Introduction to Processor Architecture (EC2.204)

LECTURE 12 – VIRTUAL MEMORY (SECTION 9.1-9.3 AND 9.6)

Deepak Gangadharan
Computer Systems Group (CSG), IIIT Hyderabad

Slide Contents: Adapted from slides by Randal Bryant

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”

Process provides each program with two key abstractions:

- Logical control flow
  - Each program seems to have exclusive use of the CPU
- Private virtual address space
  - Each program seems to have exclusive use of main memory

How are these Illusions maintained?

- Process executions interleaved (multitasking) or run on separate cores
- Address spaces managed by virtual memory system

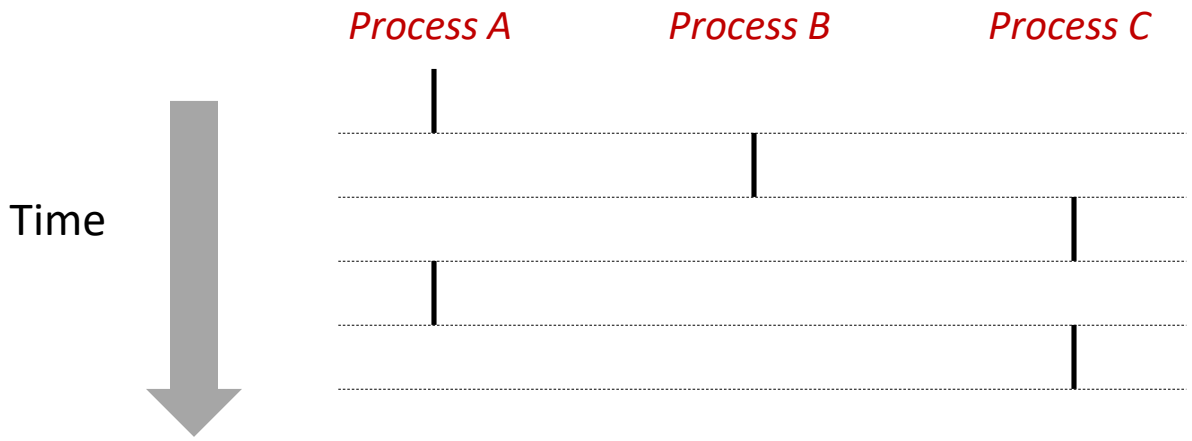
Concurrent Processes

Two processes *run concurrently* (*are concurrent)* if their flows overlap in time

Otherwise, they are *sequential*

Examples (running on single core):

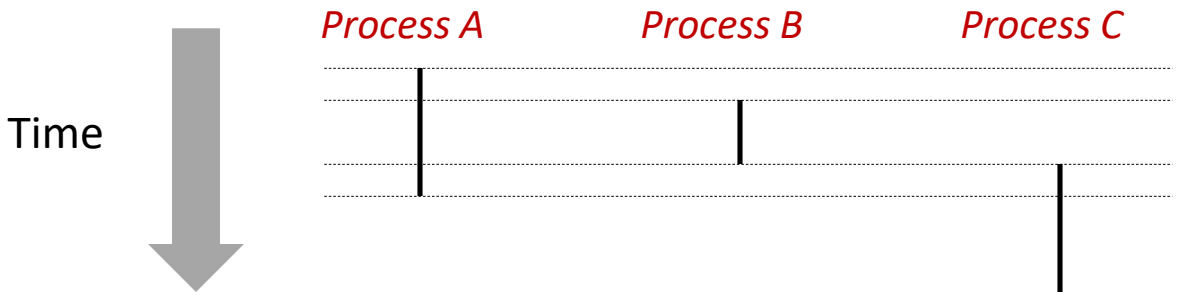
- Concurrent: A & B, A & C
- Sequential: B & C



User View of Concurrent Processes

Control flows for concurrent processes are physically disjoint in time

However, we can think of concurrent processes are running in parallel with each other



