Virtual Memory I

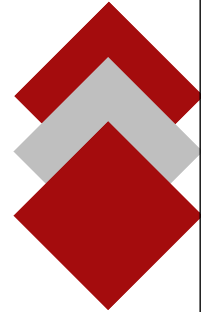
15-213/18-243: Introduction to Computer Systems

15th Lecture, 16 March 2010

Instructors:

Bill Nace and Gregory Kesden

(c) 1998 - 2010. All Rights Reserved. All work contained herein is copyrighted and used by permission of the authors. Contact 15-213-staff@cs.cmu.edu for permission or for more information.

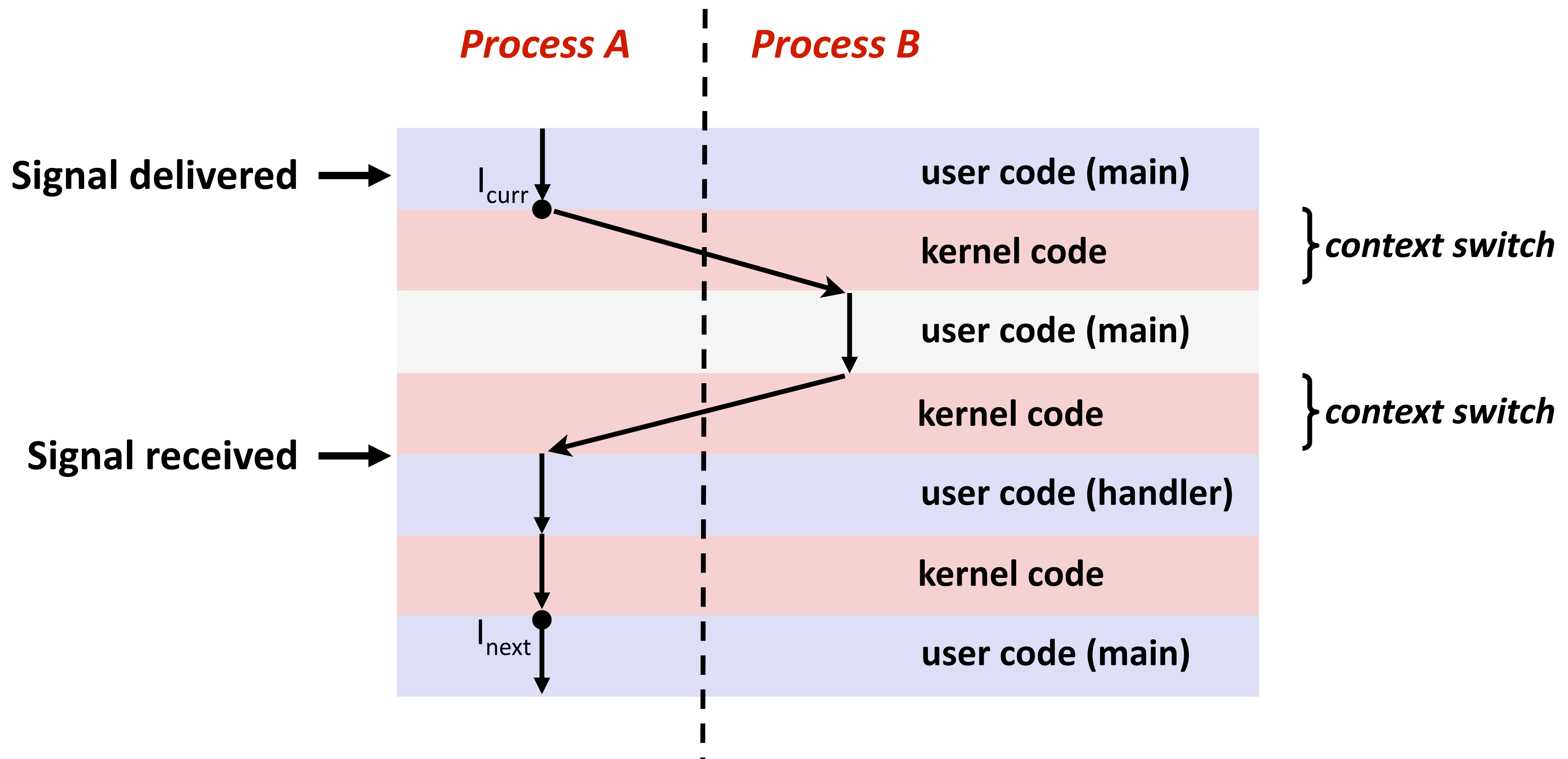


Signals

- Kernel → Process
- Process → Process (using `kill`)
- 32 types of signals
- Sent by updating bit in `pending` vector
- You can write your own signal handlers

ID	*Name*	*Default Action*	*Corresponding Event*
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Signal Handlers as Concurrent Flows



Nonlocal Jumps: `setjmp/longjmp`

- Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location
  - Controlled to way to break the procedure call / return discipline
  - Useful for error recovery and signal handling

- **`int setjmp(jmp_buf buf)`**
  - Must be called before longjmp
  - Identifies a return site for a subsequent longjmp
  - Called once, returns one or more times

- Implementation:
  - Remember where you are by storing the current ***register context***, ***stack pointer***, and ***PC value*** in **`jmp_buf`**
  - Return 0

`setjmp/longjmp` (cont)

- `void longjmp(jmp_buf buf, int i)`
  - Meaning:
