CSE 153
Design of Operating Systems

Fall 2018

Lecture 09: Paging/Virtual Memory (1)

Some slides modified from originals by Dave O'hallaron

Sharing Memory

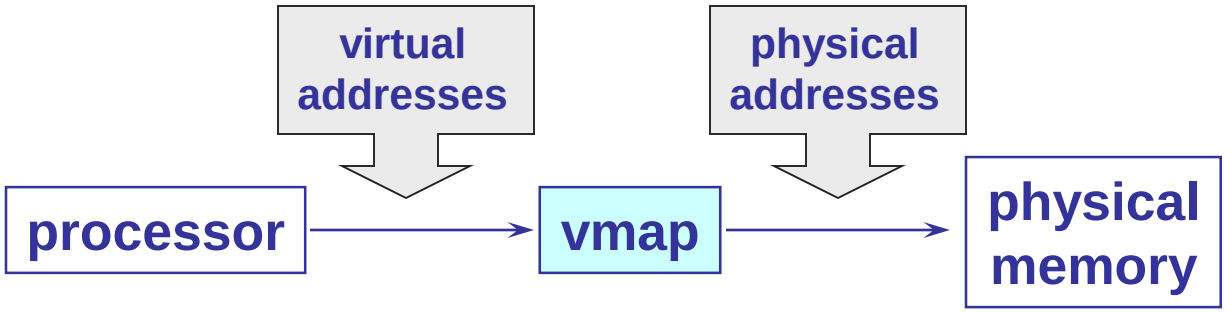
- Rewind to the days of "second-generation" computers
  - Programs use physical addresses directly
  - OS loads job, runs it, unloads it

- Multiprogramming changes all of this
  - Want multiple processes in memory at once
    - » Overlap I/O and CPU of multiple jobs
  - How to share physical memory across multiple processes?
    - » Many programs do not need all of their code and data at once (or ever) – no need to allocate memory for it
    - » A program can run on machine with less memory than it "needs"

Virtual Addresses

- To make it easier to manage the memory of processes running in the system, we're going to make them use virtual addresses (logical addresses)
  - Virtual addresses are independent of the actual physical location of the data referenced
  - OS determines location of data in physical memory
- Instructions executed by the CPU issue virtual addresses
  - Virtual addresses are translated by hardware into physical addresses (with help from OS)
  - The set of virtual addresses that can be used by a process comprises its virtual address space

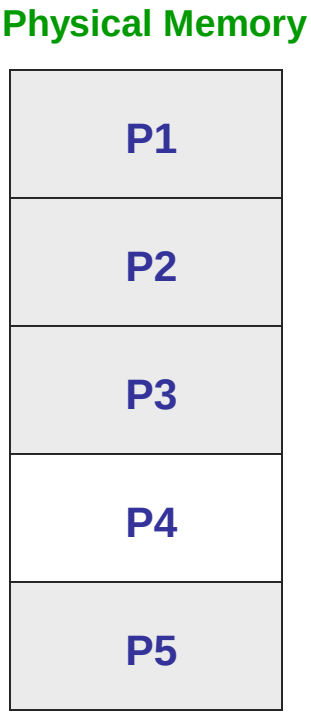
Virtual Addresses



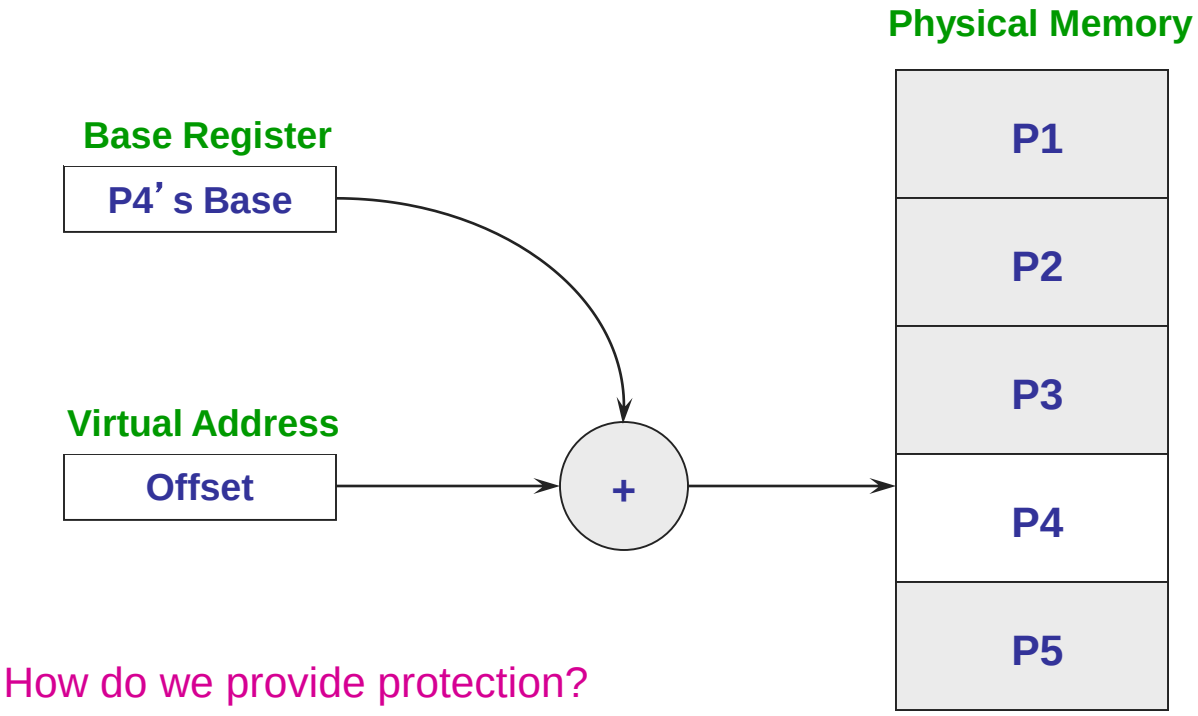
- Many ways to do this translation…
  - Need hardware support and OS management algorithms
- Requirements
  - Need protection – restrict which addresses jobs can use
  - Fast translation – lookups need to be fast
  - Fast change – updating memory hardware on context switch

Fixed Partitions

- Physical memory is broken up into fixed partitions
  - Size of each partition is the same and fixed
  - Hardware requirements: base register
  - Physical address = virtual address + base register
  - Base register loaded by OS when it switches to a process



Fixed Partitions



How do we provide protection?

Fixed Partitions

- Advantages
  - Easy to implement
    - » Need base register
    - » Verify that offset is less than fixed partition size
  - Fast context switch

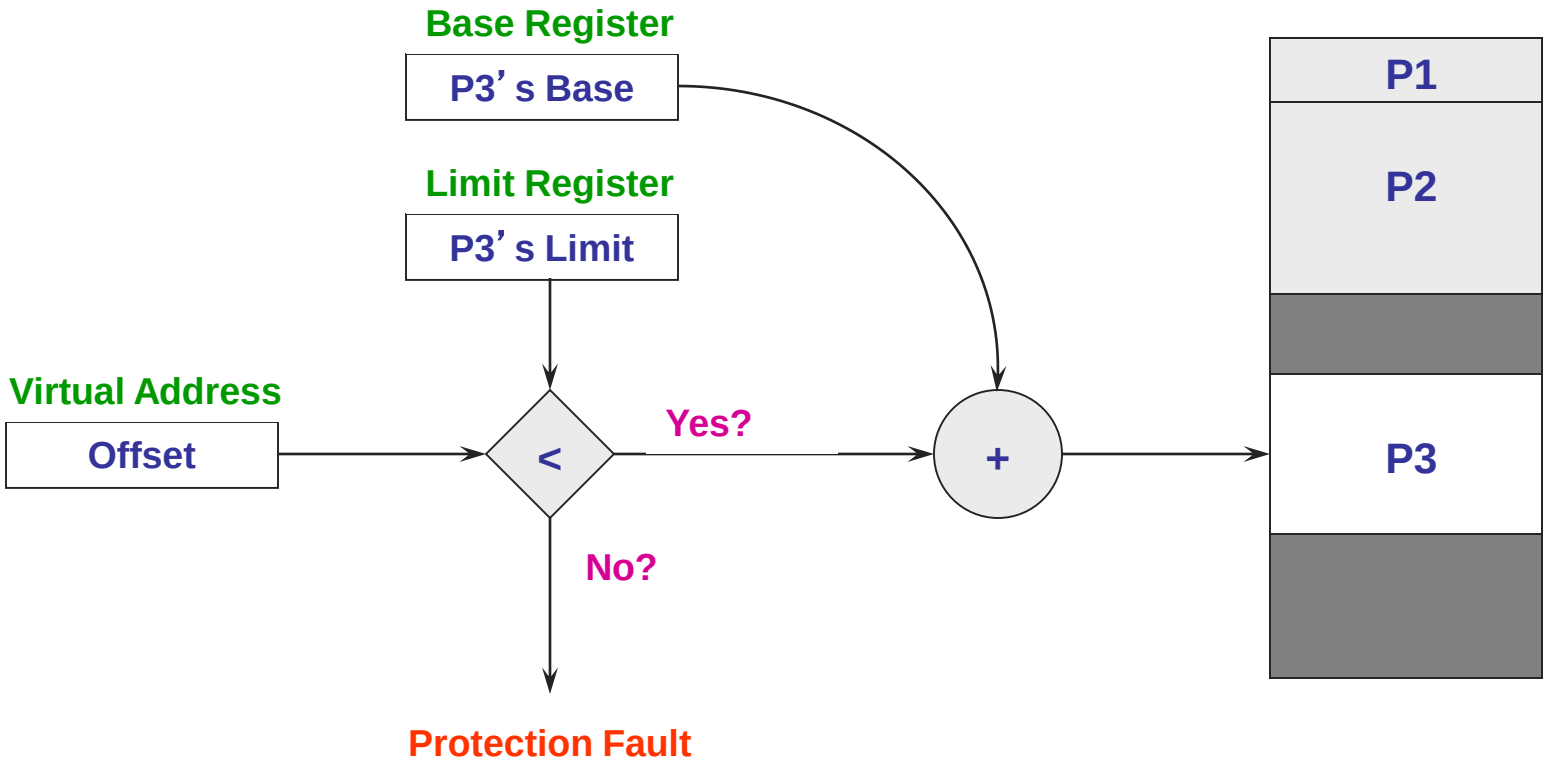
- Problems?
  - Internal fragmentation: memory in a partition not used by a process is not available to other processes
  - Partition size: one size does not fit all (very large processes?)

