Virtual Memory: Concepts

15-213: Introduction to Computer Systems
15th Lecture, Oct. 14, 2010

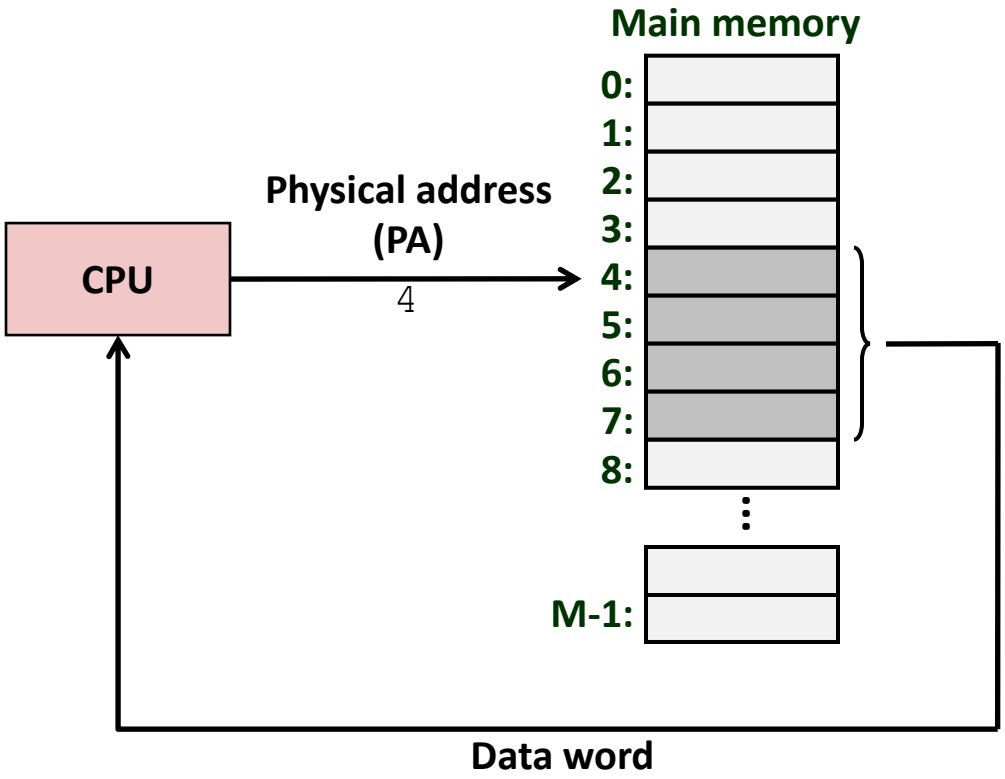
Instructors:

Randy Bryant and Dave O'Hallaron

Today

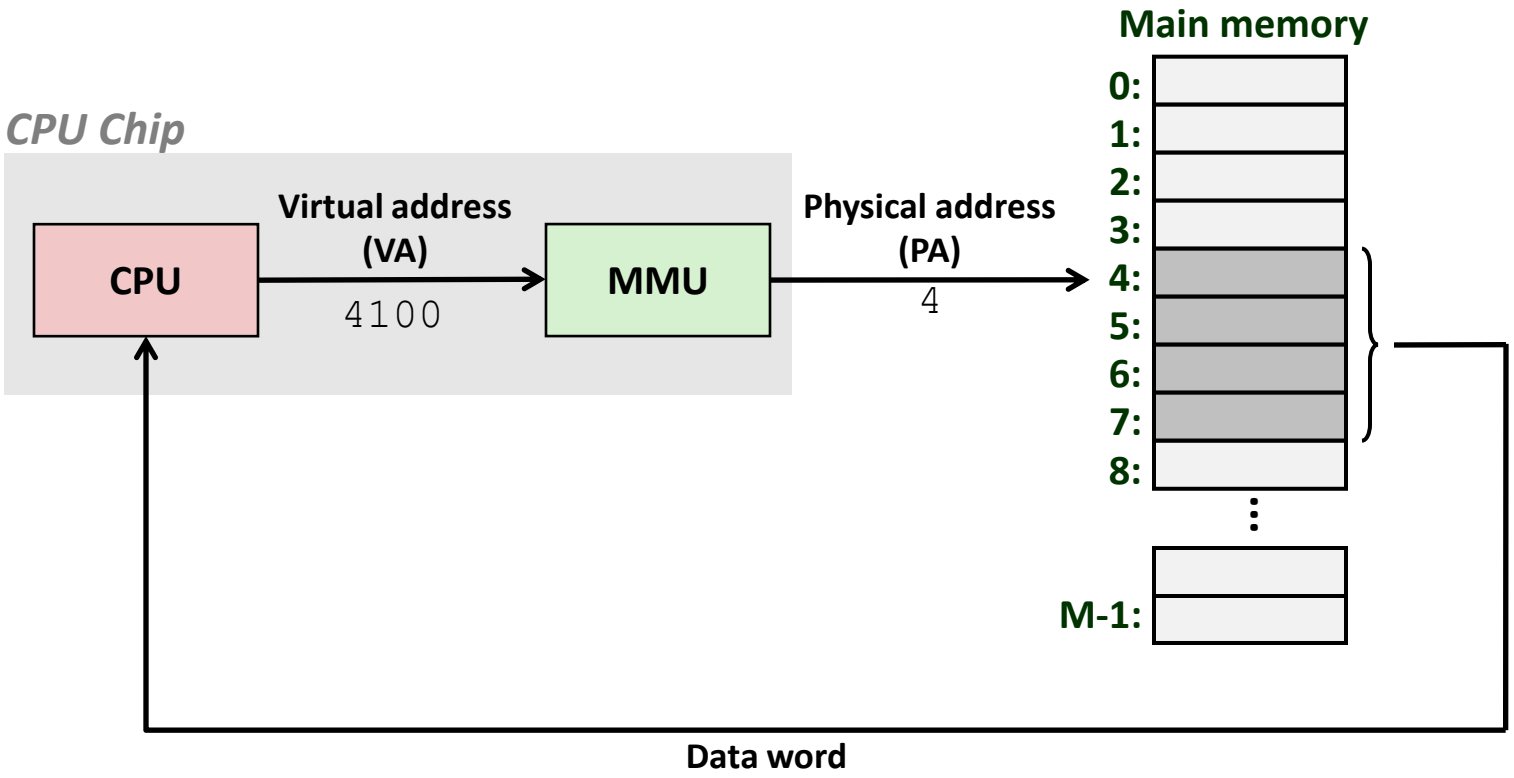
- **Address spaces**
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation

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)?

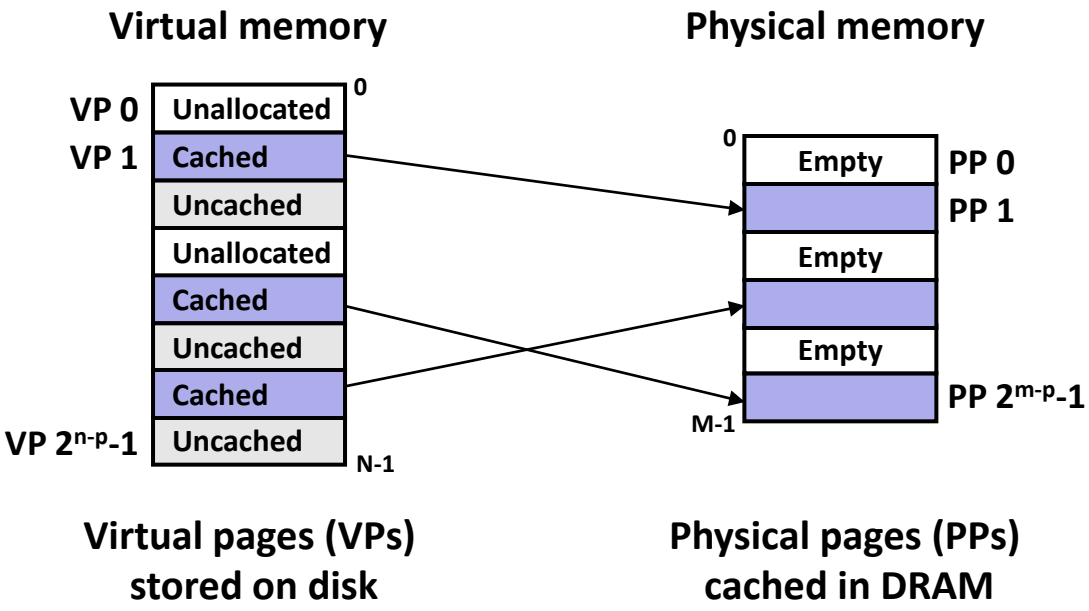
- **Uses main memory efficiently**
  - Use DRAM as a cache for the parts of a virtual address space
- **Simplifies memory management**
  - Each process gets the same uniform linear address space
- **Isolates address spaces**
  - One process can't interfere with another's memory
  - User program cannot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

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

- ***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)