    - return from the **`setjmp`** remembered by jump buffer **`buf`** again ...
    - ... this time returning **`i`** instead of 0
  - Called after **`setjmp`**
  - Called once, but never returns

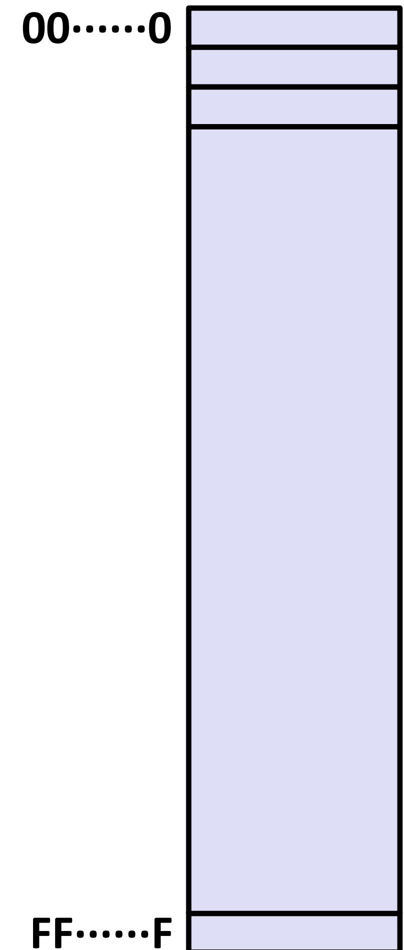
- **`longjmp`** Implementation:
  - Restore register context (stack pointer, base pointer, PC value) from jump buffer **`buf`**
  - Set `%`**`eax`** (the return value) to **`i`**
  - Jump to the location indicated by the PC stored in jump buf **`buf`**

Today

- Virtual Memory (VM) overview and motivation
- VM as tool for caching
- VM as tool for memory management
- VM as tool for memory protection
- Address translation

Virtual Memory Abstraction

- **Programs refer to virtual memory addresses**
  - **movl (%ecx),%eax**
  - Conceptually very large array of bytes
  - Each byte has its own address
  - Actually implemented with hierarchy of different memory types
  - System provides address space private to particular "process"
- **Allocation: Compiler and run-time system**
  - Where different program objects should be stored
  - All allocation within single virtual address space
- ***But why virtual memory?***
- ***Why not physical memory?***



Problem 1: How Does Everything Fit?

64-bit addresses:
16 Exabyte

Physical main memory:
Few Gigabytes

And there are many processes ….

