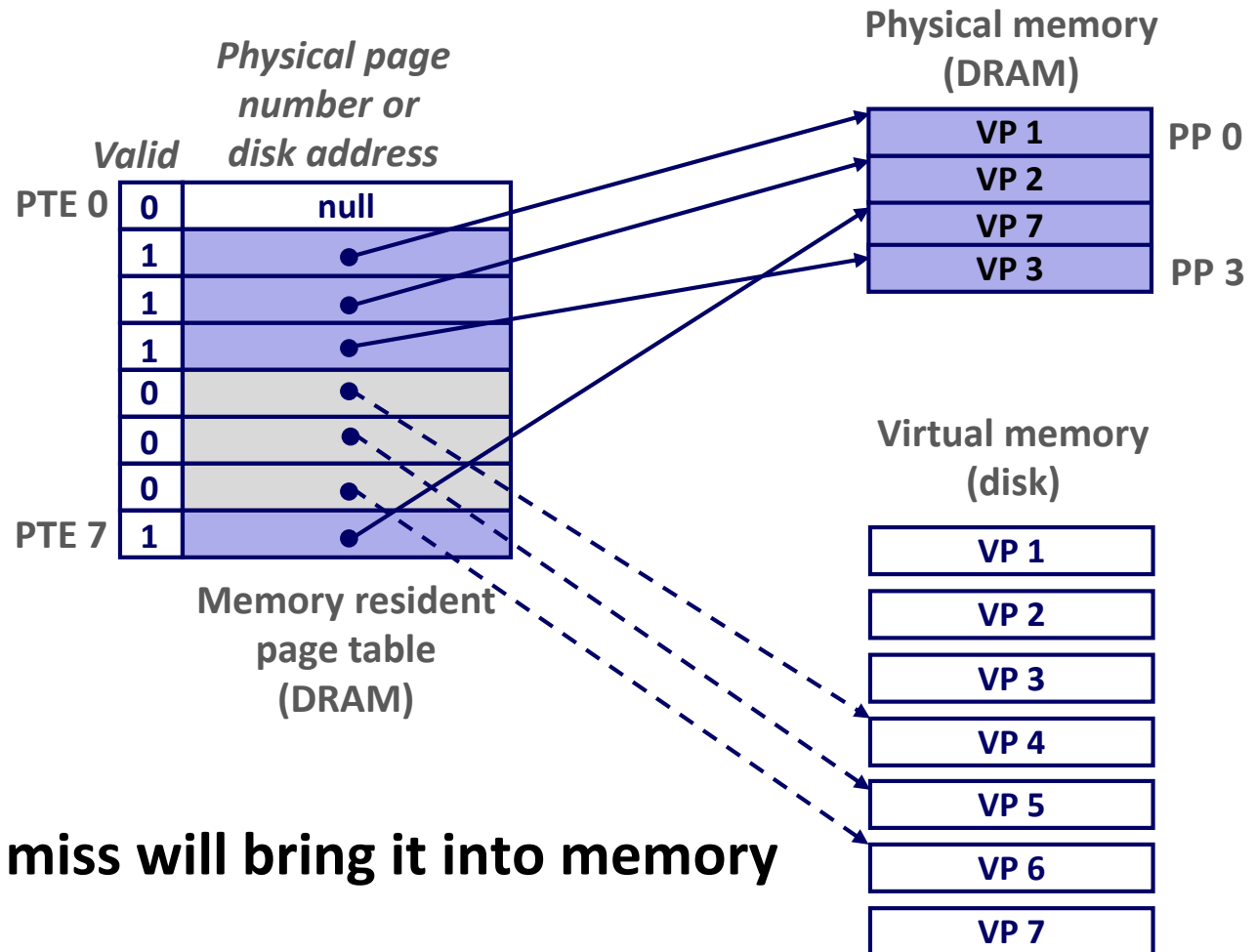
```
int a[1000];
main ()
{
    a[500] = 13;
}
```

80483b7: c7 05 10 9d 04 08 0d movl \$0xd,0x8049d10



Allocating Pages

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



- Subsequent miss will bring it into memory

Locality to the Rescue Again!

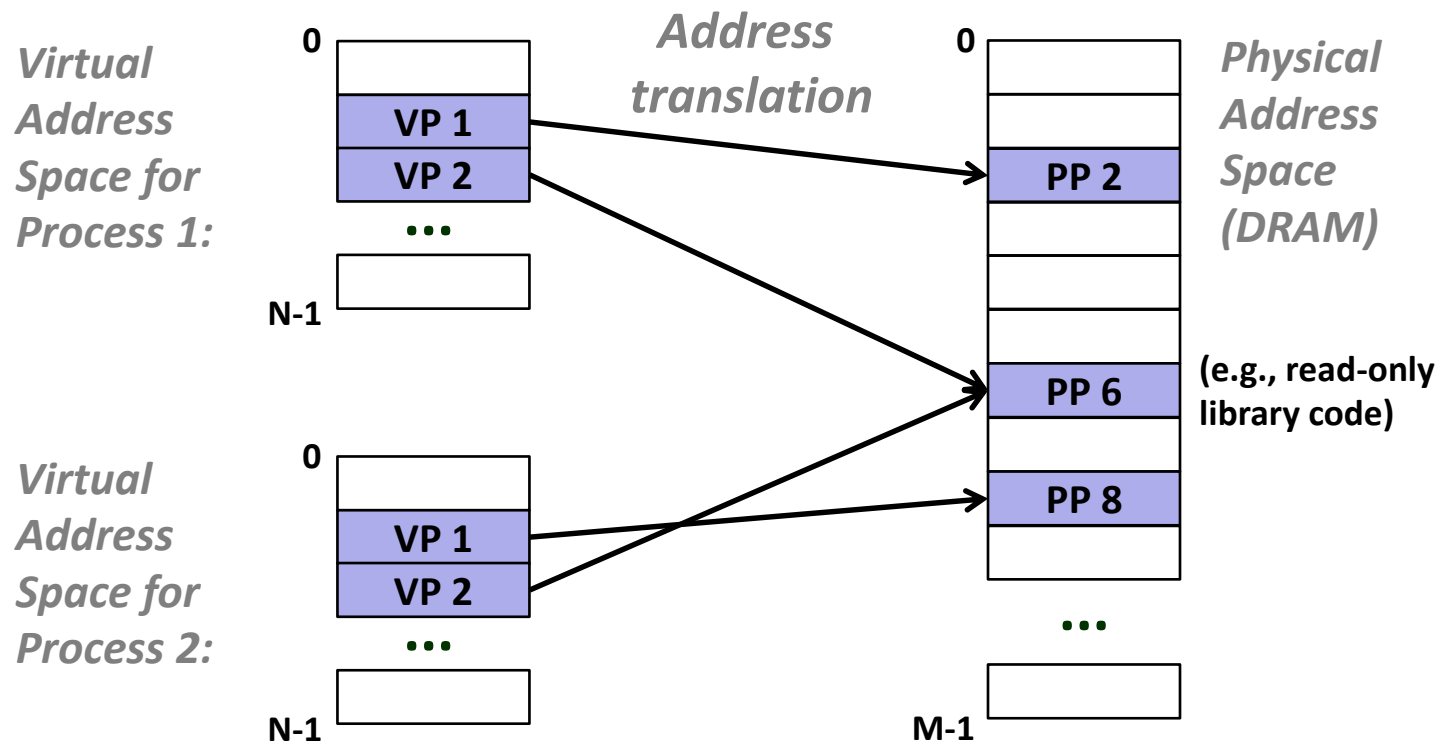
- Virtual memory seems terribly inefficient, but it works because of locality.
- At any point in time, programs tend to access a set of active virtual pages called the *working set*
 - Programs with better temporal locality will have smaller working sets
- If (working set size < main memory size)
 - Good performance for one process (after cold misses)
- 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
 - Mapping function scatters addresses through physical memory
 - Well-chosen mappings can improve locality