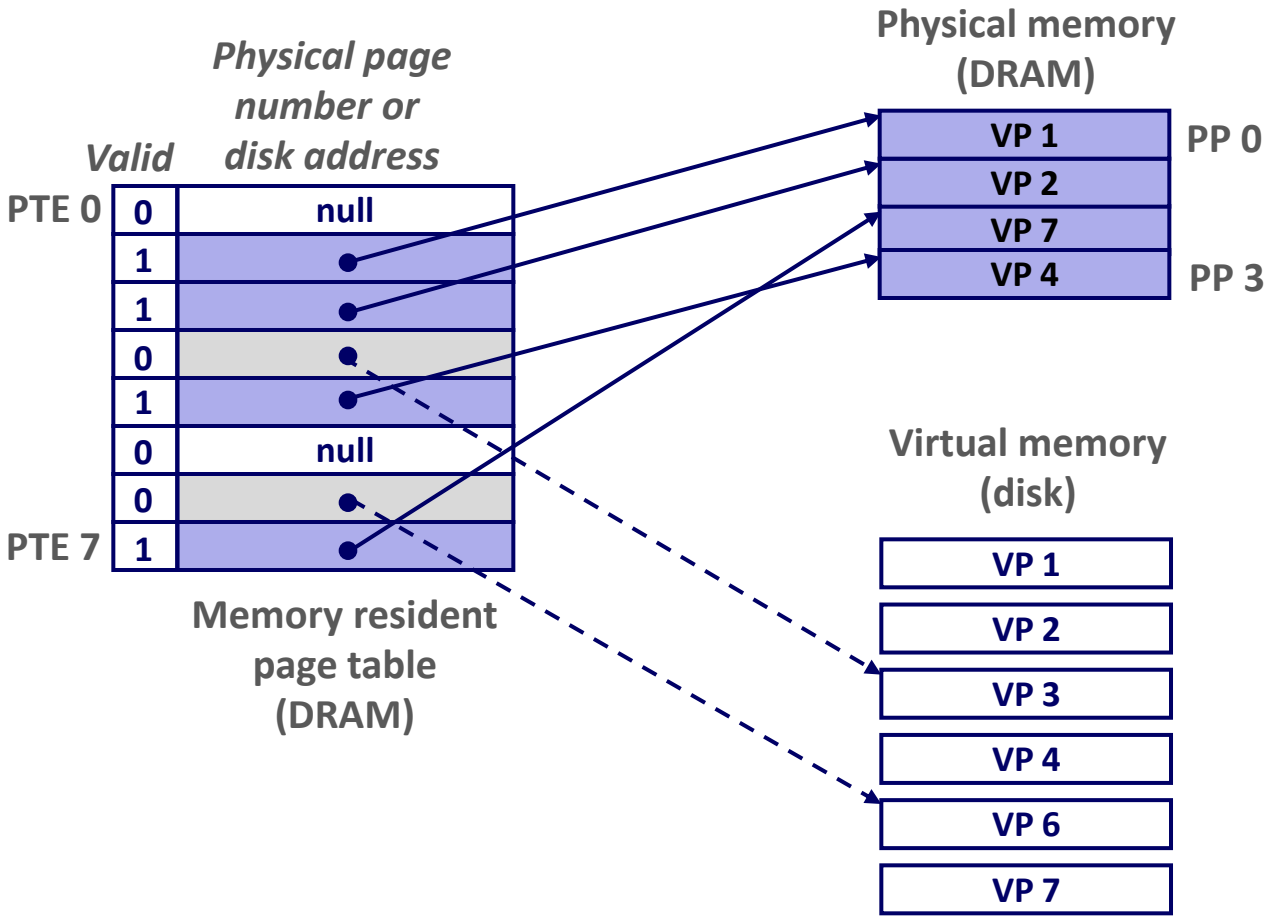
DRAM Cache Organization

- **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

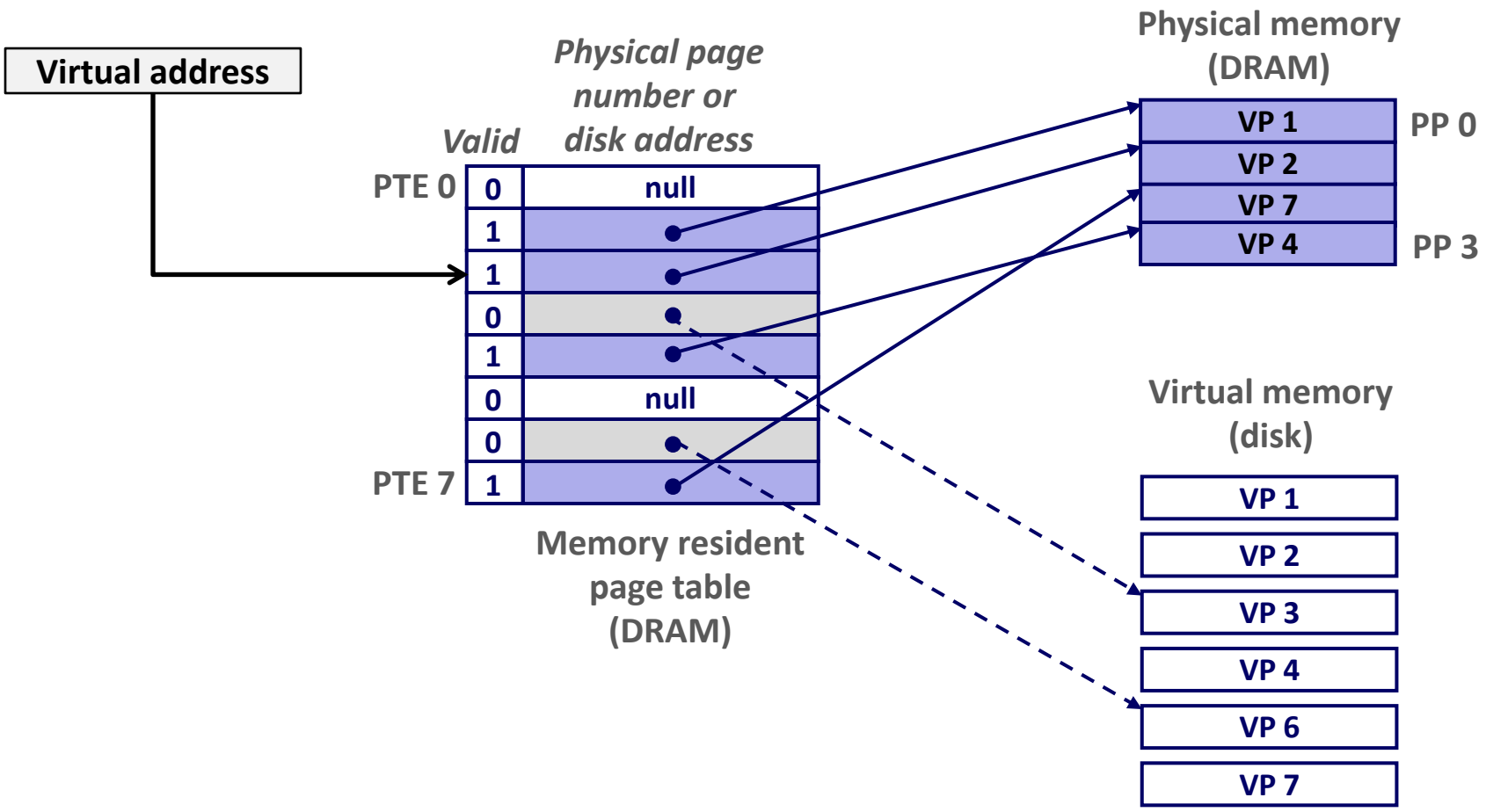
Page Tables

- **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



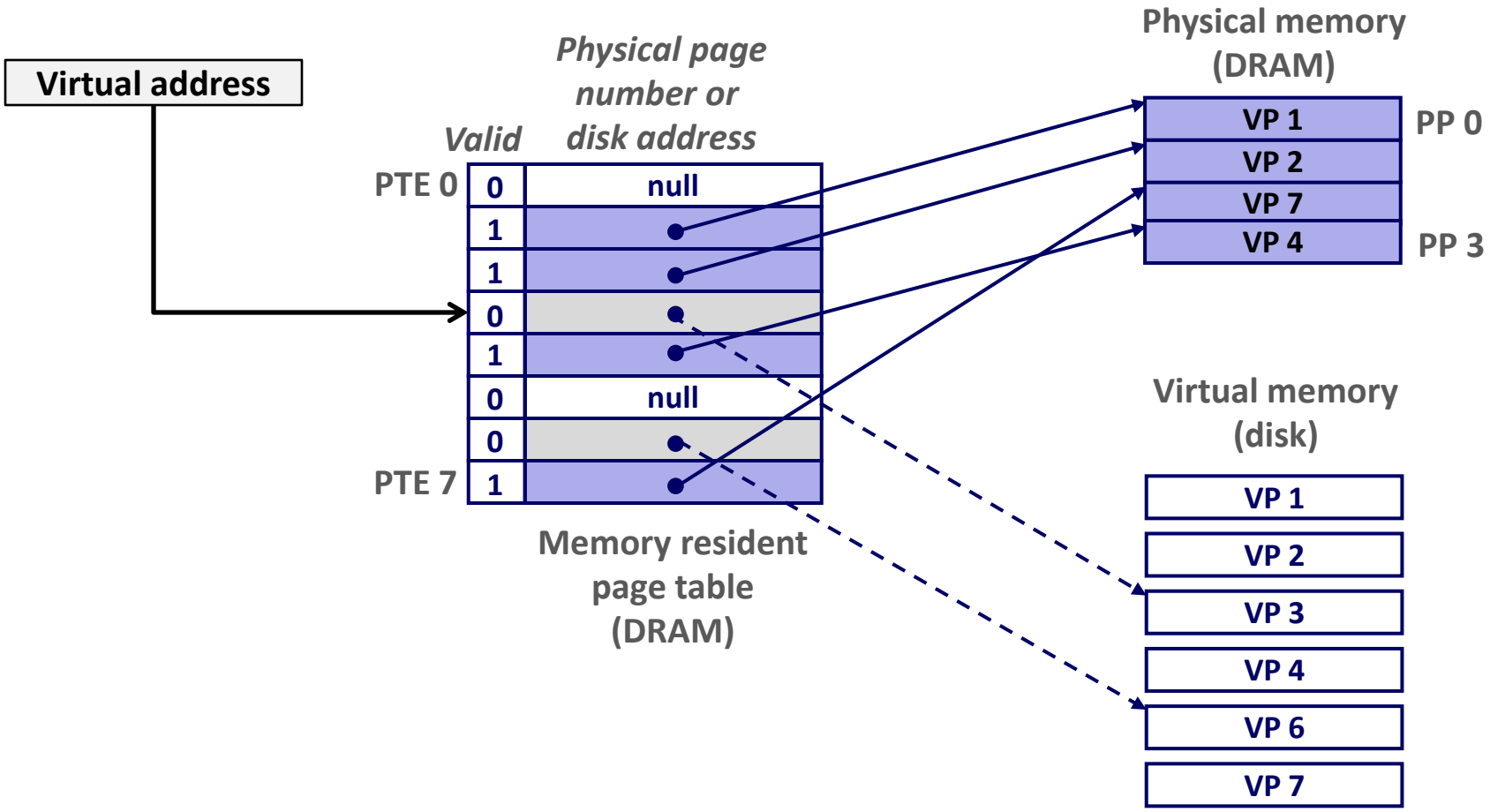