Variable Partitions

- Natural extension – physical memory is broken up into variable sized partitions
  - Hardware requirements: base register and limit register
  - Physical address = virtual address + base register

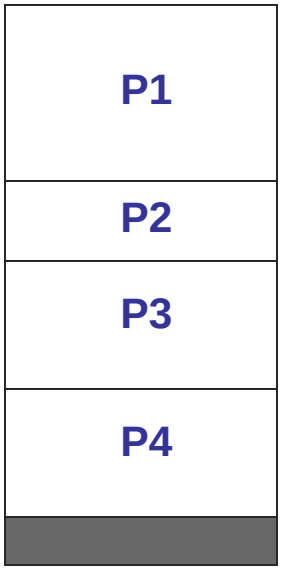
- Why do we need the limit register?
  - Protection: if (virtual address > limit) then fault

Variable Partitions



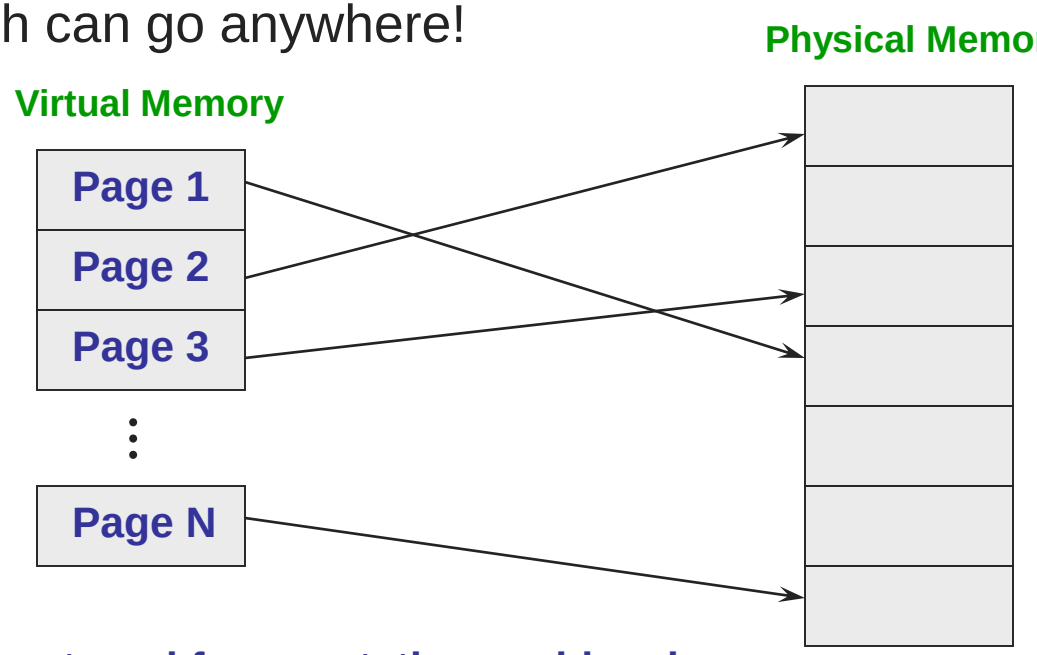
Variable Partitions

- Advantages
  - No internal fragmentation: allocate just enough for process
- Problems?
  - External fragmentation: job loading and unloading produces empty holes scattered throughout memory



Paging

- New Idea: split virtual address space into multiple partitions
  - Each can go anywhere!



Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory

But need to keep track of where things are!

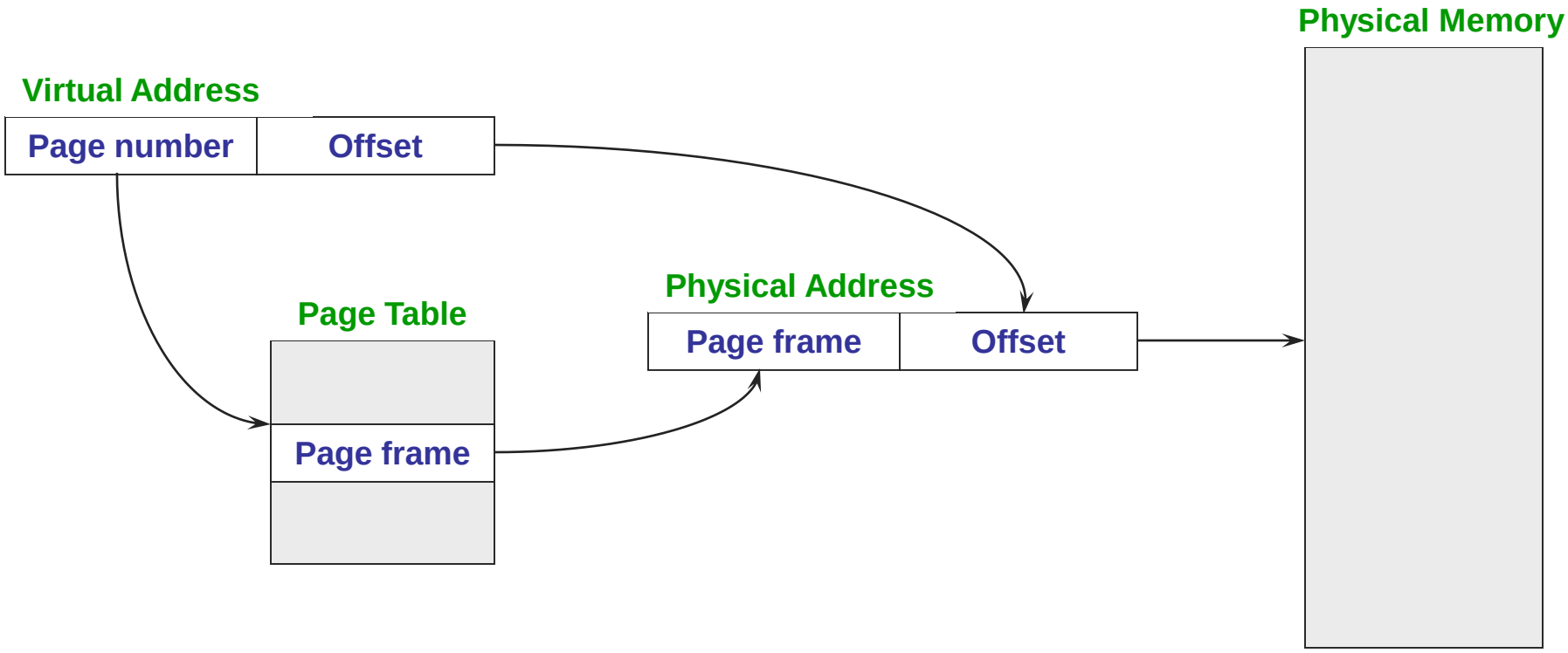
Process Perspective

- Processes view memory as one contiguous address space from 0 through N
  - Virtual address space (VAS)
- In reality, pages are scattered throughout physical storage
- The mapping is invisible to the program
- Protection is provided because a program cannot reference memory outside of its VAS
  - The address “0x1000” maps to different physical addresses in different processes

Paging

- Translating addresses
  - Virtual address has two parts: virtual page number and offset
  - Virtual page number (VPN) is an index into a page table
  - Page table determines page frame number (PFN)
  - Physical address is PFN::offset
- Page tables
  - Map virtual page number (VPN) to page frame number (PFN)
    - » VPN is the index into the table that determines PFN
  - One page table entry (PTE) per page in virtual address space
    - » Or, one PTE per VPN

Page Lookups



Paging Example

- Pages are 4KB
  - Offset is 12 bits (because 4KB = 2^{12} bytes)
  - VPN is 20 bits (32 bits is the length of every virtual address)
- Virtual address is 0x7468
  - Virtual page is 0x7, offset is 0x468
- Page table entry 0x7 contains 0x2000
  - Page frame number is 0x2000
  - Seventh virtual page is at address 0x2000 (2nd physical page)
- Physical address = 0x2000 + 0x468 = 0x2468

Page Table Entries (PTEs)

- Page table entries control mapping
  - The Modify bit says whether or not the page has been written
    - » It is set when a write to the page occurs
  - The Reference bit says whether the page has been accessed
    - » It is set when a read or write to the page occurs
  - The Valid bit says whether or not the PTE can be used
    - » It is checked each time the virtual address is used (Why?)
  - The Protection bits say what operations are allowed on page
    - » Read, write, execute (Why do we need these?)
  - The page frame number (PFN) determines physical page

Paging Advantages

- Easy to allocate memory
  - Memory comes from a free list of fixed size chunks
  - Allocating a page is just removing it from the list
  - External fragmentation not a problem
    - » All pages of the same size
- Simplifies protection
  - All chunks are the same size
  - Like fixed partitions, don't need a limit register
- Simplifies virtual memory – later

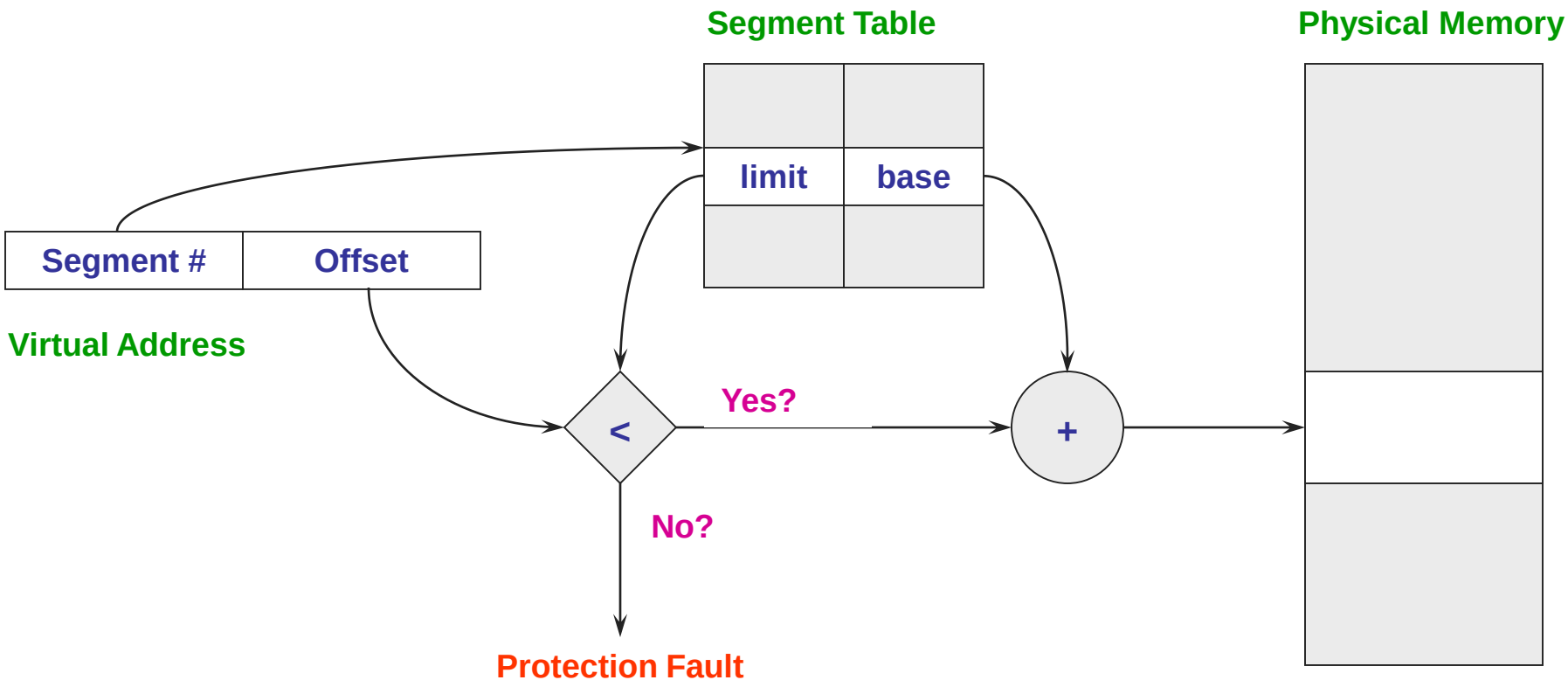
Paging Limitations

- Can still have internal fragmentation
  - Process may not use memory in multiples of a page
- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - What can we do?
- Memory required to hold page table can be significant
  - Need one PTE per page
  - 32 bit address space w/ 4KB pages = 2^{20} PTEs
  - 4 bytes/PTE = 4MB/page table
  - 25 processes = 100MB just for page tables!
  - What can we do?

