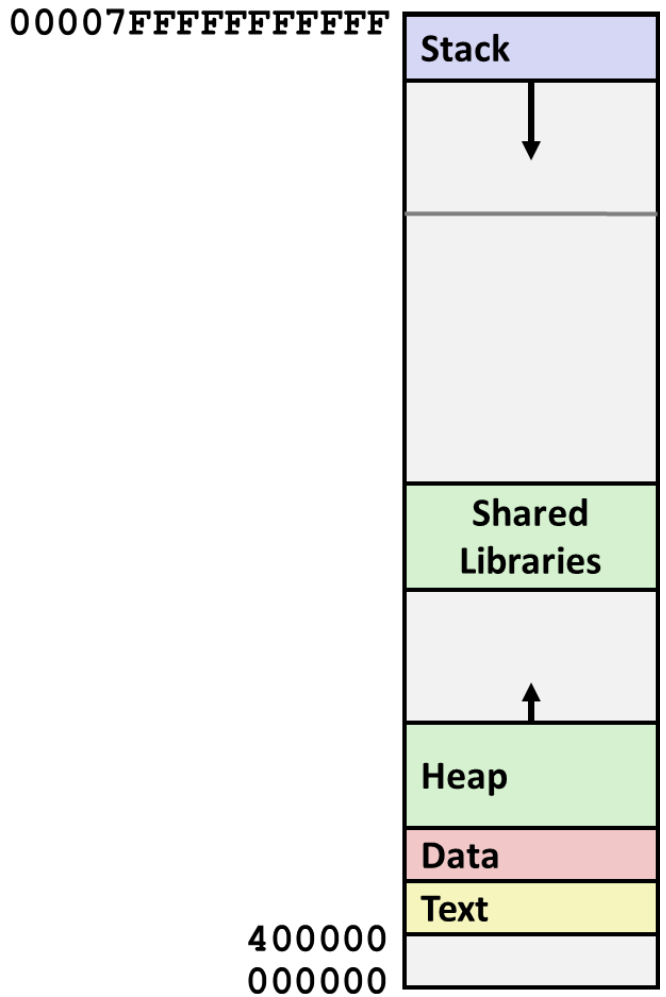
Virtual Memory: Concepts

CS2011: Introduction to Computer Systems
Lecture 14 (**9.1, 9.2, 9.3, 9.4, 9.5, 9.6**)

This Picture is a Lie

- This is RAM, we said…
- But the computer can run more than one program at a time!
- Where are all the other programs?
- Let's *investigate*.



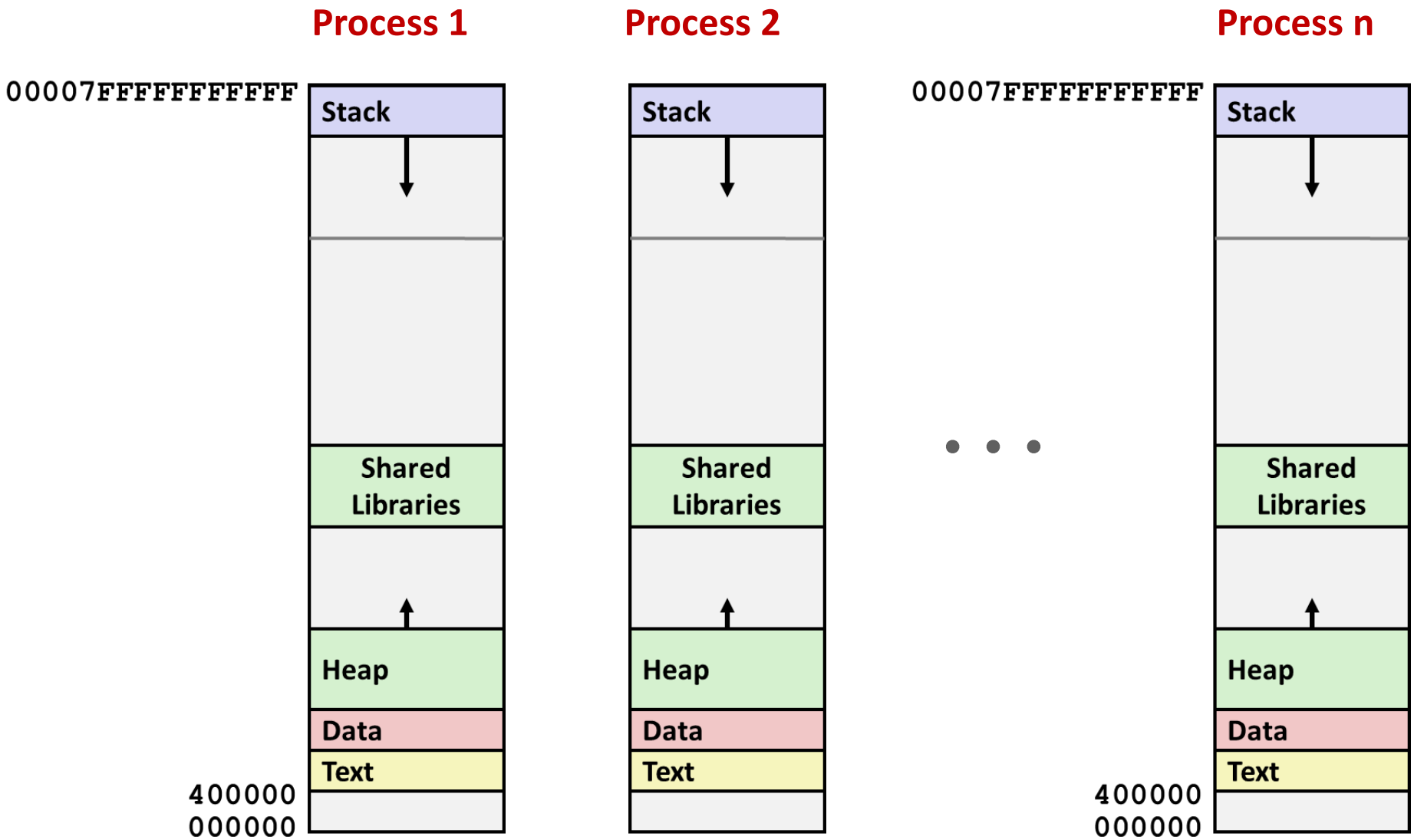
Processes

- **Definition: A *process* is an instance of a running program.**
  - One of the most profound ideas in computer science
  - Not the same as "program" or "processor"

- **Unix: A *parent process* creates a new *child process* by calling `fork`**
  - Child is (sort of) a copy of the parent
  - `fork` returns *twice*—once in each process
    - Different return value in each

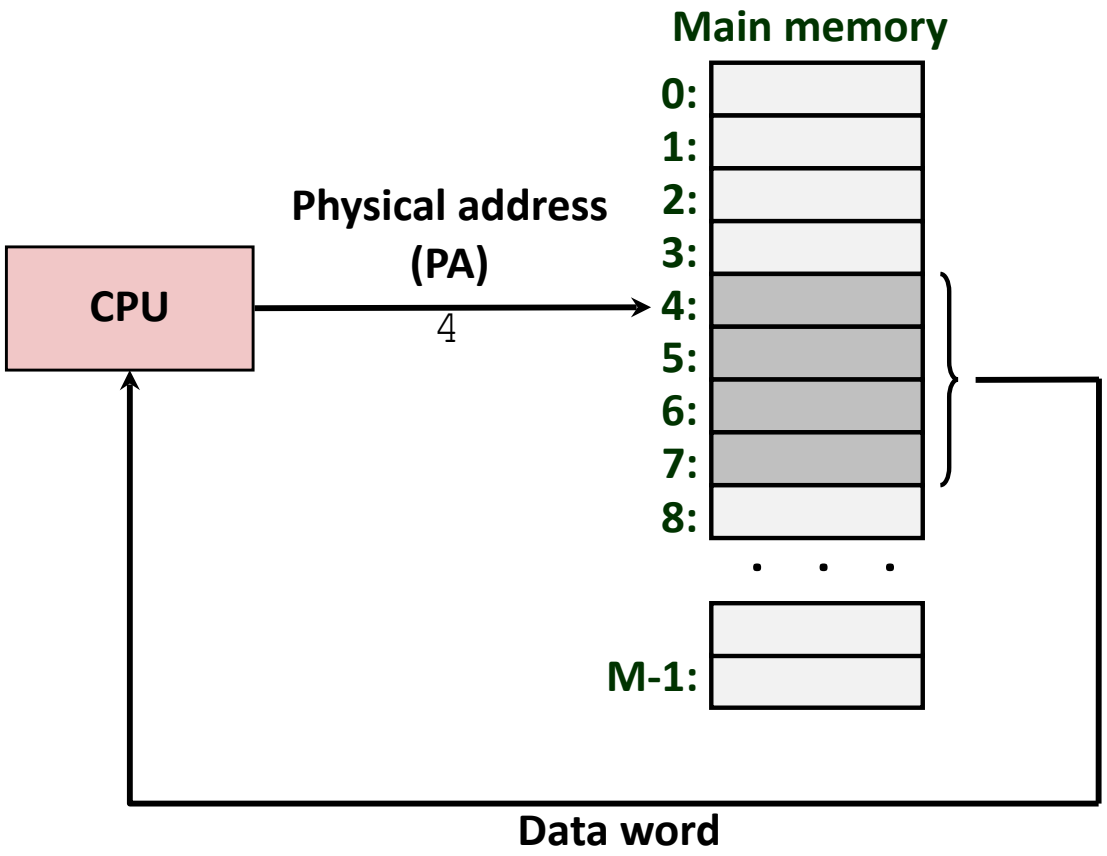
- **Parent can wait for child to finish by calling `waitpid`**
  - For now, think of this as "what `main` returns to"

Hmmm, How Does This Work?!



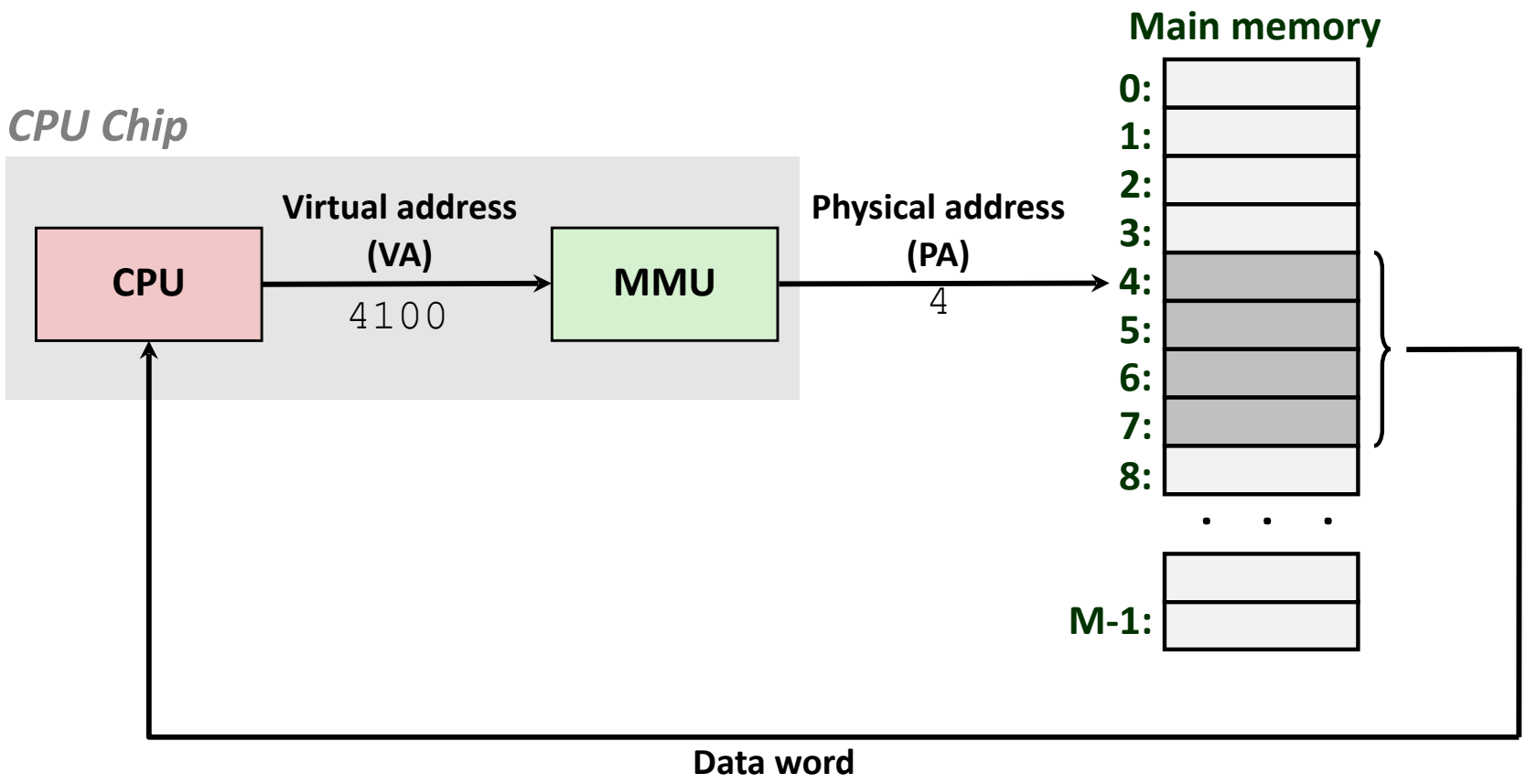
Solution: Virtual Memory

A System Using Physical Addressing



- Used in "simple" systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

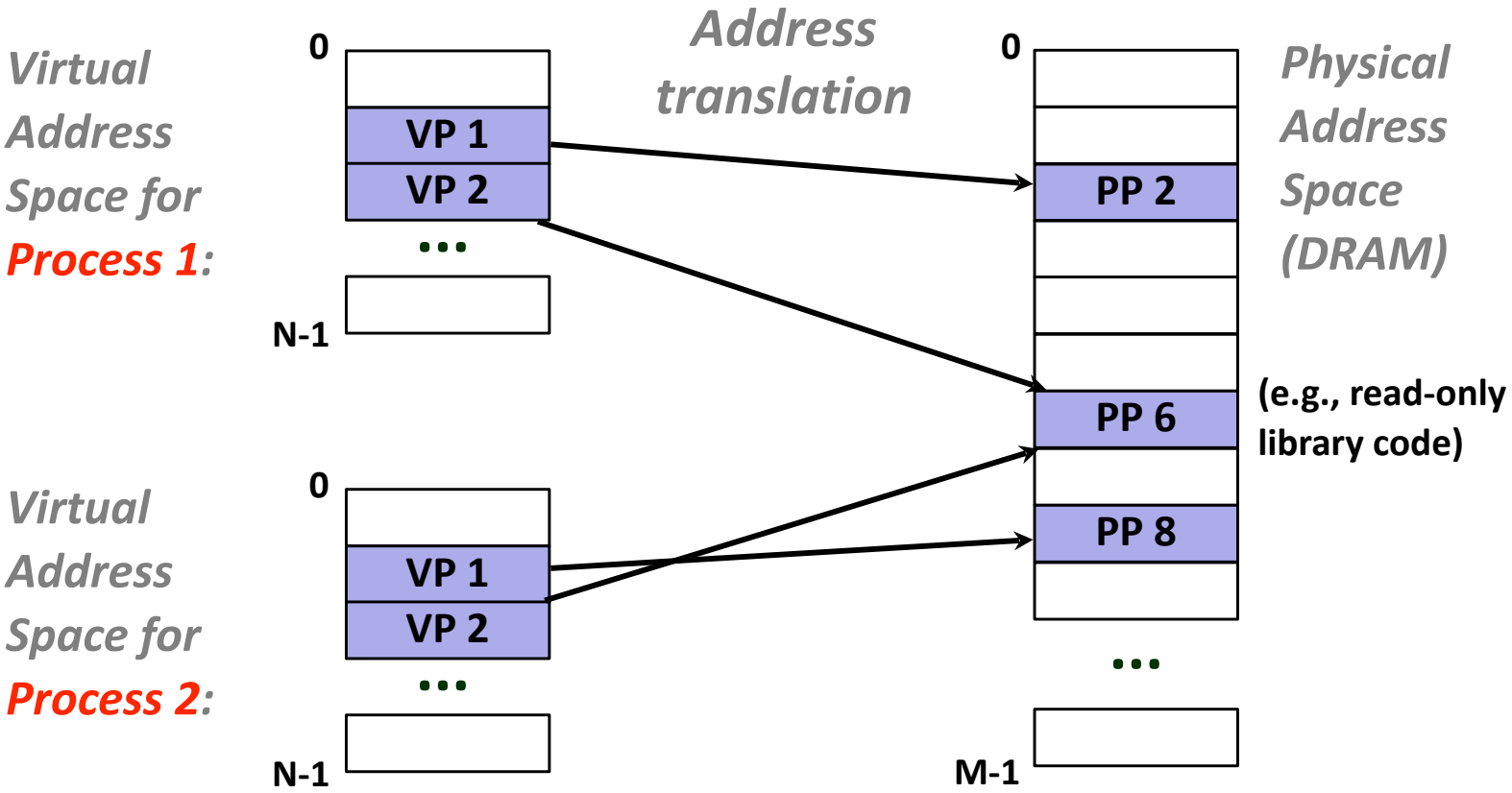
A System Using Virtual Addressing



- MMU (Memory Management Unit)
- Used in all modern servers, laptops, and smart phones
- One of the great ideas in computer science