Page Hit

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



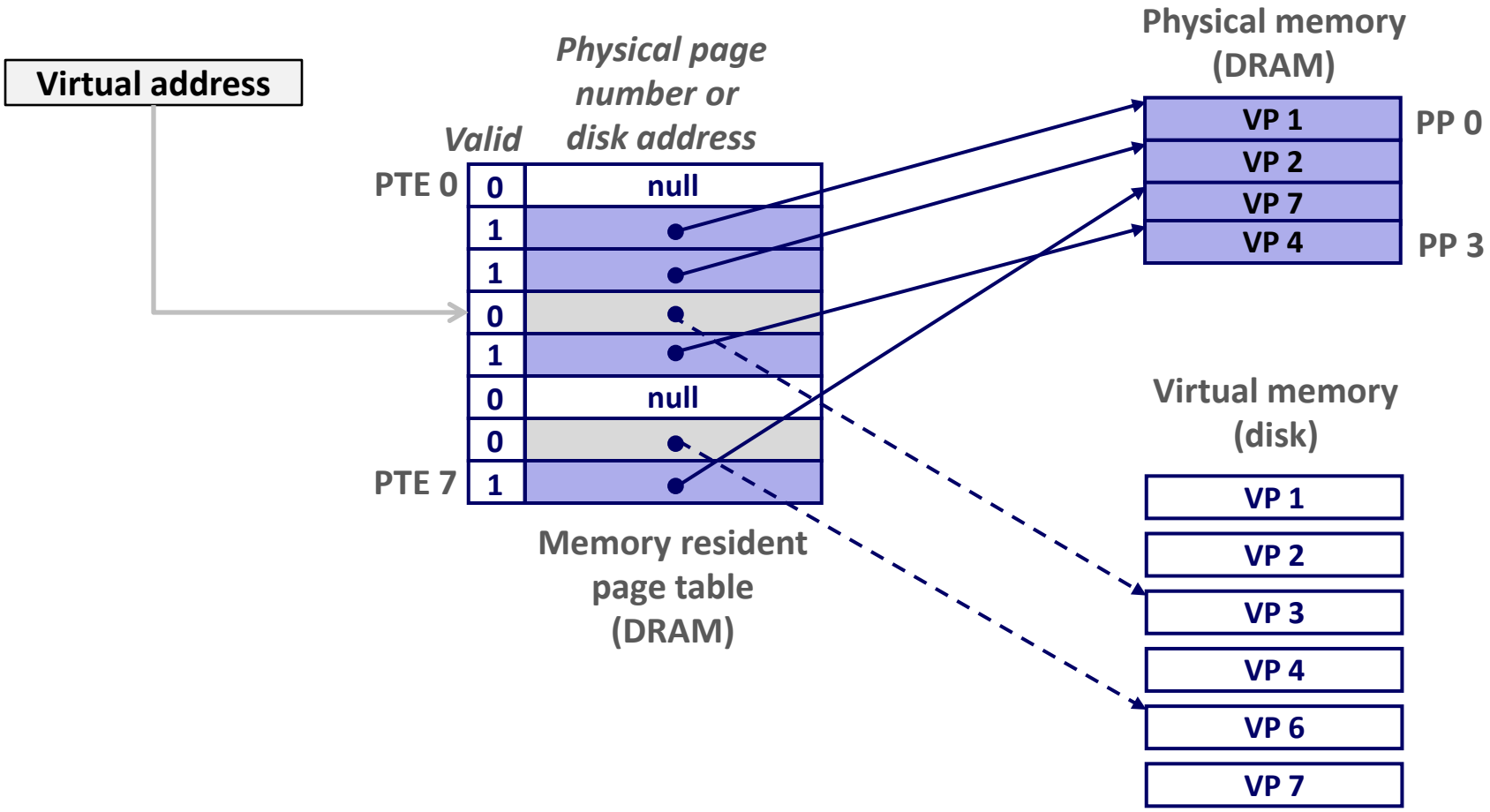
Page Fault

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



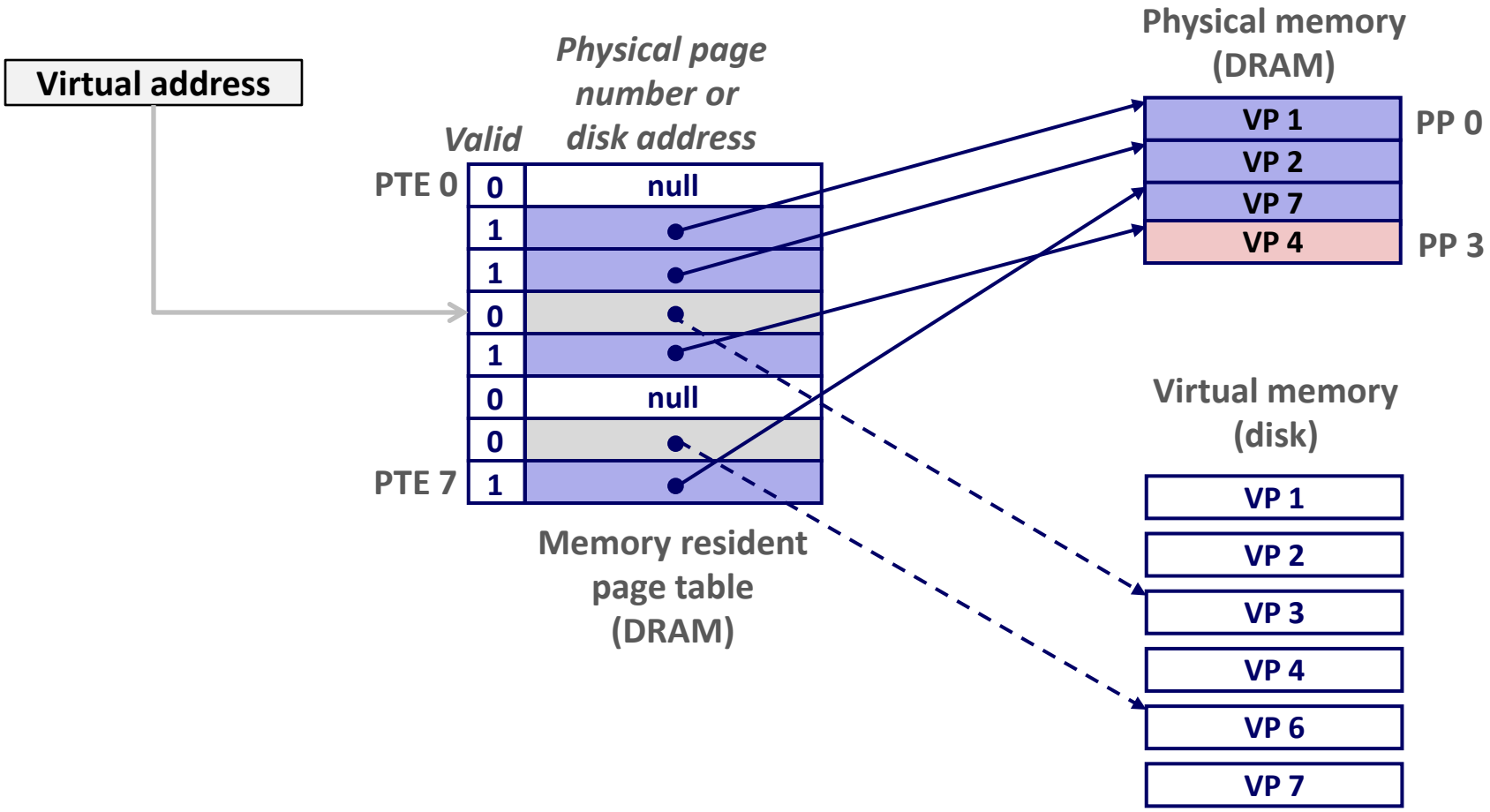
Handling Page Fault

- Page miss causes page fault (an exception)