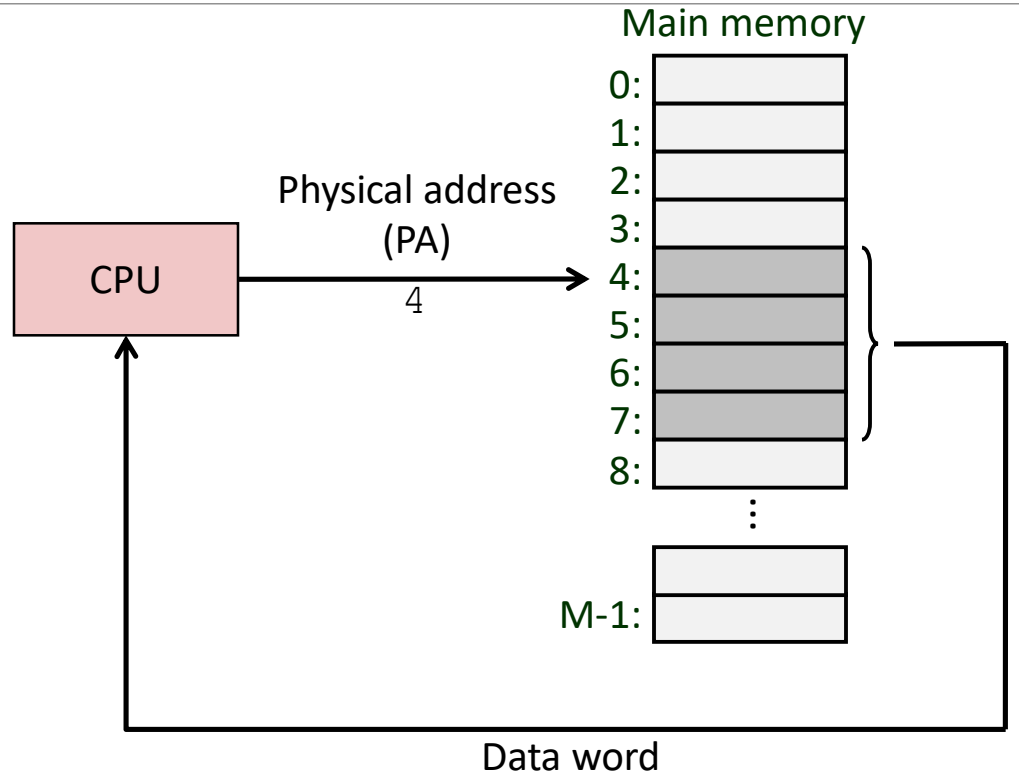
Topics

Address spaces

VM as a tool for caching

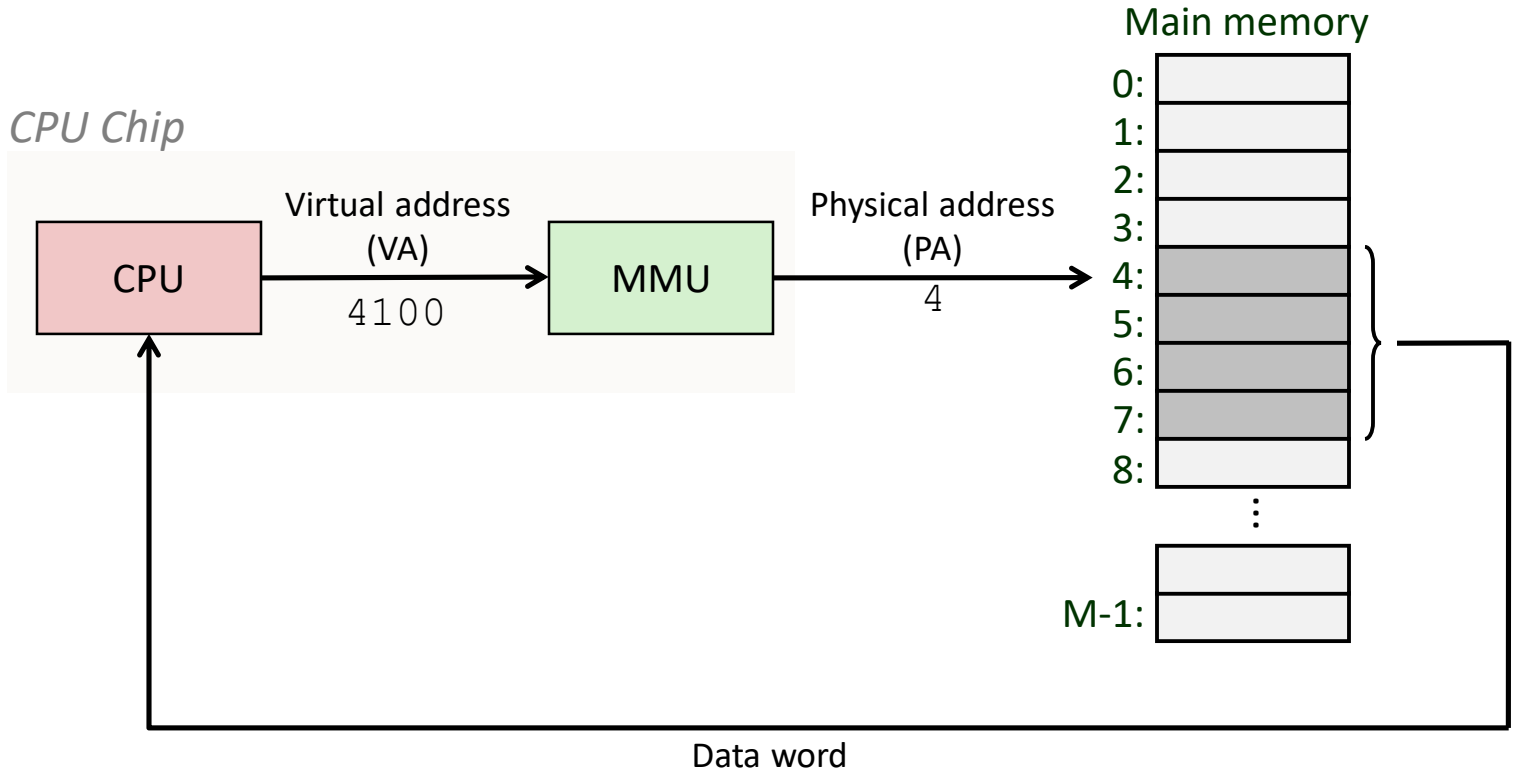
Address translation

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, desktops, and laptops

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 … }

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}

Clean distinction between data (bytes) and their attributes (addresses)

Each object 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 the 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

Topics

Address spaces

VM as a tool for caching

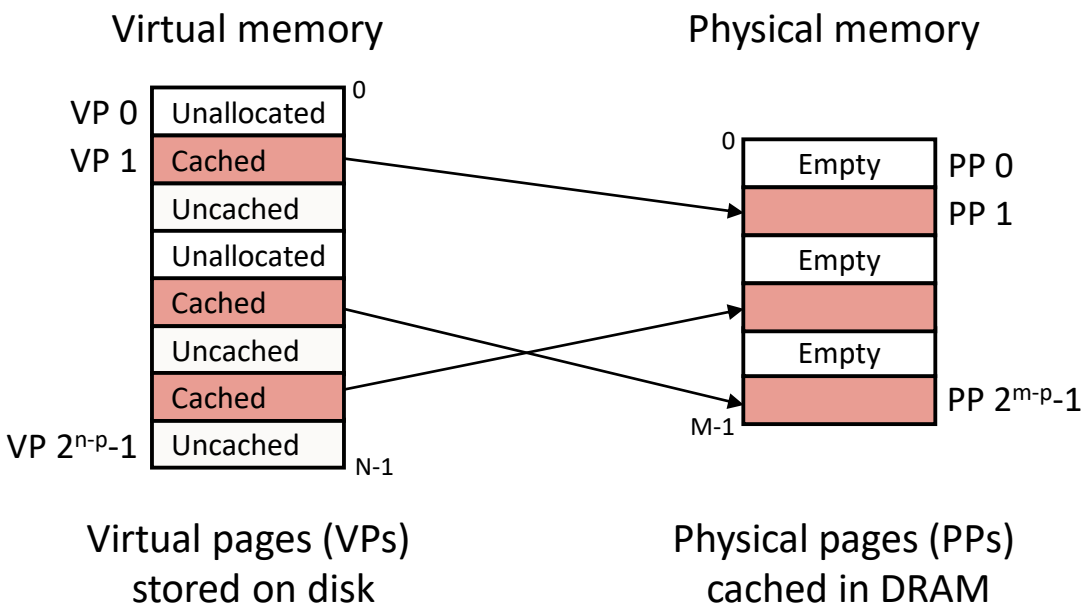
Address translation

VM as a Tool for Caching

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

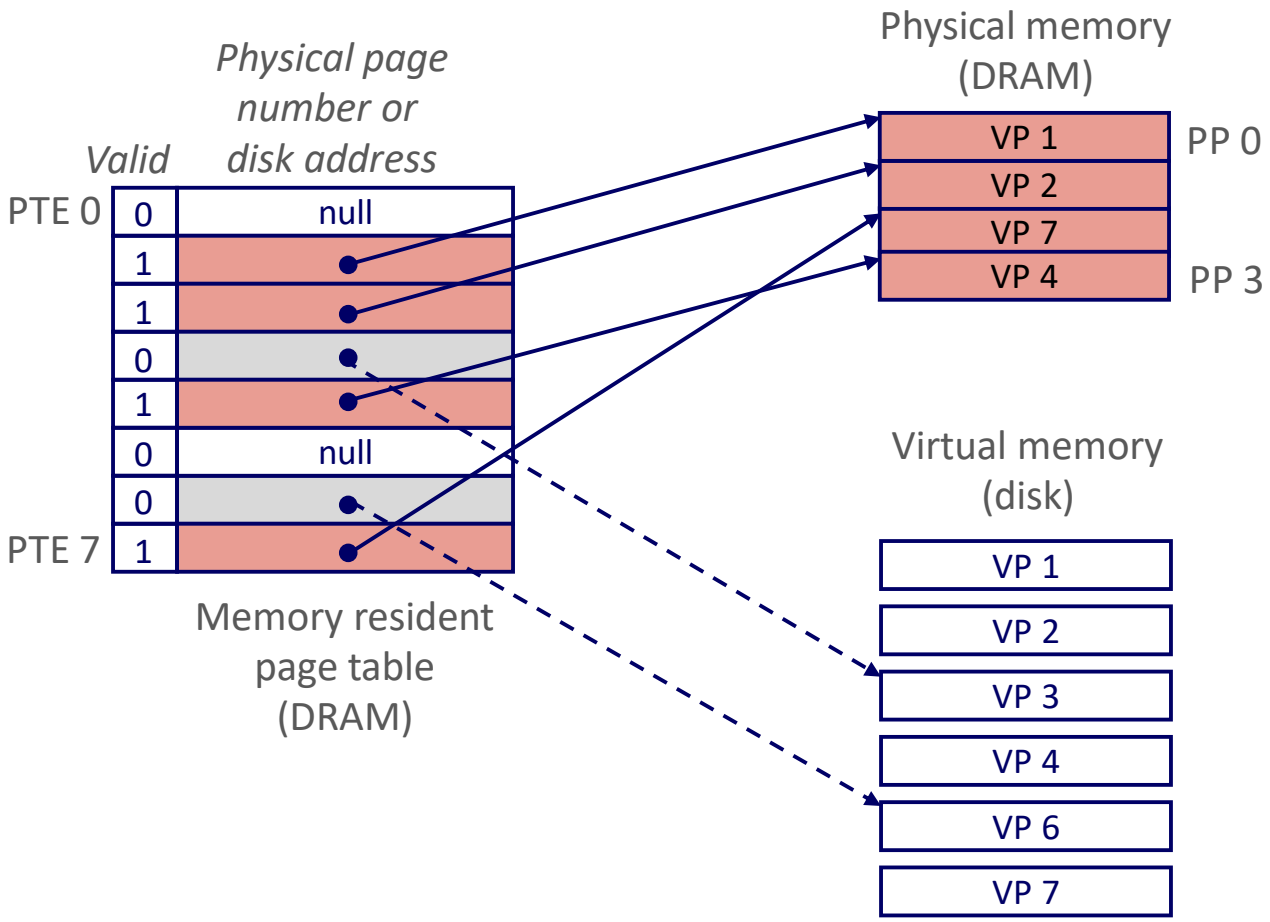
DRAM cache organization driven by the enormous miss penalty