Segmentation

- Segmentation: partition memory into logically related units
  - Module, procedure, stack, data, file, etc.
  - Units of memory from user's perspective
- Natural extension of variable-sized partitions
  - Variable-sized partitions = 1 segment/process
  - Segmentation = many segments/process
  - Fixed partition : Paging :: Variable partition : Segmentation
- Hardware support
  - Multiple base/limit pairs, one per segment (segment table)
  - Segments named by #, used to index into table
  - Virtual addresses become <segment #, offset>

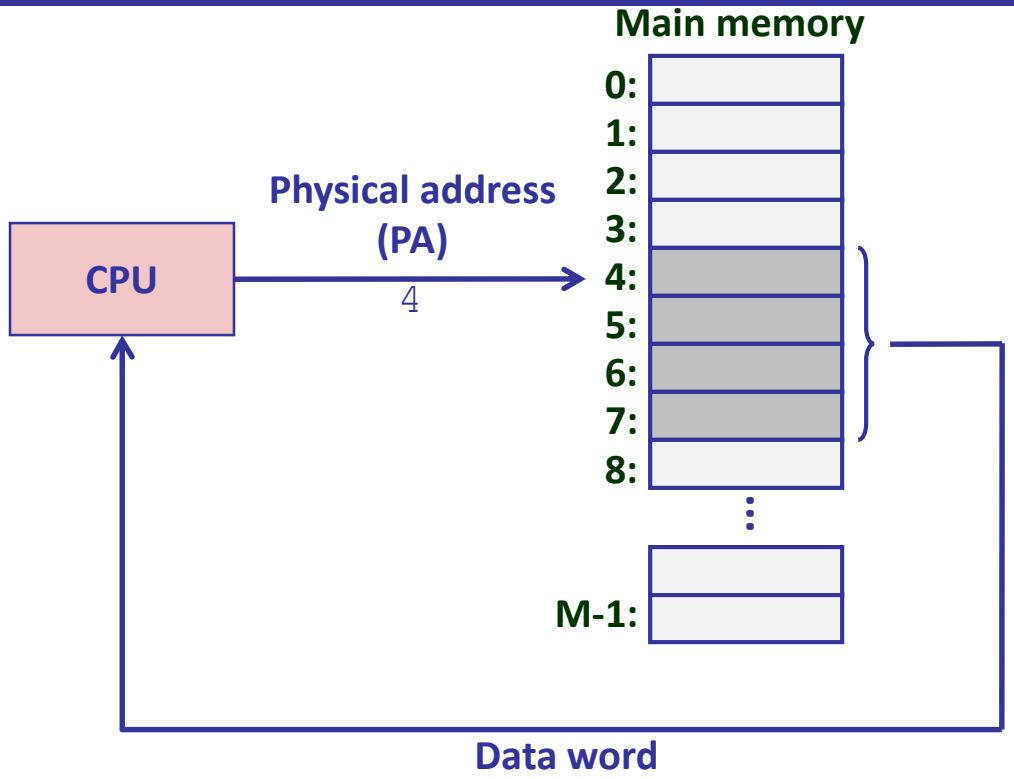
Segment Lookups



Today

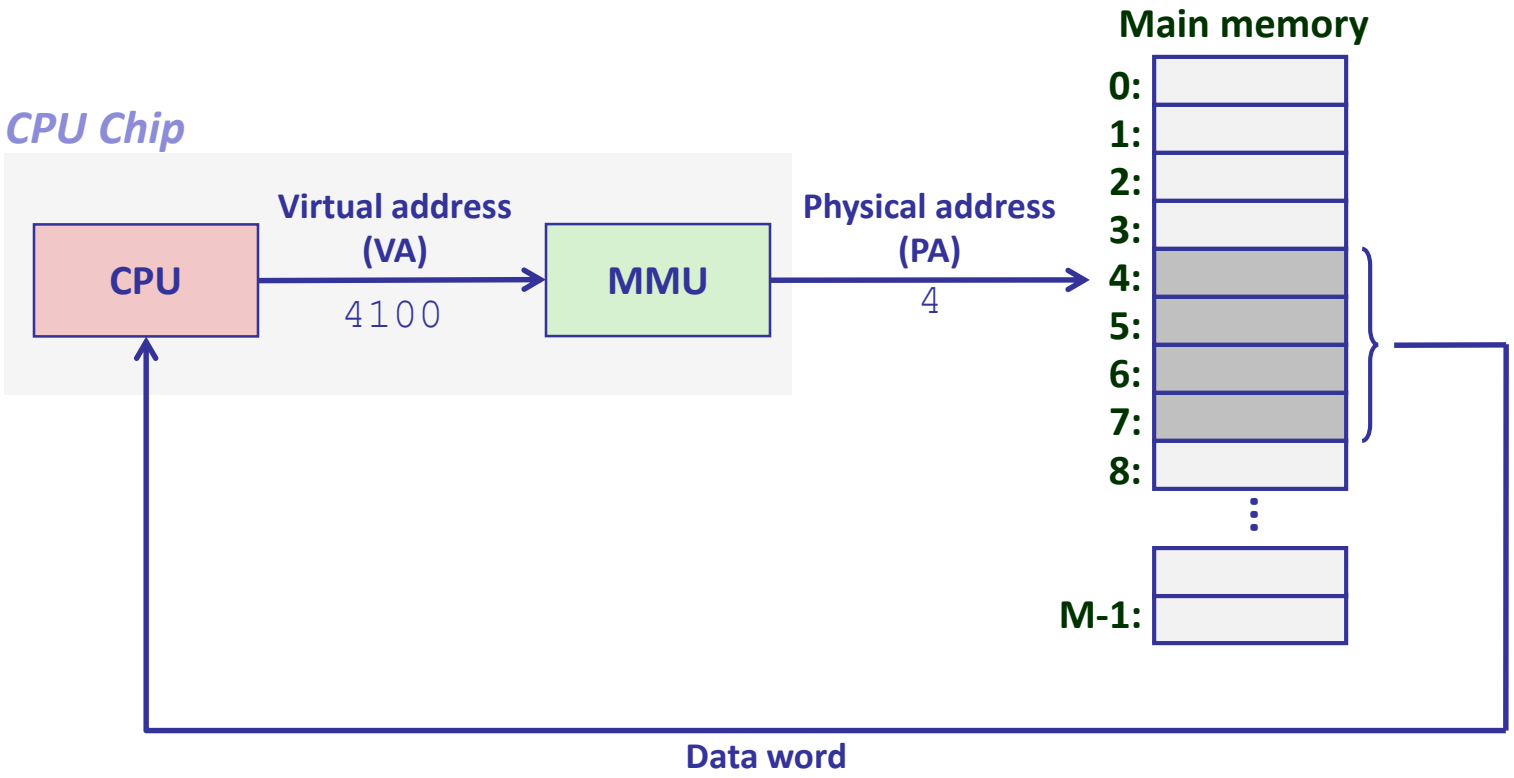
- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation

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)?

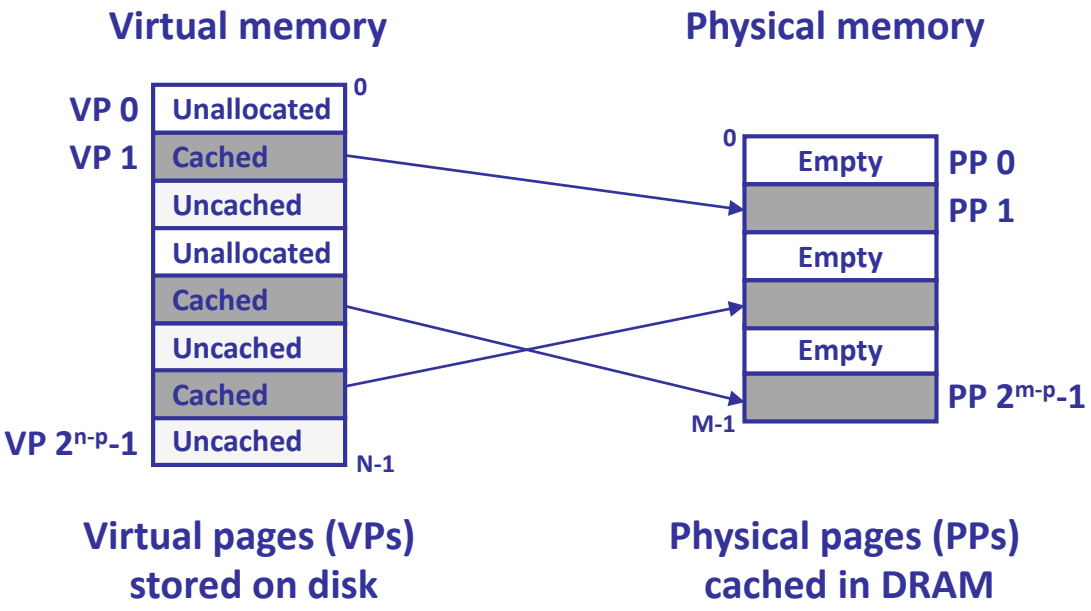
- Virtual memory is page with a new ingredient
  - Allow pages to be on disk
    - » In a special partition (or file) called swap
- Motivation?
  - Uses main memory efficiently
  - Use DRAM as a cache for the parts of a virtual address space
- Simplifies memory management
  - Each process gets the same uniform linear address space
  - With VM, this can be big!

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

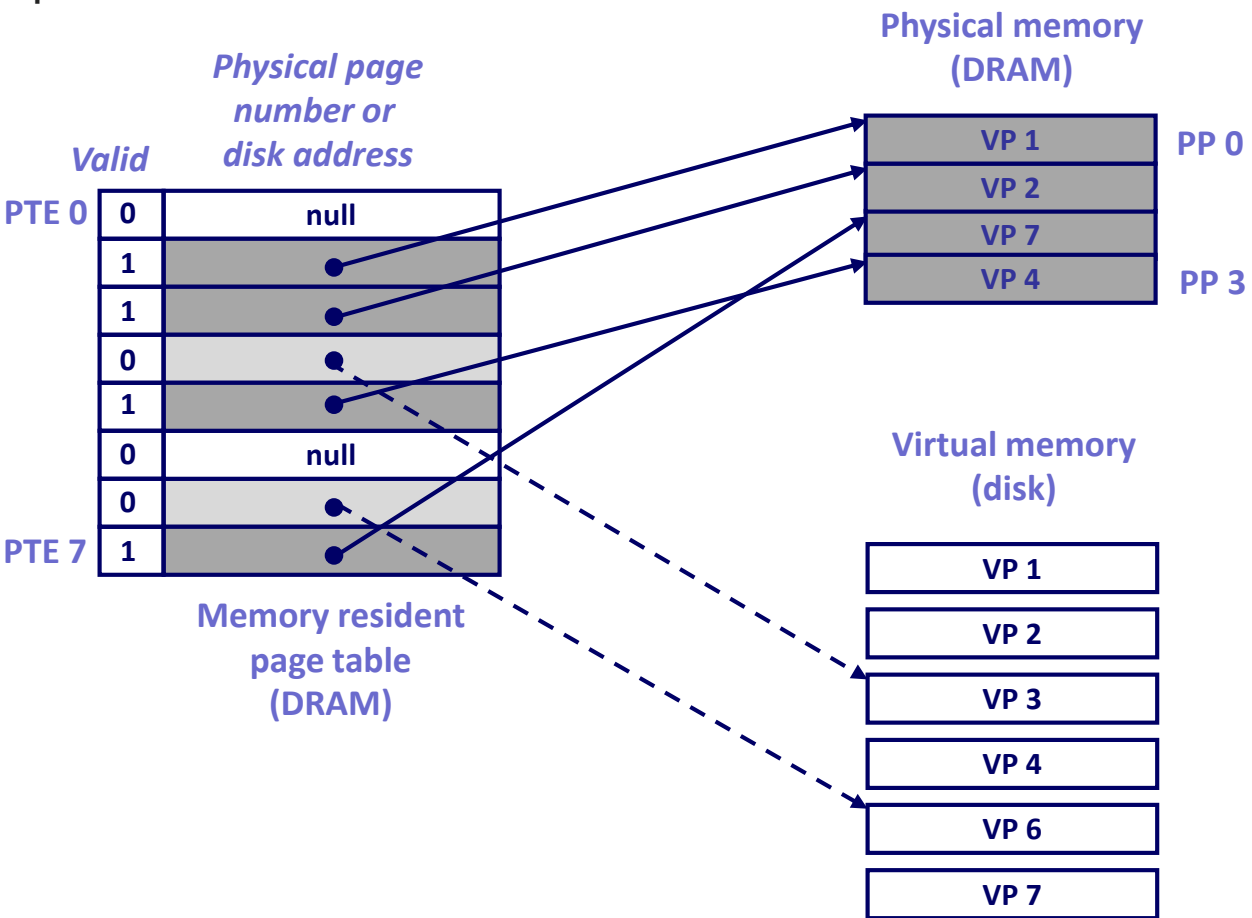
- DRAM cache organization driven by the enormous miss penalty
  - DRAM is about ***10x*** slower than SRAM
  - Disk is about ***10,000x*** slower than DRAM