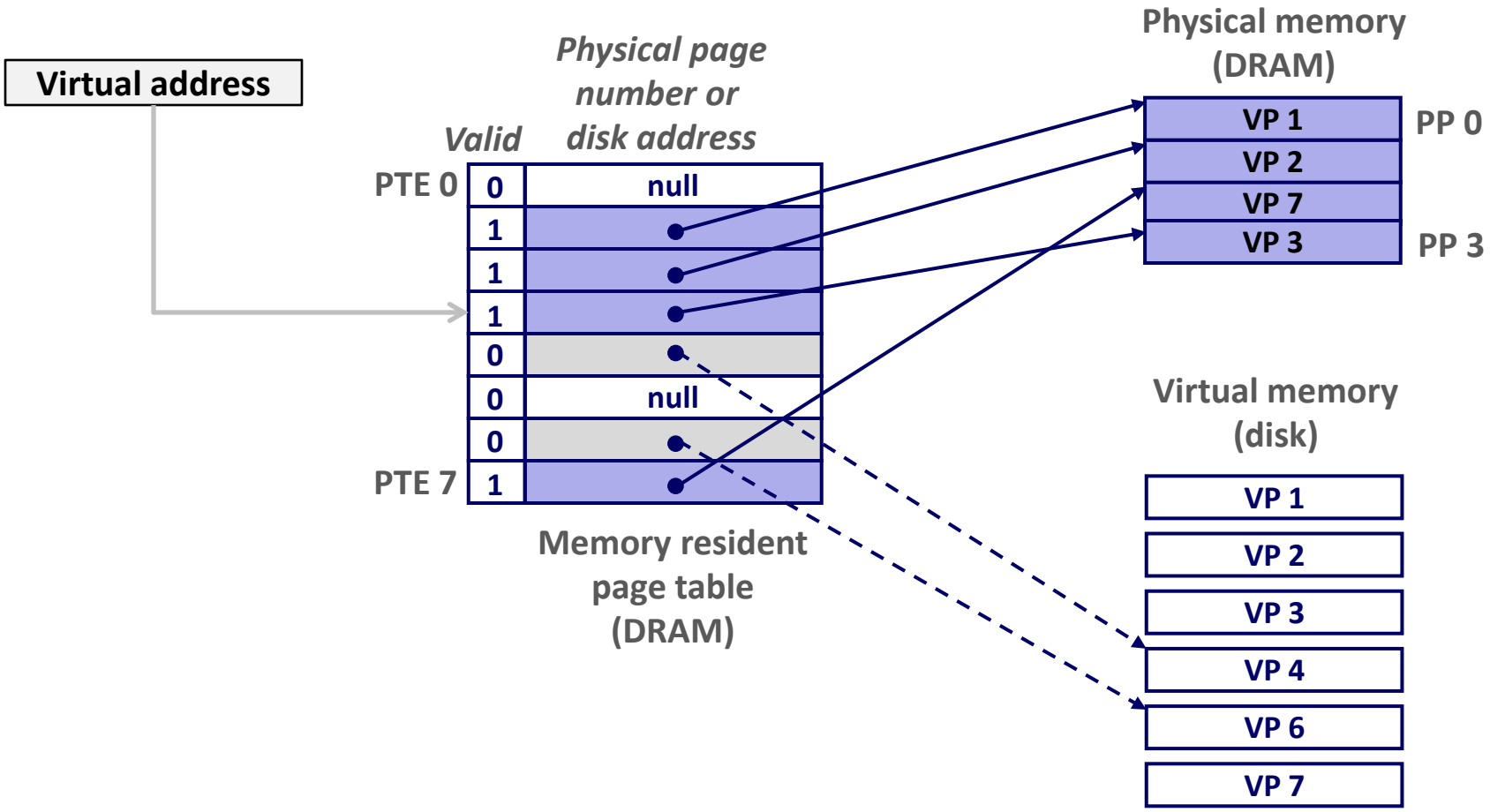
Handling Page Fault

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



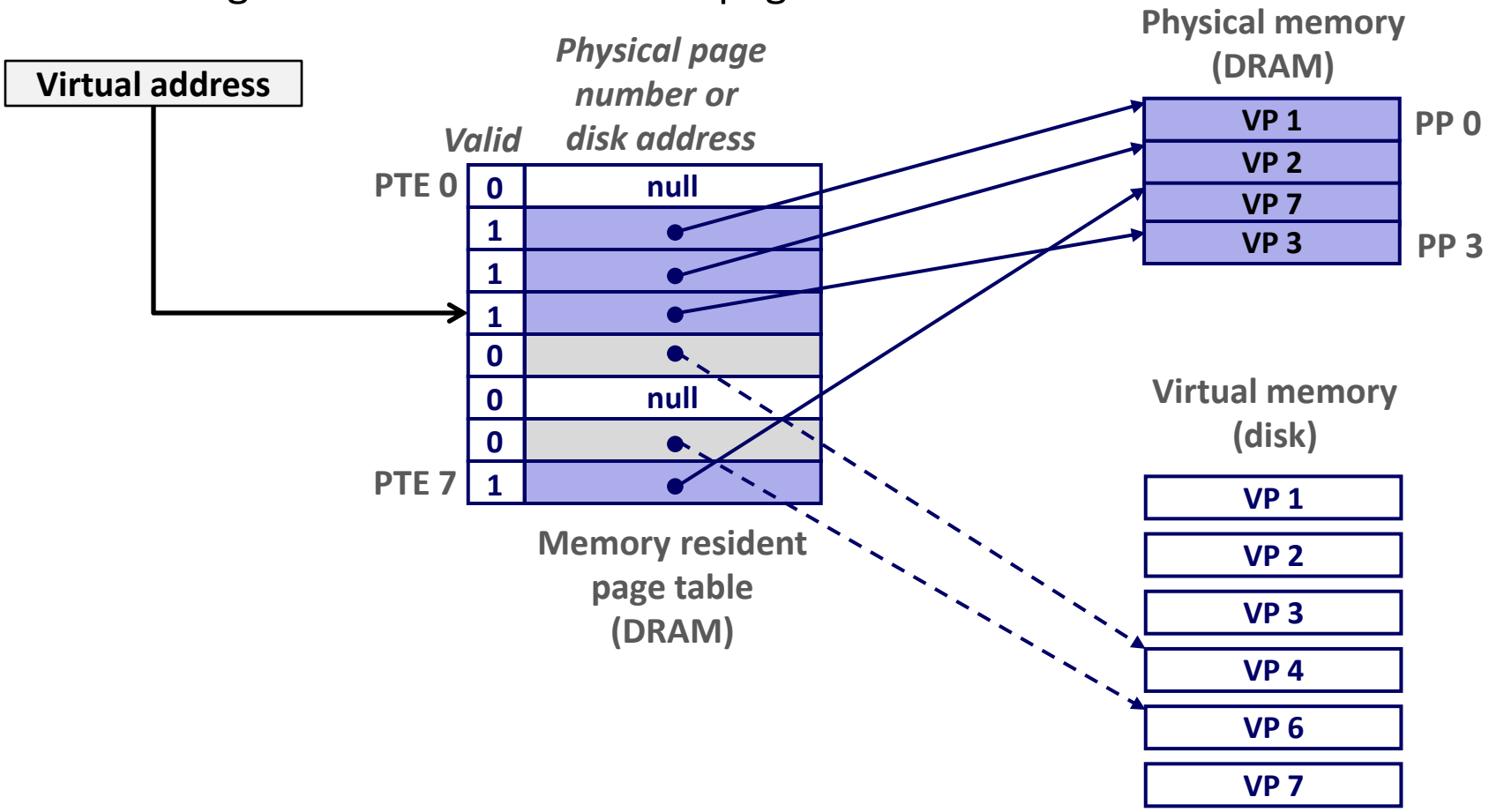
Handling Page Fault

- 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)
- Offending instruction is restarted: page hit!



Locality to the Rescue Again!

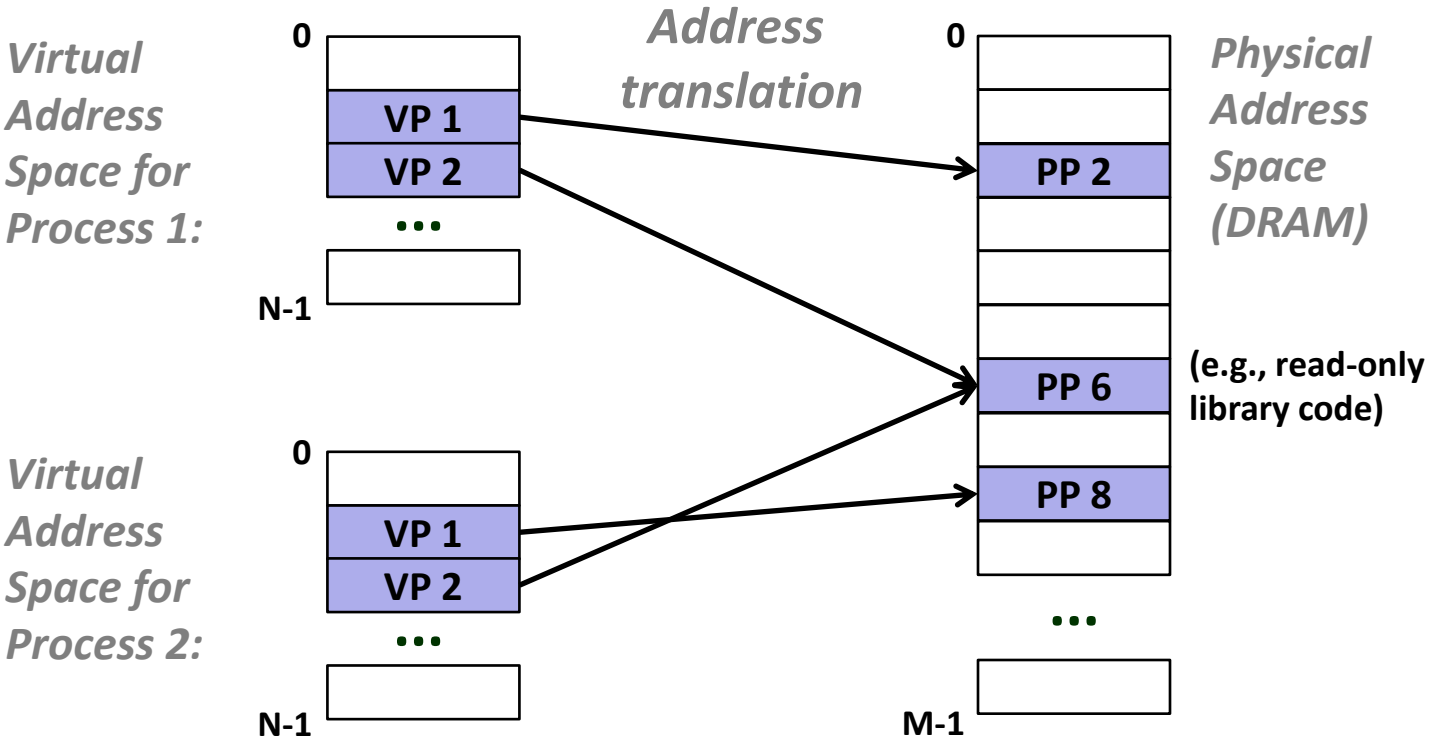
- **Virtual memory works because of locality**
- **At any point in time, programs tend to access a set of active virtual pages called the *working set***
  - Programs with better temporal locality will have smaller working sets
- **If (working set size < main memory size)**
  - Good performance for one process after compulsory misses
- **If (SUM(working set sizes) > main memory size)**
  - *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously

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

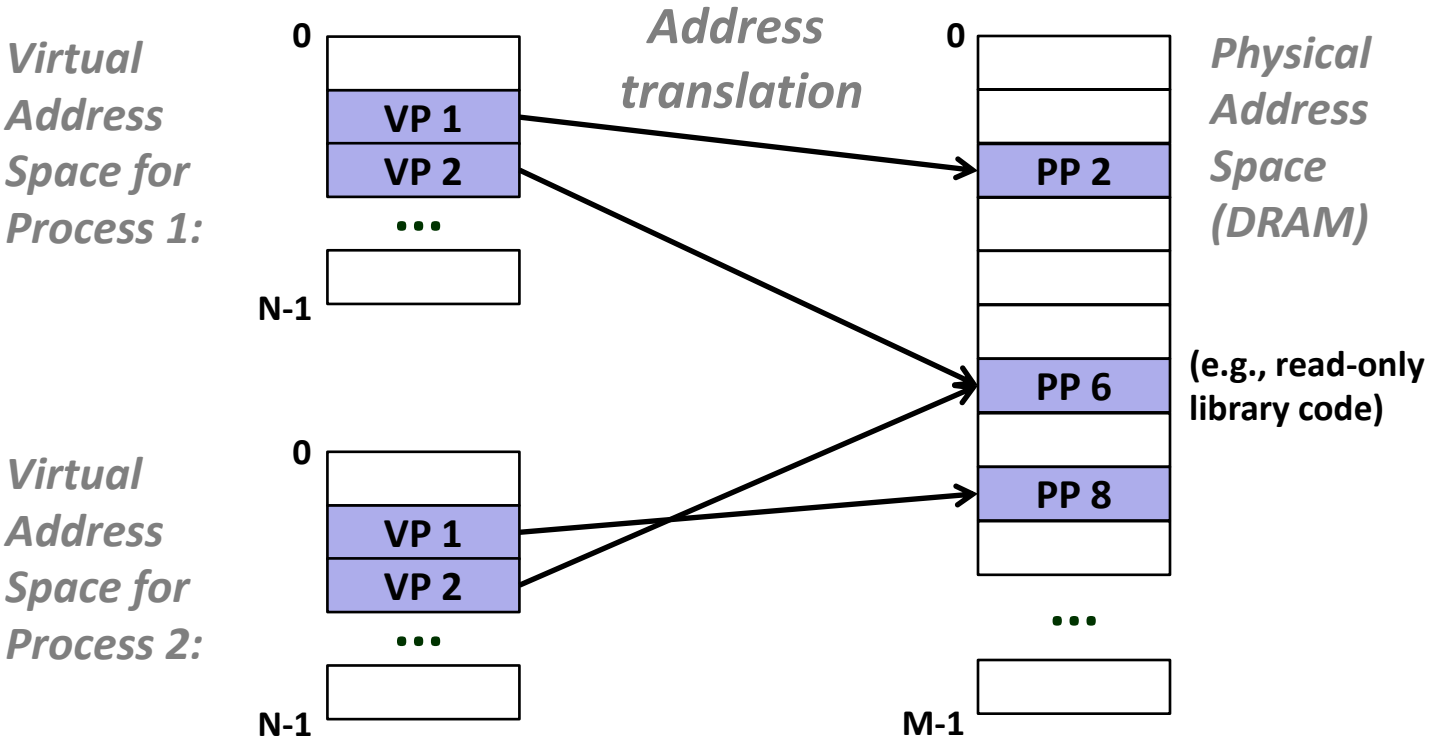
VM as a Tool for Memory Management

- **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 simplify memory allocation and management