- DRAM is about ***10x*** slower than SRAM
- Disk is about ***10,000x*** slower than DRAM

Consequences

- Large page (block) size: typically 4-8 KB, sometimes 4 MB
- Fully associative
  - Any VP can be placed in any 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

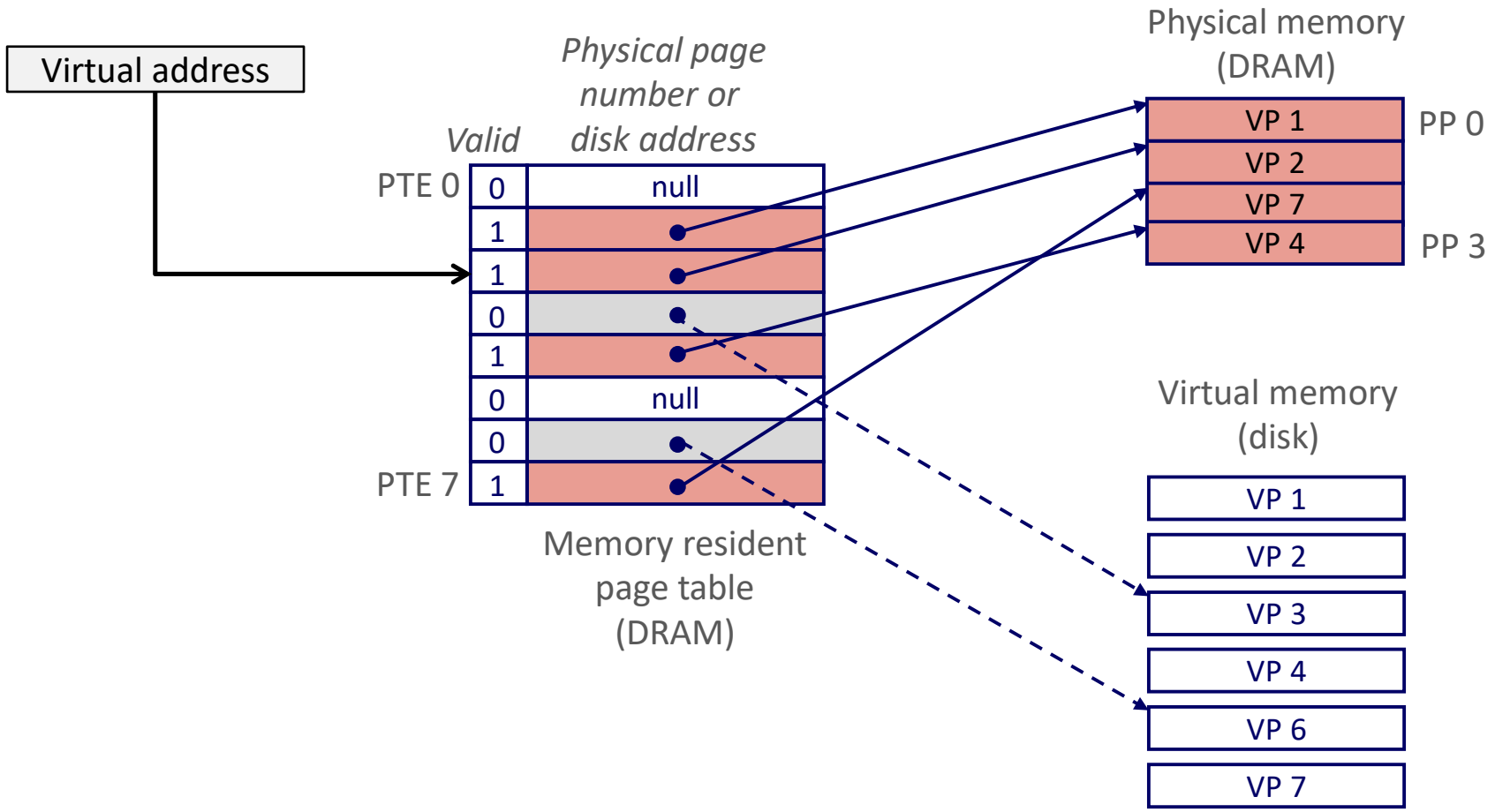
Page Tables



A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages.

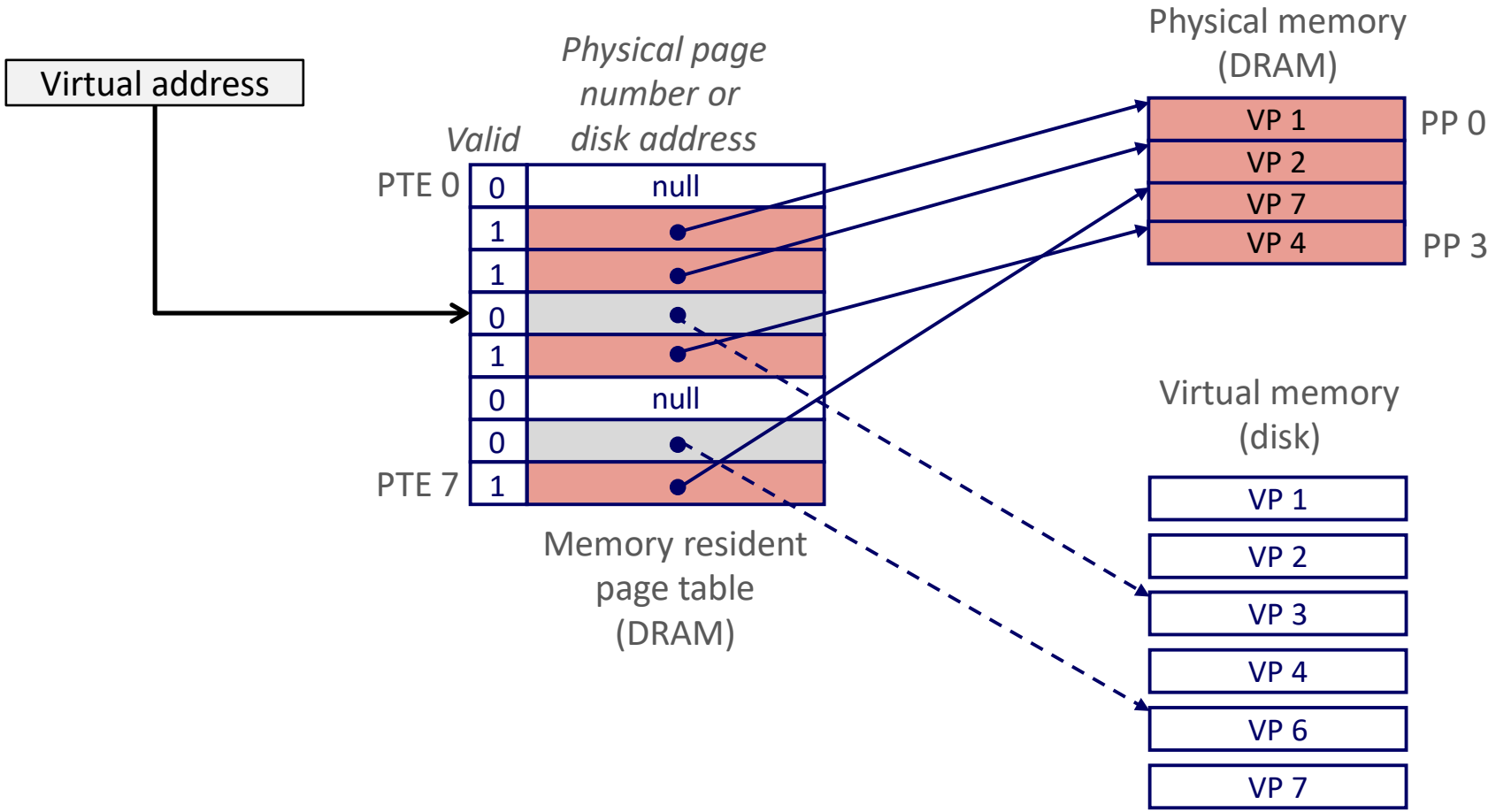
- Per-process kernel data structure in DRAM