- 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

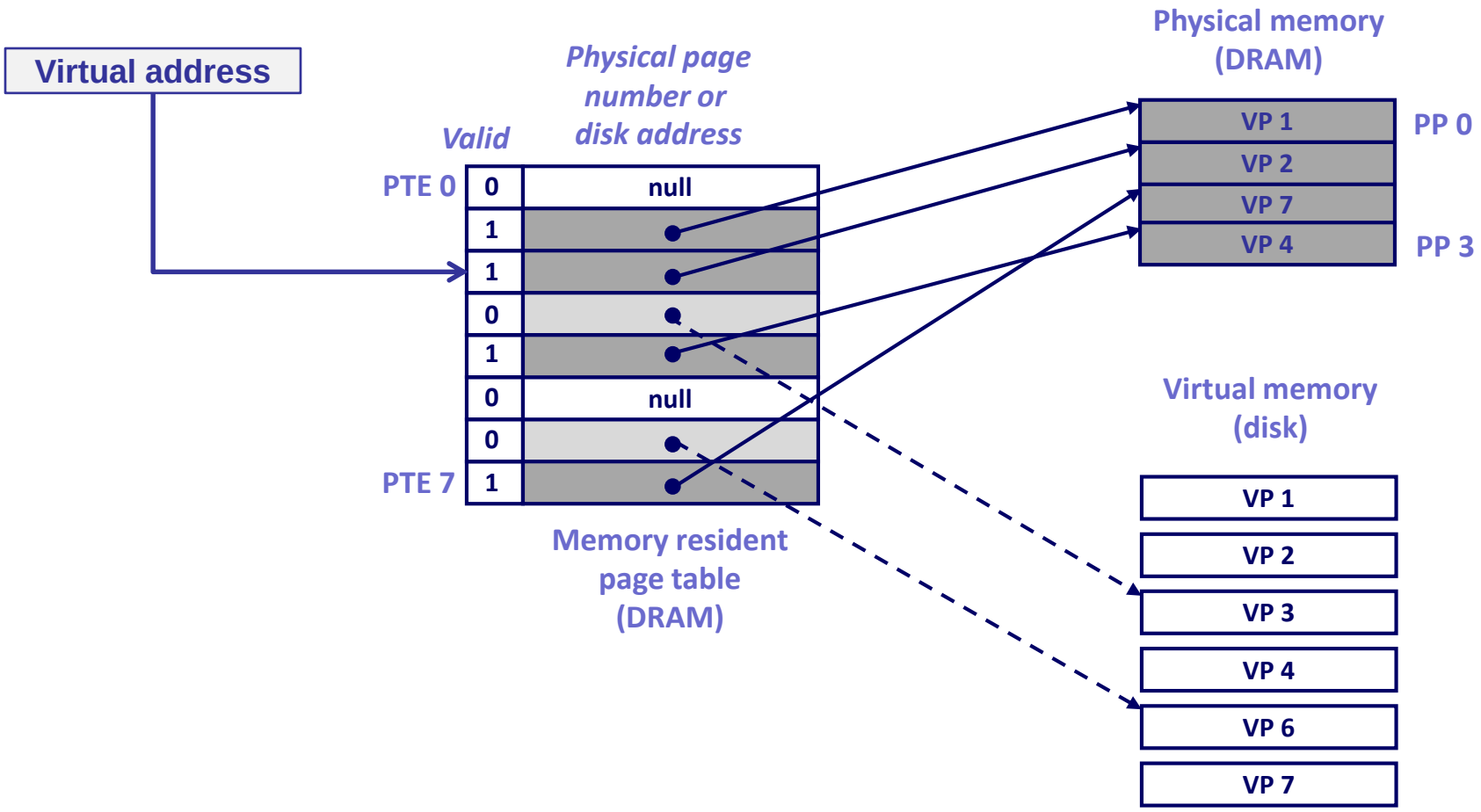
Page Tables

- 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



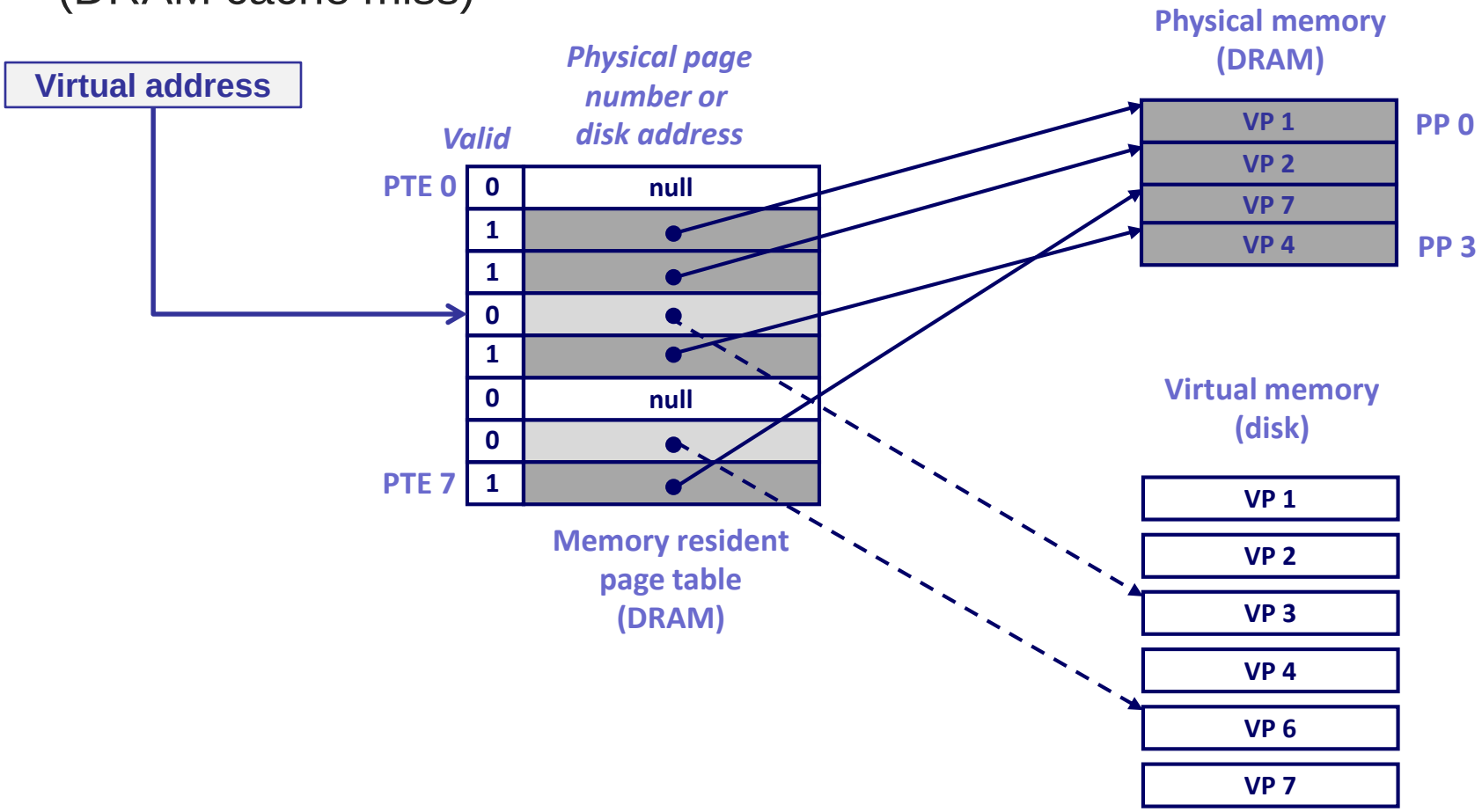
Page Hit

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



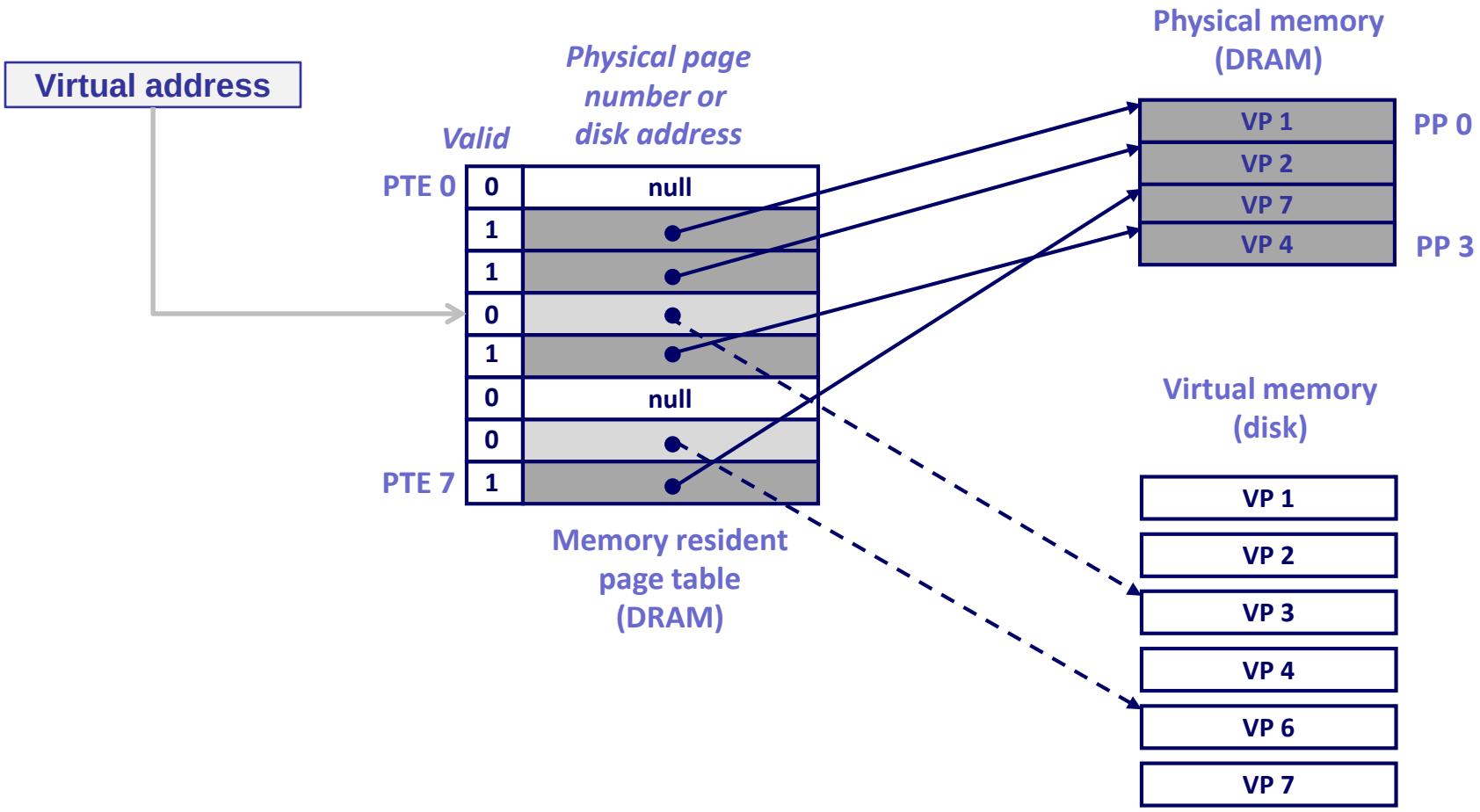
Page Fault

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