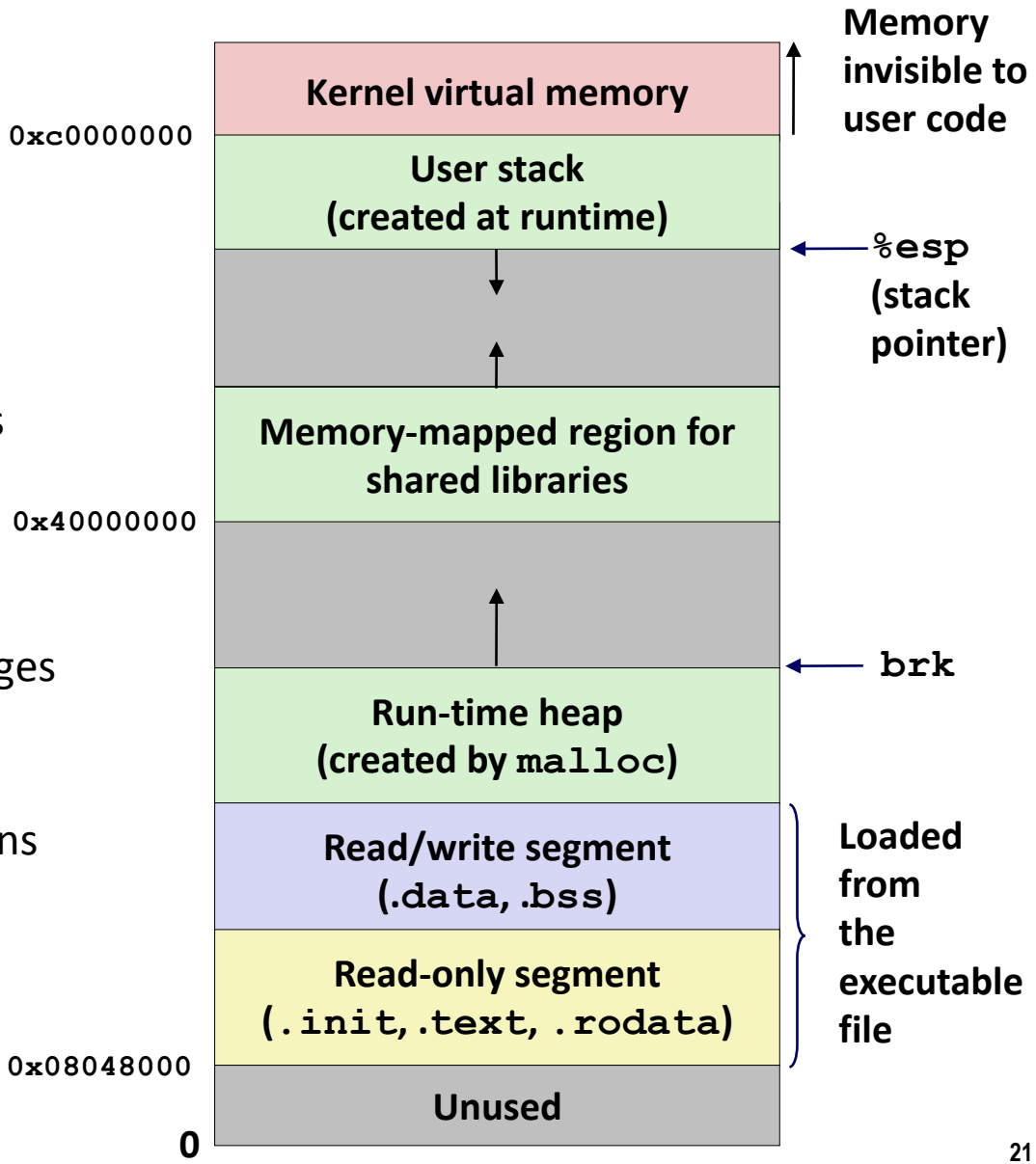
VM as a Tool for Memory Management

- **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, stack, and shared libraries always start at the same address
- **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

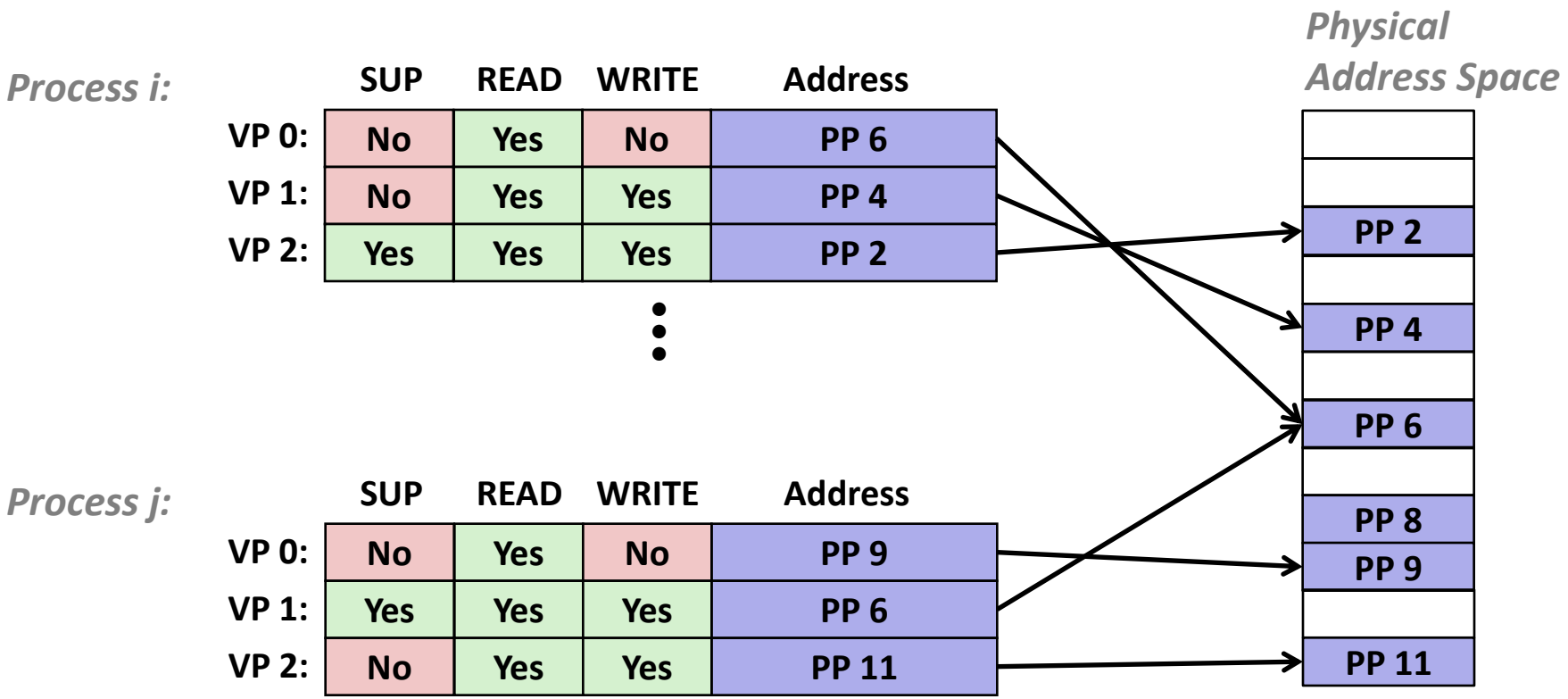


Today

- 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
- Page fault handler checks these before remapping
  - If violated, send process SIGSEGV (segmentation fault)