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 of **V**irtual **P**ages (**VP**s)
  - Mapping function scatters addresses through physical memory
    - Well-chosen mappings can improve locality



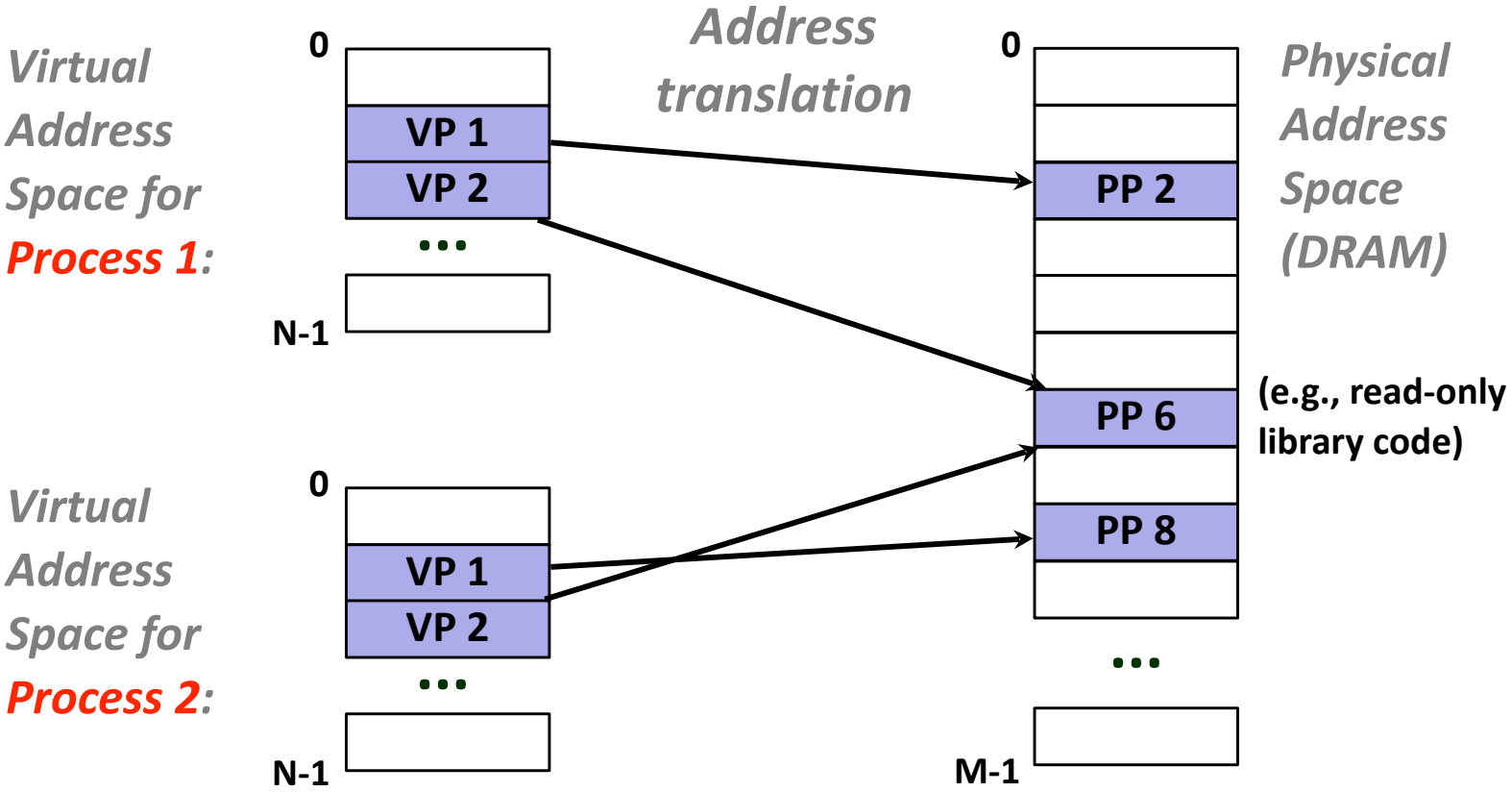
VM as a Tool for Memory Management

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)



Why Virtual Memory (VM)?

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

Most importantly: Uses main memory efficiently

- Use DRAM as a **cache** for parts of a virtual address space

Address Spaces

- **Linear address space:** Ordered set of contiguous non-negative integer addresses:

  {0, 1, 2, 3 … }

- **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}

N often much bigger than M

E.g., N = **2^{48} (64-bit machine and excluding kernel space) ≅ 256TB**

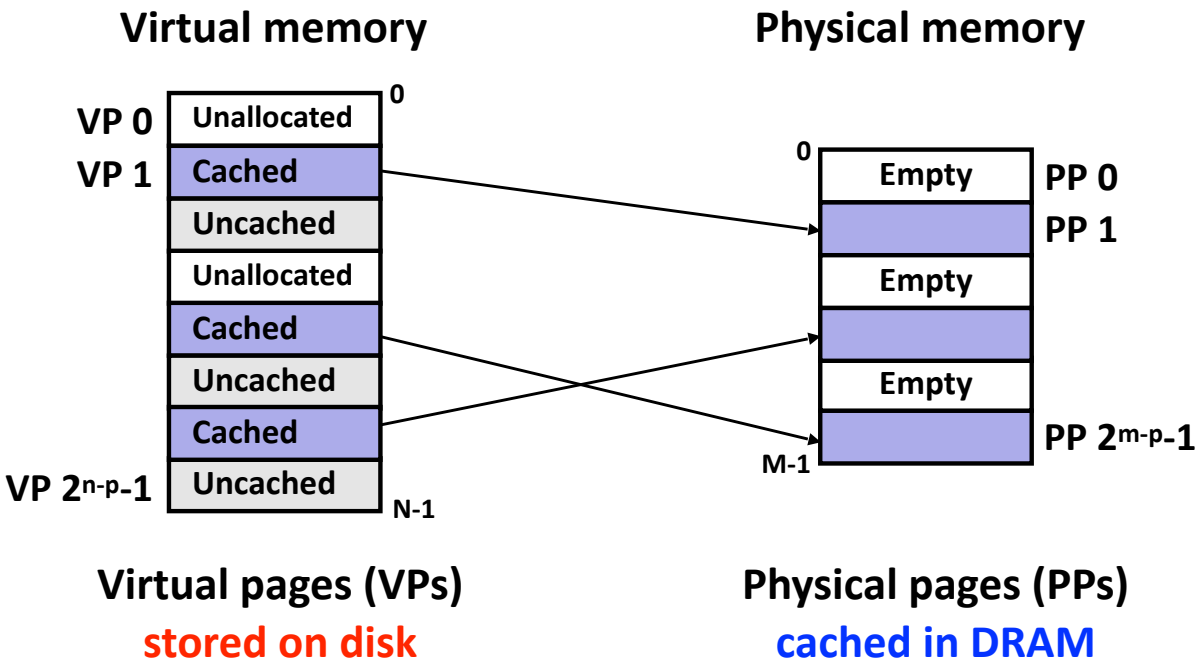
M = **2^{35} (a 32GB DRAM)**

VM Address Translation

- **Virtual Address Space**
  - *VAS = {0, 1, …, N–1}*
- **Physical Address Space**
  - *PAS = {0, 1, …, M–1}*
- **Address Translation**
  - ***MAP: VAS → PAS U {∅}***
  - For virtual address ***a***:
    - ***MAP(a) = a'*** if data at virtual address ***a*** is at physical address ***a'*** in ***PAS***
    - ***MAP(a) =*** ∅ if data at virtual address ***a*** is not in physical memory
      - Either invalid or stored on disk

VM as a Tool for Caching

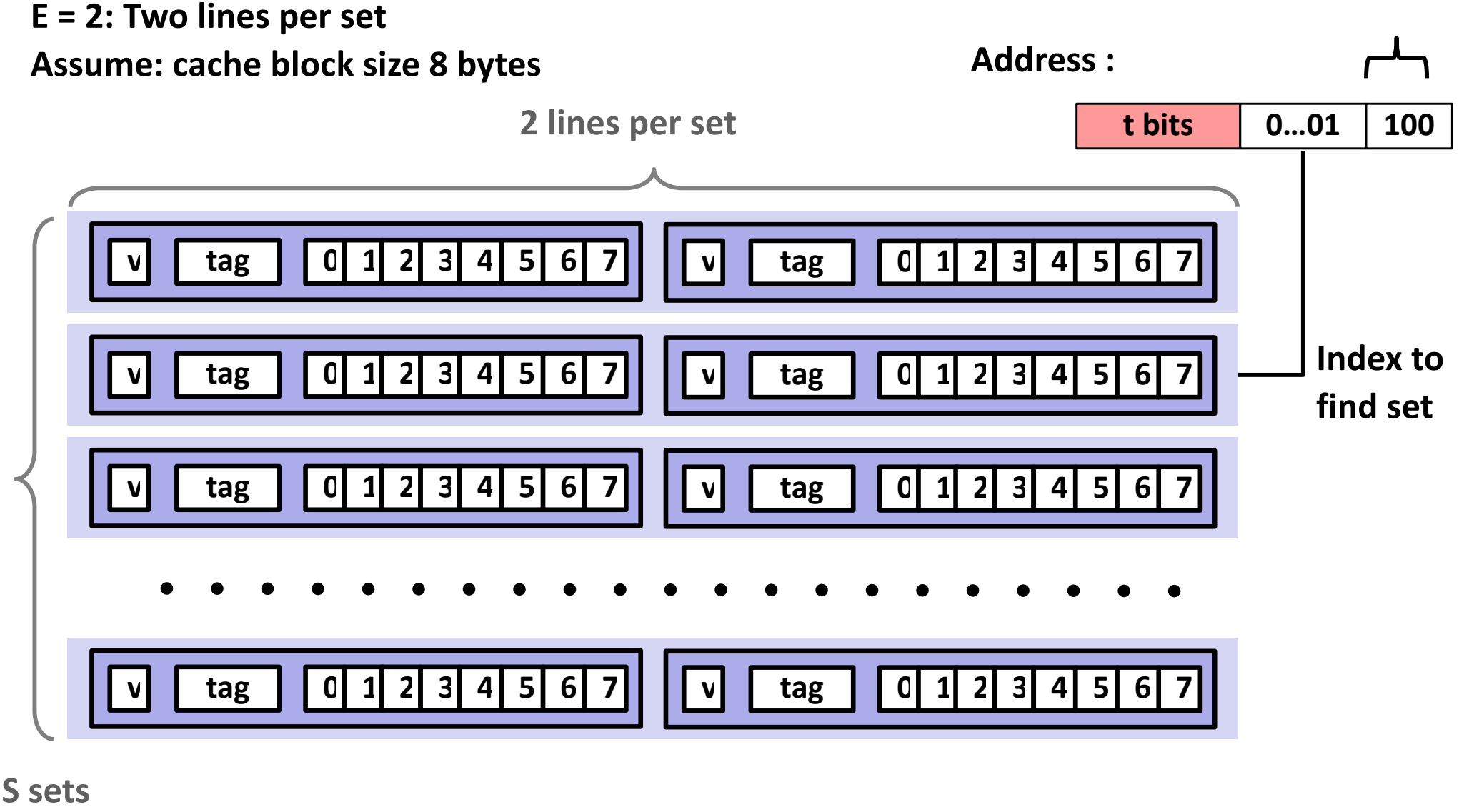
- Conceptually, *virtual memory* is an array of N contiguous bytes stored on disk.
- The contents of the array on disk are cached in *physical memory* (*DRAM cache*)
  - These cache blocks are called *pages* (size is P = 2^p bytes, e.g., 2^{12} = 4096 bytes = 4KB)



Remember: Set Associative Cache

E = 2: Two lines per set

Assume: cache block size 8 bytes



DRAM Cache Organization

DRAM cache organization driven by the enormous miss penalty

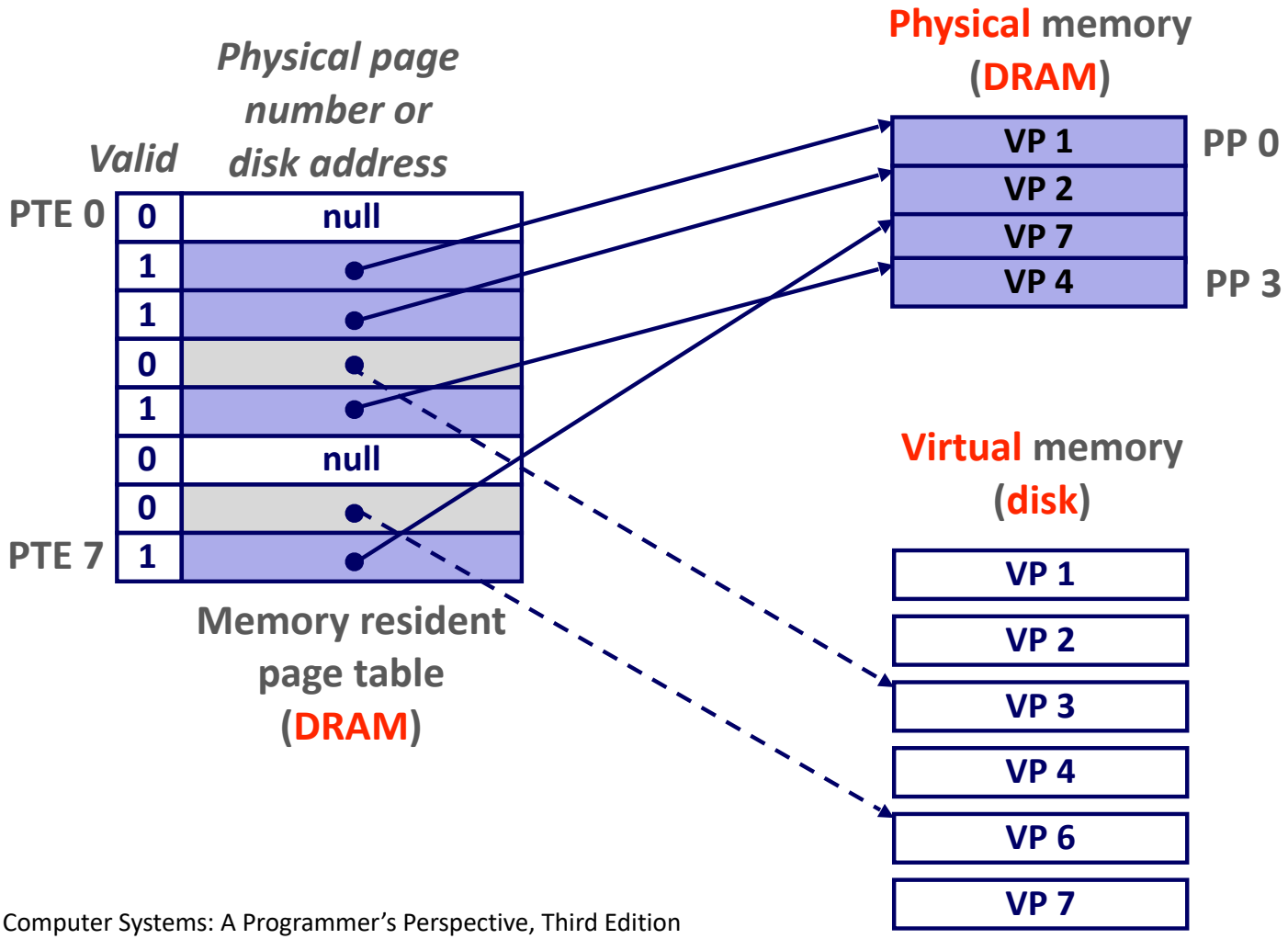
- DRAM is about ***10x*** slower than SRAM
- Disk is about ***10,000x*** slower than DRAM
  - Misses in DRAM are much more expensive
  - Accessing first byte on disk (especially if magnetic disk) is very slow