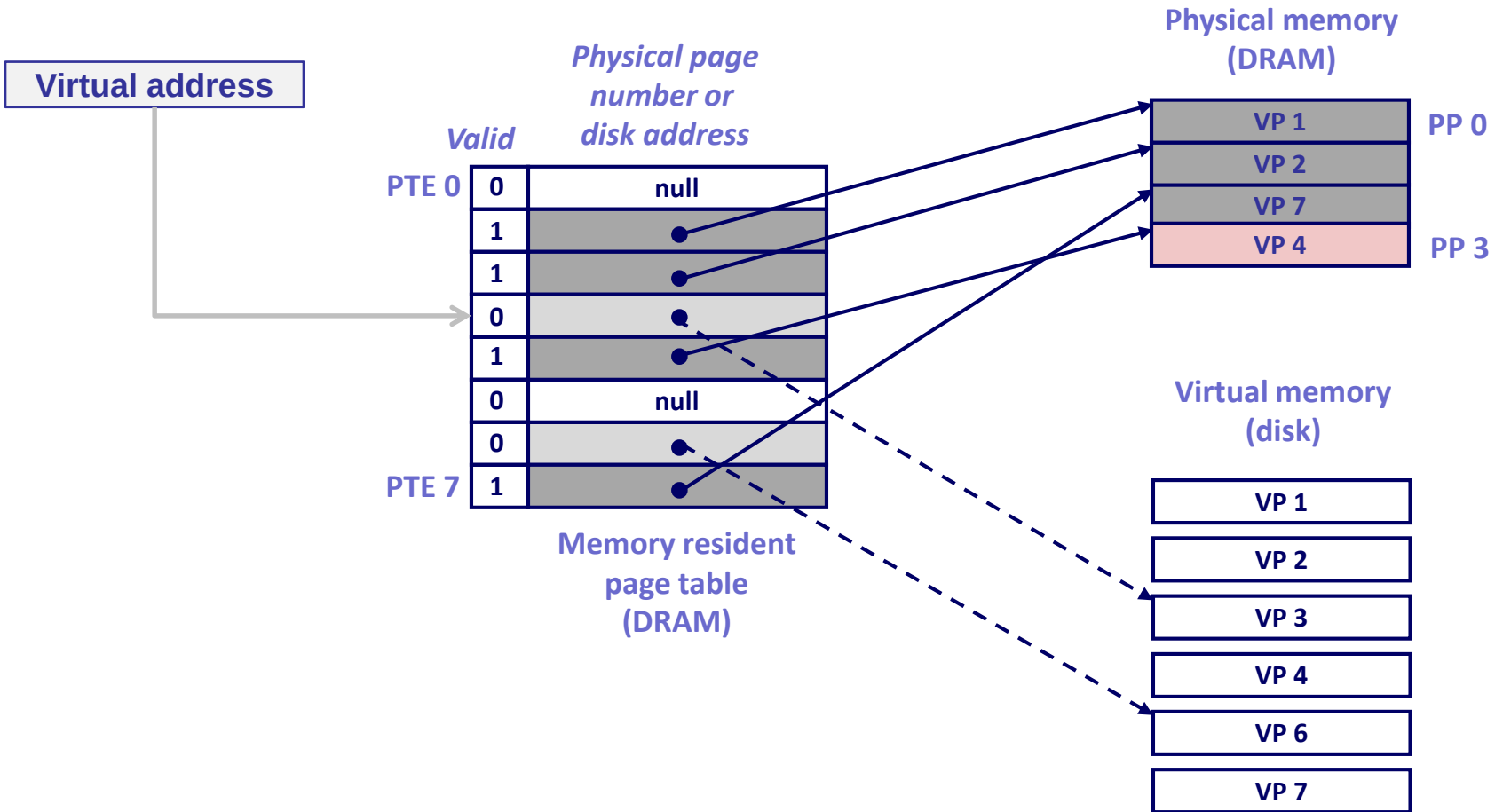
Handling Page Fault

- Page miss causes page fault (an exception)



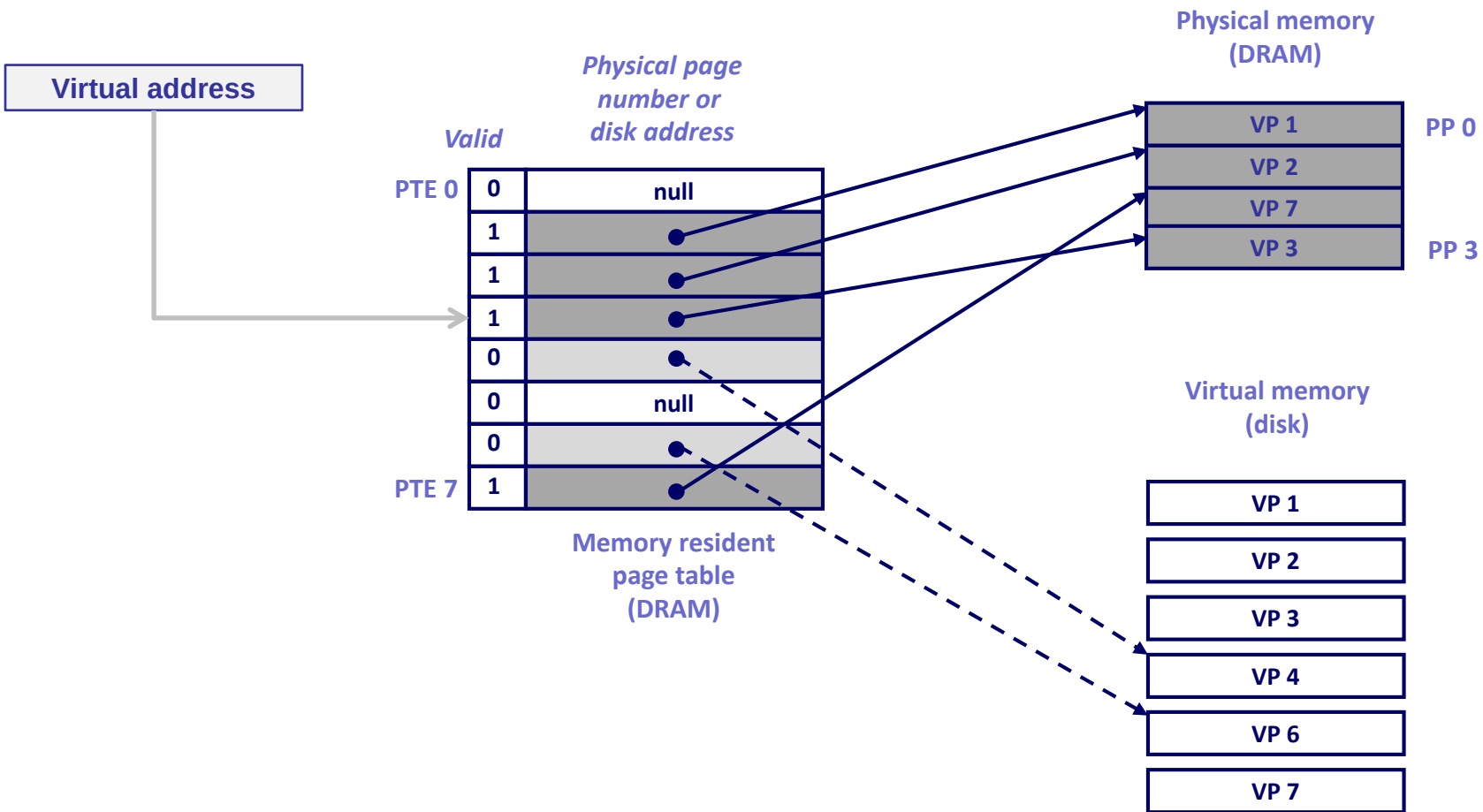
Handling Page Fault

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



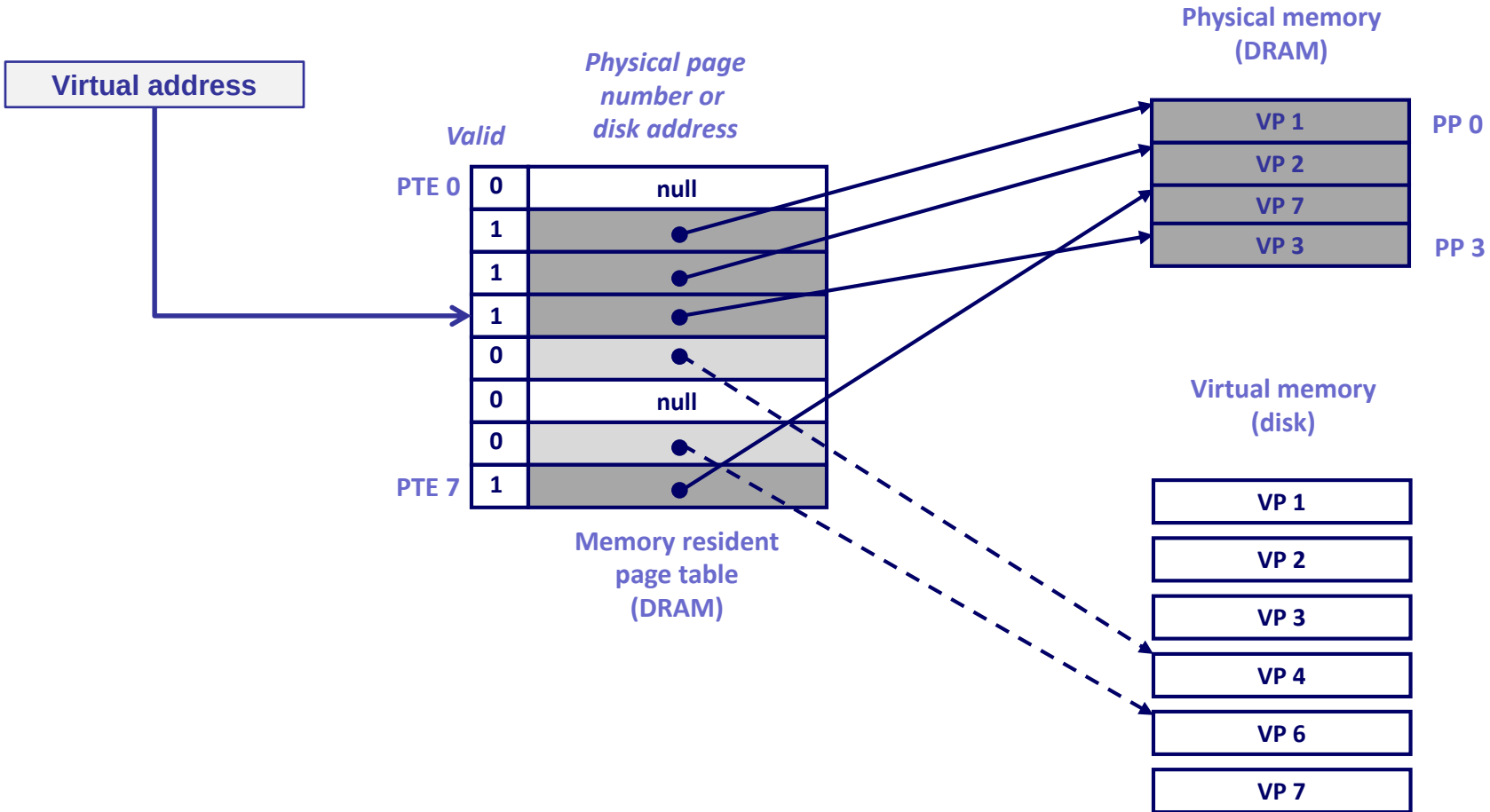
Handling Page Fault

- 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!