Problem 2: Memory Management

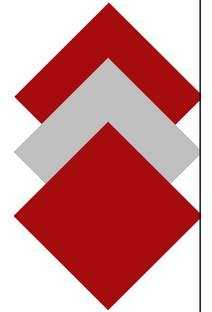
Process 1
Process 2
Process 3
…
Process n

X

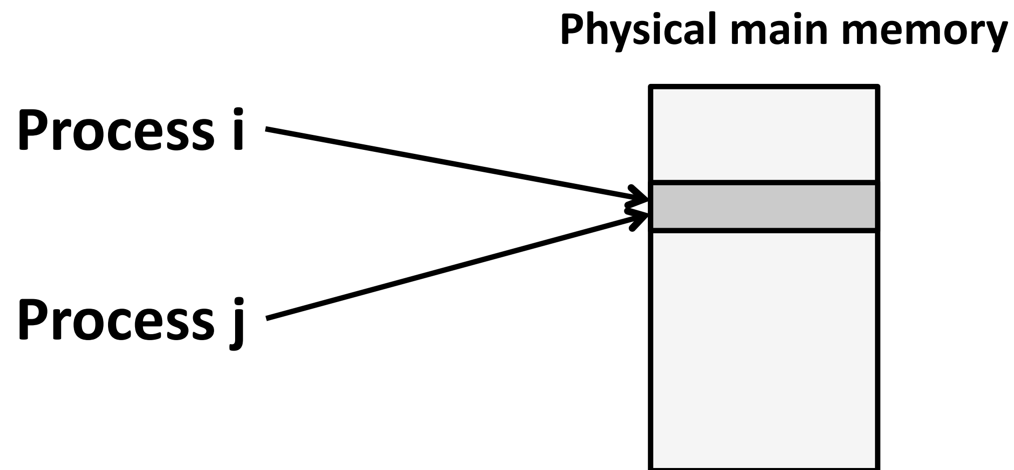
stack
heap
`.text`
`.data`
…

What goes where?