Design Consequences

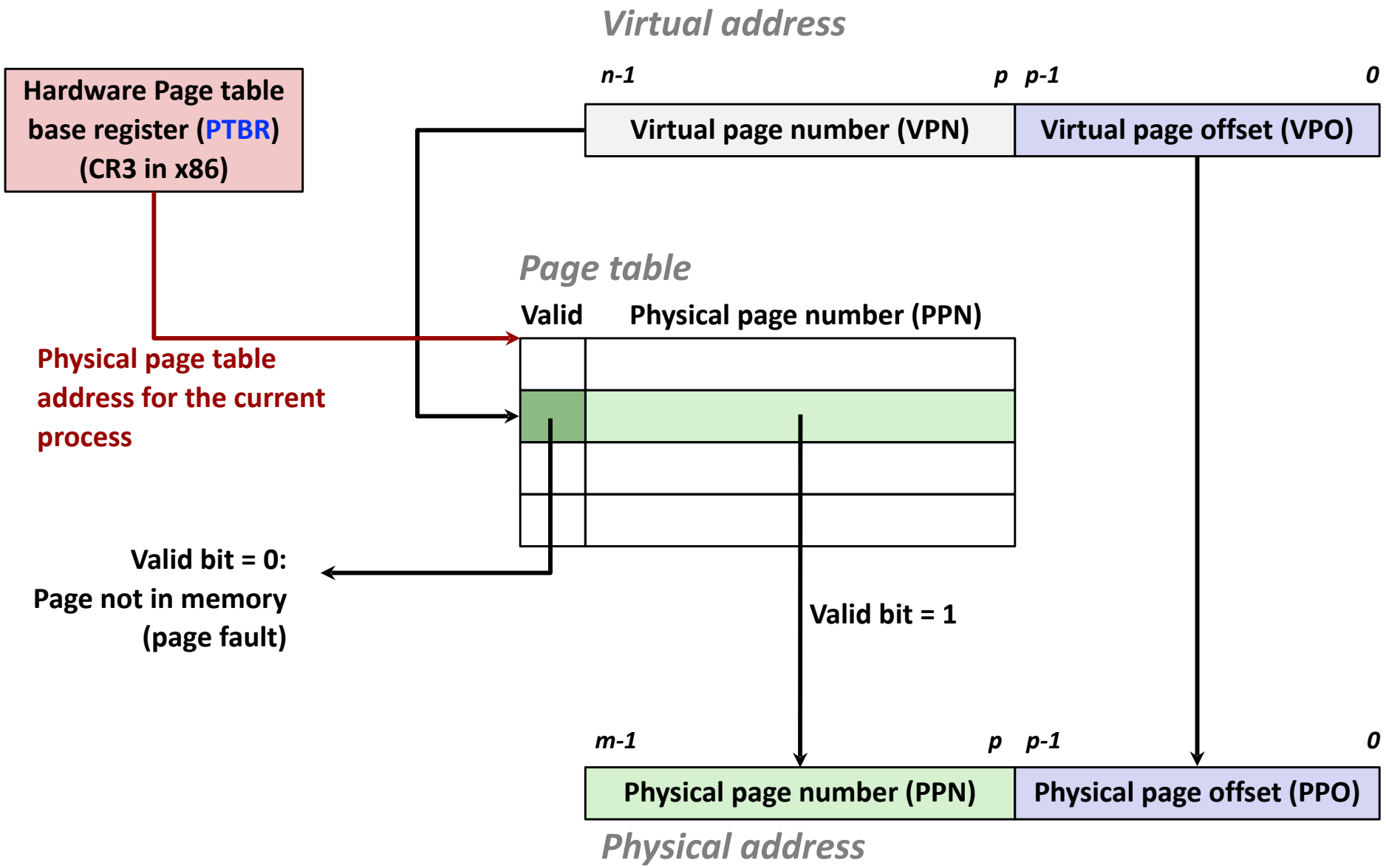
- **Large page (block) size**: typically 4 KB, sometimes 4 MB
- **Fully** associative
  - Any VP can be placed in any PP
  - Requires a "large" mapping function – different from cache memories
- Highly sophisticated, **expensive replacement algorithms**
  - The penalty of replacing the wrong VP is very high
  - Too complicated and open-ended to be implemented in hardware (*will not cover in this class*)
- Always **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

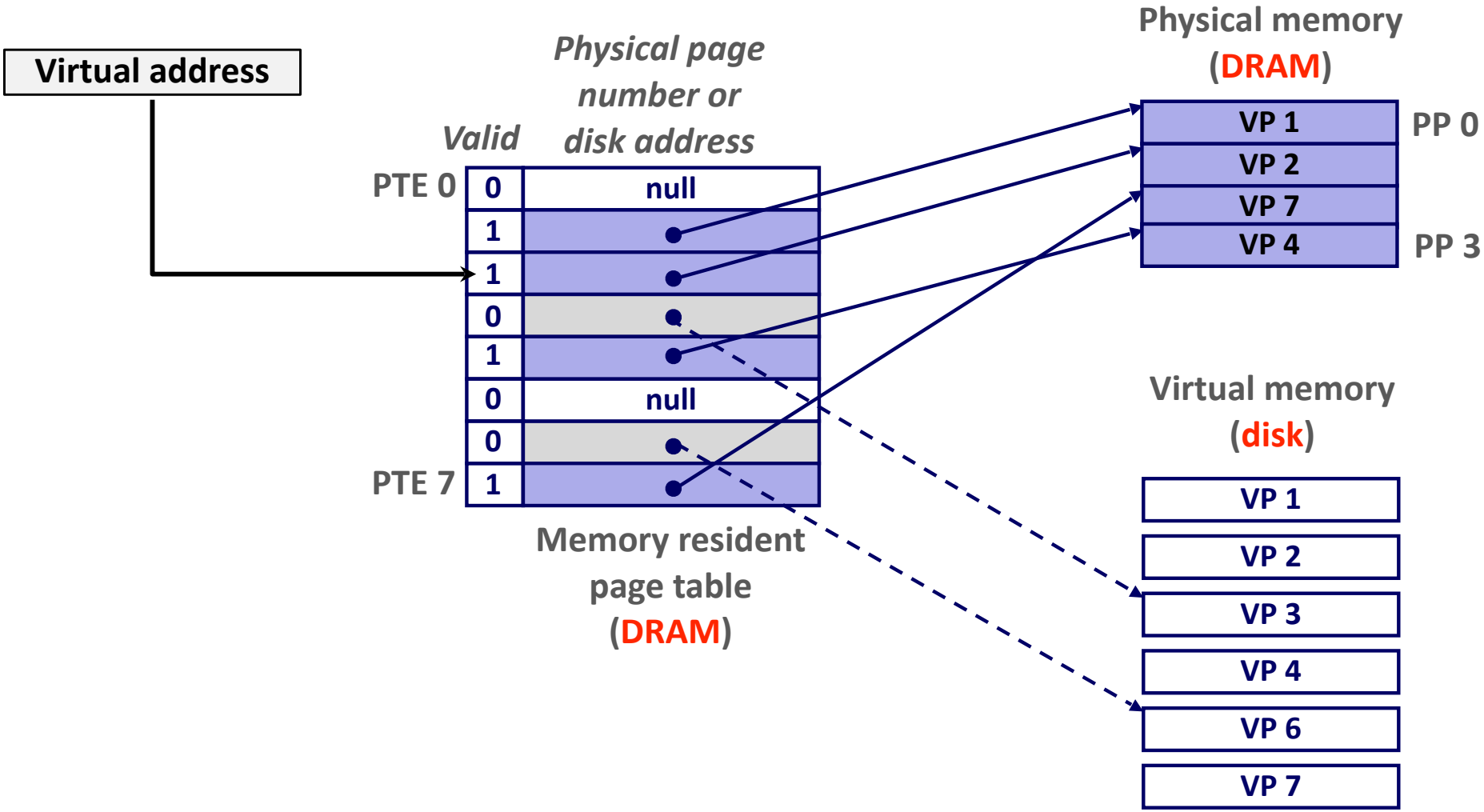


Address Translation With a Page Table

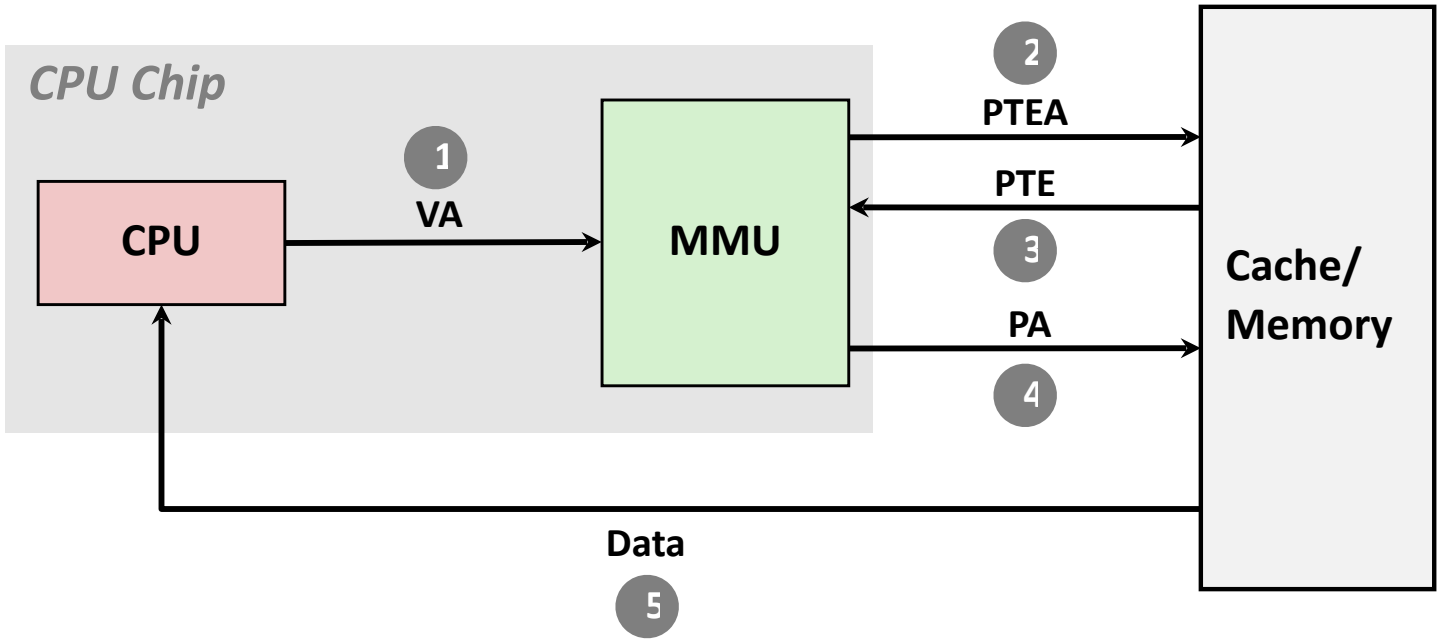


Page Hit

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



Address Translation: Page Hit



1) Processor sends virtual address to MMU

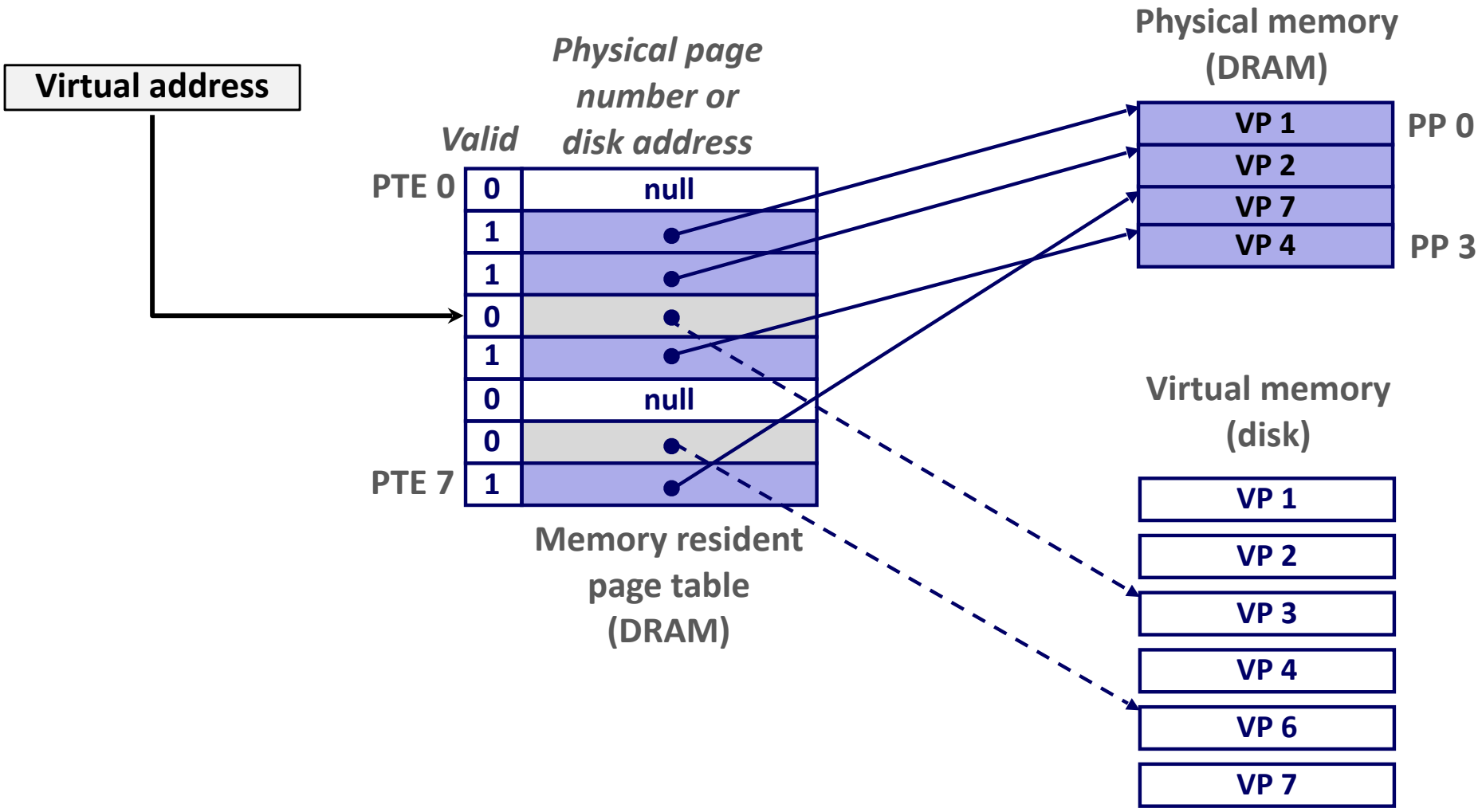
2-3) MMU fetches PTE from page table in cache/memory