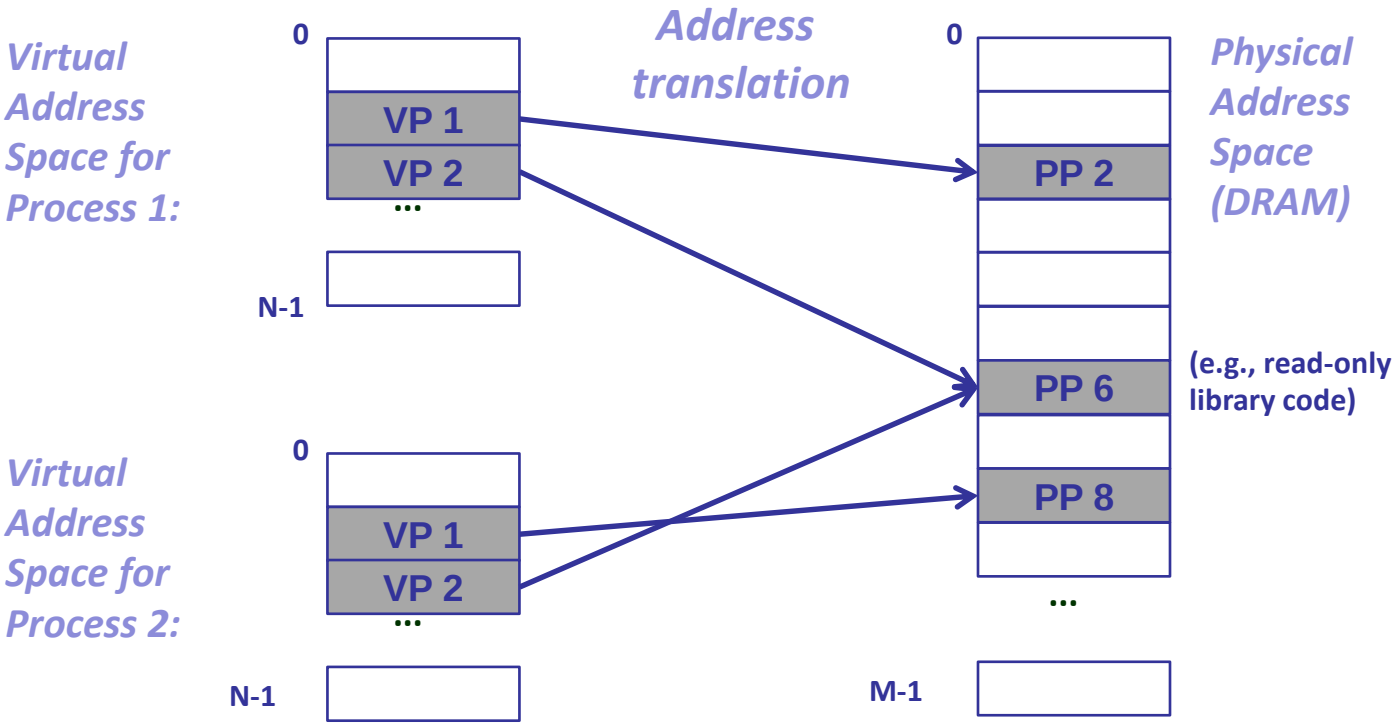
- Virtual memory works because of locality
- At any point in time, programs tend to access a set of active virtual pages called the *working set*
  - Programs with better temporal locality will have smaller working sets
- If (working set size < main memory size)
  - Good performance for one process after 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

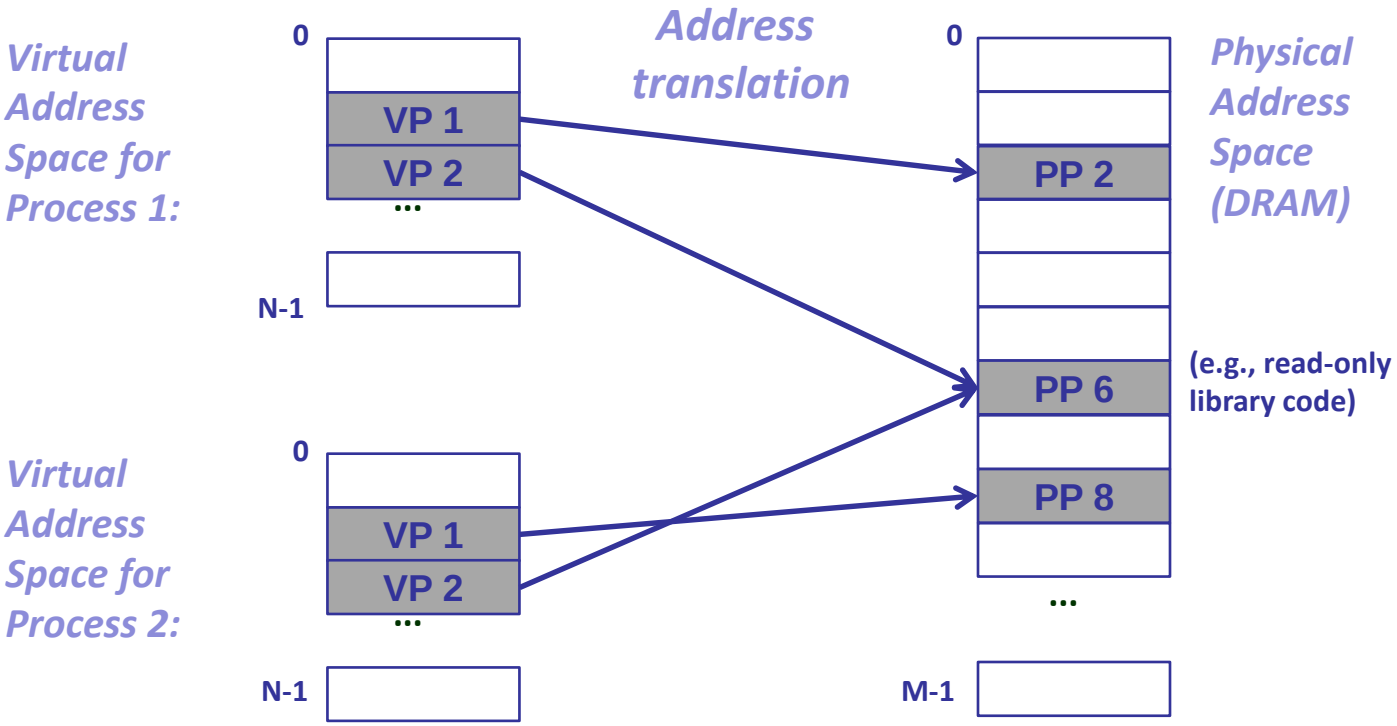
VM as a Tool for Memory Management

- 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)



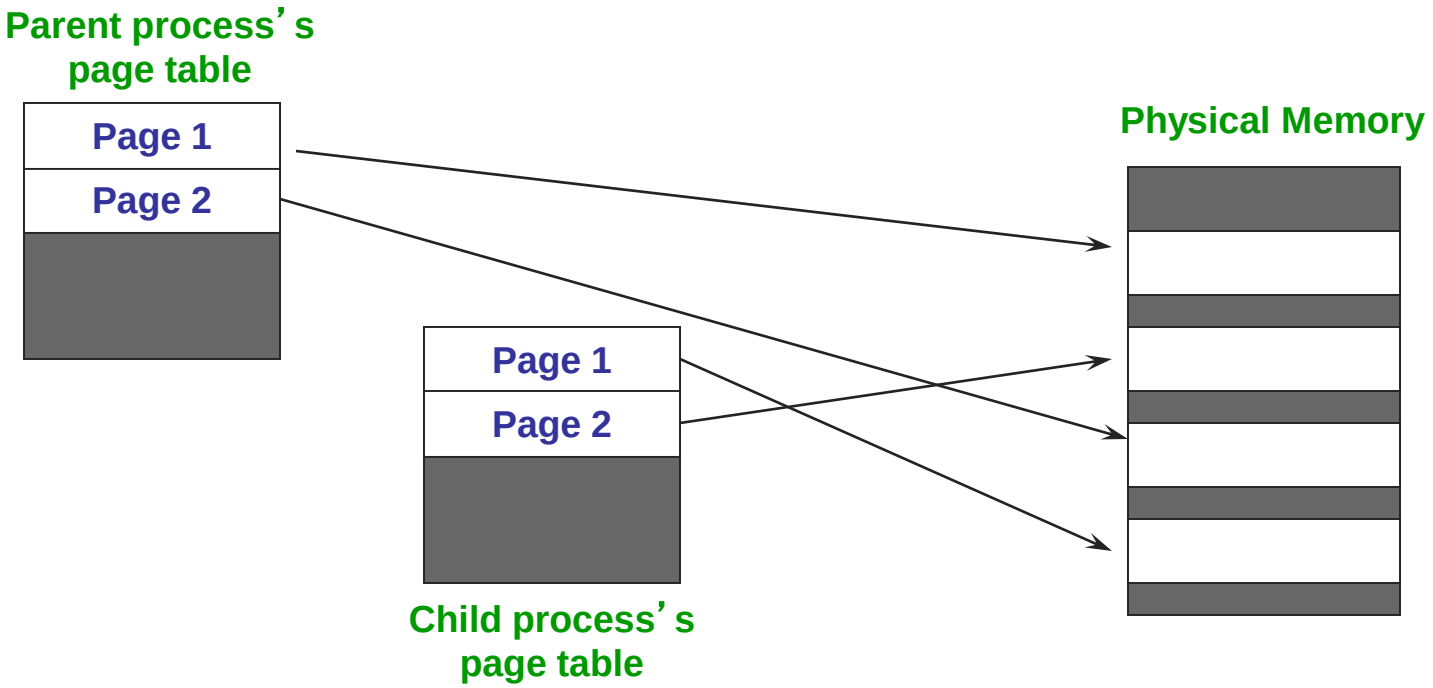
Sharing

- Can map shared memory at same or different virtual addresses in each process' address space
  - Different:
    - » 10^{th} virtual page in P1 and 7^{th} virtual page in P2 correspond to the 2^{nd} physical page
    - » Flexible (no address space conflicts), but pointers inside the shared memory segment are invalid
  - Same:
    - » 2^{nd} physical page corresponds to the 10^{th} virtual page in both P1 and P2
    - » Less flexible, but shared pointers are valid