Page Hit



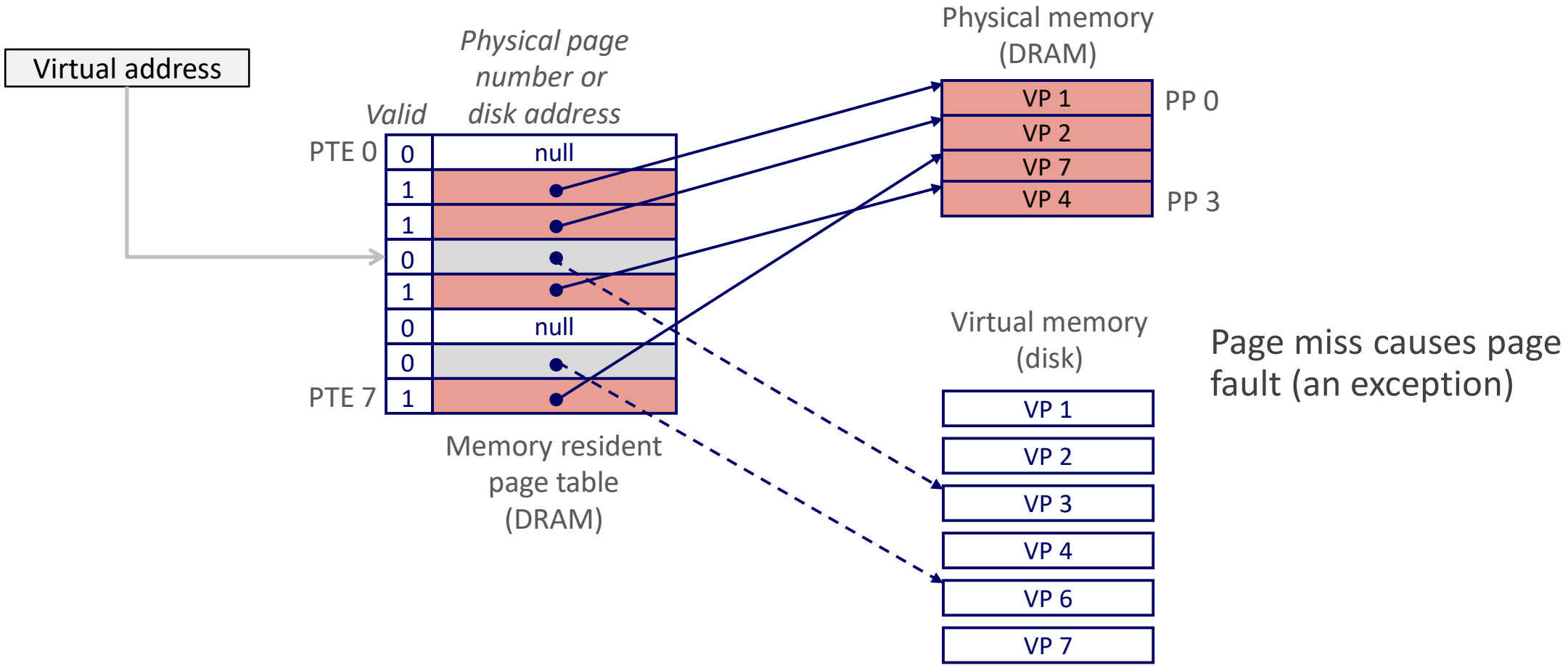
Page hit: reference to VM word that is in physical memory (DRAM cache hit)

Page Fault



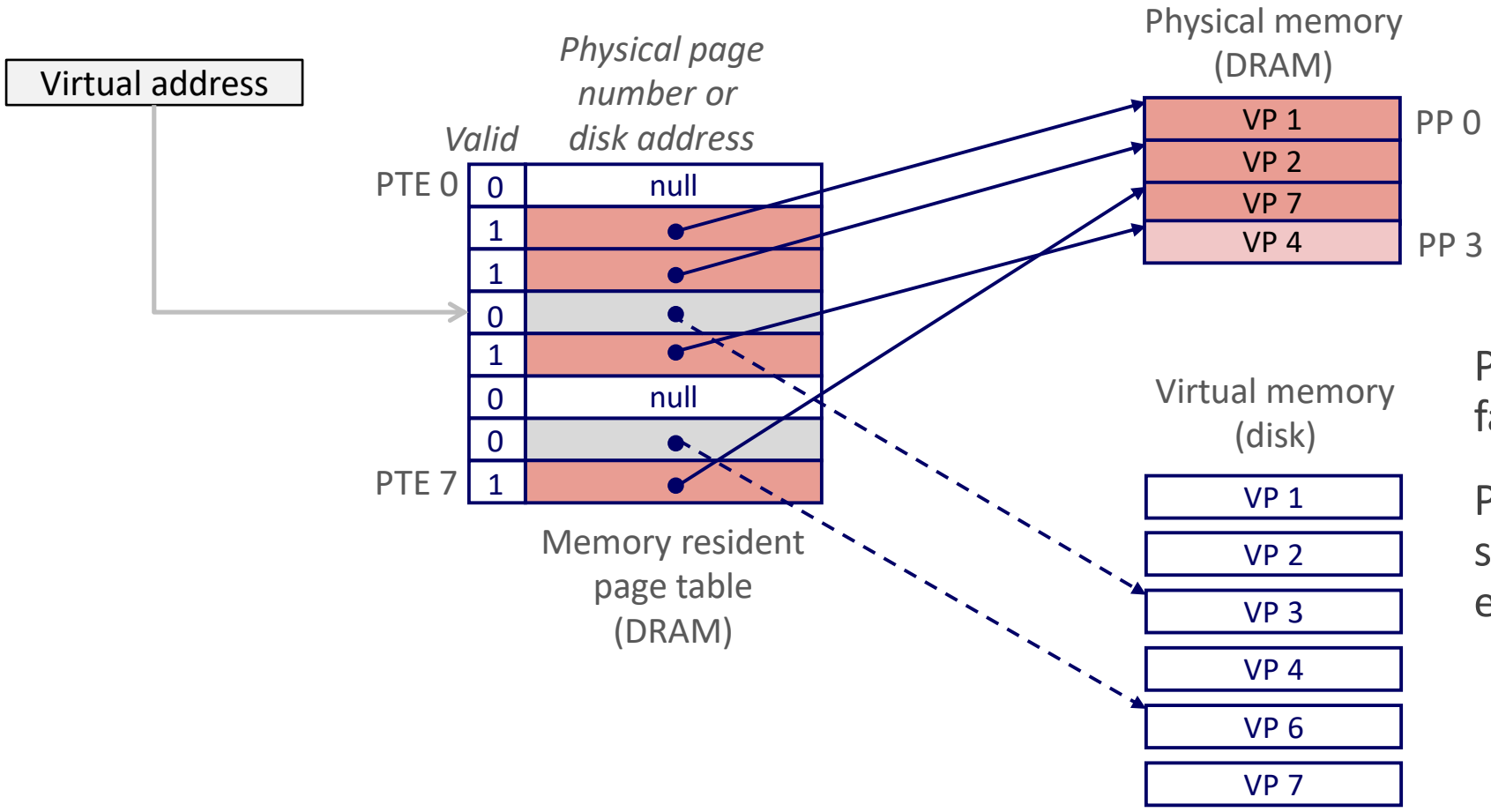
Page fault: reference to VM word that is not in physical memory (DRAM cache miss)

Handling Page Fault



Page miss causes page fault (an exception)