4) MMU constructs physical address and sends physical address to cache/memory

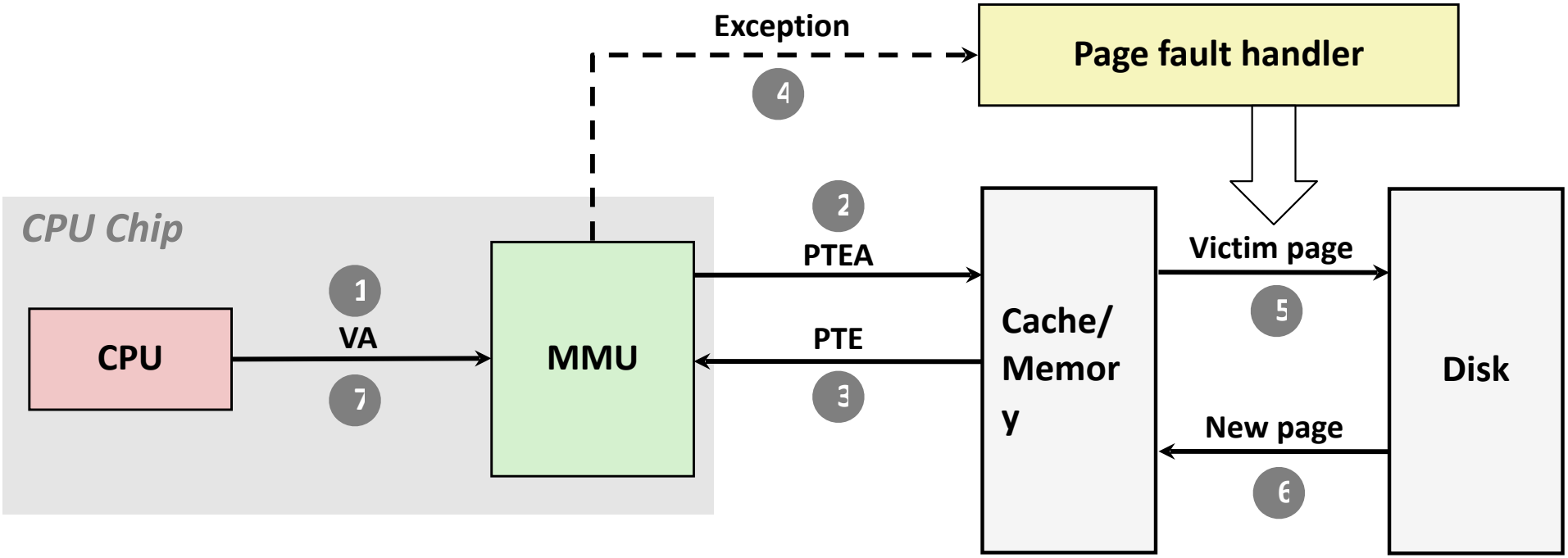
5) Cache/memory sends data word to processor

Page Fault

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



Address Translation: Page Fault



1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in cache/memory

4) Valid bit is zero, so MMU triggers **page fault exception**

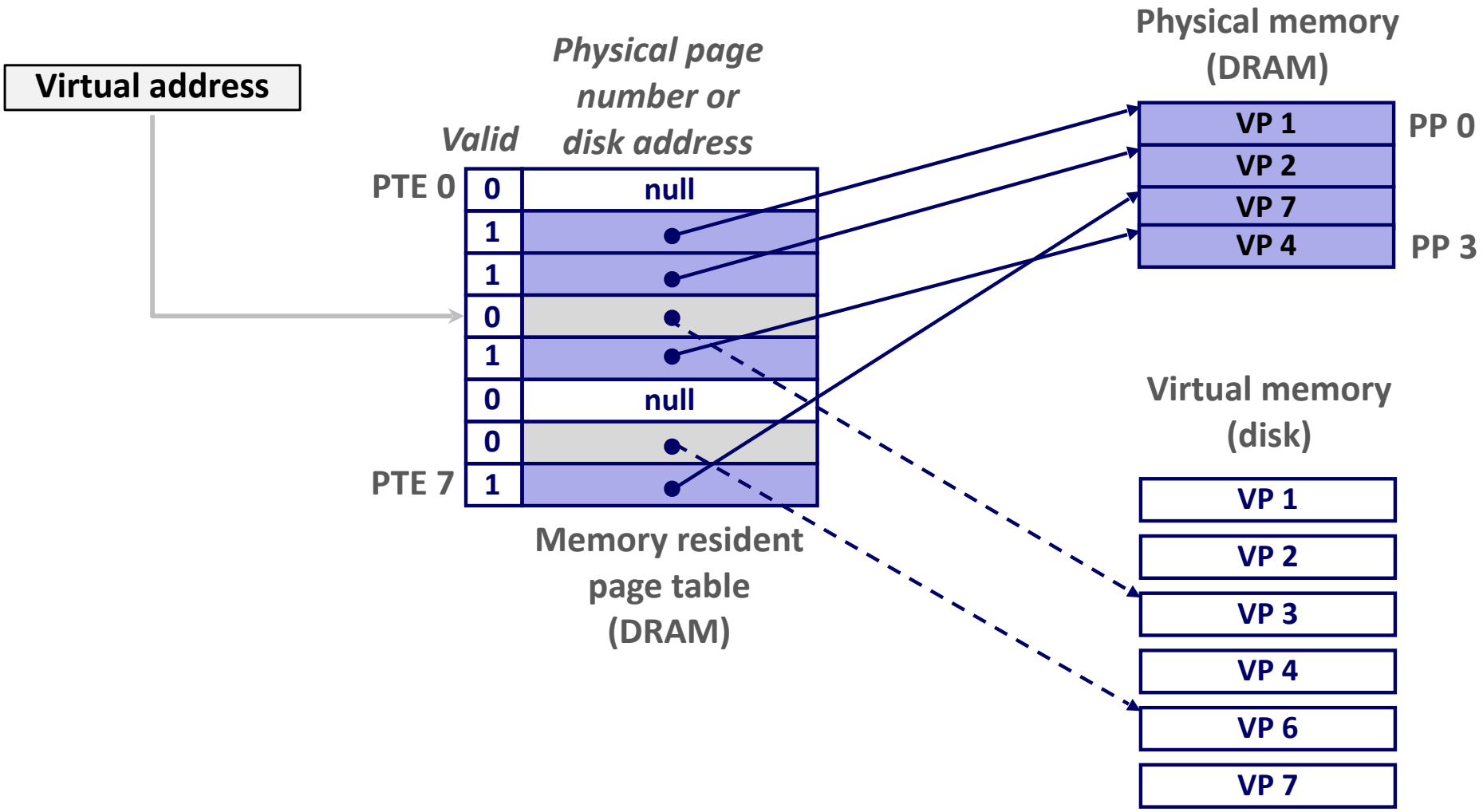
5) Handler identifies victim (and, if dirty, pages it out to disk)

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

7) Handler returns to original process, restarting faulting instruction (which will result in a hit now)

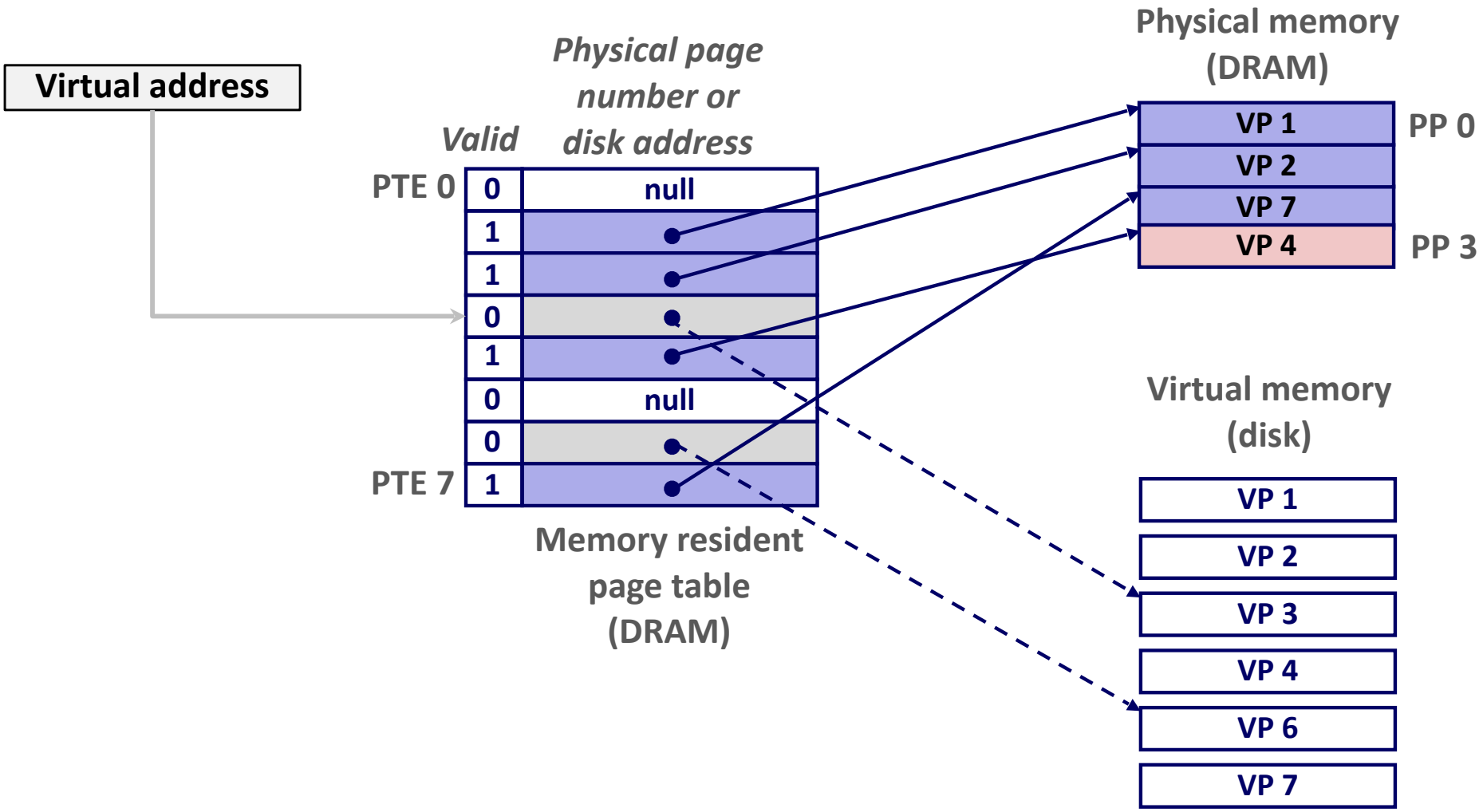
Handling Page Fault

- Page miss causes page fault (an exception)