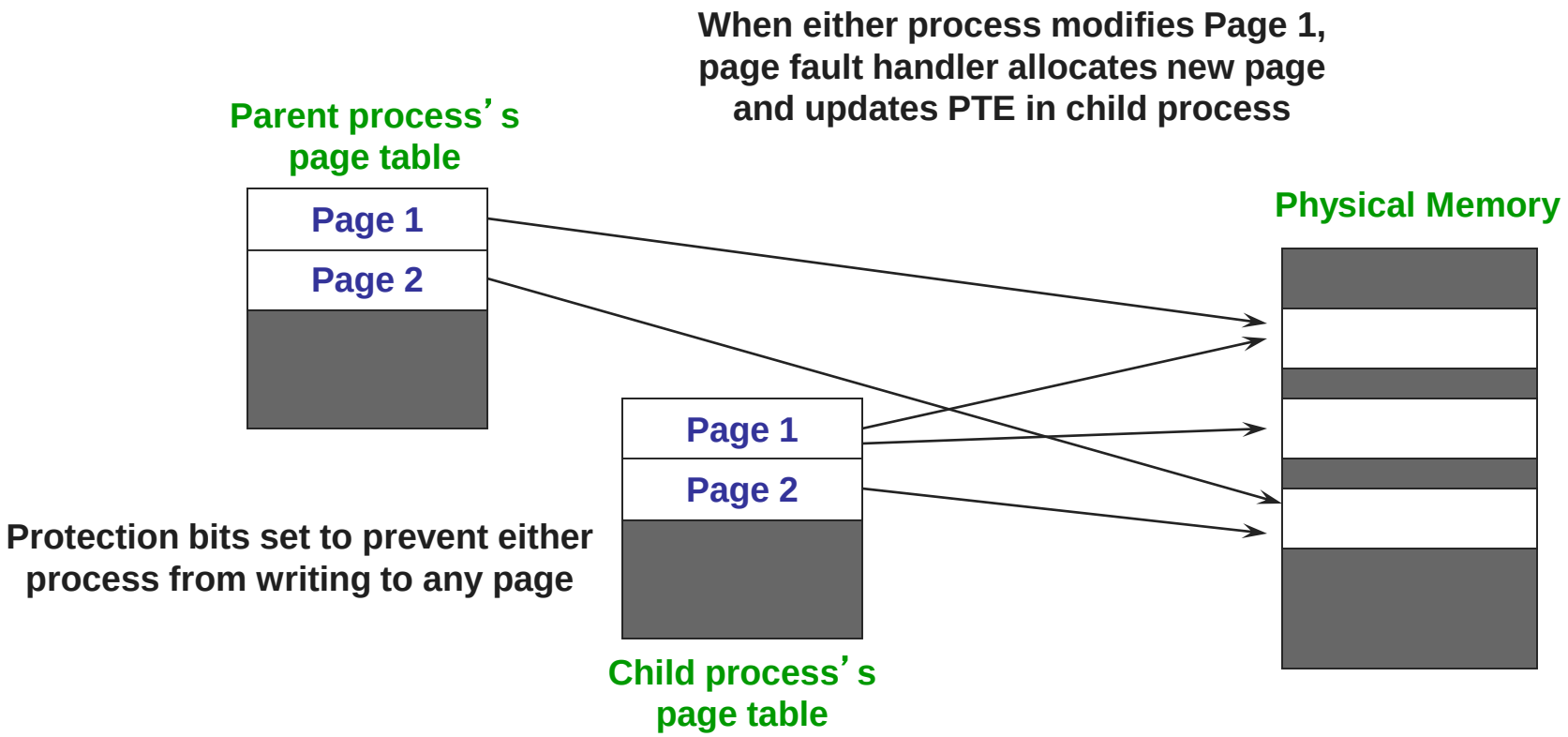
Copy on Write

- OSes spend a lot of time copying data
  - System call arguments between user/kernel space
  - Entire address spaces to implement fork()
- Use Copy on Write (CoW) to defer large copies as long as possible, hoping to avoid them altogether
  - Instead of copying pages, create shared mappings of parent pages in child virtual address space
  - Shared pages are protected as read-only in parent and child
    - » Reads happen as usual
    - » Writes generate a protection fault, trap to OS, copy page, change page mapping in client page table, restart write instruction
  - How does this help fork()?

Execution of fork()