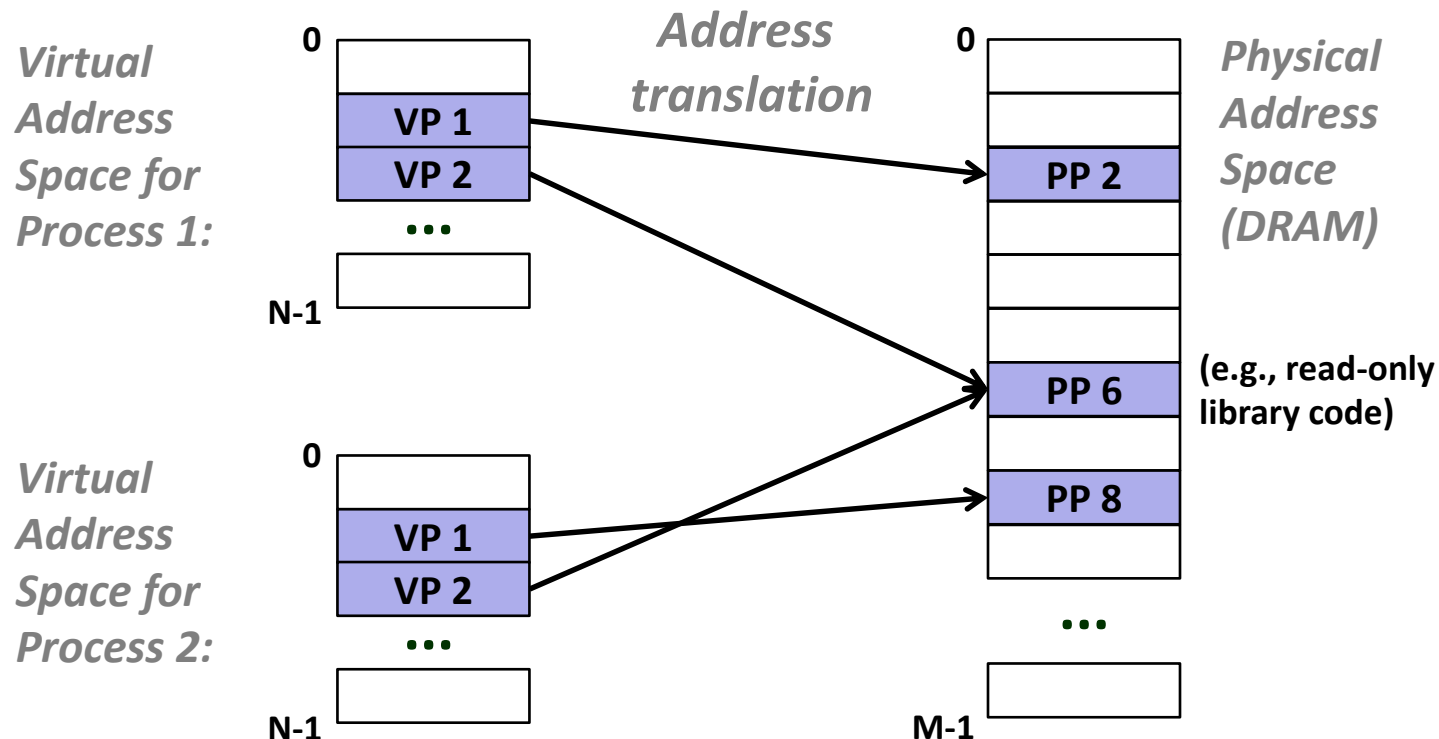
VM as a Tool for Memory Management

■ 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)