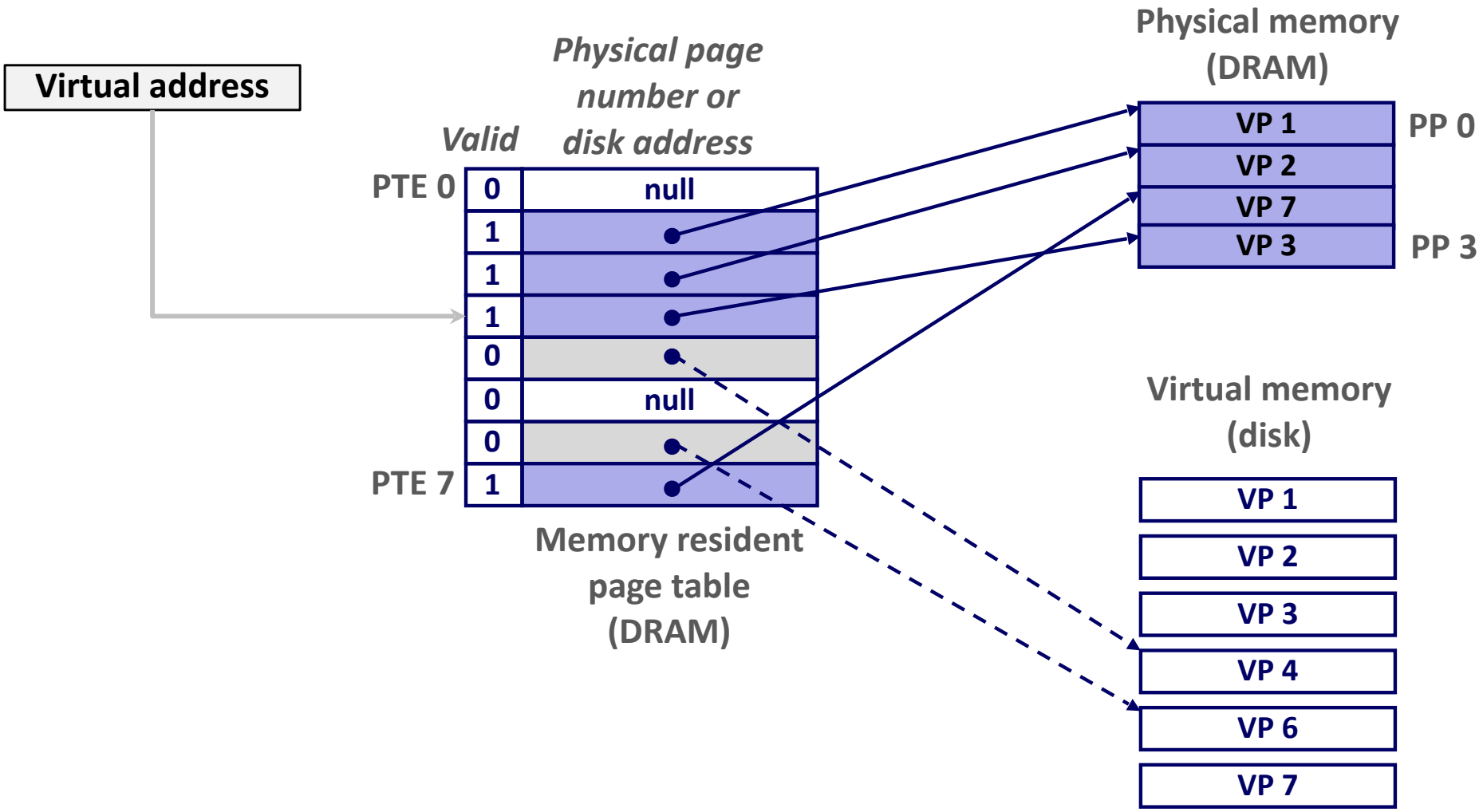
Handling Page Fault

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



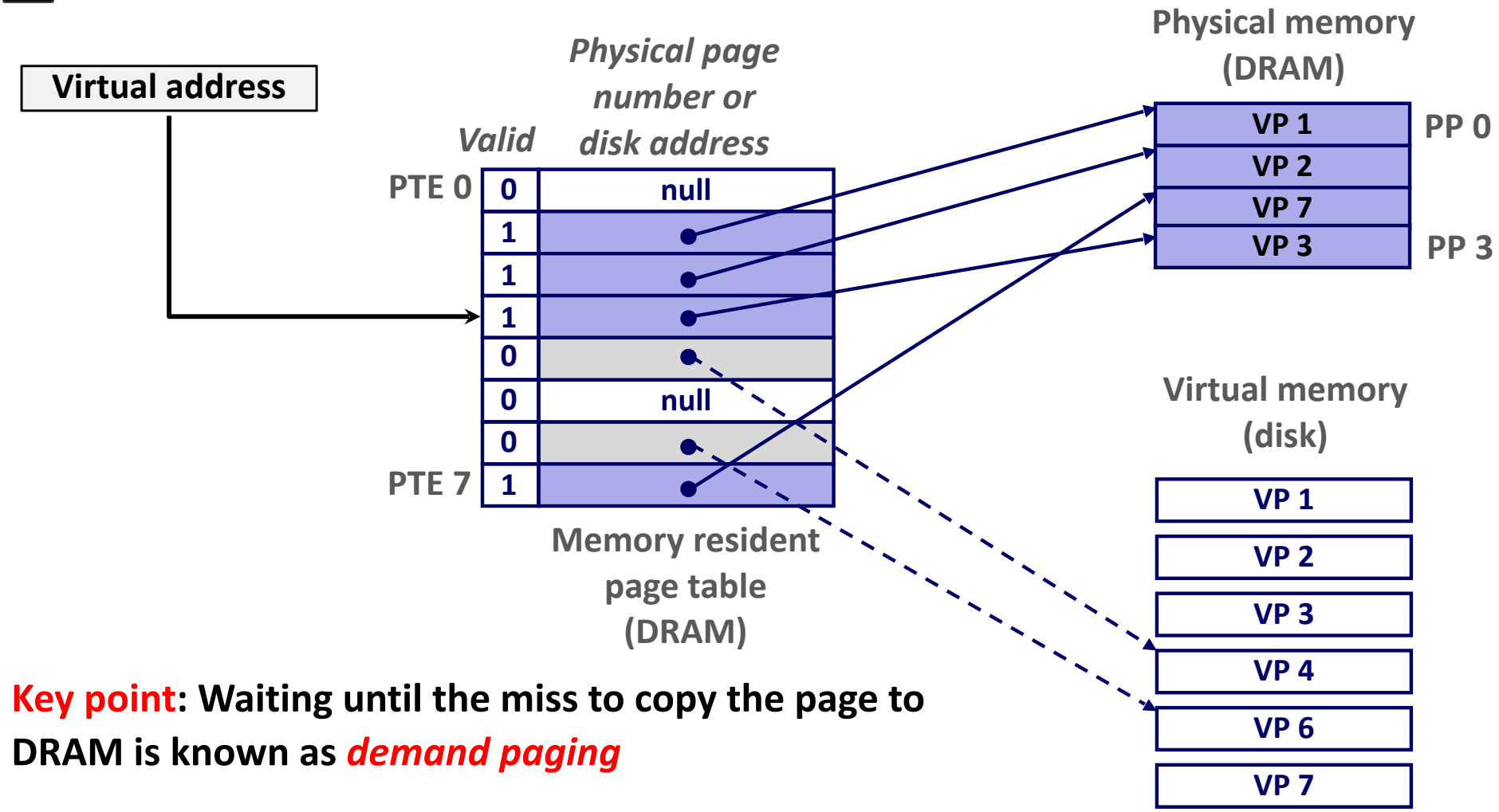
Handling Page Fault

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



Handling Page Fault

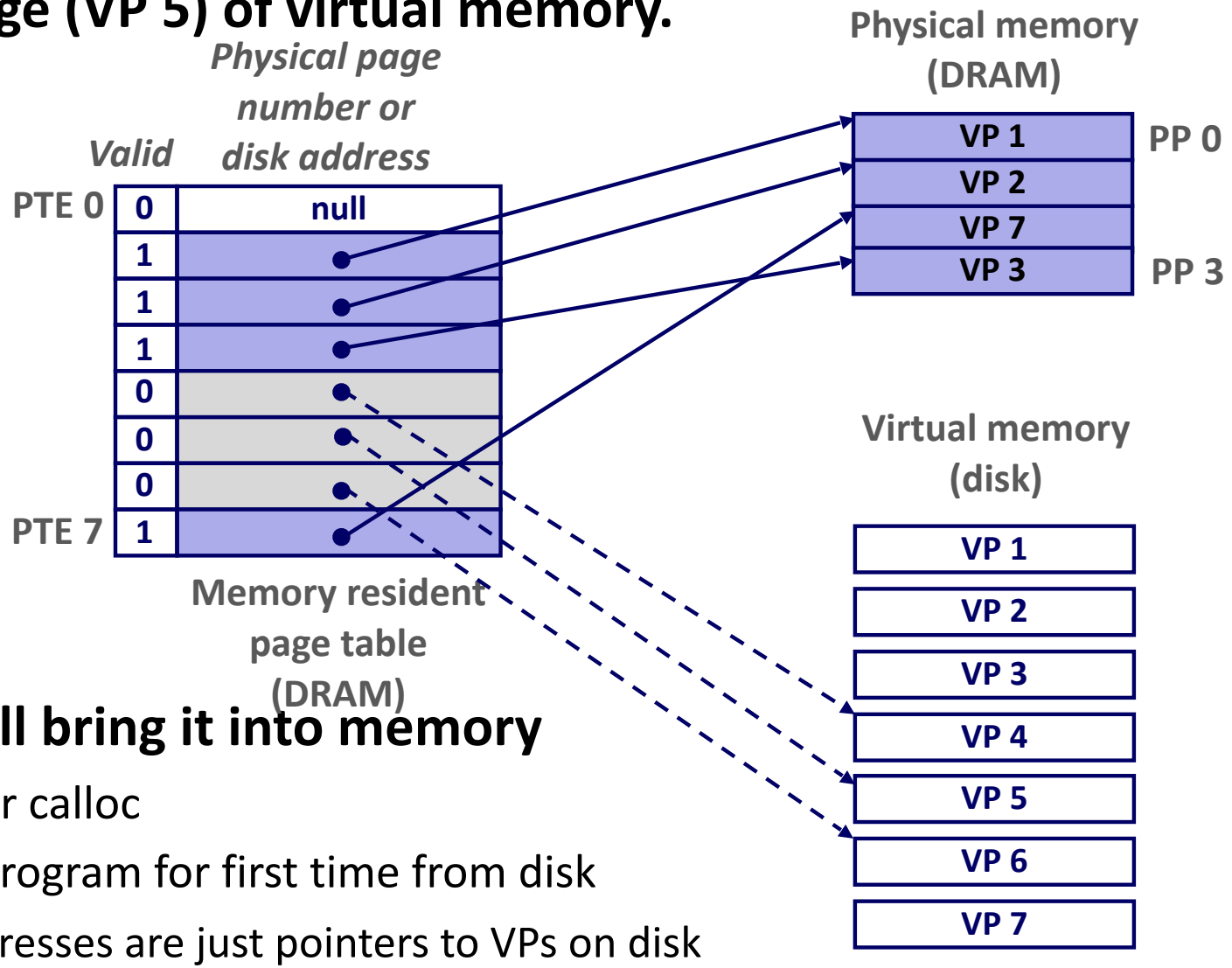
- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!



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.



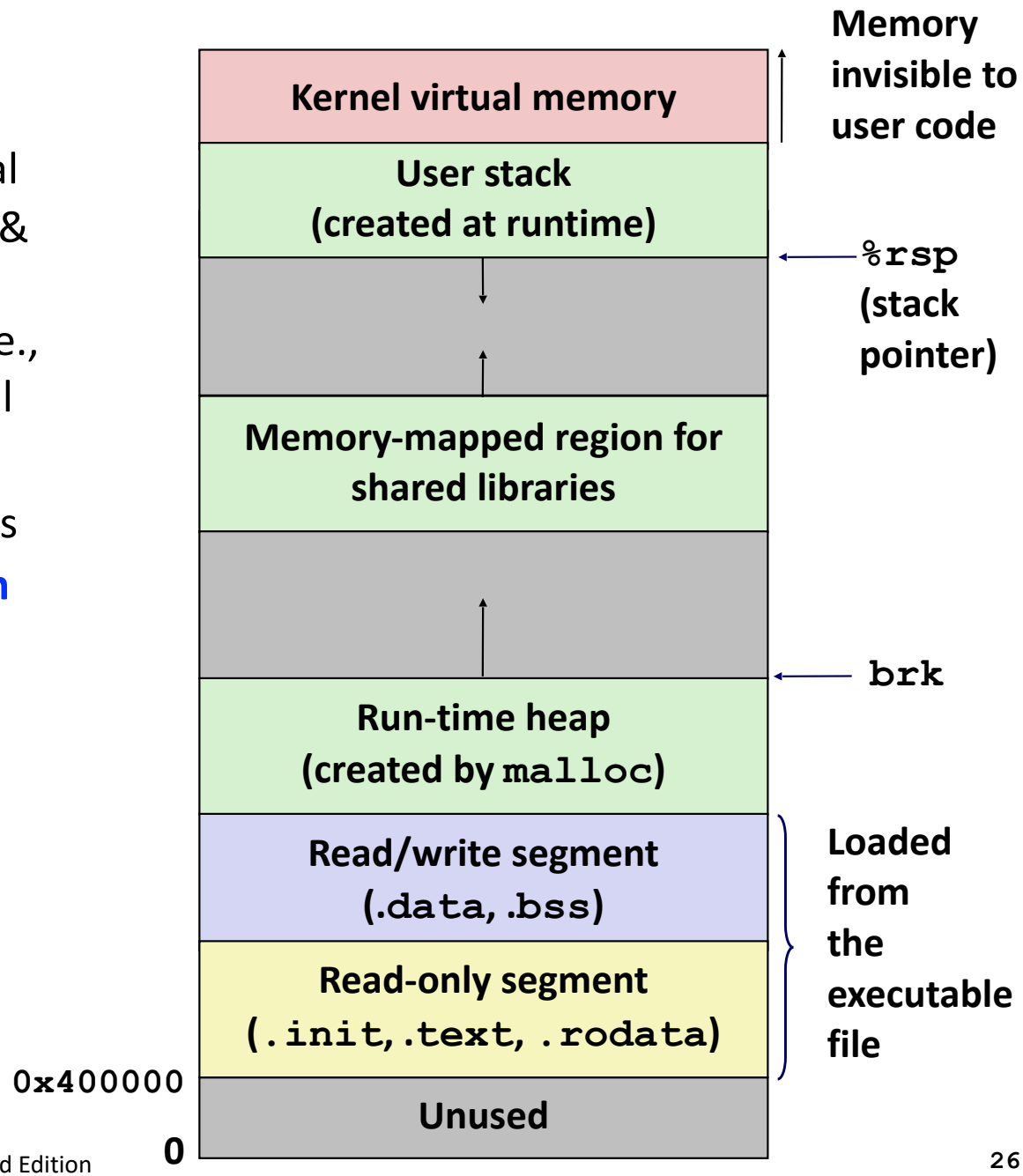
Subsequent miss will bring it into memory

- E.g., calling malloc or calloc
- E.g., when loading program for first time from disk
  - Code/data addresses are just pointers to VPs on disk
  - As code first executes, a sequence of page faults will bring in actual code/data

Virtual Memory Simplifies Loading

Loading

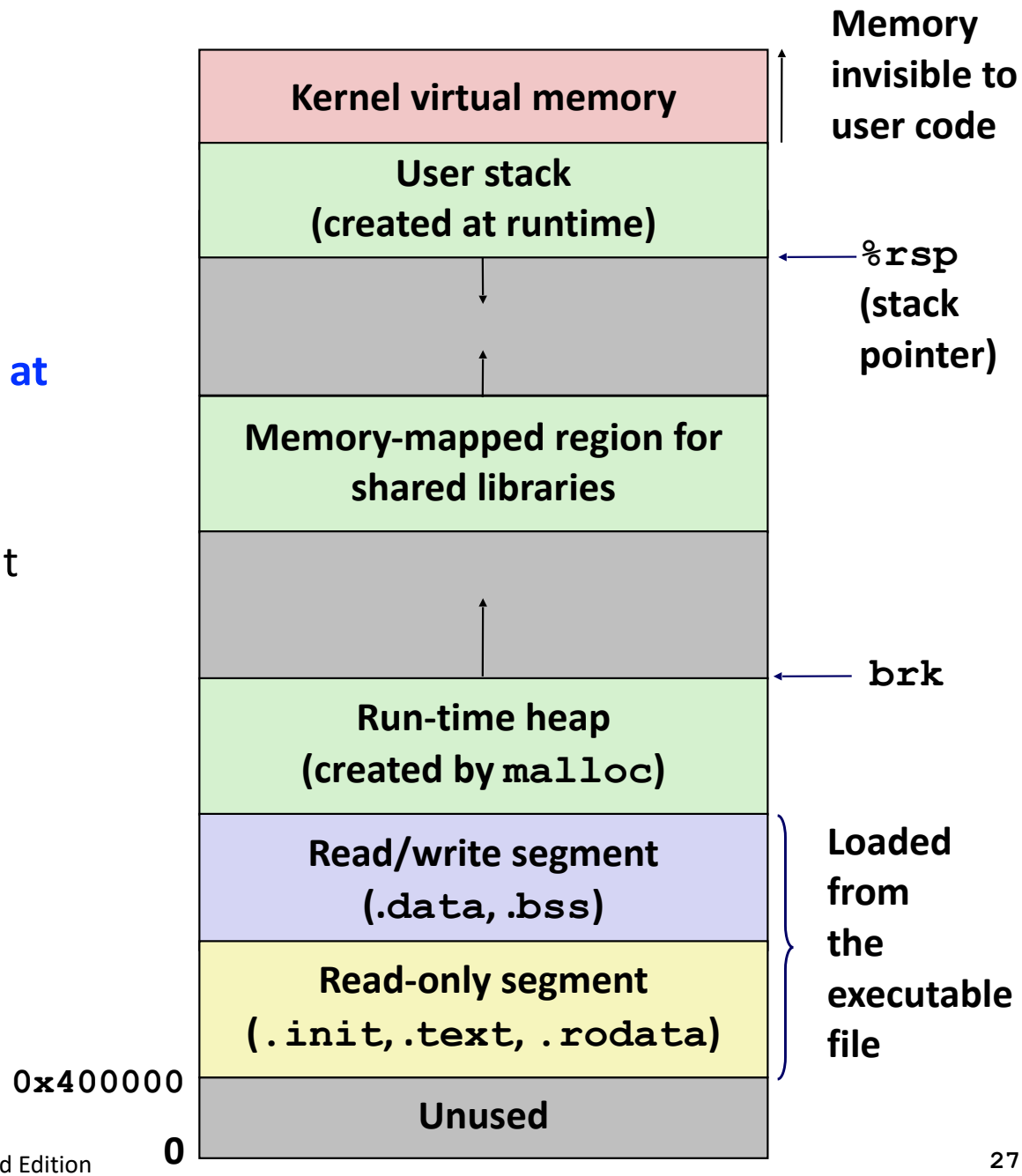
- `execve` (loader) allocates virtual pages for .text and .data sections & creates **PTE**s (**P**age **T**able **E**ntry) marked as **invalid or uncached** (i.e., only creates **pointers** to the actual code/data)
- The `.text` and `.data` sections are **then copied**, page by page, **on demand** by the virtual memory system



Virtual Memory Simplifies Linking

Linking

- Each program has **similar virtual address space**
- Code, data, and heap always **start at the same addresses**.
- Code segment always starts at virtual address **0x400000** on 64-bit linux system
- Linkers can produce fully linked executables independent of the ultimate physical location of code and data