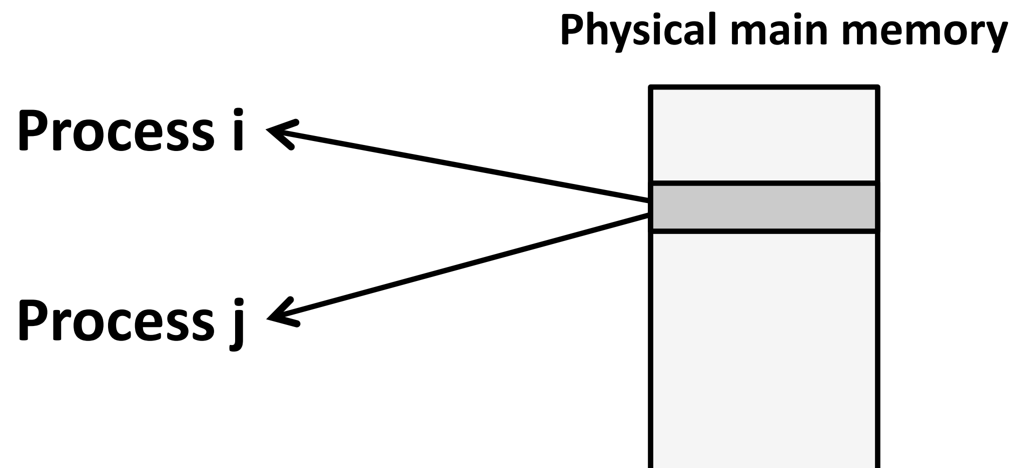
Physical main memory



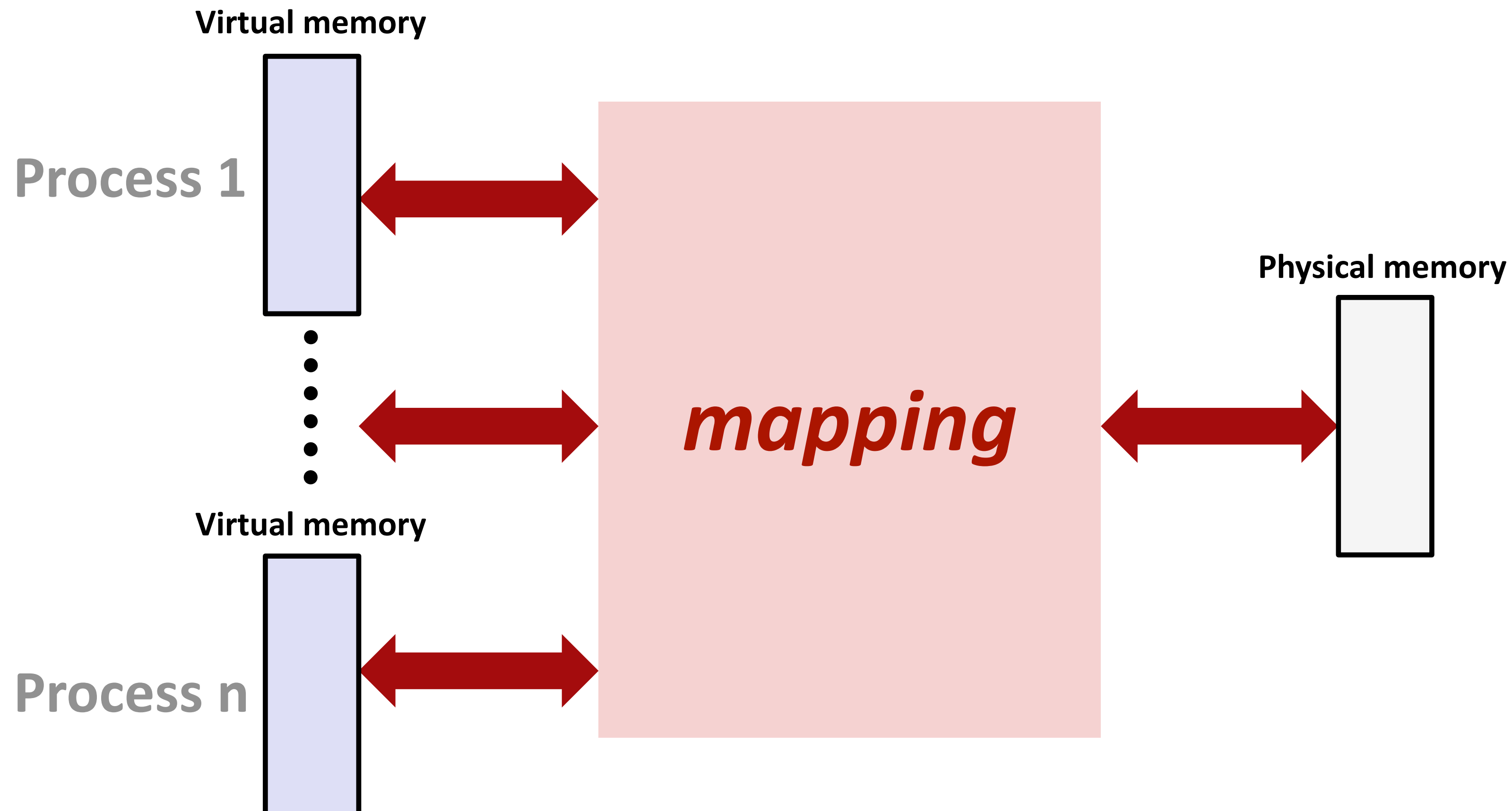
Problem 3: How To Protect



Problem 4: How To Share?