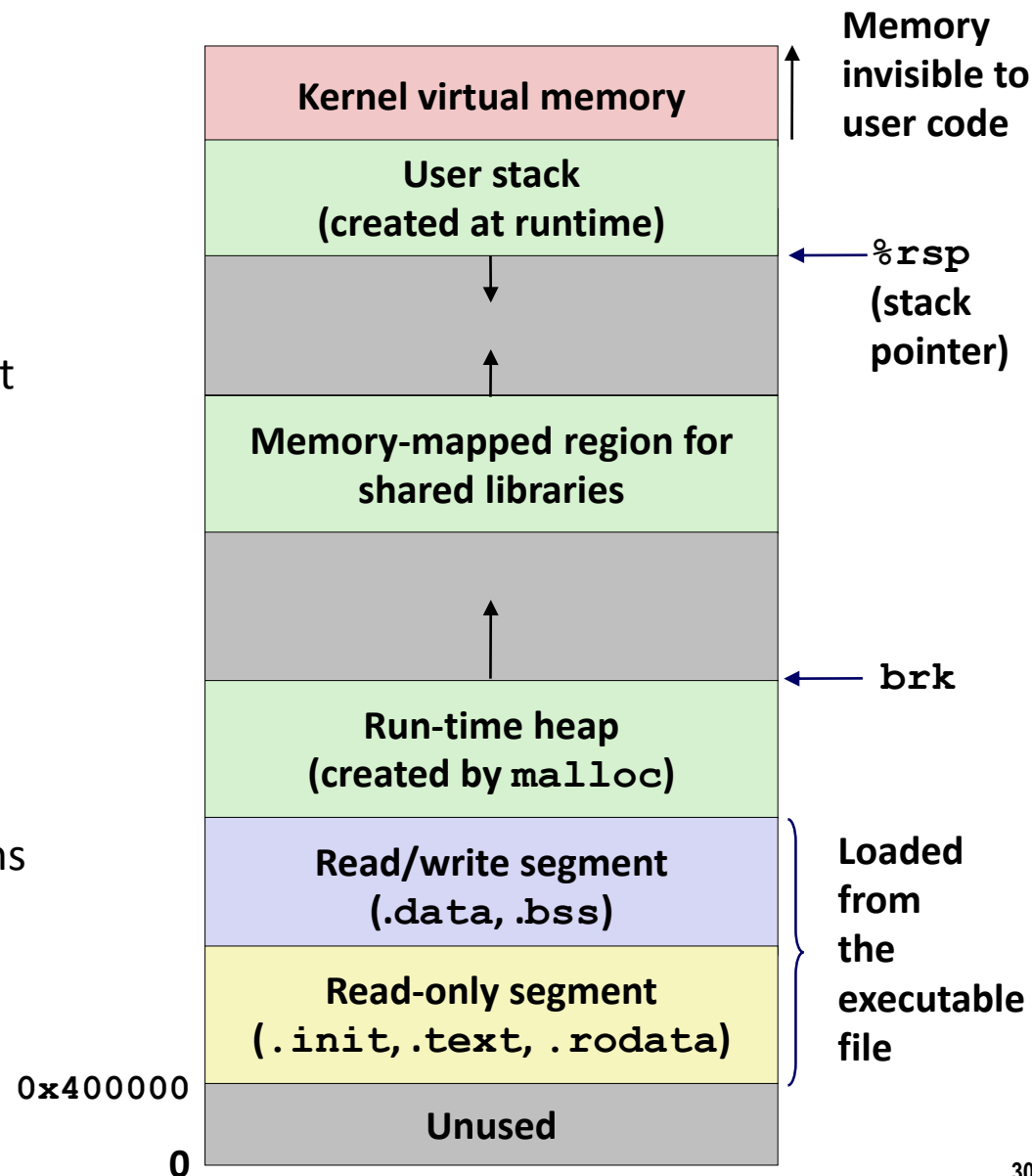
Simplifying Linking and Loading

■ Linking

- Each program has similar virtual address space
- Code, data, and heap always start at the same addresses.

■ 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

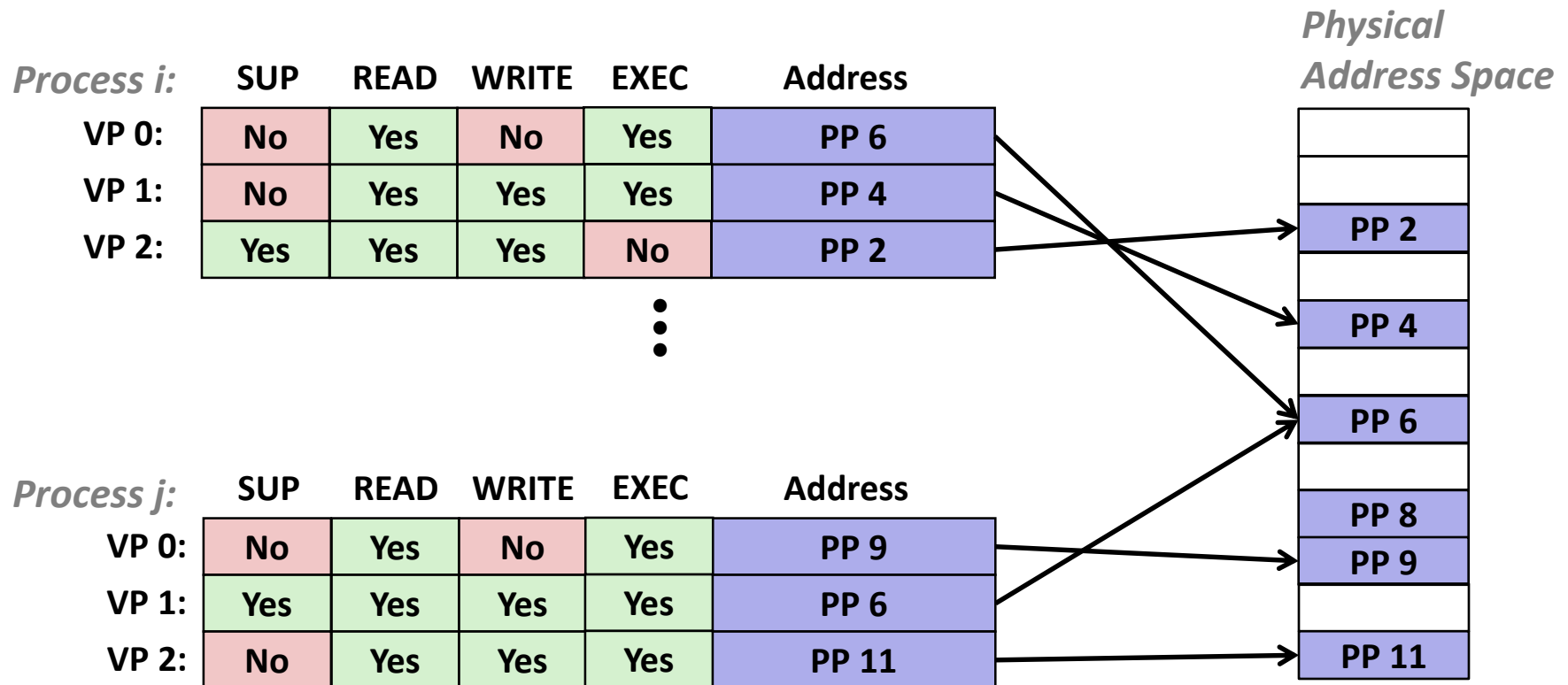


Today

- Address spaces
- 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



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**

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)