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**

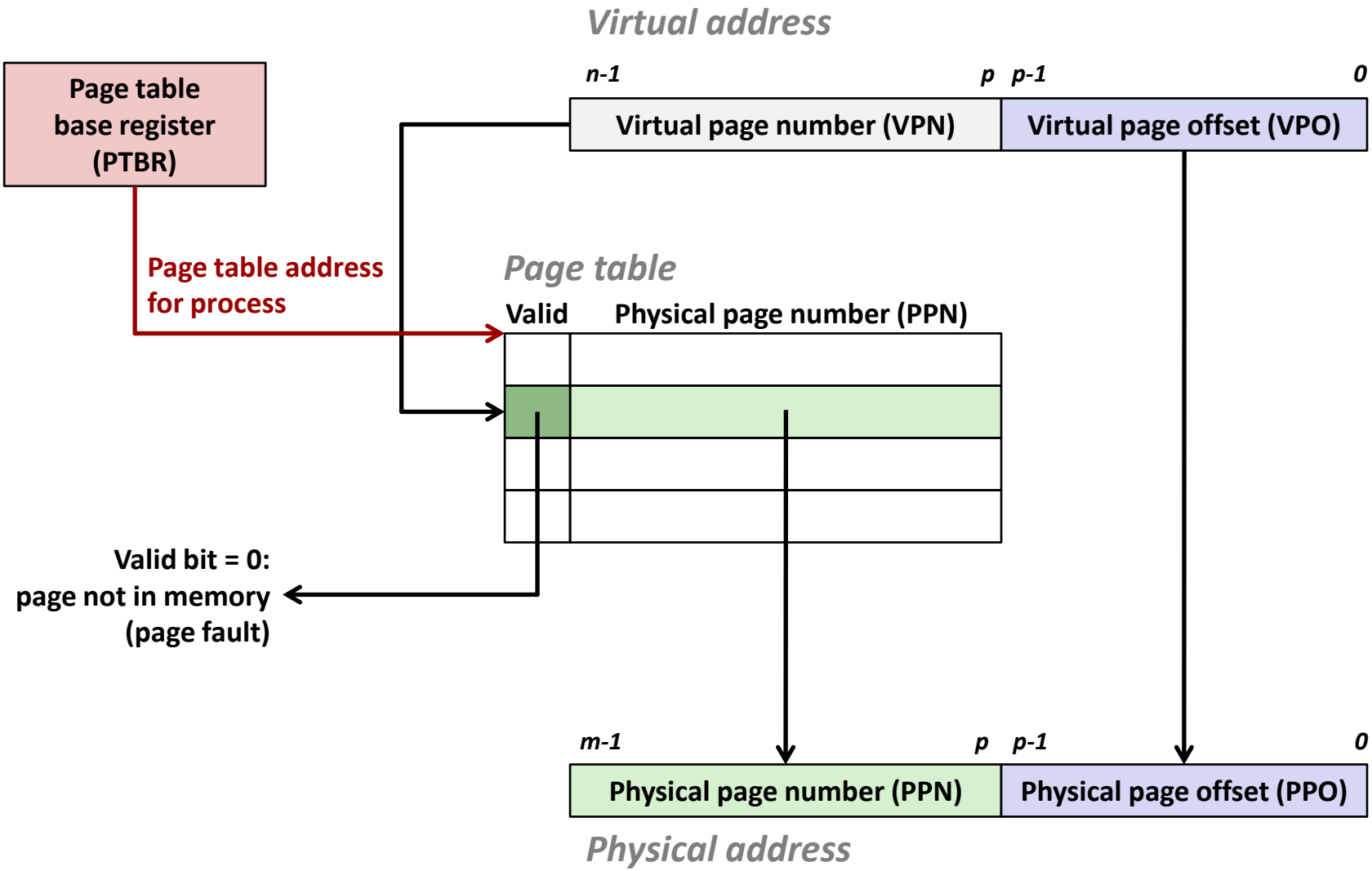
VM Address Translation

- **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

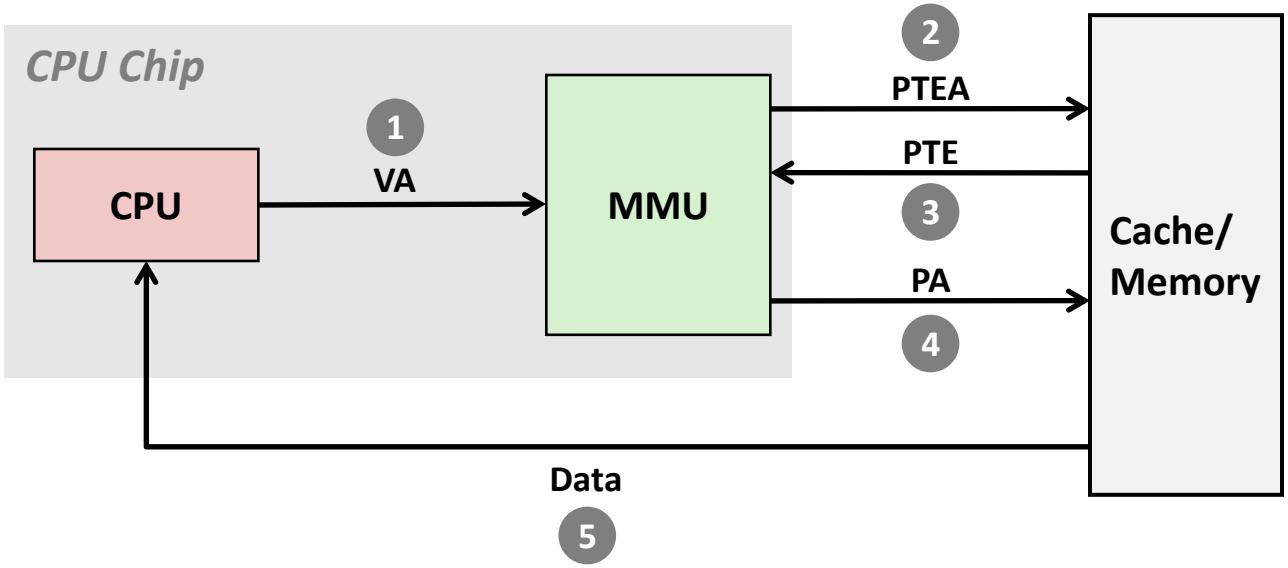
Summary of Address Translation Symbols

- **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
  - **CO**: Byte offset within cache line
  - **CI:** Cache index
  - **CT**: Cache tag

Address Translation With a Page Table



Address Translation: Page Hit



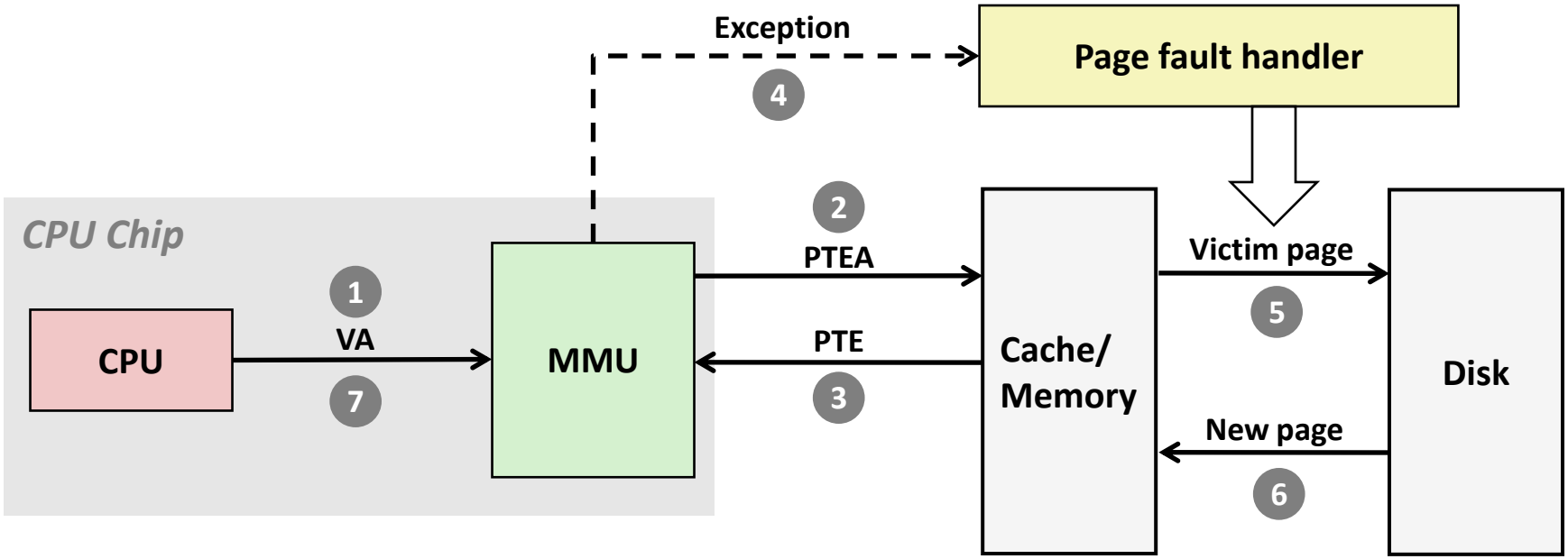
1) Processor sends virtual address to MMU

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

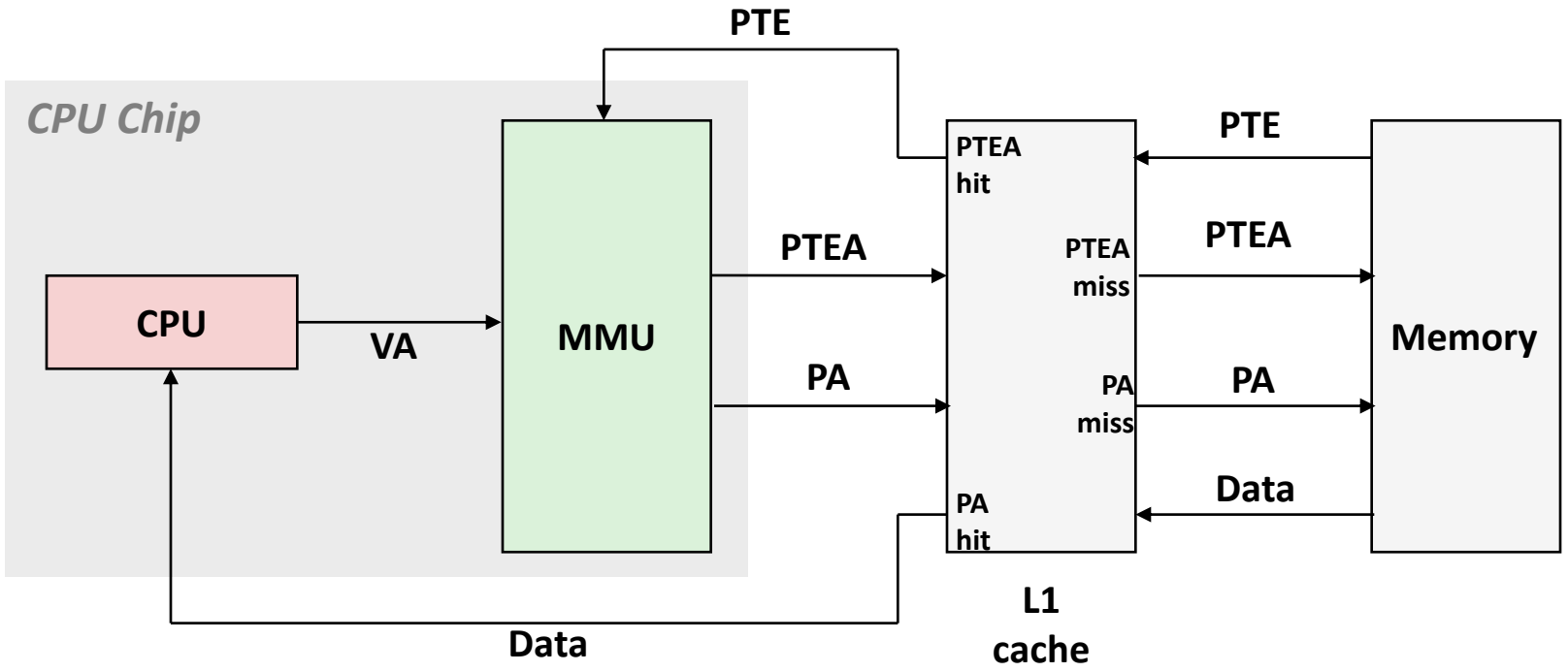
4) Valid bit is zero, so MMU triggers page fault exception