Solution: Level Of Indirection



- Each process gets its own private memory space
- Solves the previous problems

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

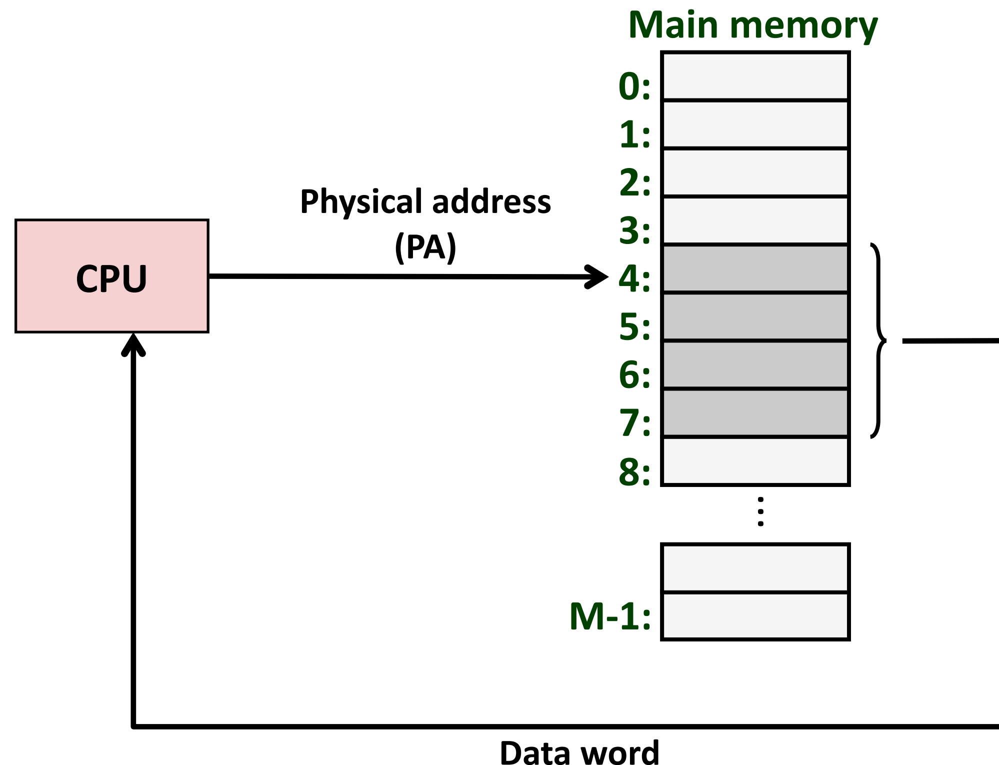
  {0, 1, 2, 3, …, N-1}

- **Physical address space:** Set of $M = 2^m$ physical addresses

  {0, 1, 2, 3, …, M-1}

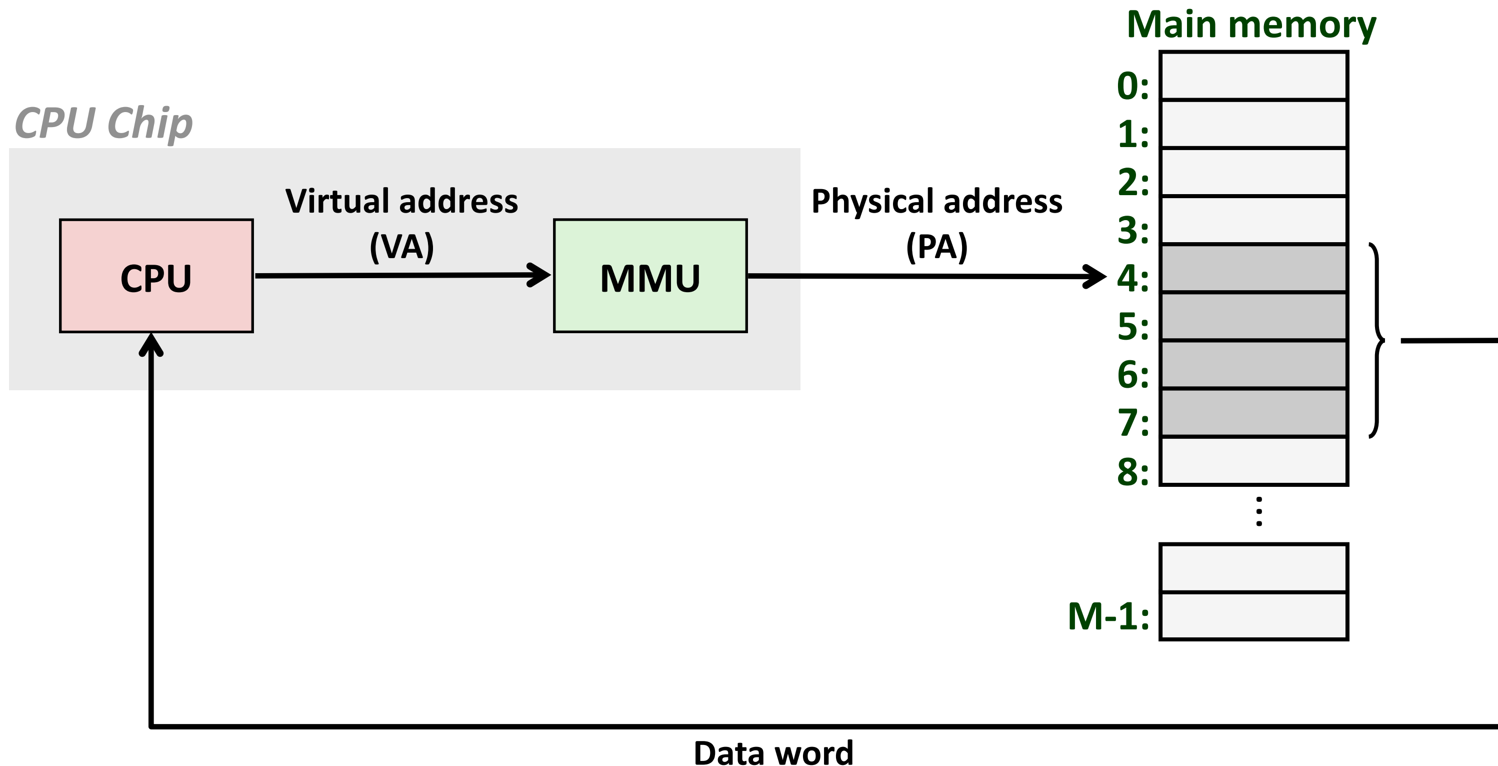
- Clean distinction between data (bytes) and their attributes (addresses)
- Each object can now have multiple addresses
- Every byte in main memory:
  one physical address, one (or more) virtual addresses

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 desktops, laptops, workstations**
- **One of the great ideas in computer science**
- ***MMU checks the cache***

Why Virtual Memory (VM)?