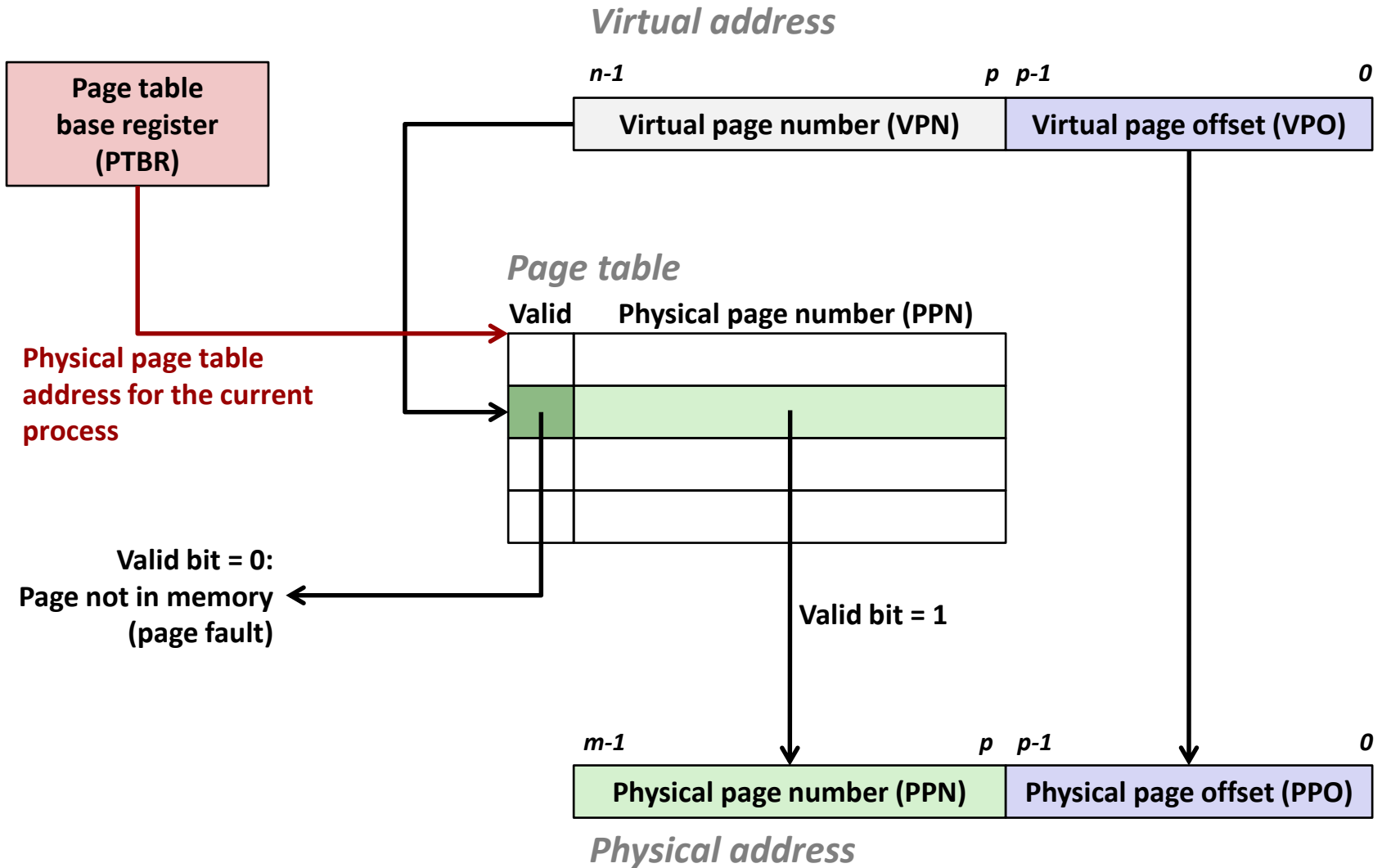
■ Components of the virtual address (VA)

- VPO: Virtual page offset
- VPN: Virtual page number

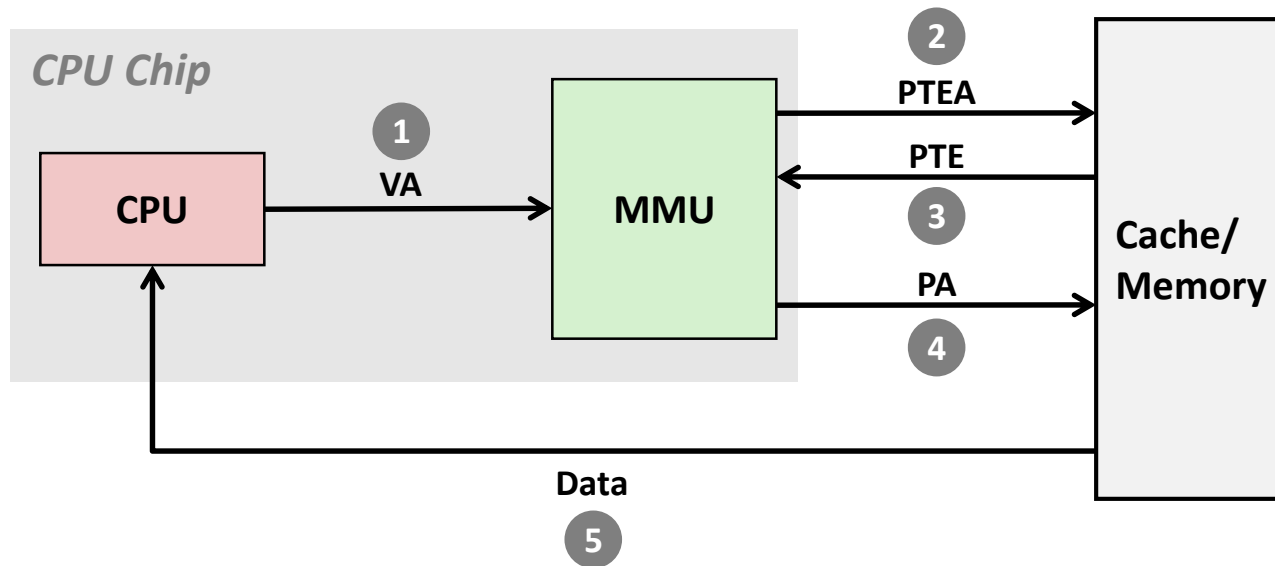
■ Components of the physical address (PA)