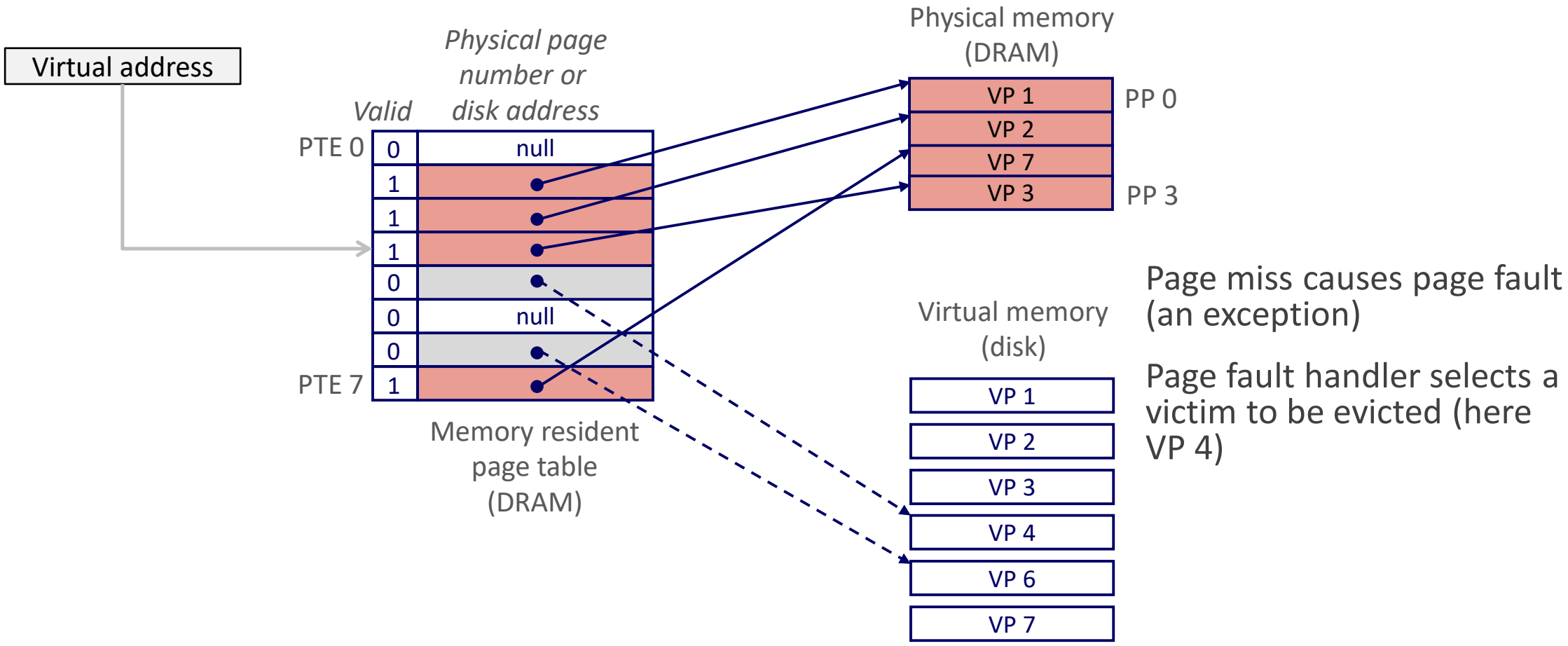
Handling Page Fault



Page miss causes page fault (an exception)

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

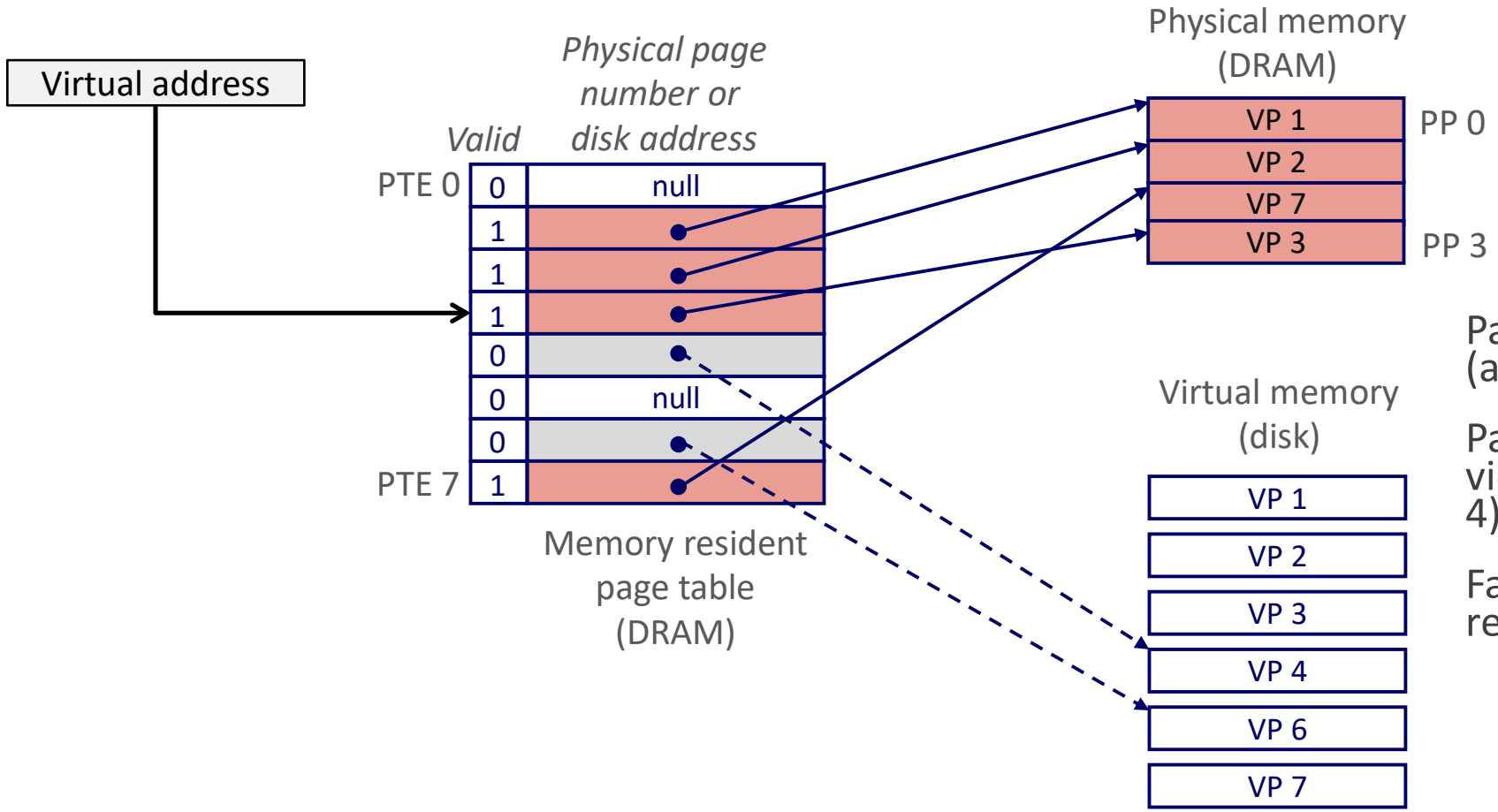
Handling Page Fault



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)

Faulting instruction is restarted: page hit!

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 swapped (copied) in and out continuously

Topics

Address spaces

VM as a tool for caching

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 → P U {∅}***
- For virtual address ***a***:
  - ***MAP(a) = a'*** if data at virtual address ***a*** is at physical address ***a'*** in ***P***
  - ***MAP(a) =*** ∅ 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)