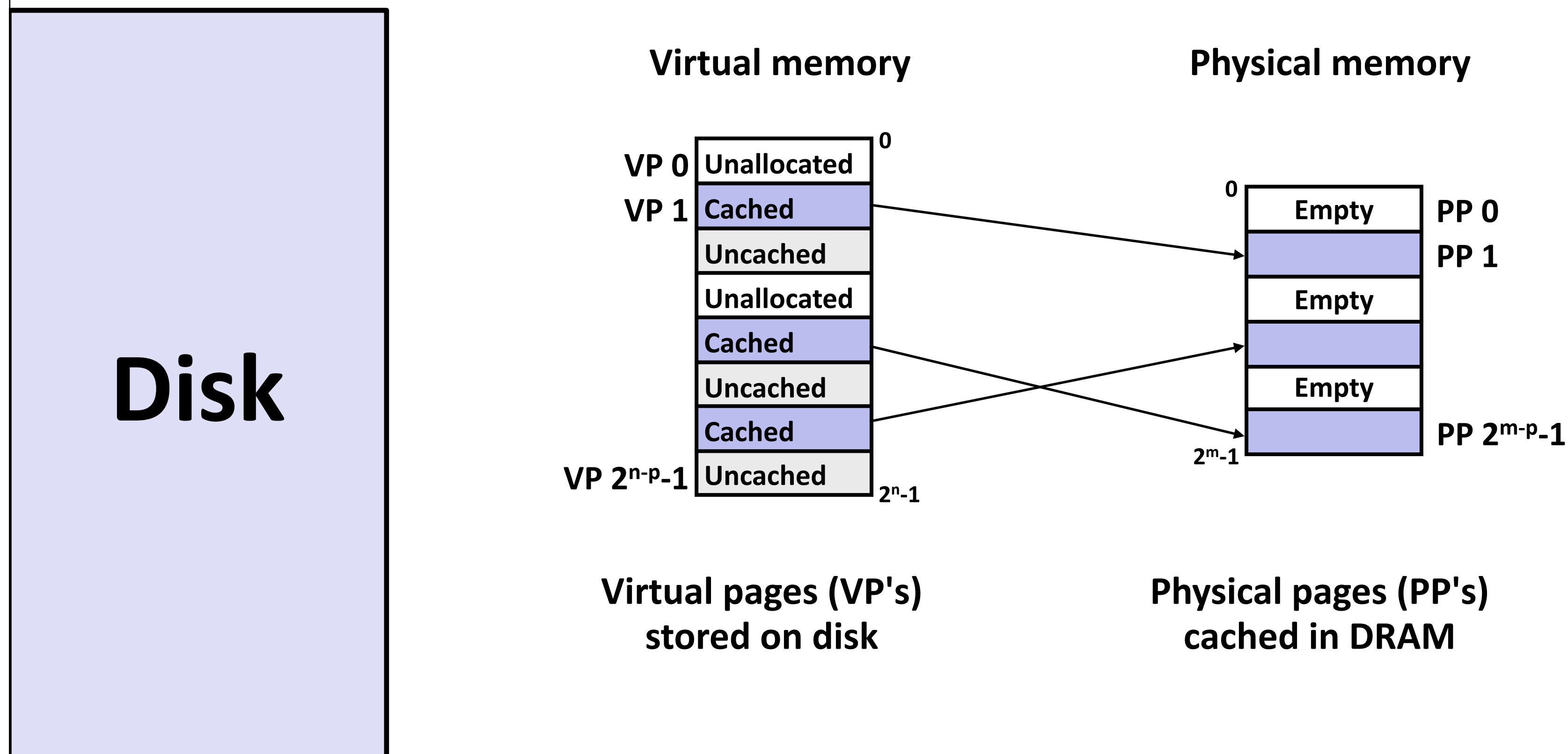
- **Efficient use of limited main memory (RAM)**
  - Use RAM as a cache for the parts of a virtual address space
    - some non-cached parts stored on disk
    - some (unallocated) non-cached parts stored nowhere
  - Keep only active areas of virtual address space in memory
    - transfer data back and forth as needed
- **Simplifies memory management for programmers**
  - Each process gets the same full, private linear address space
- **Isolates address spaces**
  - One process can't interfere with another's memory
    - because they operate in different address spaces
  - User process cannot access privileged information
    - different sections of address spaces have different permissions

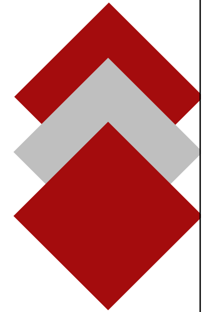
Today

- Virtual Memory (VM) overview and motivation
- **VM as tool for caching**
- VM as tool for memory management
- VM as tool for memory protection
- Address translation

VM as a Tool for Caching

- **Virtual memory: array of N = 2^n contiguous bytes**
  - think of the array (allocated part) as being stored on disk
- **Physical memory (DRAM) = cache for allocated virtual memory**
- **Blocks are called pages; size = 2^p**