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

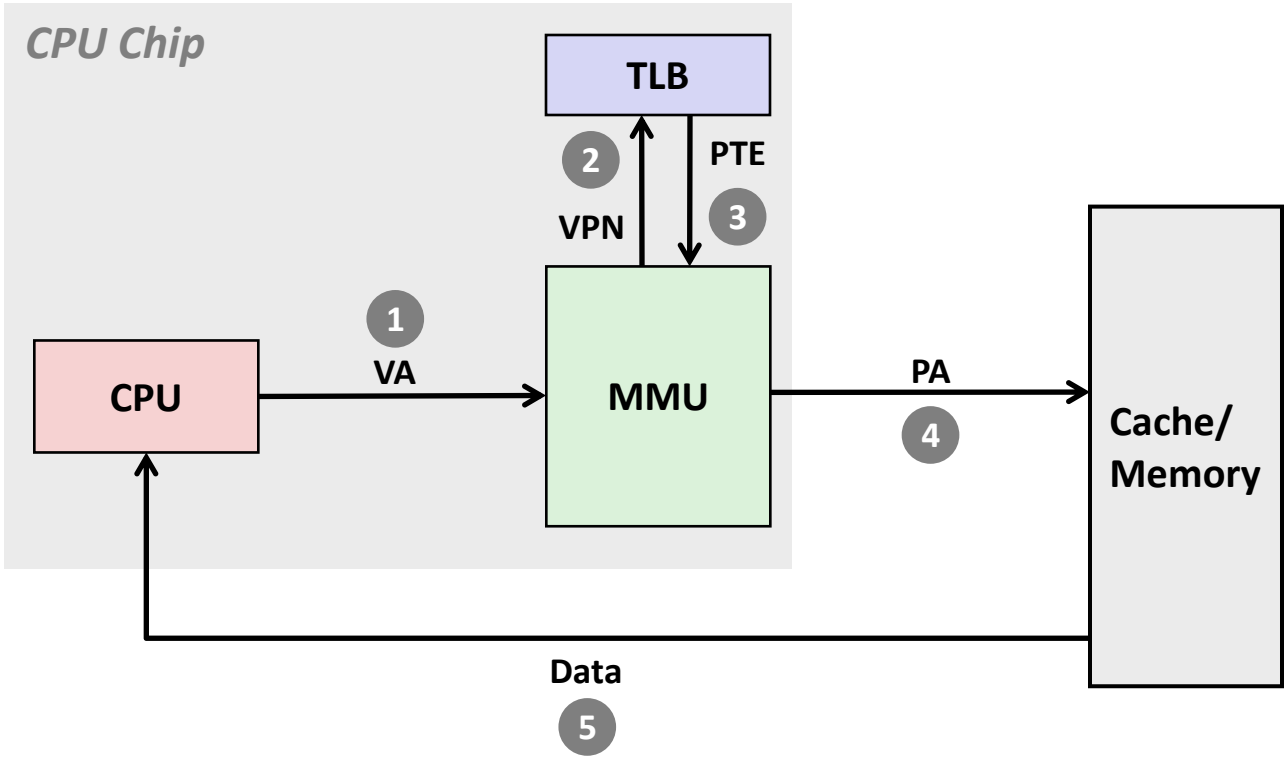


VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Speeding up Translation with a TLB

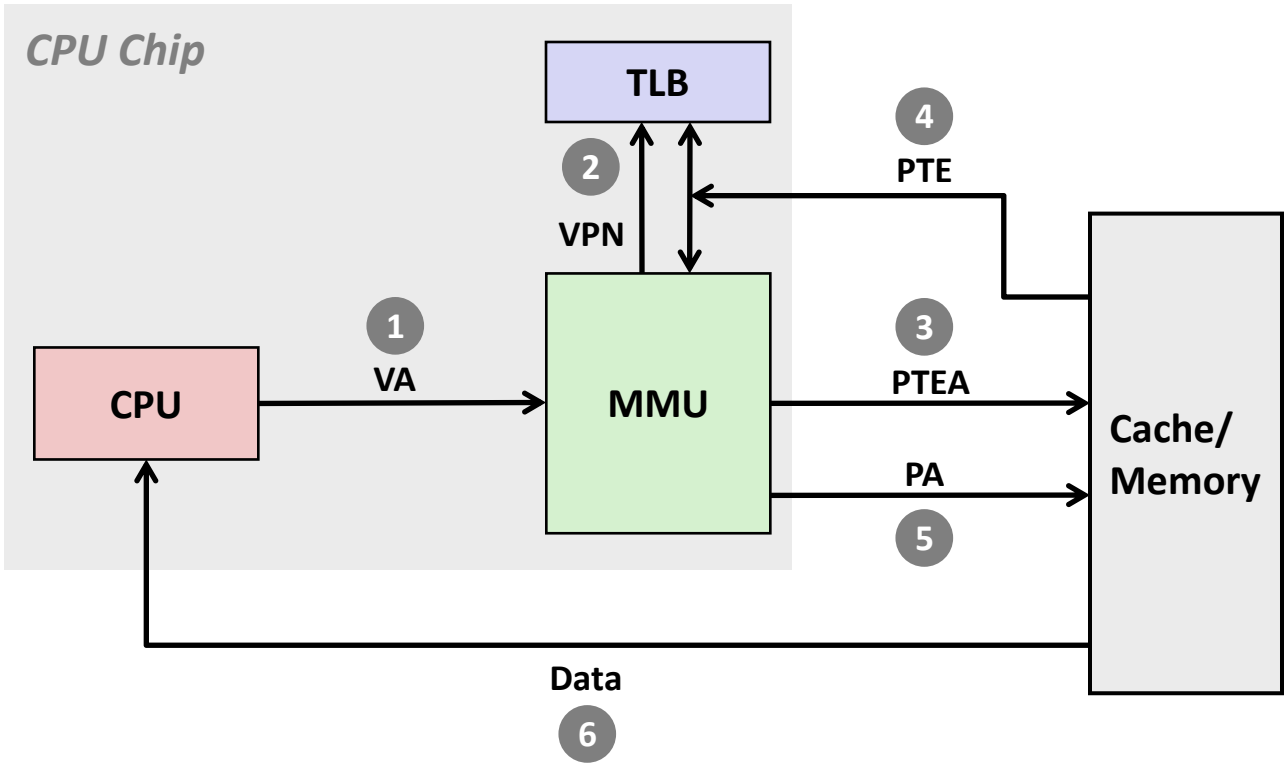
- **Page table entries (PTEs) are cached in L1 like any other memory word**
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay
- **Solution: *Translation Lookaside Buffer* (TLB)**
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

TLB Hit



A TLB hit eliminates a memory access

TLB Miss

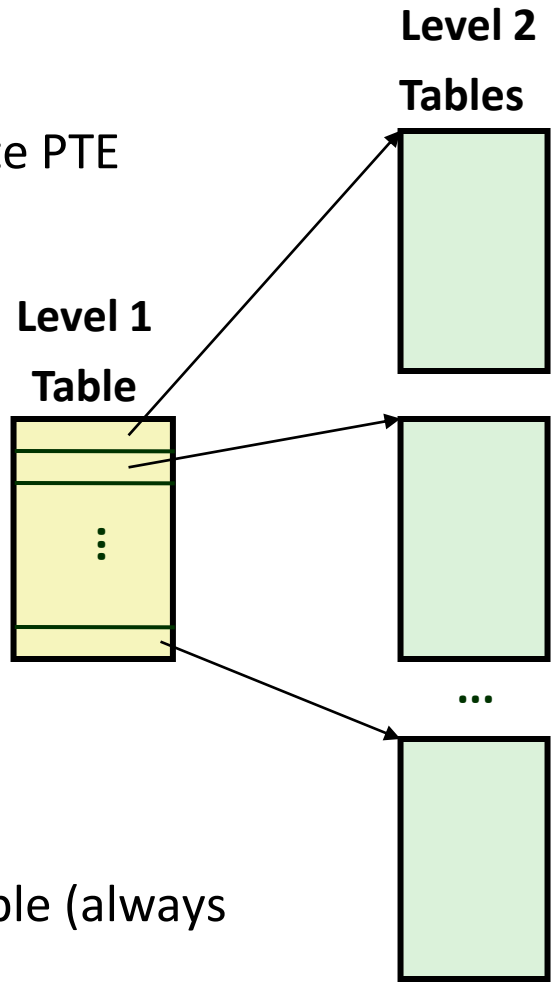


A TLB miss incurs an additional memory access (the PTE)

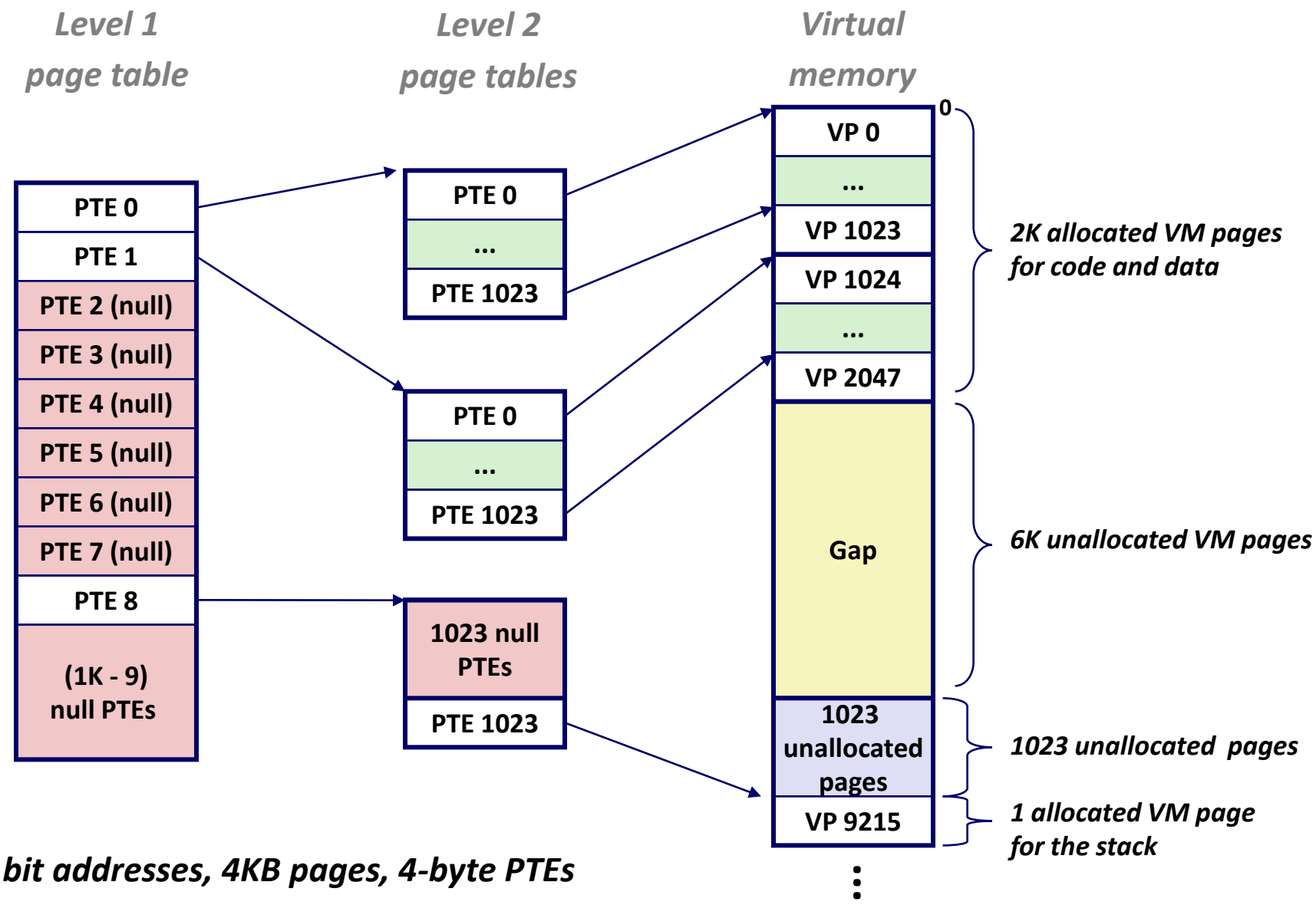
Fortunately, TLB misses are rare. Why?