- PPO: Physical page offset (same as VPO)
- PPN: Physical page number

Address Translation With a Page Table

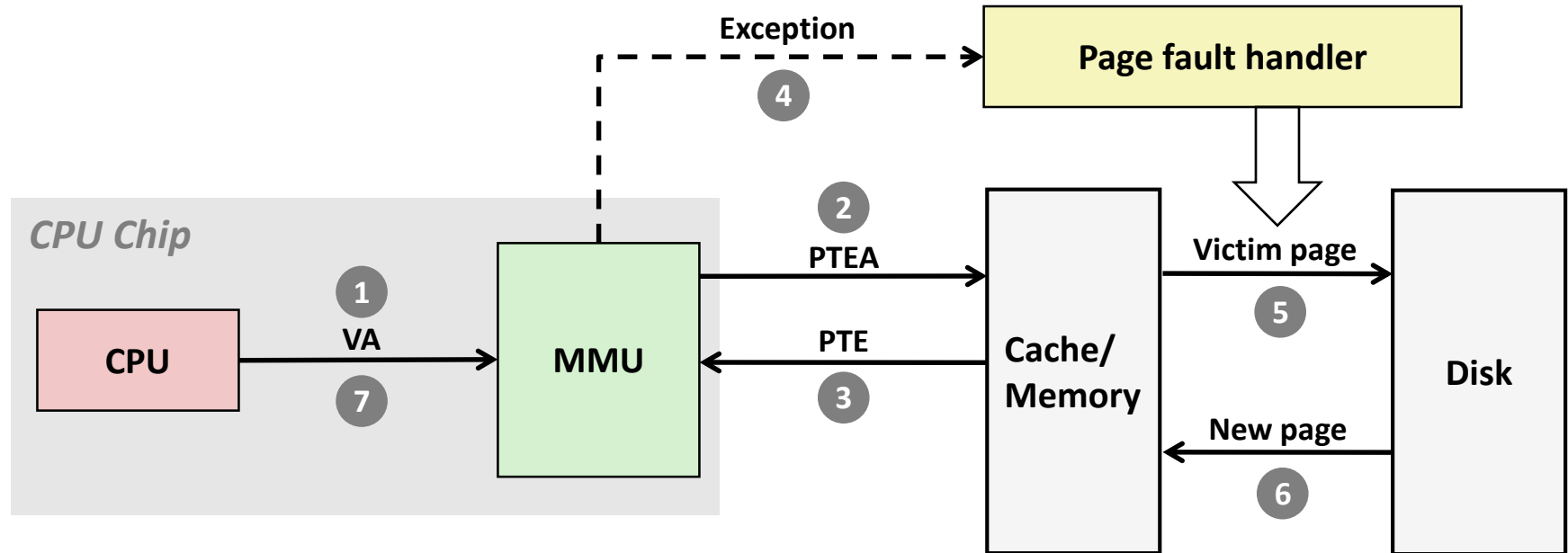


Address Translation: Page Hit



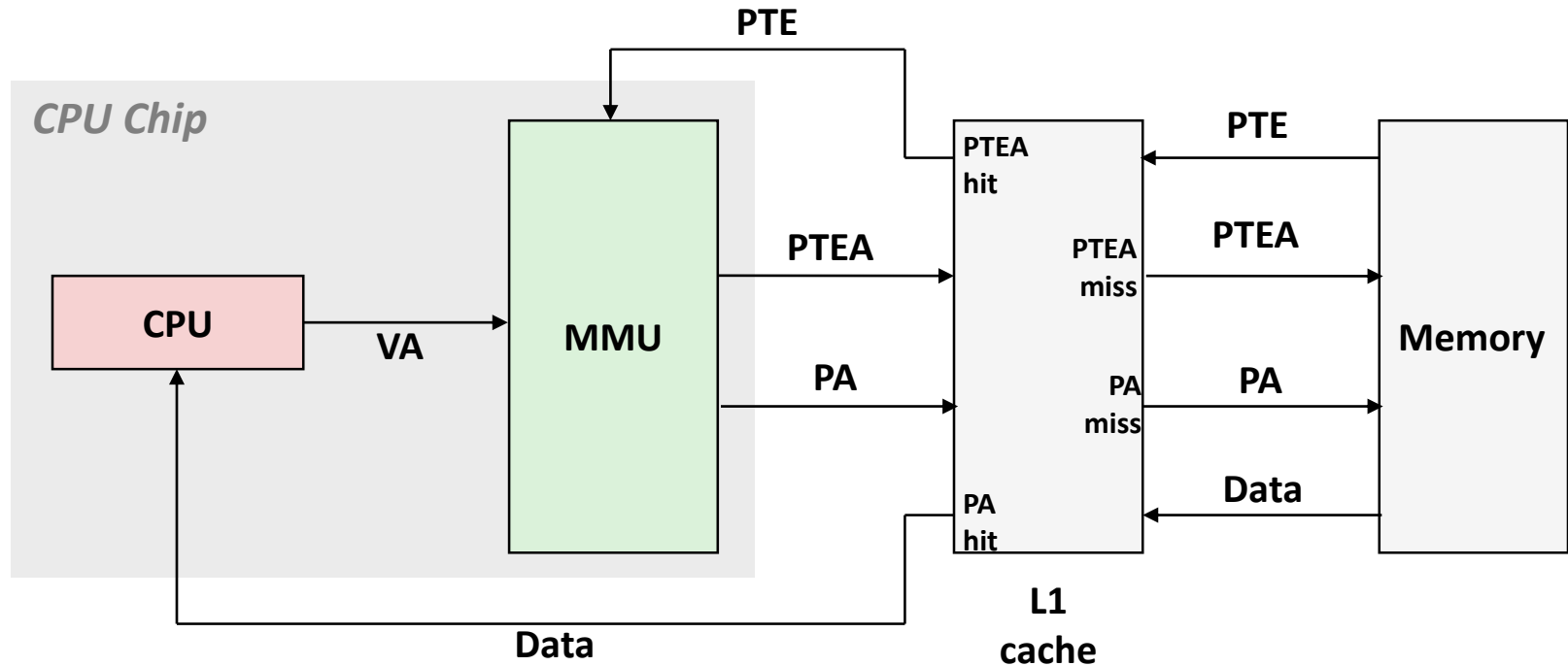
- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

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 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

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)