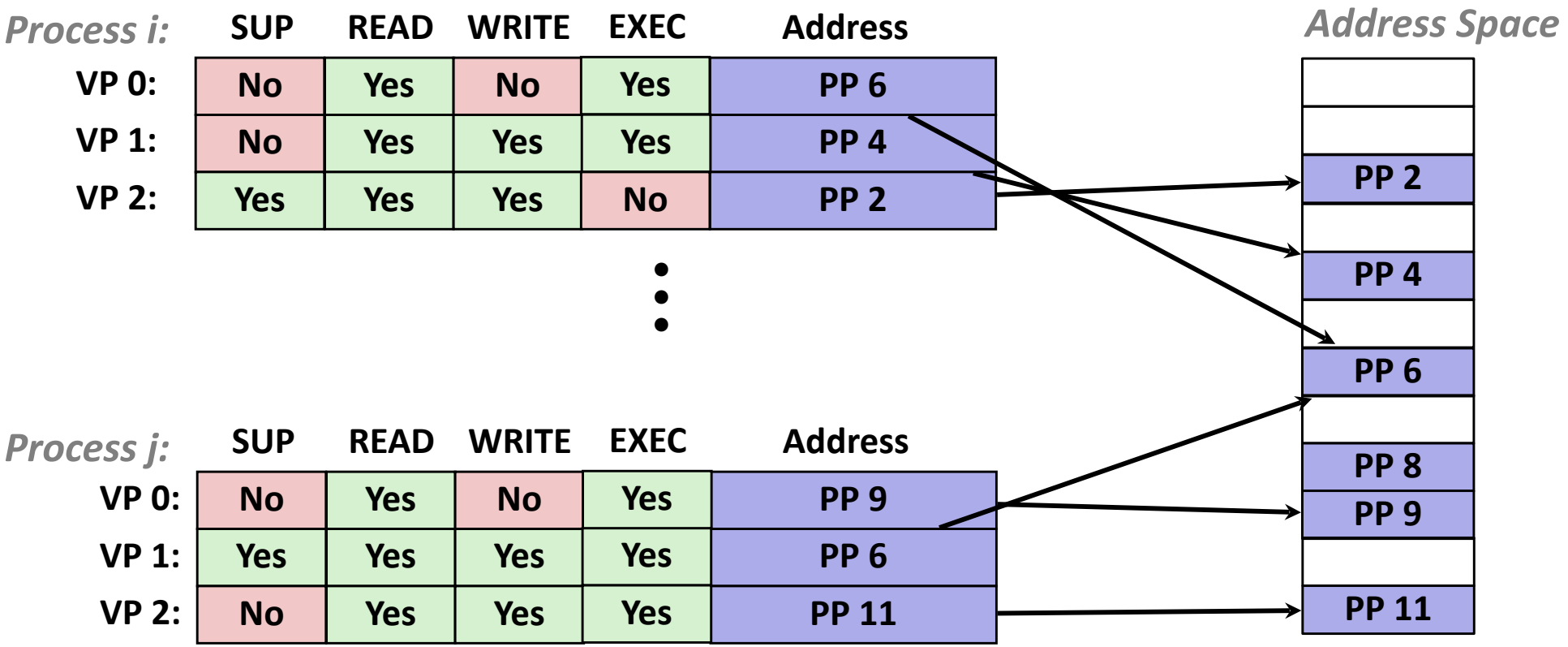


VM as a Tool for Memory Protection

Extend PTEs with permission bits

- SUP bit (If set, processes must be running in supervisor/**kernel mode** in order to access the page)
  - i.e., only when SUP = 0, processes running in **user mode** can access the page
- READ, WRITE bits (Process in user mode is allowed access to this page for read/write)
- EXEC bit (Process can execute code in this page - added in recent x86-64 systems that can mark stack un-executable)

MMU checks these bits on each access



Locality to the Rescue Again!

- **Virtual memory seems terribly inefficient due to large miss penalties, but it works because of locality.**

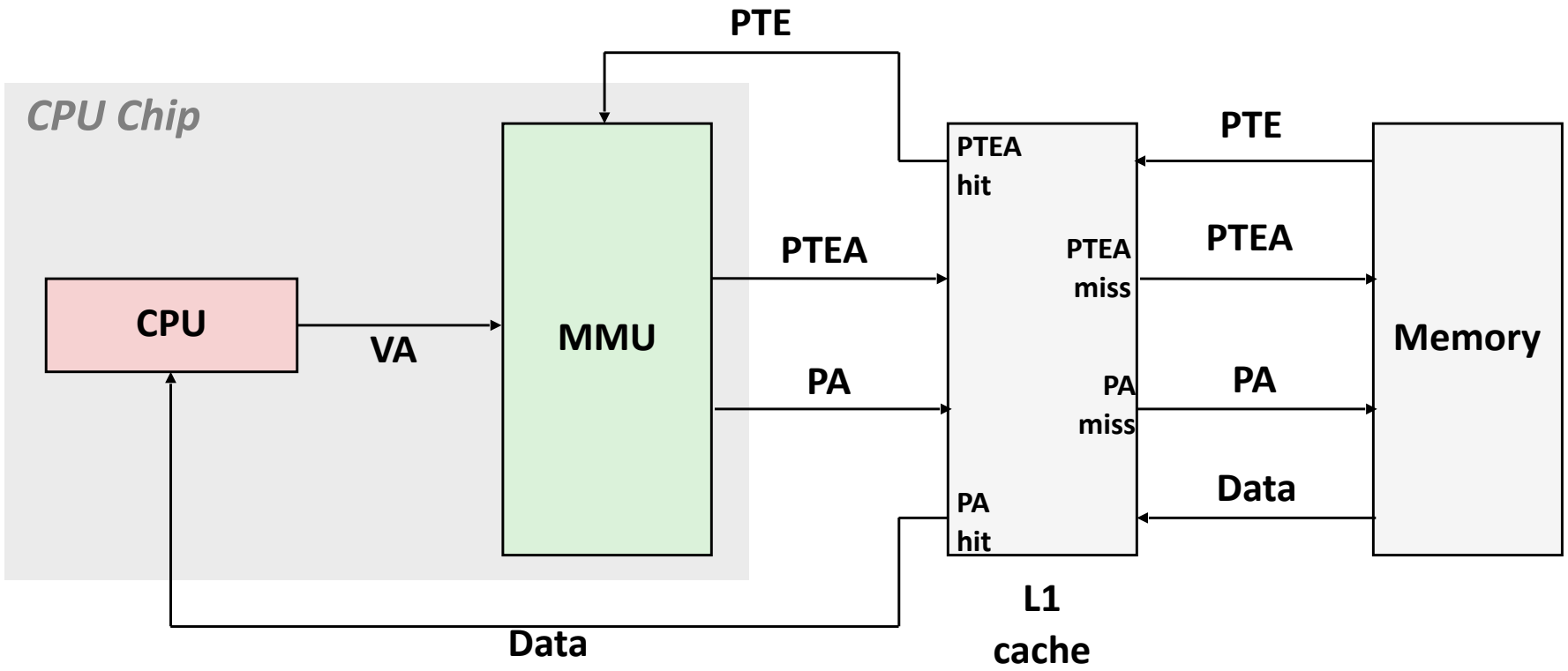
- **At any point in time, programs tend to access a set of active virtual pages called the *working set***
  - Programs with better temporal locality will have smaller working sets

- **If (working set size < main/physical memory size)**
  - Good performance for one process after compulsory misses
    - Subsequent references will result in a hit and no need to evict any victim page from physical memory

- **If (SUM(working set sizes) > main/physical memory size)**
  - *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously

Integrating VM and SRAM Cache

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



In physically addressed SRAM caches, the MMU first translates VA to PA then a cache lookup is performed using the PA

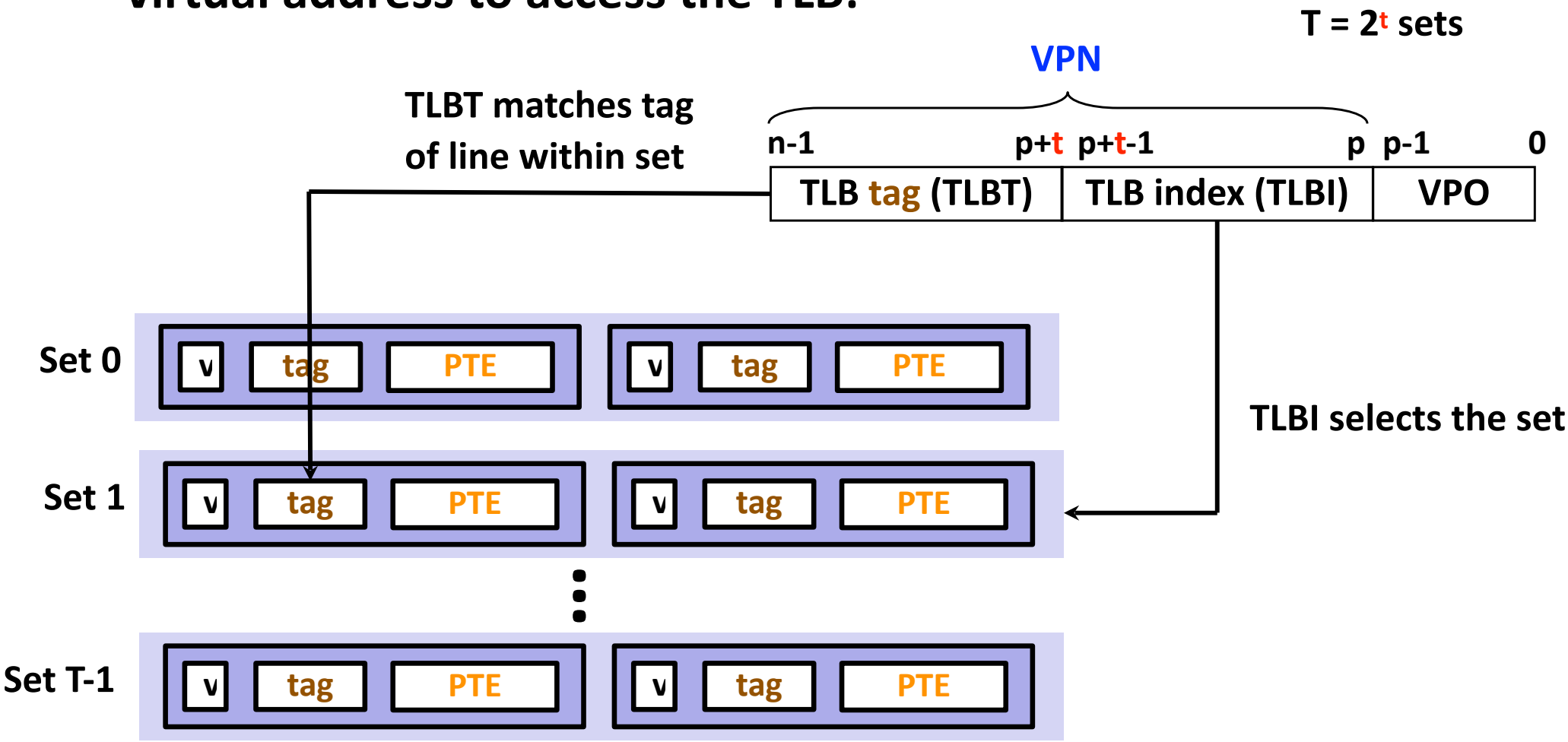
PTE is a data word, and thus can be cached

Speeding up Translation with a TLB

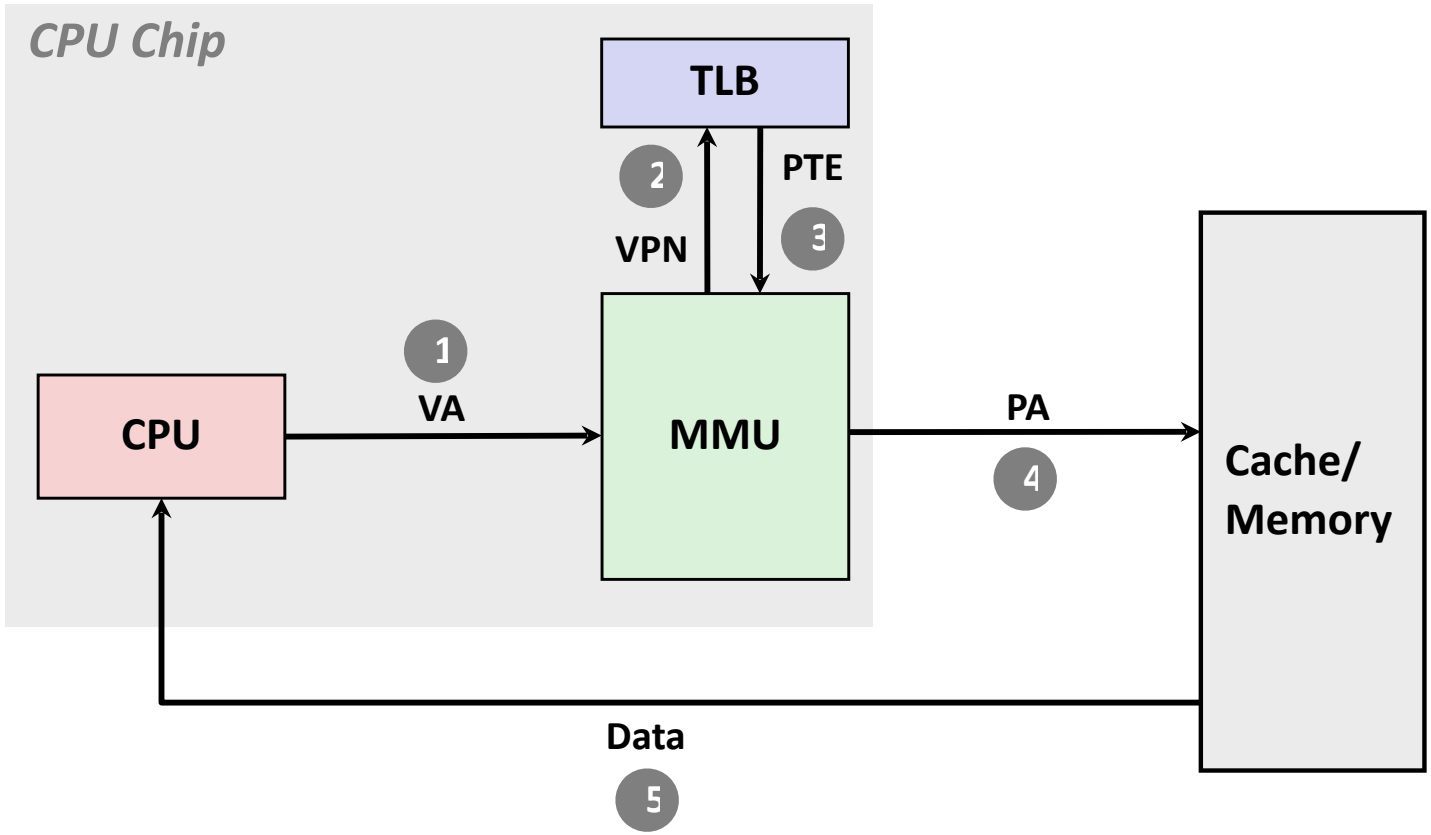
- **Page table entries (PTEs) are cached in L1 like any other memory word**
  - PTEs may be evicted by other data references
  - **Problem**: PTE hit **still requires a small L1 cache lookup delay**
    - An additional cache access (few extra cycles) for looking up the PTE in the L1 cache
- **Solution: *Translation Lookaside Buffer* (TLB)**
  - Eliminate this small L1 delay by using a **small set-associative dedicated hardware cache in MMU for page table entries**
    - TLB hit = page table not consulted
  - Steps of mapping virtual page numbers to physical page numbers now all take place inside the MMU, and thus **much faster**
  - Contains **complete page table entries for small number of pages.**
    - Can be fairly small: one TLB entry covers 4k or more

Accessing the TLB

- MMU uses the VPN (Virtual Page Number) portion of the virtual address to access the TLB:

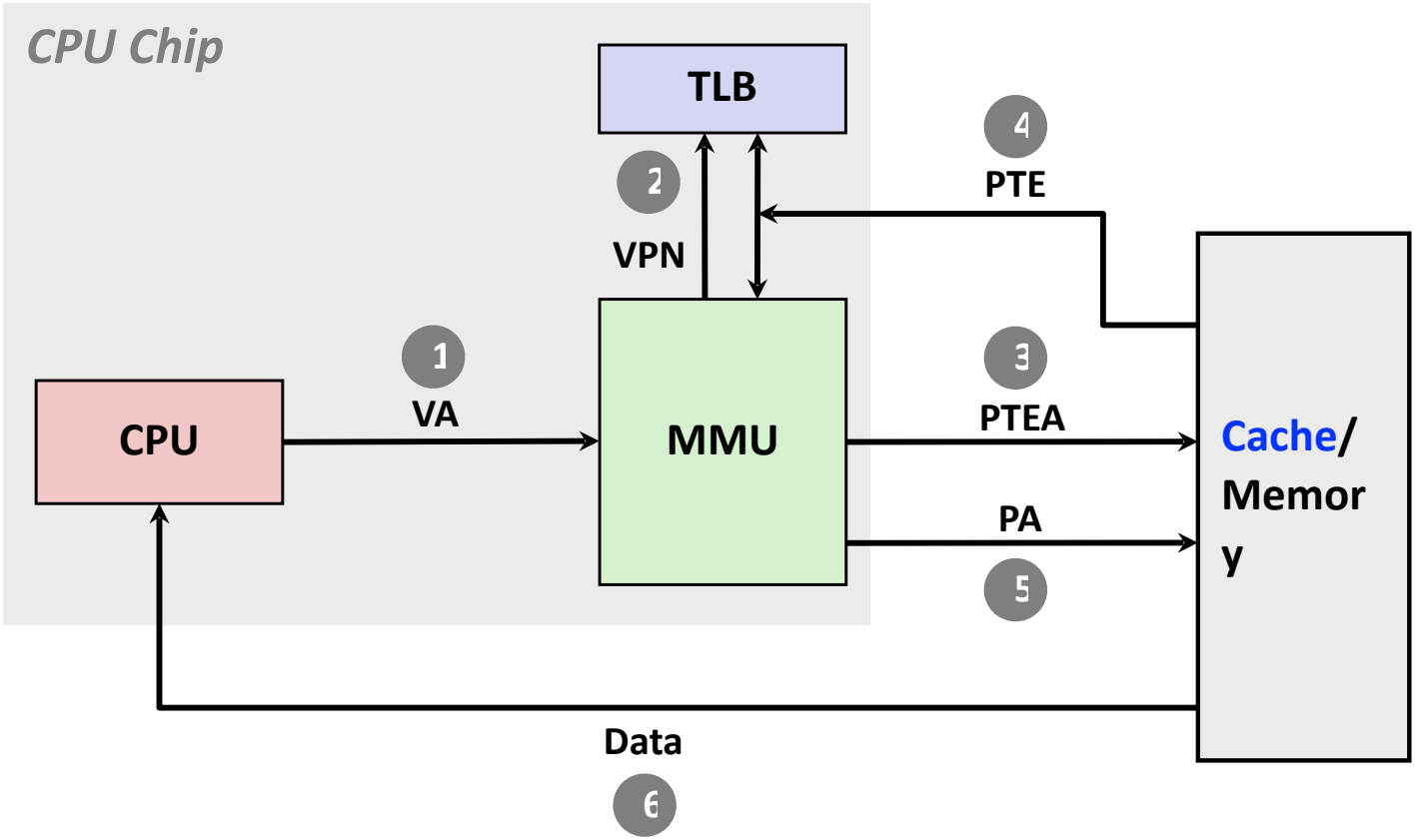


TLB Hit



A TLB hit eliminates a memory access

TLB Miss



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

MMU first checks if PTE is in **L1 cache** and that can incur only a **small delay**

Fortunately, TLB misses are rare.

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)

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

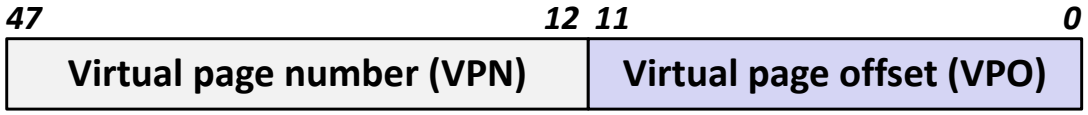
Components of the physical address (PA)

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

Multi-Level Page Tables

Suppose:

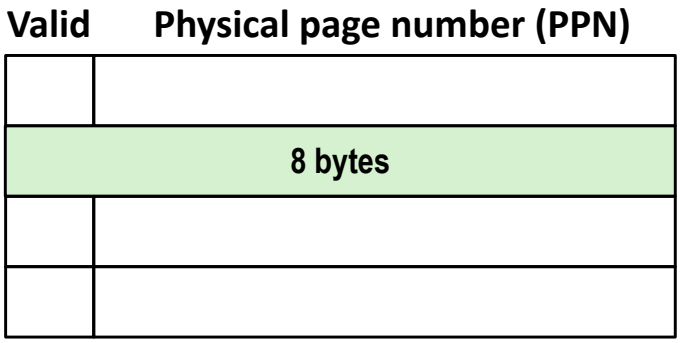
- 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE



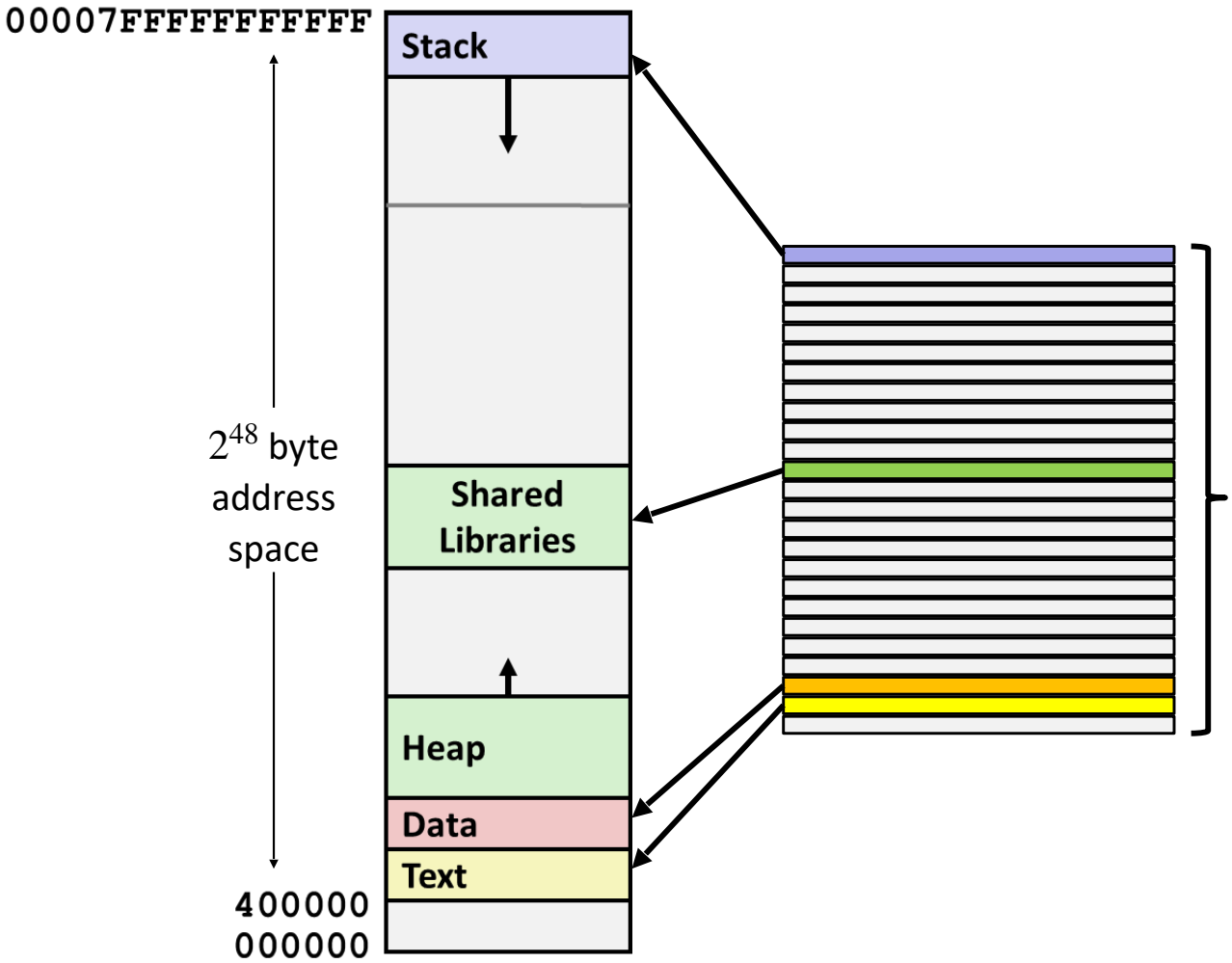
Problem:

- Would need a 512 GB page table resident in memory at any given time (even if process referenced small chunk of virtual address space)!
  - $2^{48} * 2^{-12} * 2^{3} = 2^{39}$ bytes

Page table



The problem (with one-level page tables) - Program Memory View



One 64-bit array element for each 4096-byte (4KB) page

$$= \frac{2^{48}}{4096} \cdot 8 \text{ bytes}$$

$= 2^{39}$ bytes

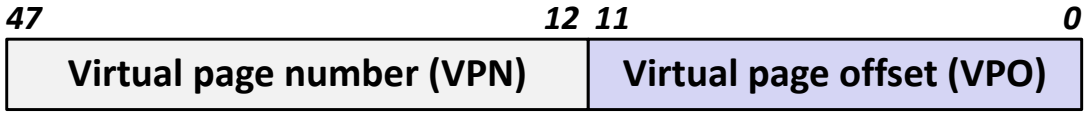
= 512 **gigabytes**

for **one** page table

Multi-Level Page Tables

Suppose:

- 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE



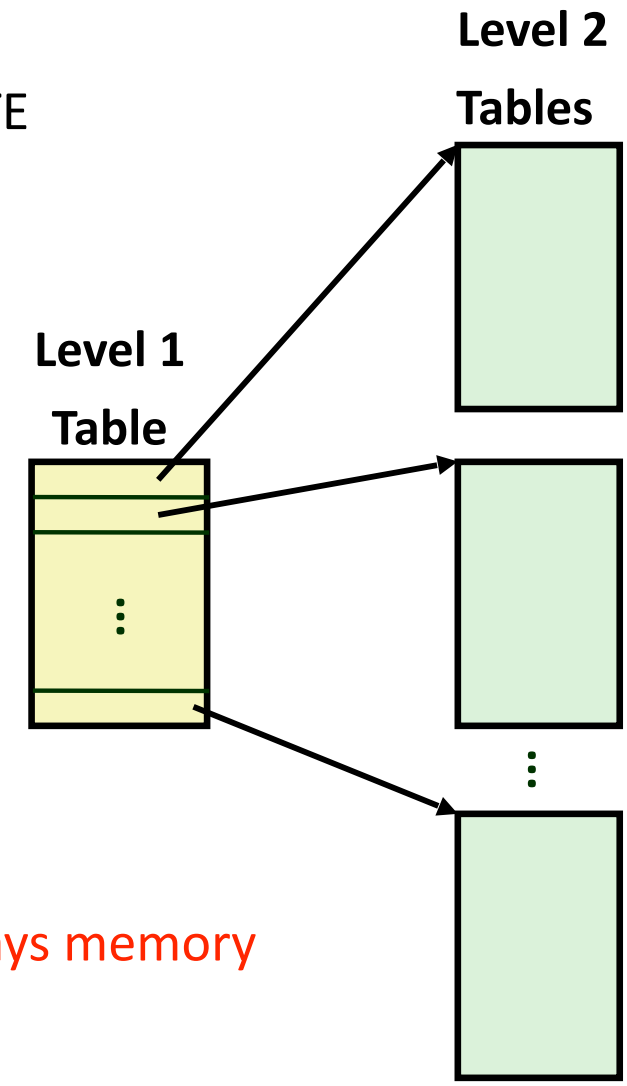
Problem:

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

Common solution: Multi-level page table

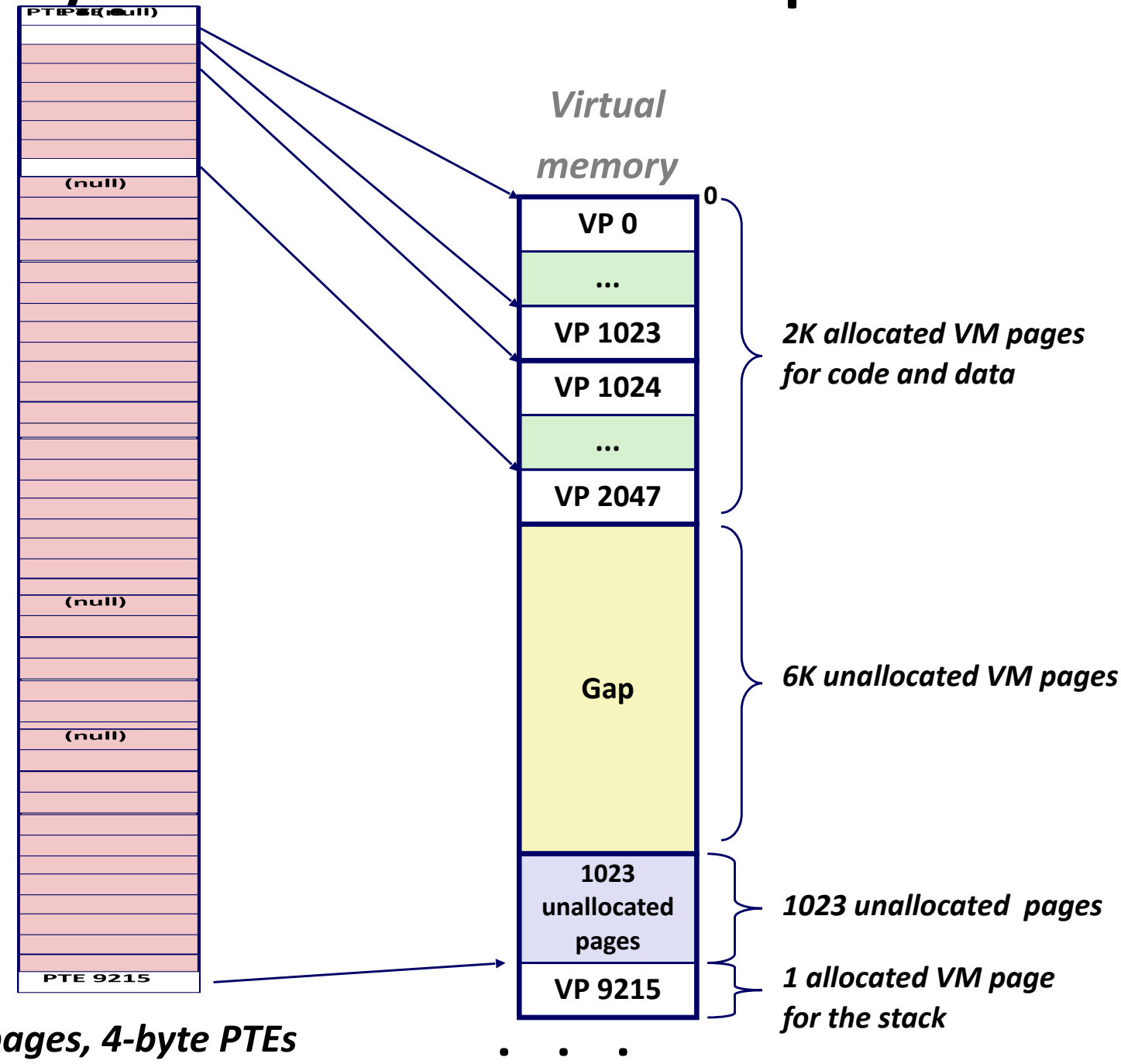
Example: 2-level page table

- 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)