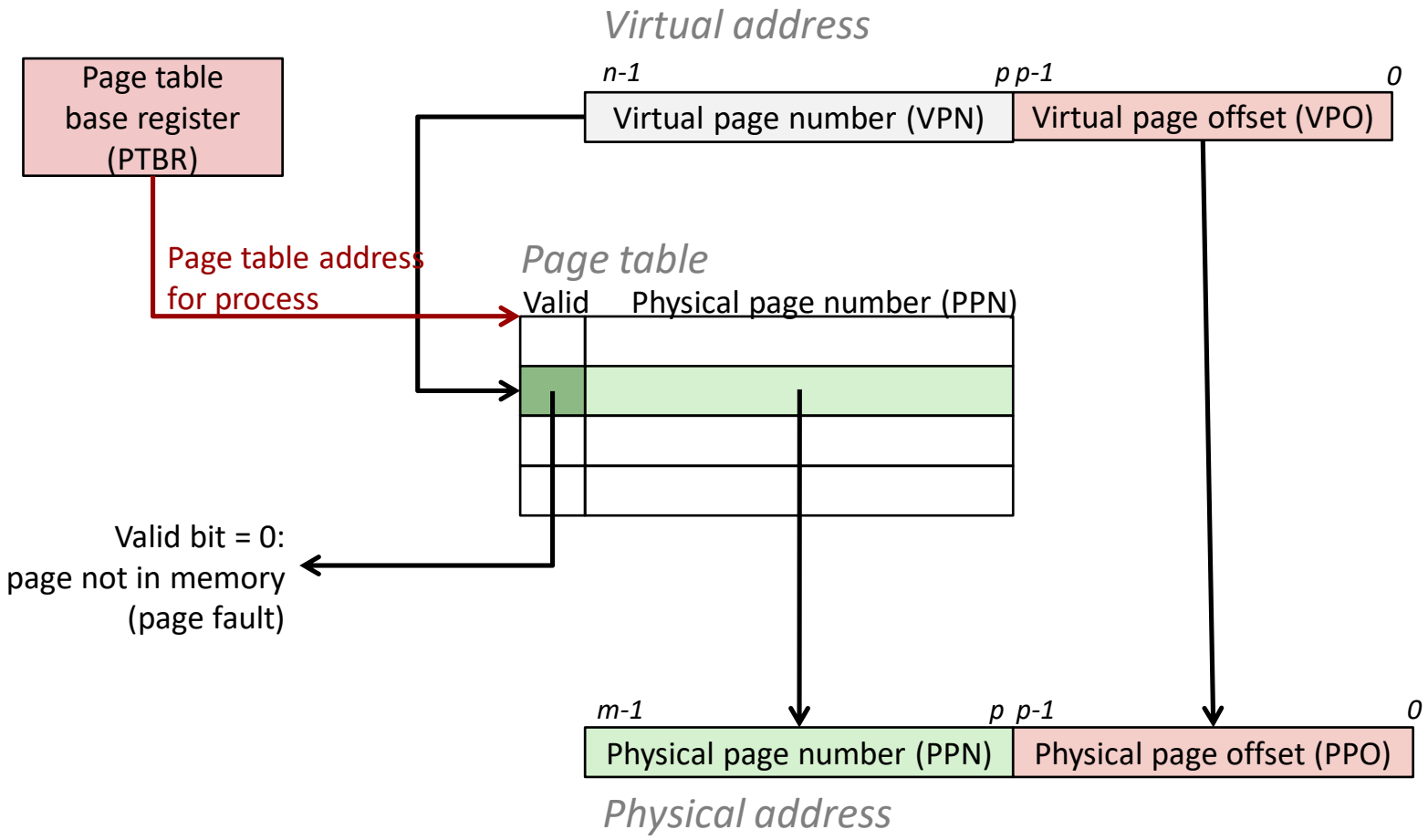
Components of the virtual address (VA)

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

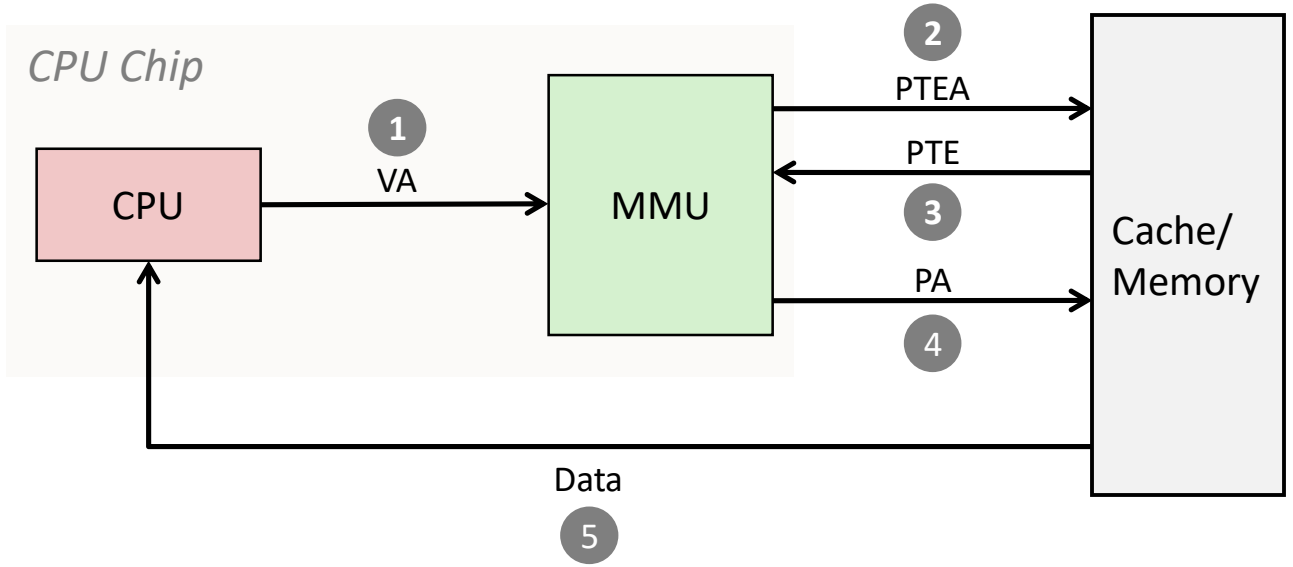
Components of the physical address (PA)

- **PPO**: Physical page offset (same as VPO)
- **PPN:** Physical page number
- **CO**: Byte offset within cache line
- **CI:** Cache index
- **CT**: Cache tag

Address Translation With a Page Table



Address Translation: Page Hit



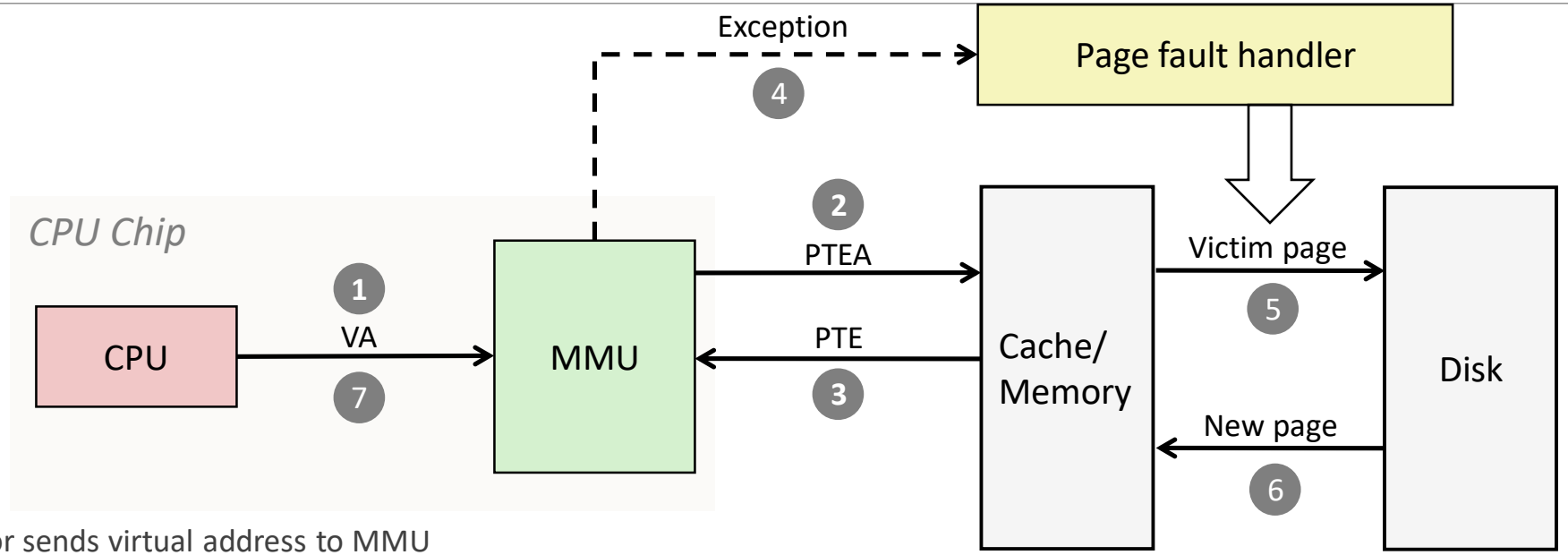
1) Processor sends virtual address to MMU