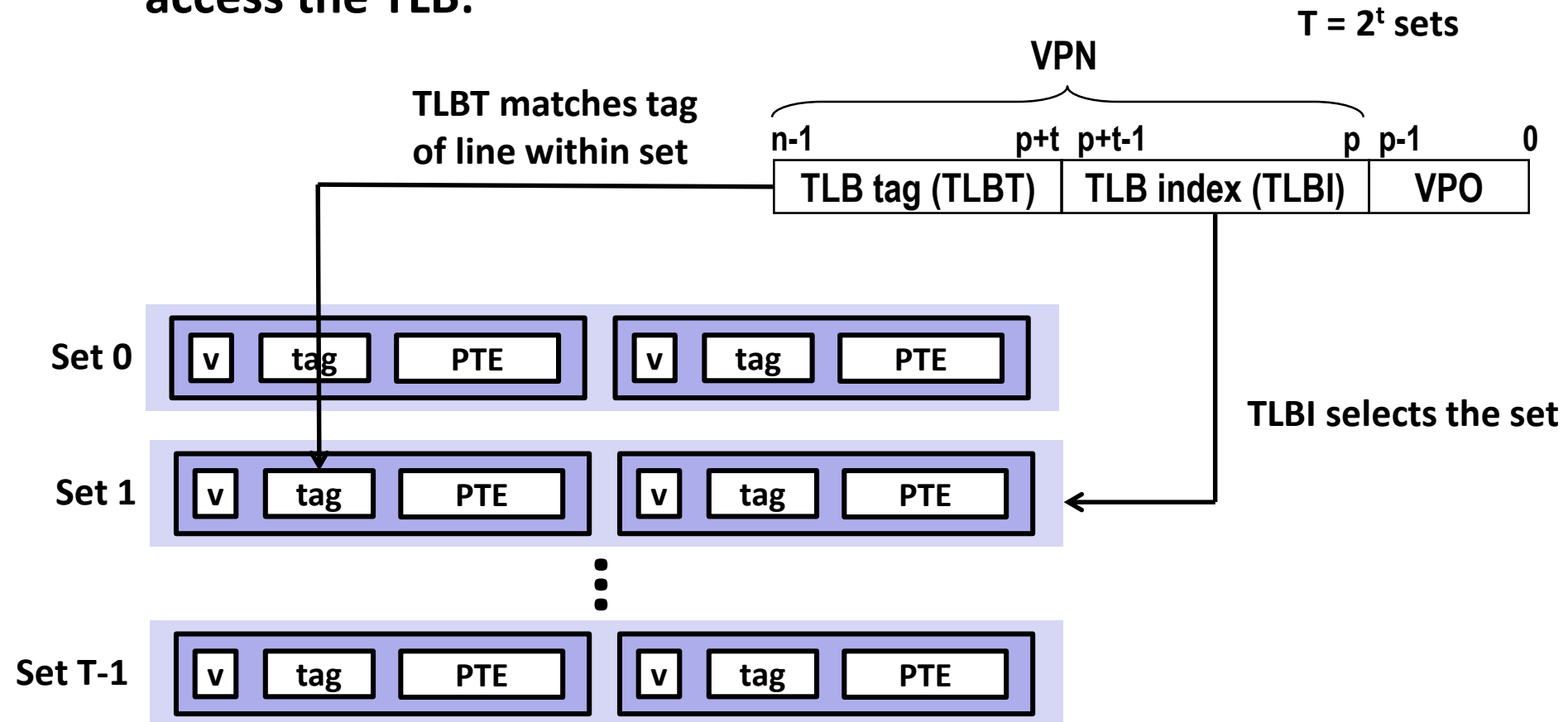
- *TLBI: TLB index*
- *TLBT: TLB tag*
- VPO: Virtual page offset
- VPN: Virtual page number

■ Components of the physical address (PA)

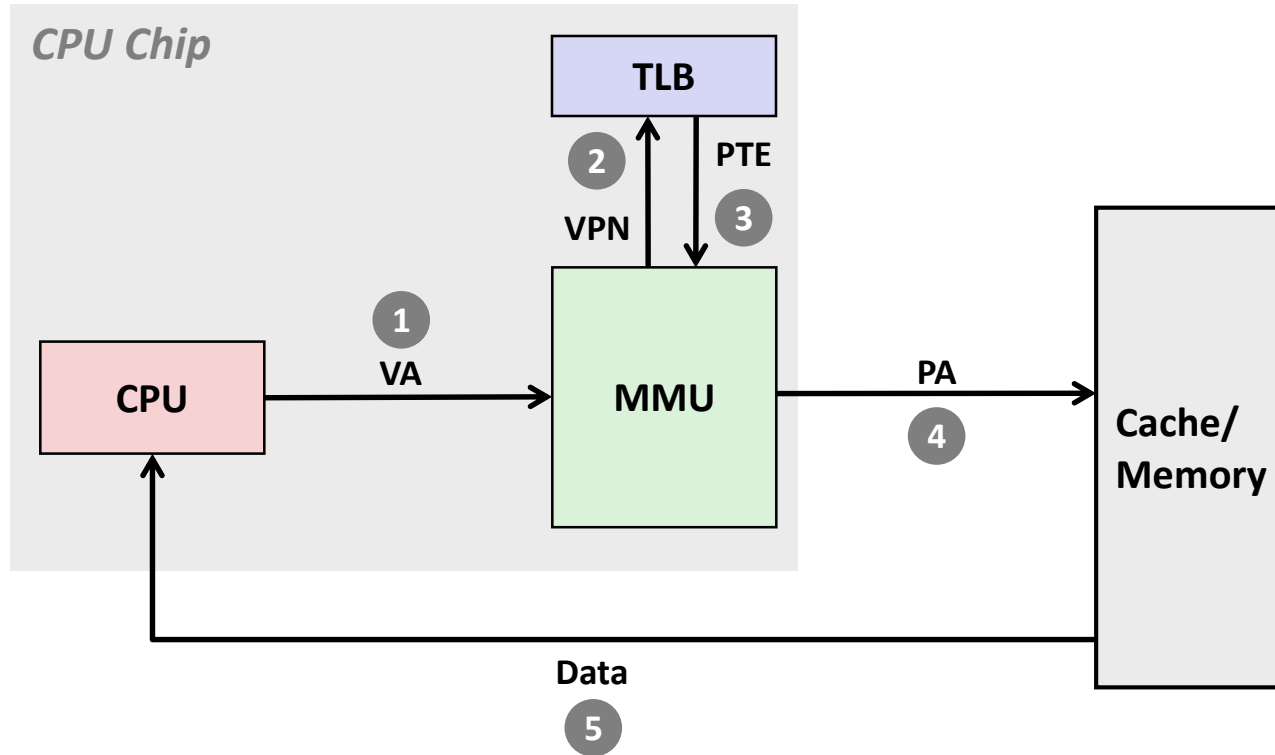
- PPO: Physical page offset (same as VPO)
- PPN: Physical page number