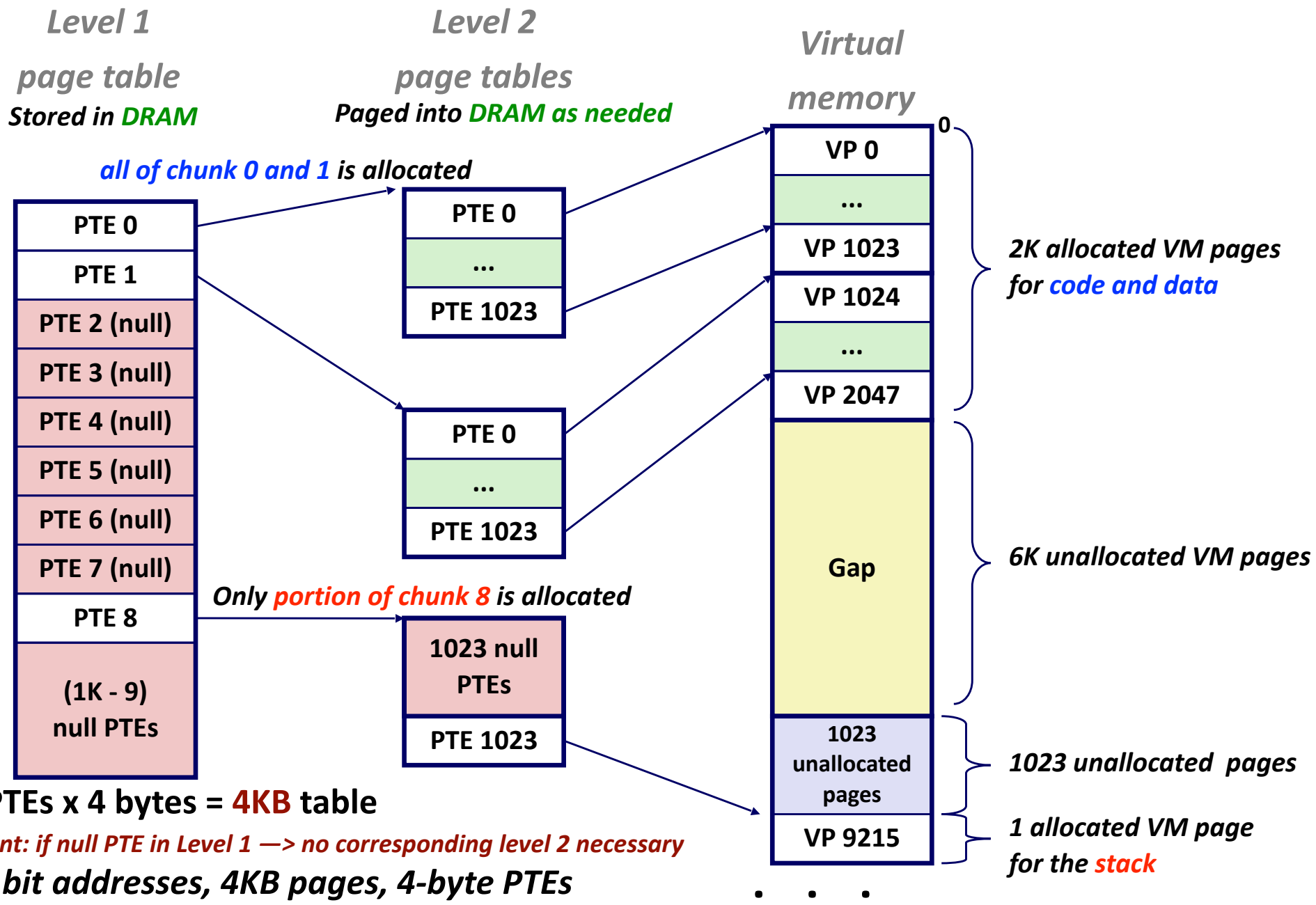
Example Program/Process VM: We have a problem

2^{20} Entries of
4 bytes each
= 4MB table



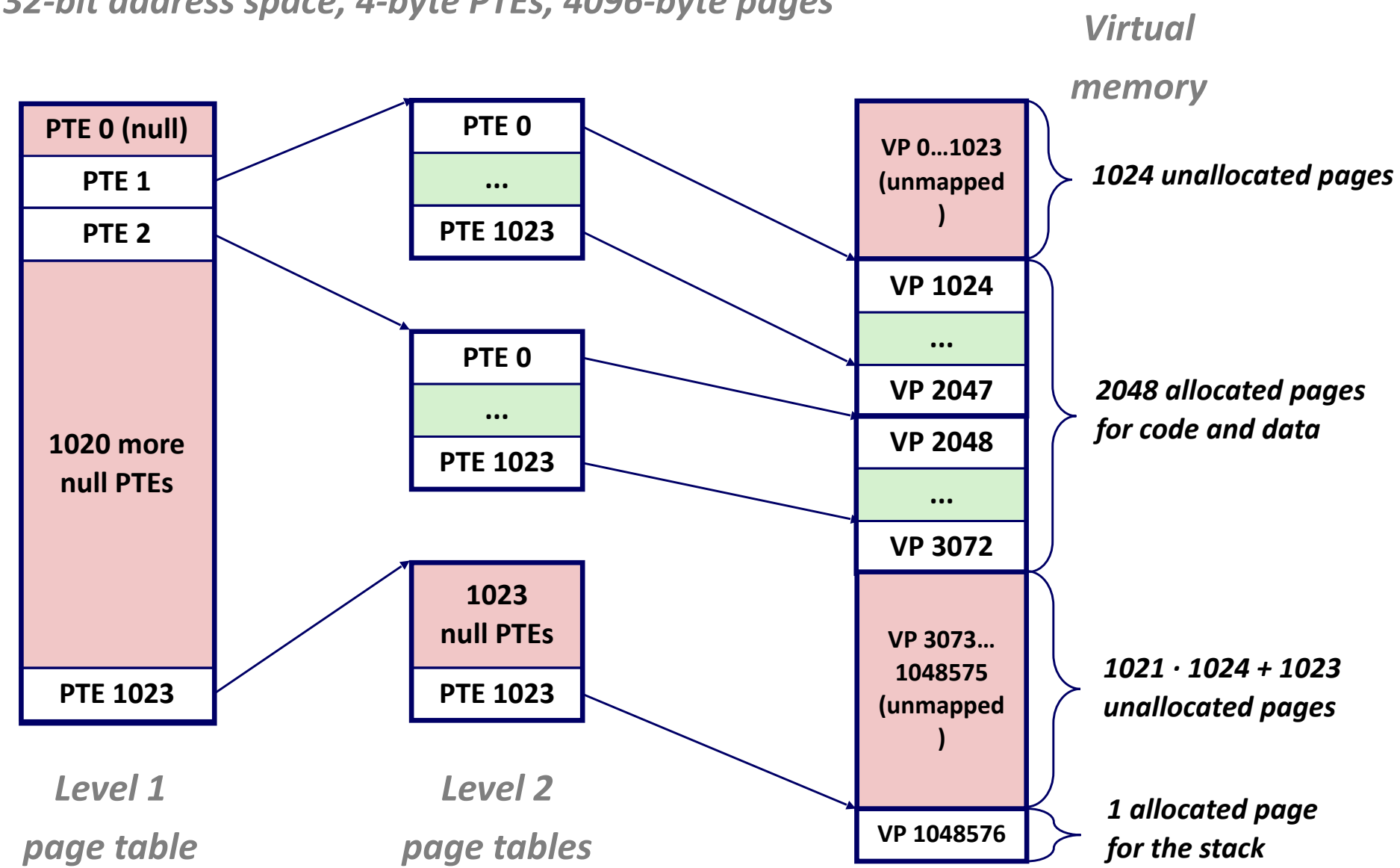
32 bit addresses, 4KB pages, 4-byte PTEs

Example Program/Process VM: A Two-Level Page Table Hierarchy

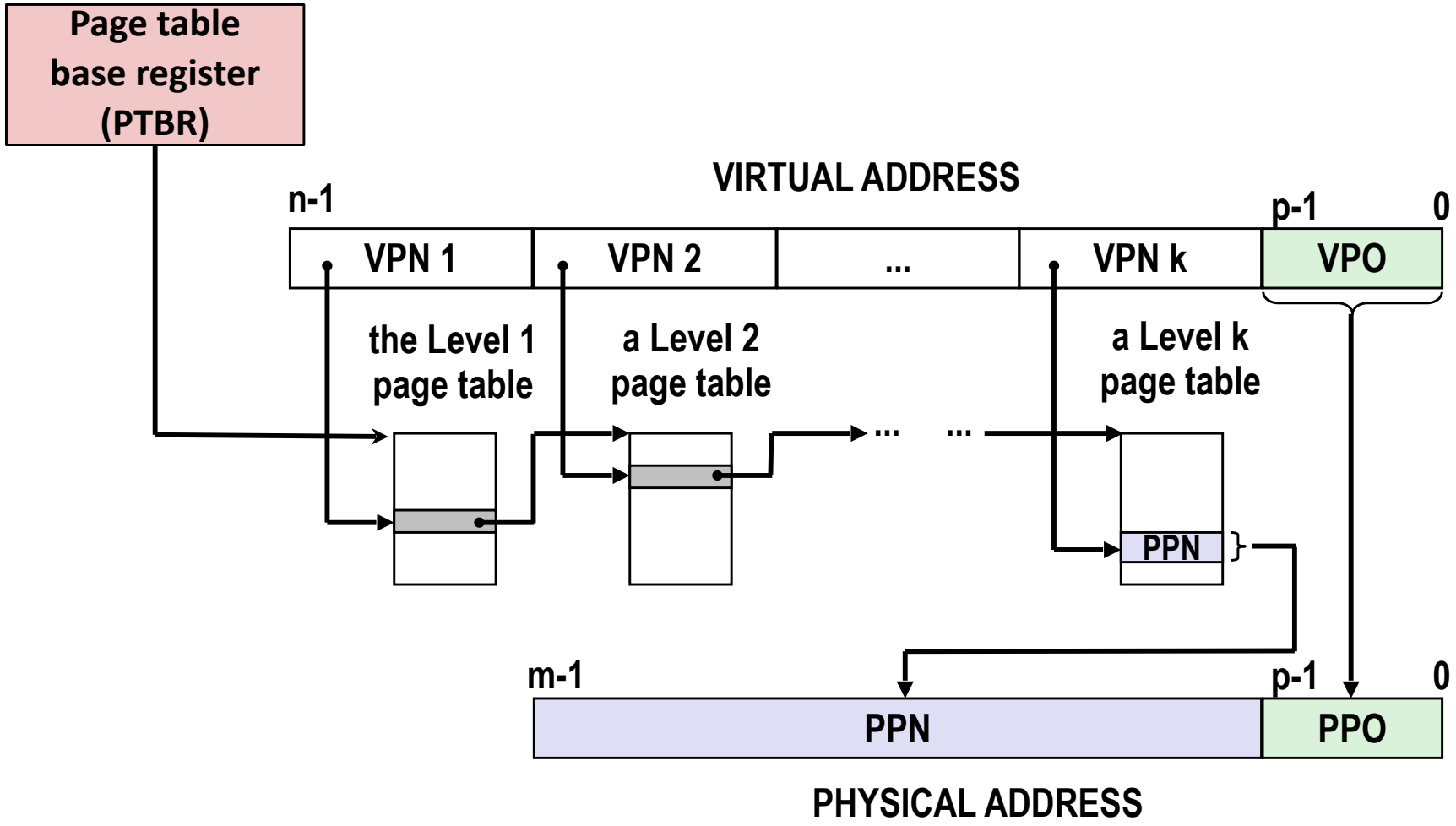


Another Example Program VM Layout

32-bit address space, 4-byte PTEs, 4096-byte pages

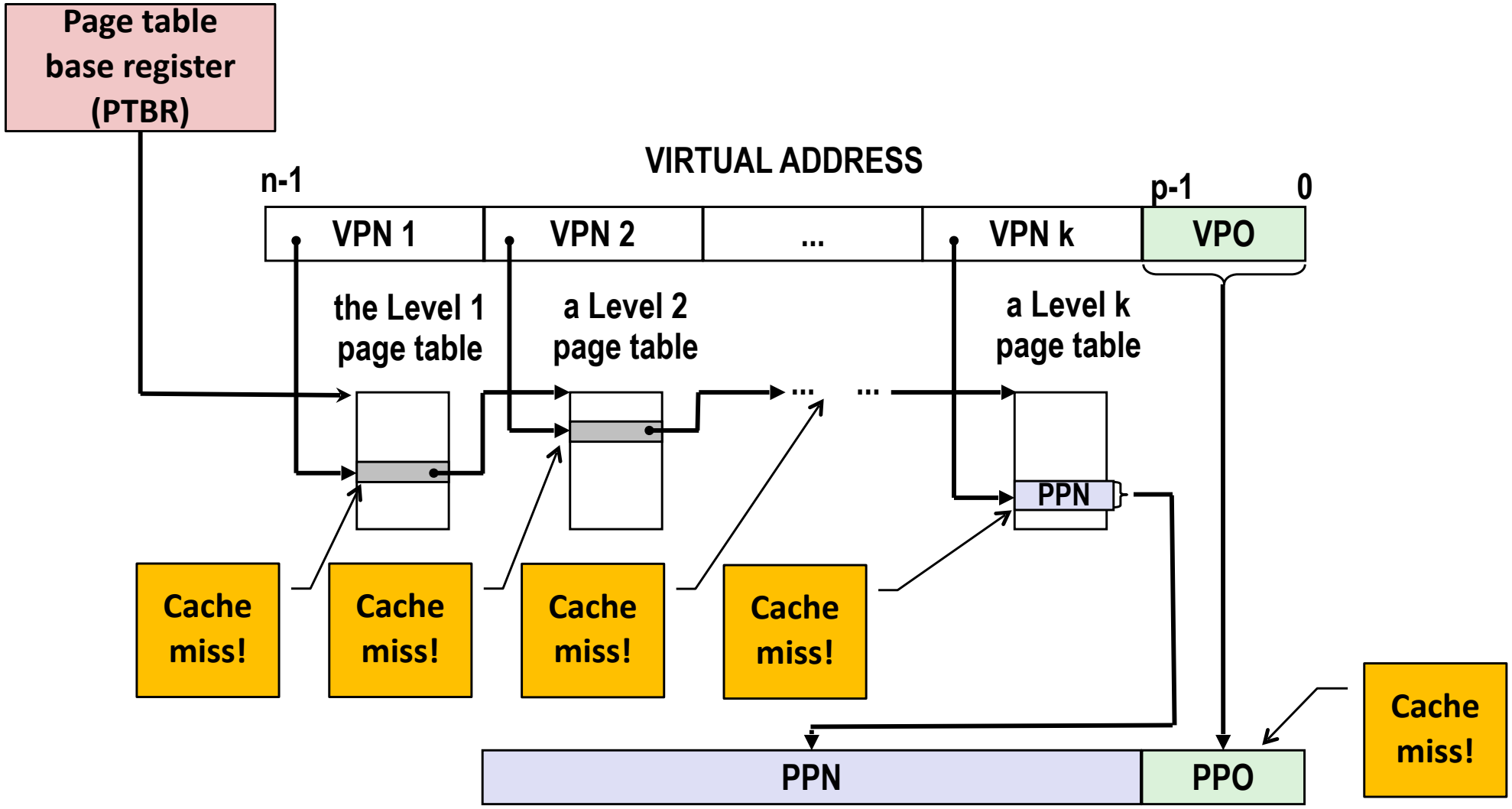


Translating with a k-level Page Table



The problem (with k-level page tables)

With k-level page table MMU must access k PTEs to construct physical address,
but since TLB caches those PTEs then it is not significantly slower than a single-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 inter-positioning point to check permissions