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

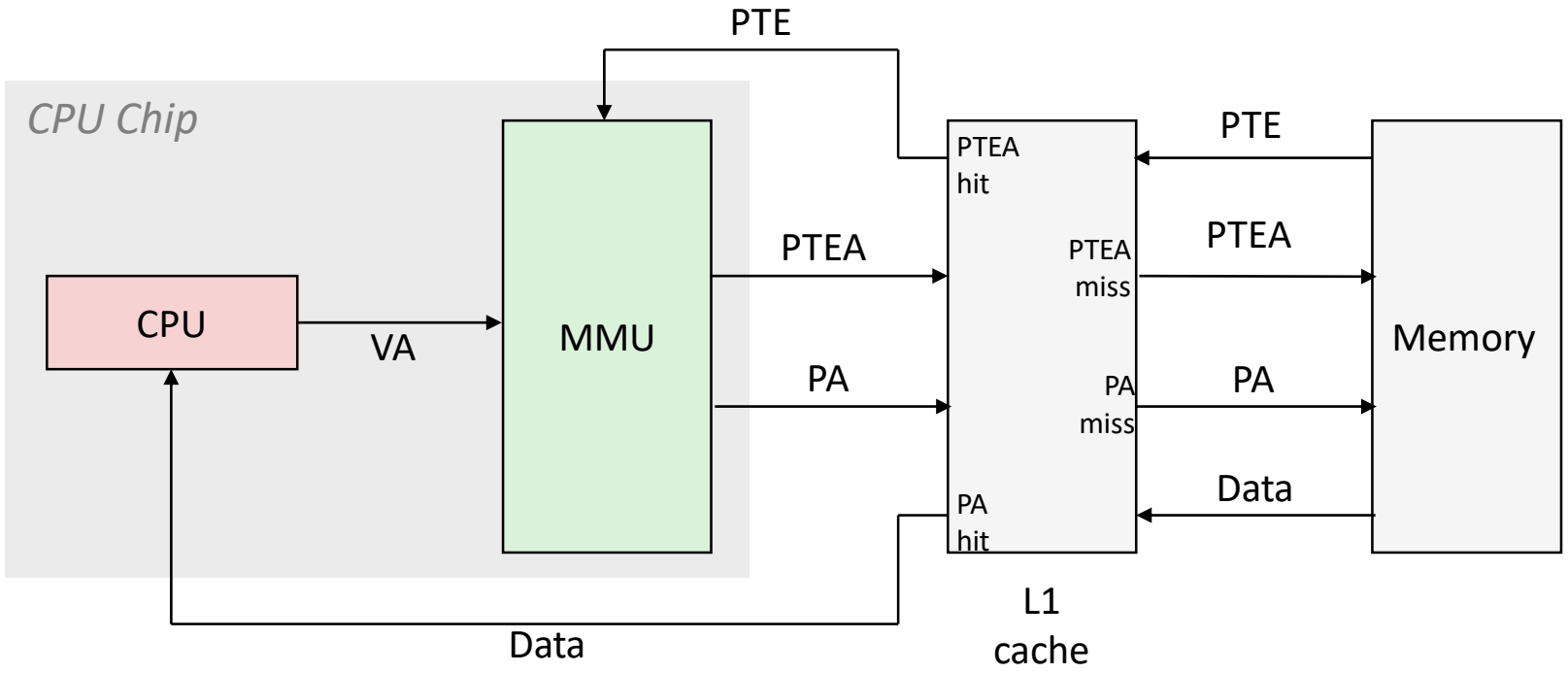
4) Valid bit is zero, so MMU triggers page fault exception

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

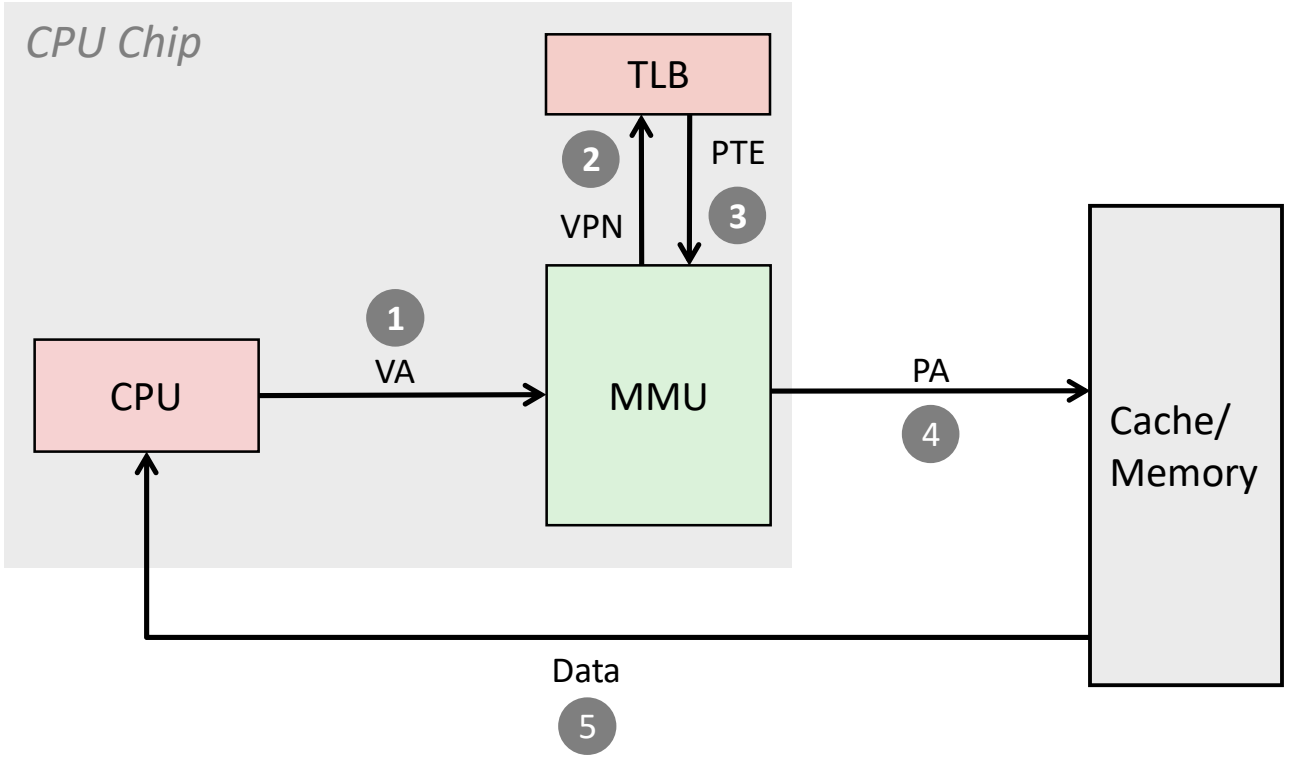
Page table entries (PTEs) are cached in L1 like any other memory word

- PTEs may be evicted by other data references
- PTE hit still requires a small L1 delay

Solution: *Translation Lookaside Buffer* (TLB)