Accessing the TLB

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

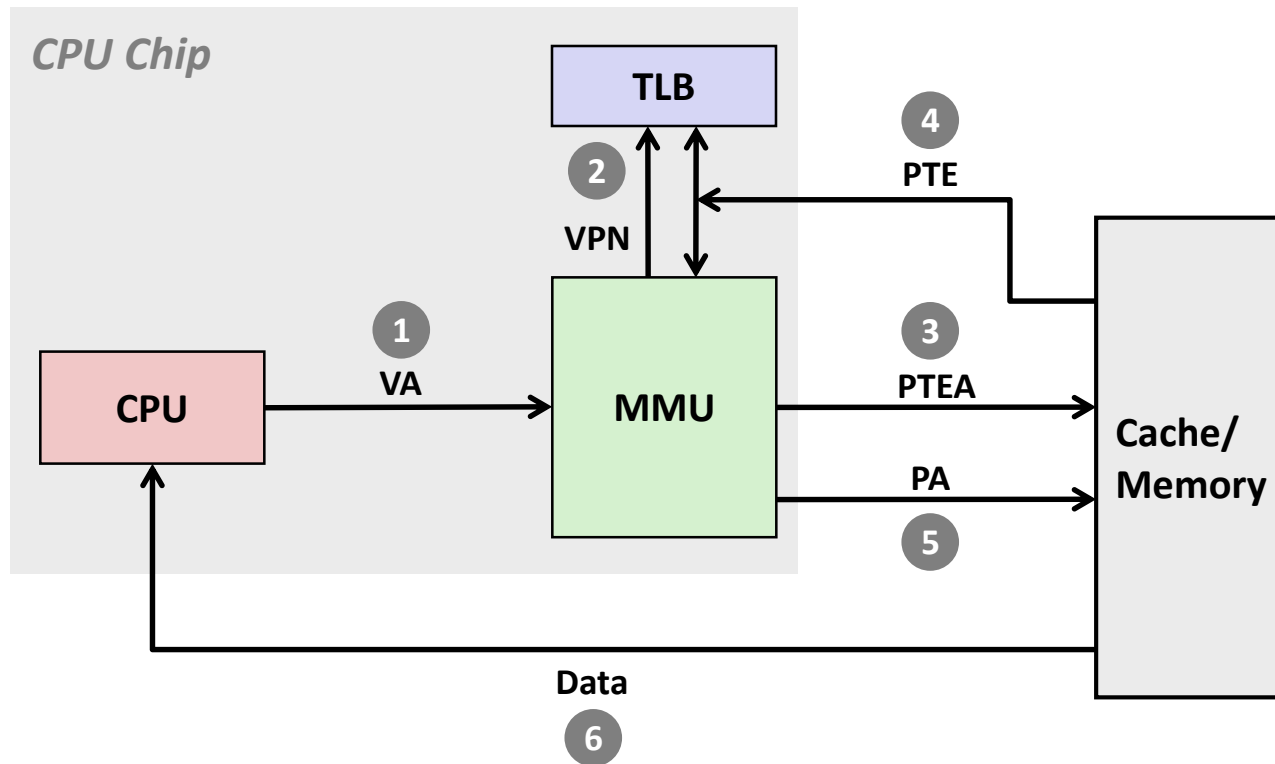


TLB Hit



A TLB hit eliminates a cache/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

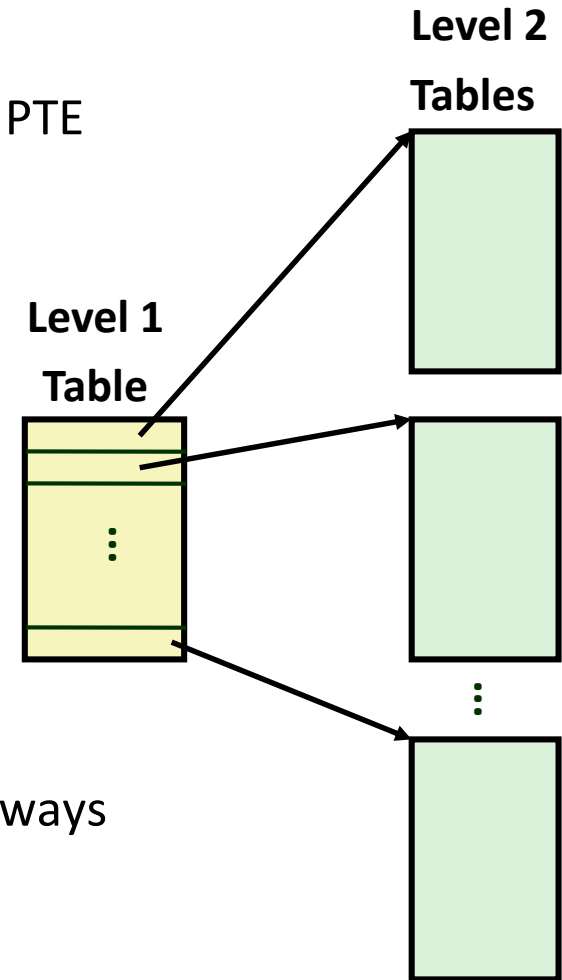
■ Problem:

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

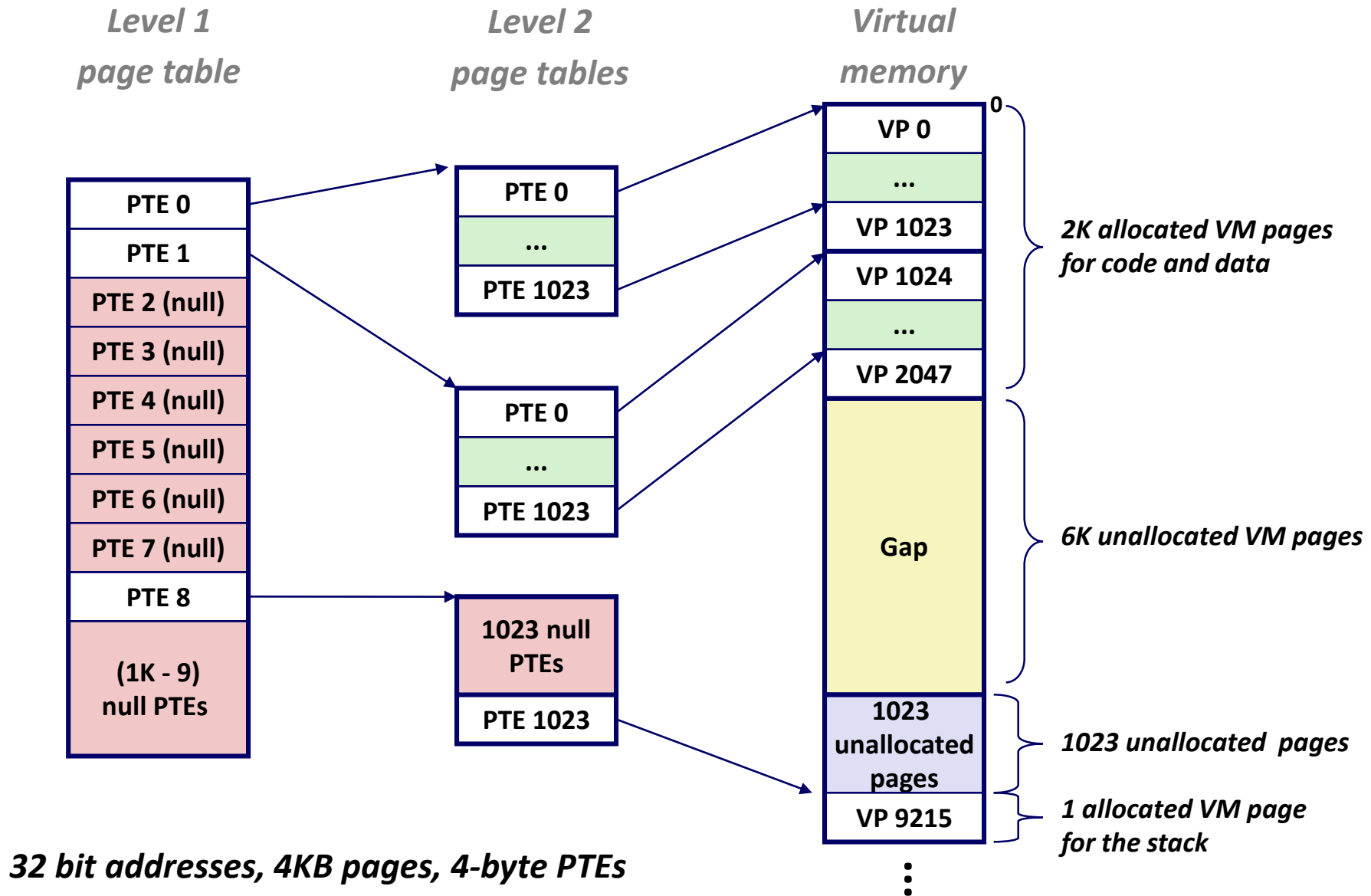
■ Common solution: Multi-level page table

■ Example: 2-level page table

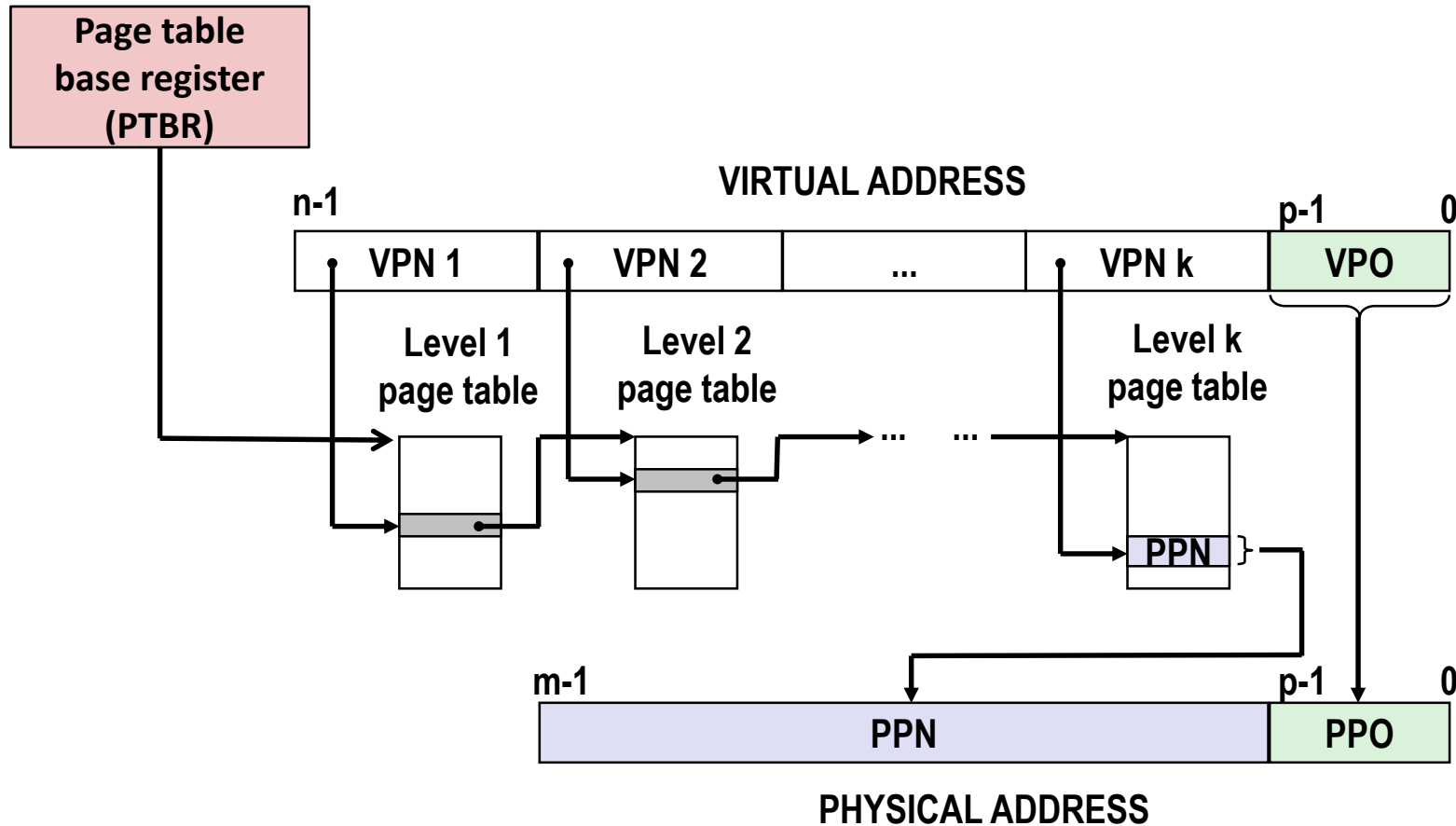
- Level 1 table: each PTE points to a page table (always memory resident)
- Level 2 table: each PTE points to a page (paged in and out like any other data)



A Two-Level Page Table Hierarchy



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

■ Implemented via combination of hardware & software

- MMU, TLB, exception handling mechanisms part of hardware
- Page fault handlers, TLB management performed in software