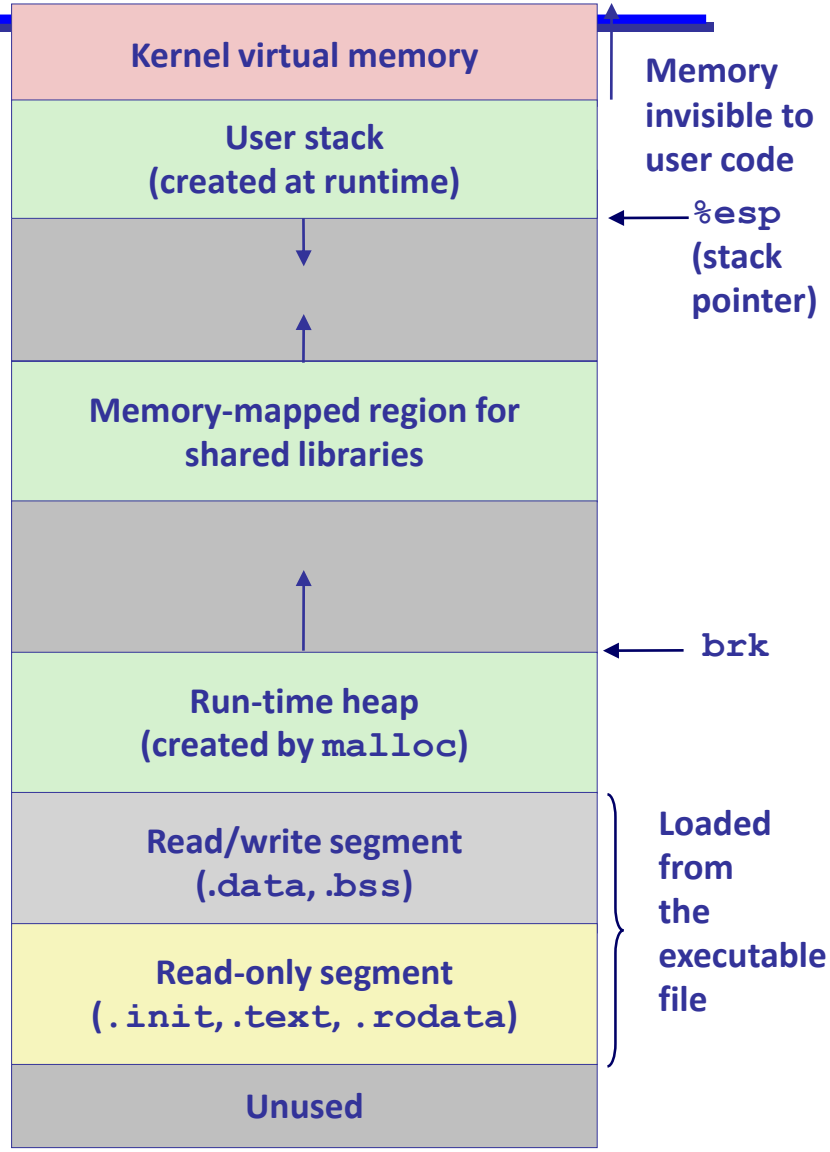


fork() with Copy on Write



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

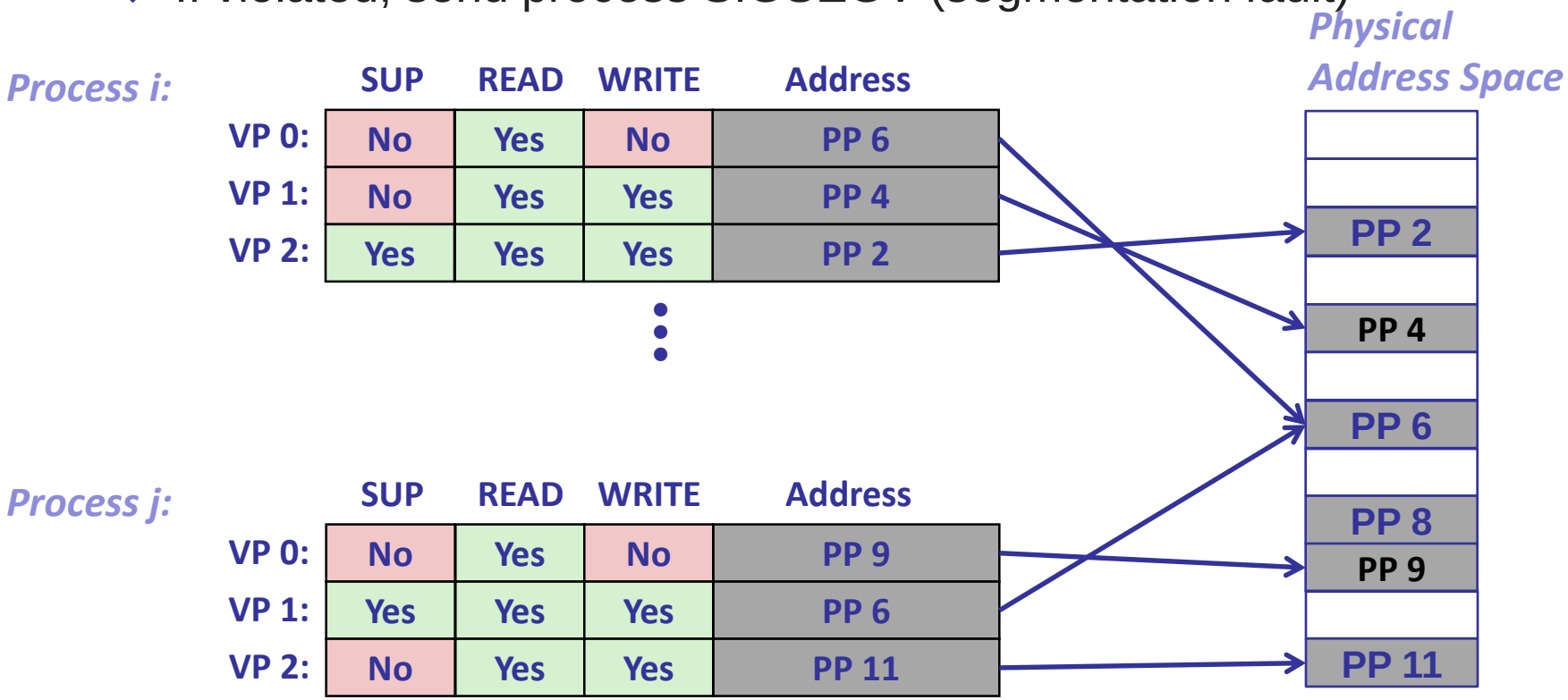


Today

- 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

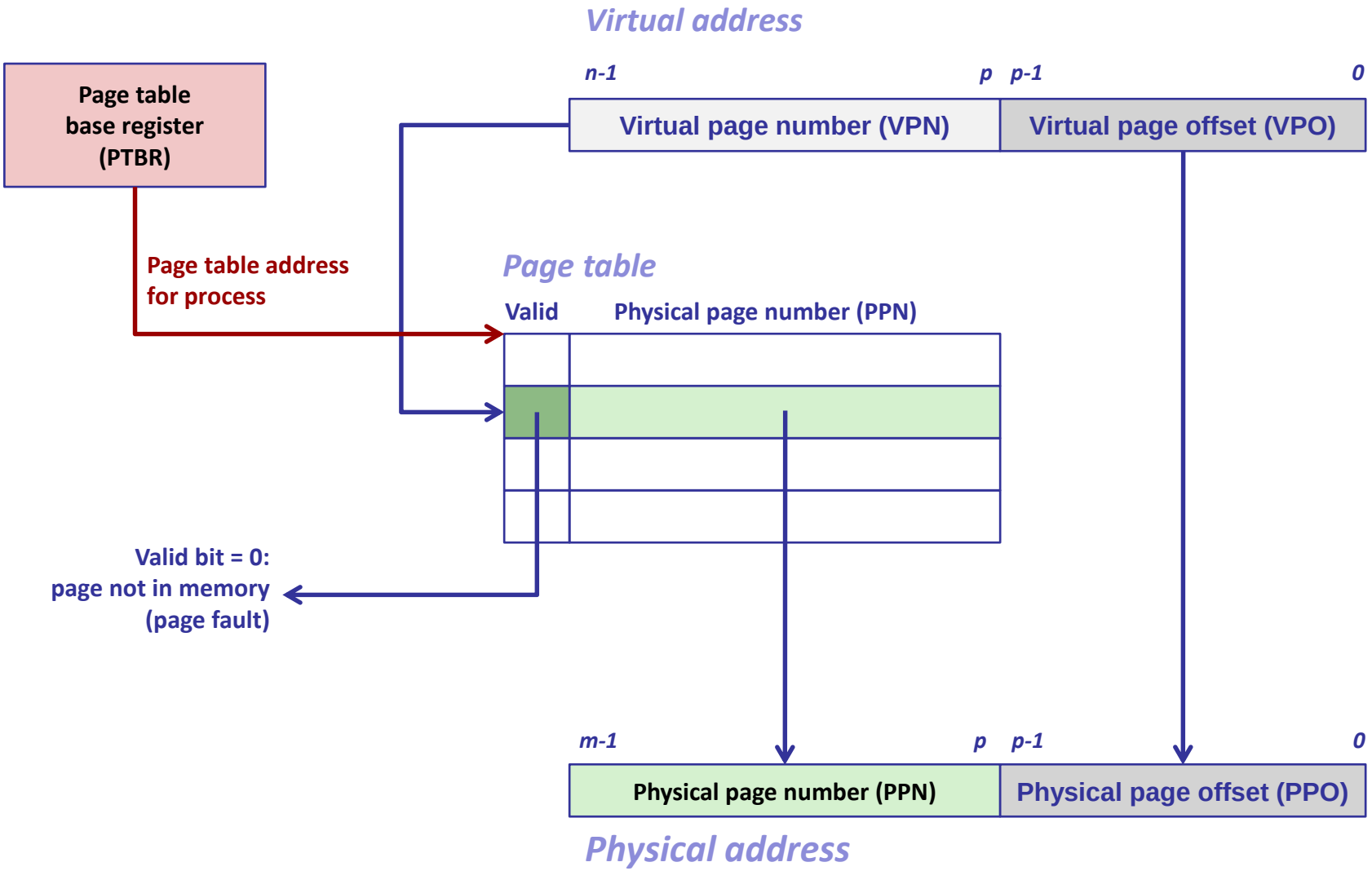
- Extend PTEs with permission bits
- Page fault handler checks these before remapping
  - If violated, send process SIGSEGV (segmentation fault)



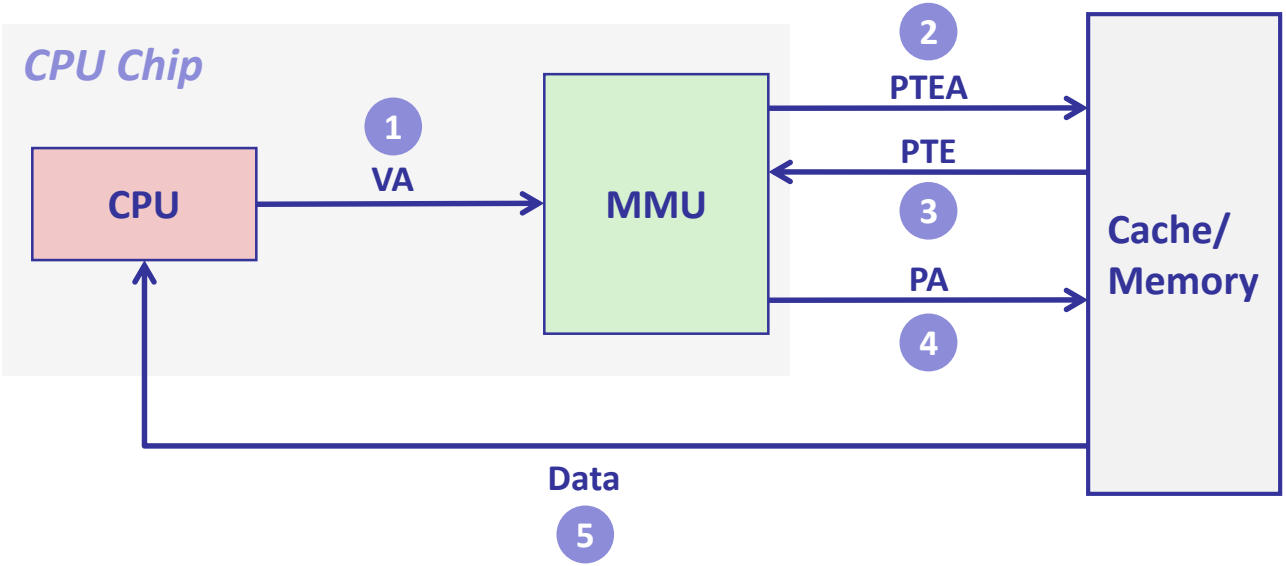
Today

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

Address Translation With a Page Table



Address Translation: Page Hit



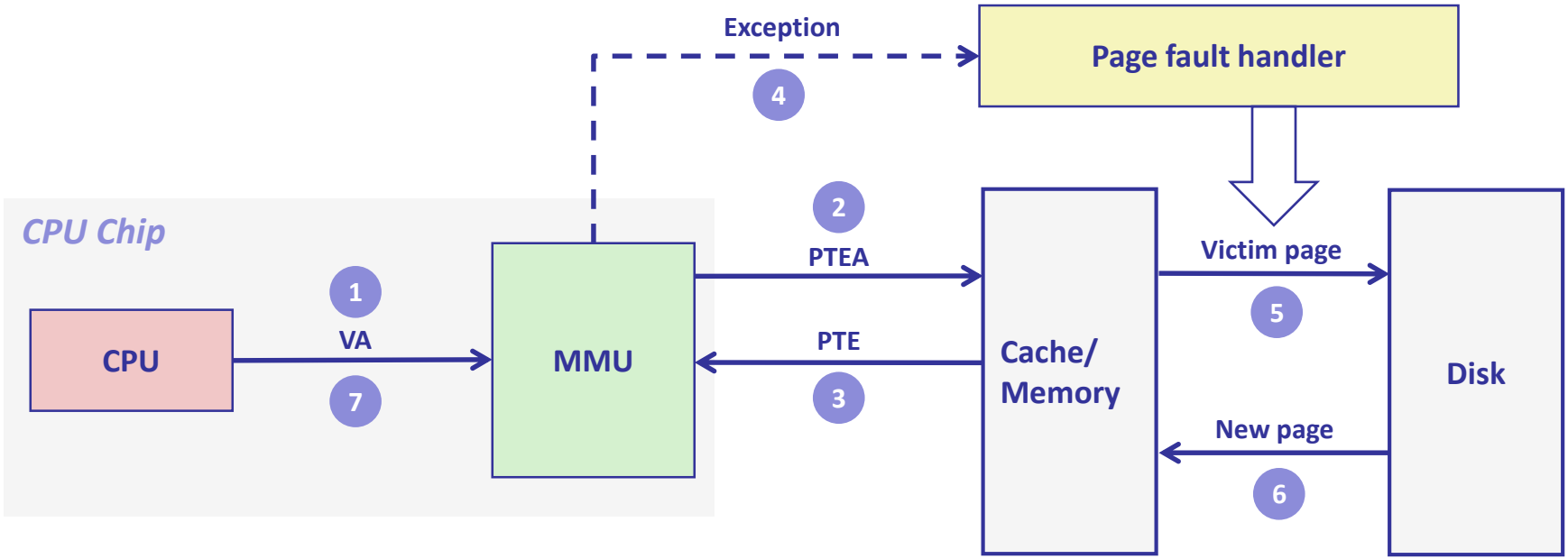
1) Processor sends virtual address to MMU

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

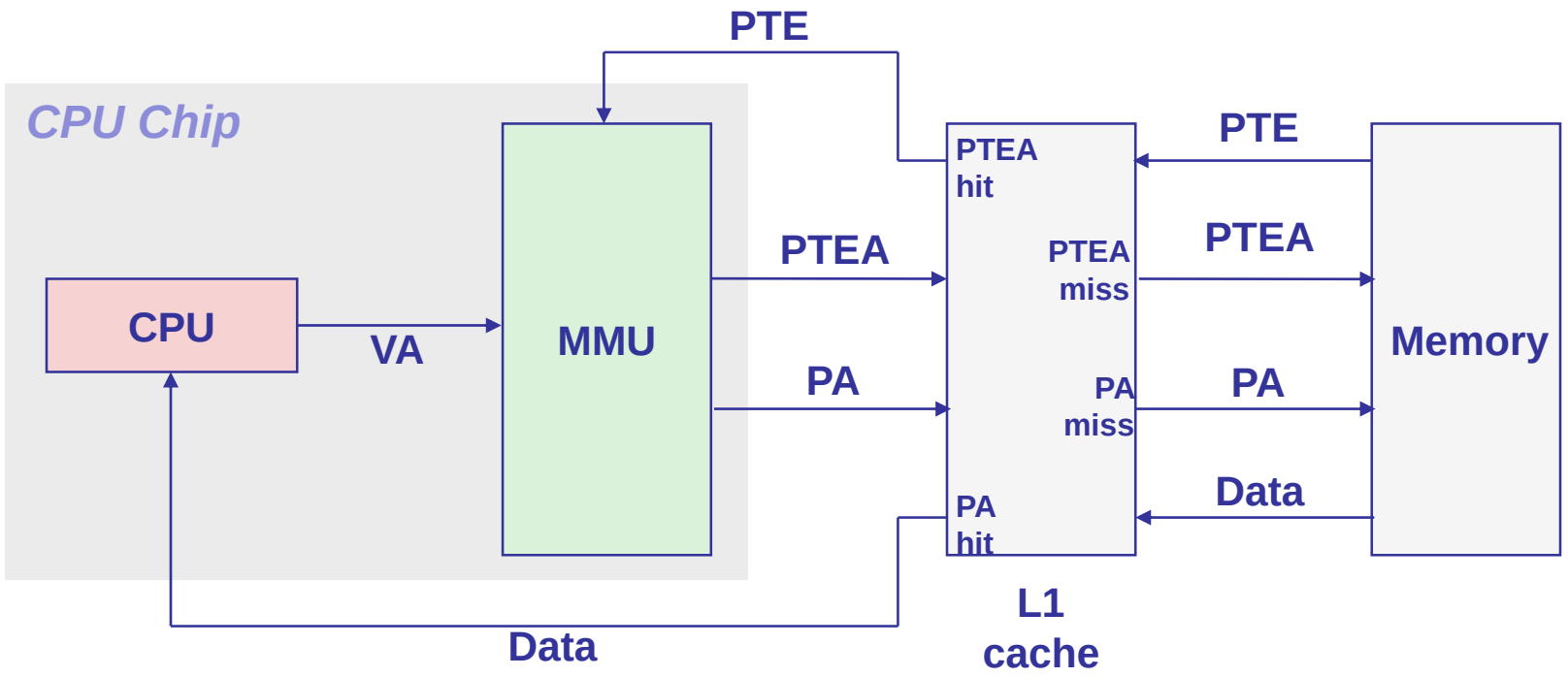
4) Valid bit is zero, so MMU triggers page fault exception

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

Elephant(s) in the room

- Problem 1: Translation is slow!
  - Many memory accesses for each memory access
  - Caches are useless!

- Problem 2: Page table can be gigantic!
- We need one for each process
- All your memory are belong to us!

"Unfortunately, there's another elephant in the room."