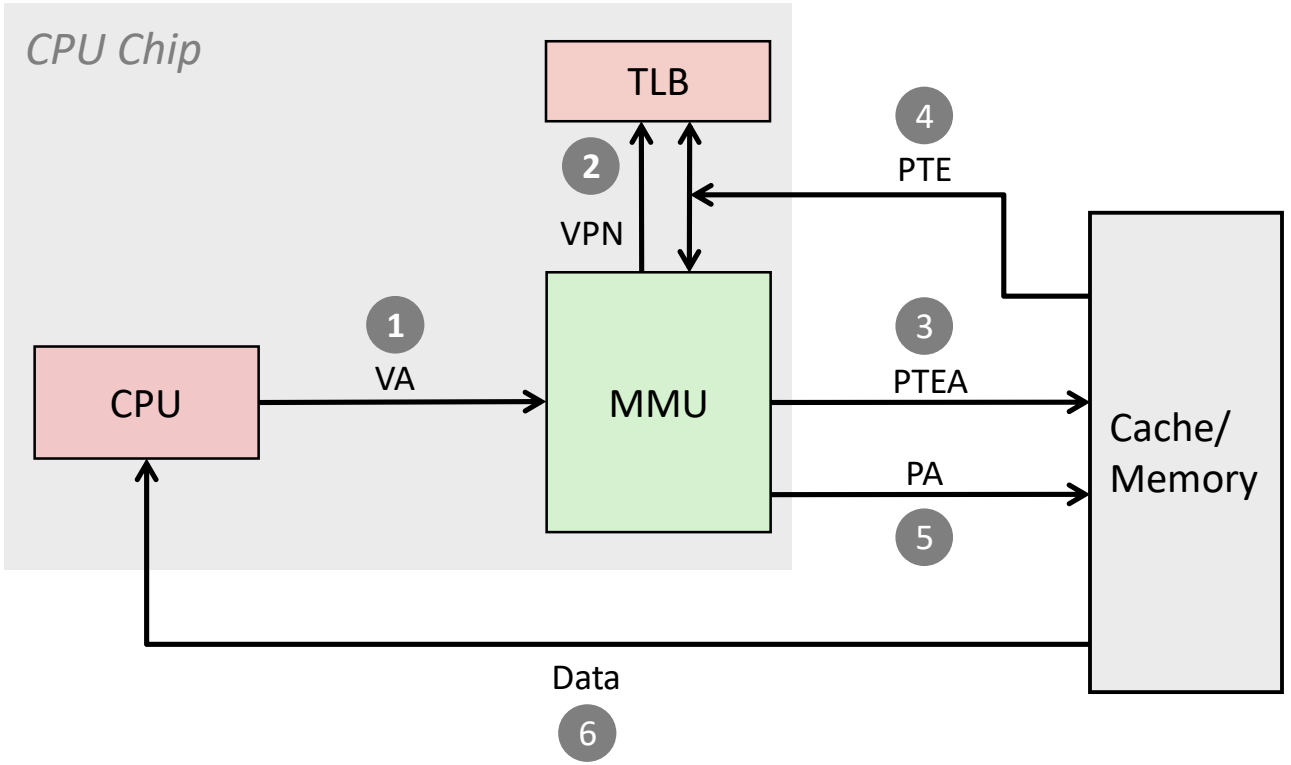
- Small hardware cache in MMU
- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages

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?

Multi-Level Page Tables

Suppose:

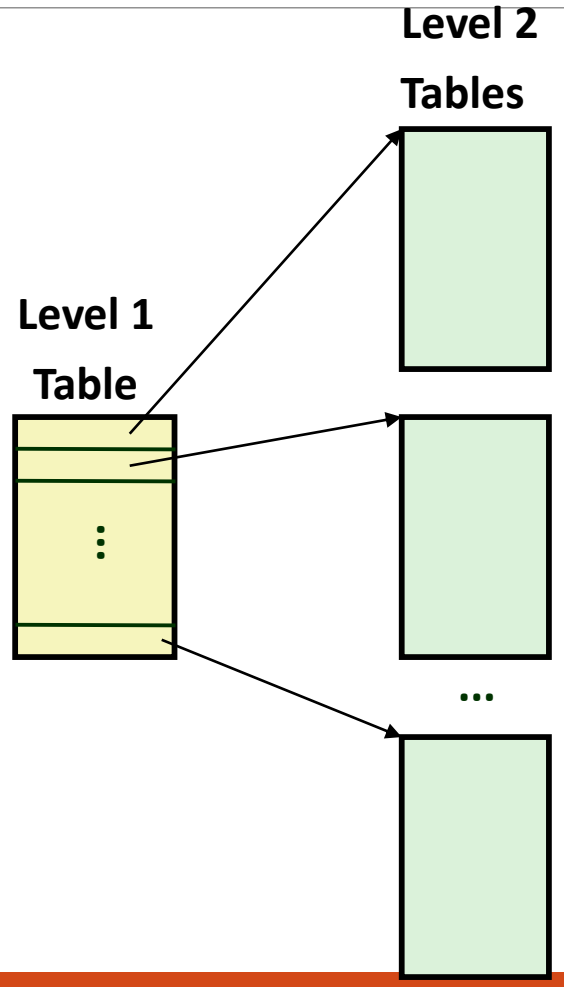
- 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE

Problem:

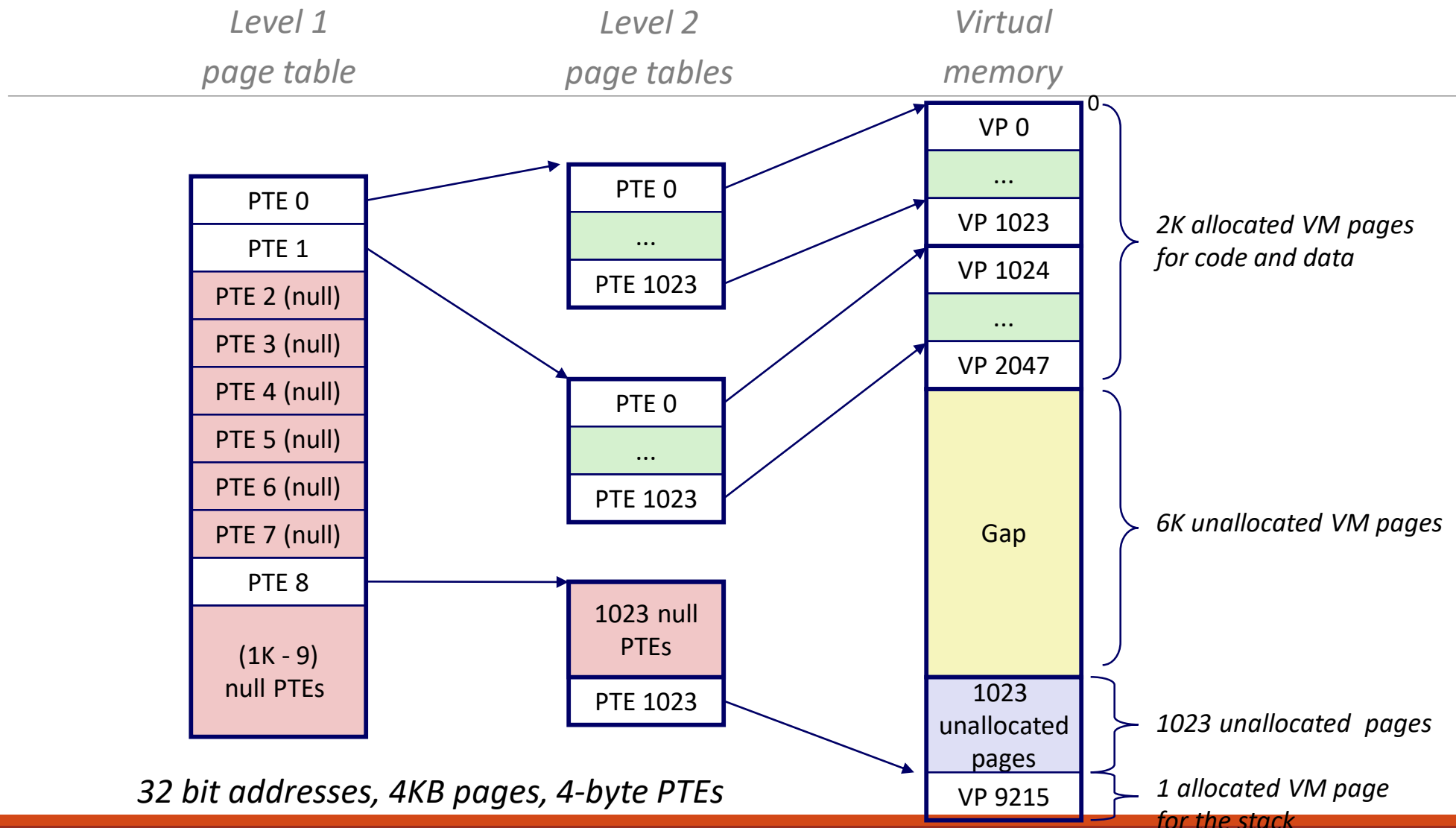
- Would need a 512 GB page table!
  - $2^{48} * 2^{-12} * 2^{3} = 2^{39}$ bytes

Common solution:

- Multi-level page tables
- 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



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

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

Thank You!