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

Single level page table

- **64bit address space, 4KB page size, 4GB physical memory**
- **One entry per virtual page**
  - 2^{64} addressable bytes/2^{12} bytes per page = 2^{52} page table entries
- **Page table entry size**
  - One page table entry contains:
    - Physical page number + Access control bits
  - 4GB physical memory = 2^{32} bytes
    - 2^{32} bytes memory / 2^{12} bytes per page = 2^{20} physical pages
    - 20 bits needed for physical page number
  - Page table entry = ~ 4 bytes
- **Page table size**
  - 2^{52} page table entries * 4 bytes = 2^{54} bytes (16 petabytes) per process

Multilevel page table

- **Let's make sure that any page table requires only a single page (4KB)**
  - Page table entry = ~ 4 bytes
  - 4KB page / 4bytes per PTE = 1024 entries
  - 10 bits of address space needed
  - We have 52 bit address for page table
  - We need ceiling(52/10) = 6 levels needed !

- **For each page table access, we need 6 memory accesses**

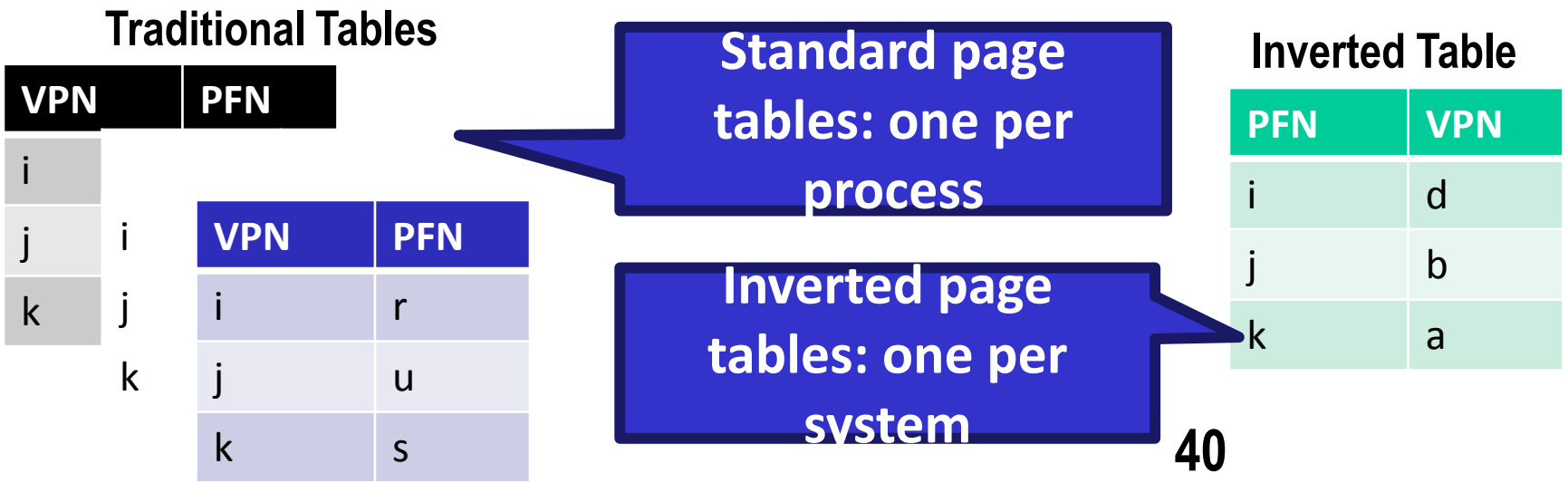
Inverted Page Table

- **We only have 4GB physical memory**
  - 2^{32} bytes memory / 2^{12} bytes per page = 2^{20} physical pages
- **If we only store a single page table entry per physical page**
- **(table is indexed by PPN)**
  - Each page table entry has process ID, VPN, and access info
  - 16 bit process ID, 52bit VPN, 12 bit access info = 80 bits = 10 bytes
  - 2^{20} PTE * 10 bytes = 10MB

- **For each (VPN, pid) pair, we need to find where it is in the page table by comparing it with all entries**
  - Linear time, up to 2^{20} memory accesses……
- **Hashing to the rescue -> Hashed Inverted Page Tables**

Inverted Page Tables

- **Our current discussion focuses on tables that map virtual pages to physical pages**
- **What if we flip the table: map physical pages to virtual pages?**
  - Since there is only one physical memory, we only need one inverted page table!



40

Normal vs. Inverted Page Tables

- **Advantage of inverted page table**
  - Only one table for the whole system
- **Disadvantages**
  - Lookups are more computationally expensive

Table must be scanned to locate a given VPN, thus O(n) lookup time

VPN serves as an index into the array, thus O(1) lookup time

Traditional Table

VPN	PFN
i	d
j	b
k	a

me y?

Inverted Tabl

PFN	VPN
i	d
j	b
k	a

41

Inverted page tables

- **Problem:**
  - Page table overhead increases with address space size
  - Page tables get too big to fit in memory!

- **Consider a computer with 64 bit addresses**
  - Assume 4 Kbyte pages (12 bits for the offset)
  - Virtual address space = 2^{52} pages!
  - Page table needs 2^{52} entries!
  - This page table is much too large for memory!
    - Many peta-bytes per process page table

Inverted page tables

- **An inverted page table**
  - Has one entry for every frame of memory
  - Records which page is in that frame
  - Is indexed by frame number not page number!
- **So how can we search an inverted page table on a TLB miss fault?**

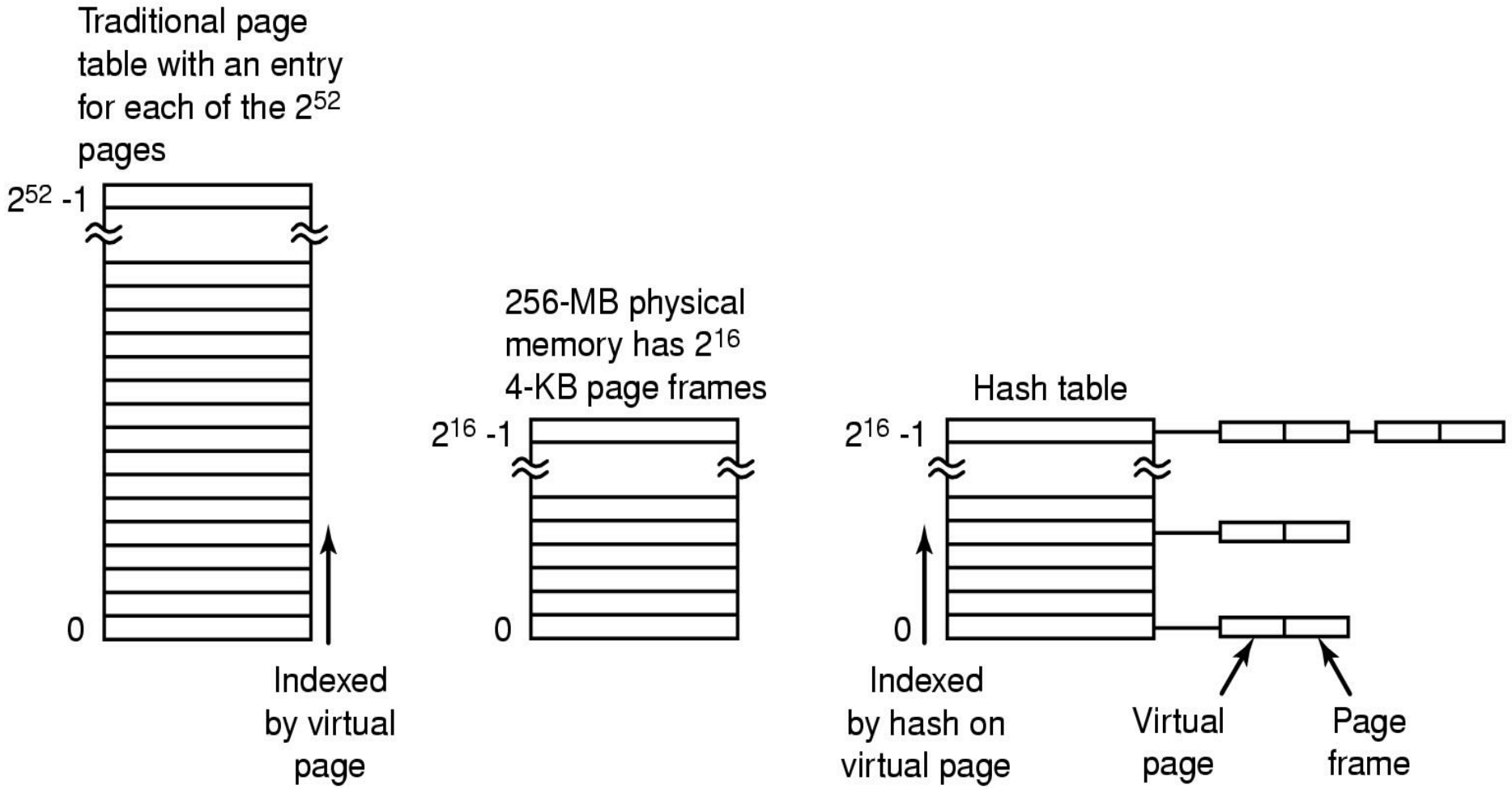
Inverted page tables

- **If we have a page number (from a faulting address) and want to find it page table entry, do we**
  - Do an exhaustive search of all entries?
  - No, that's too slow!
  - Why not maintain a hash table to allow fast access given a page number?
    - O(1) lookup time with a good hash function

Hash Tables

- **Data structure for associating a key with a value**
  - Perform hash function on key to produce a hash
  - Hash is a number that is used as an array index
  - Each element of the array can be a linked list of entries (to handle collisions)
  - The list must be searched to find the required entry for the key (entry's key matches the search key)
  - With a good hash function the list length will be very short

Inverted page table



Which page table design is best?

- The best choice depends on CPU architecture
- 64 bit systems need inverted page tables
- Some systems use a combination of regular page tables together with segmentation (later)