Memory Hierarchy: Core 2 Duo

L1/L2 cache: 64 B blocks

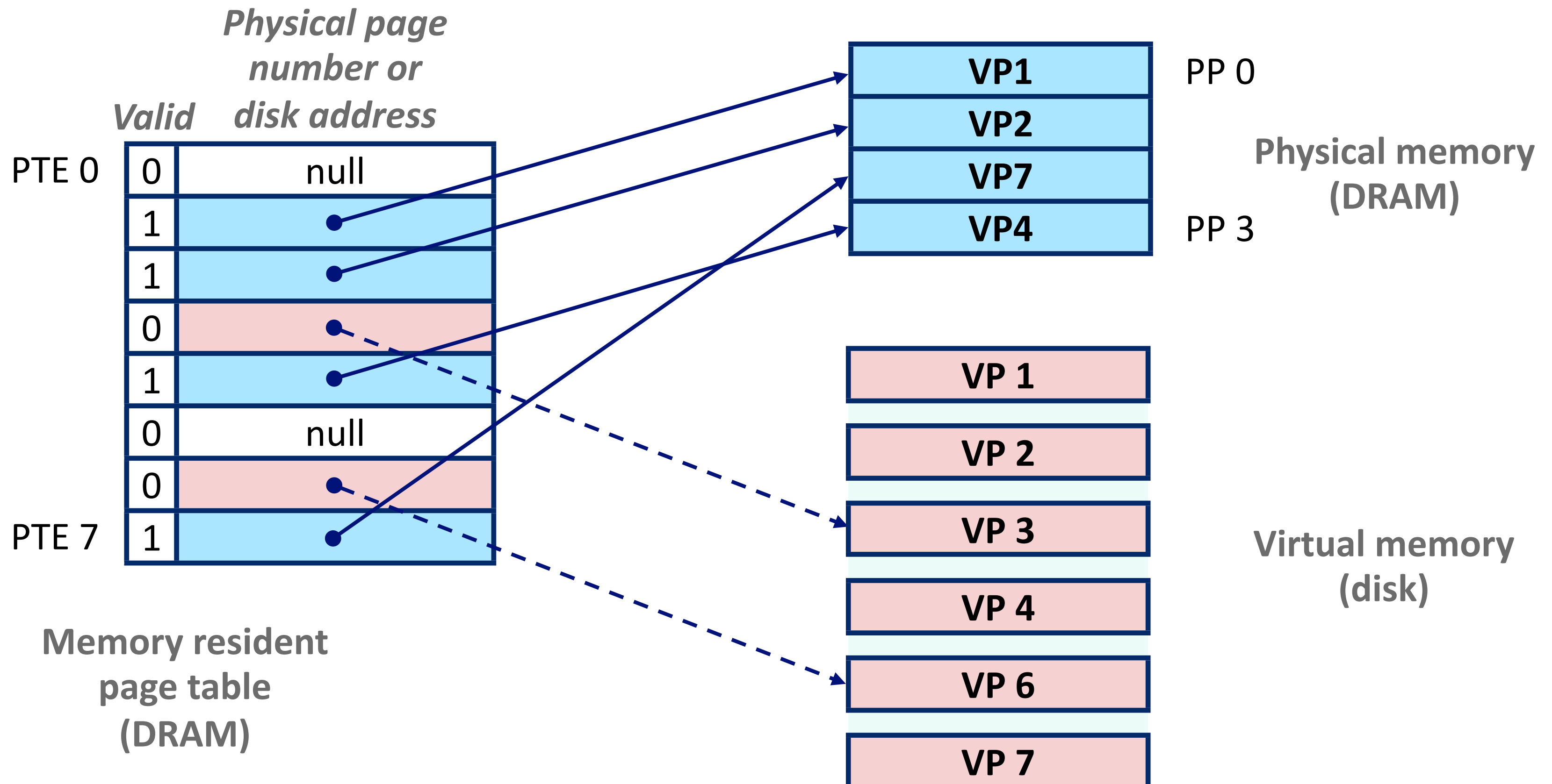
DRAM Cache Organization

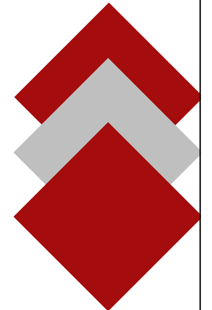
- **DRAM cache organization driven by the enormous miss penalty**
  - DRAM is about ***10x*** slower than SRAM
  - Disk is about ***10,000x*** slower than DRAM
    - For first byte, faster for next byte

- **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

Address Translation: Page Tables

- A ***page table*** is an array of page table entries (PTEs) that maps virtual pages to physical pages. Here: 8 VPs
  - Per-process kernel data structure in DRAM

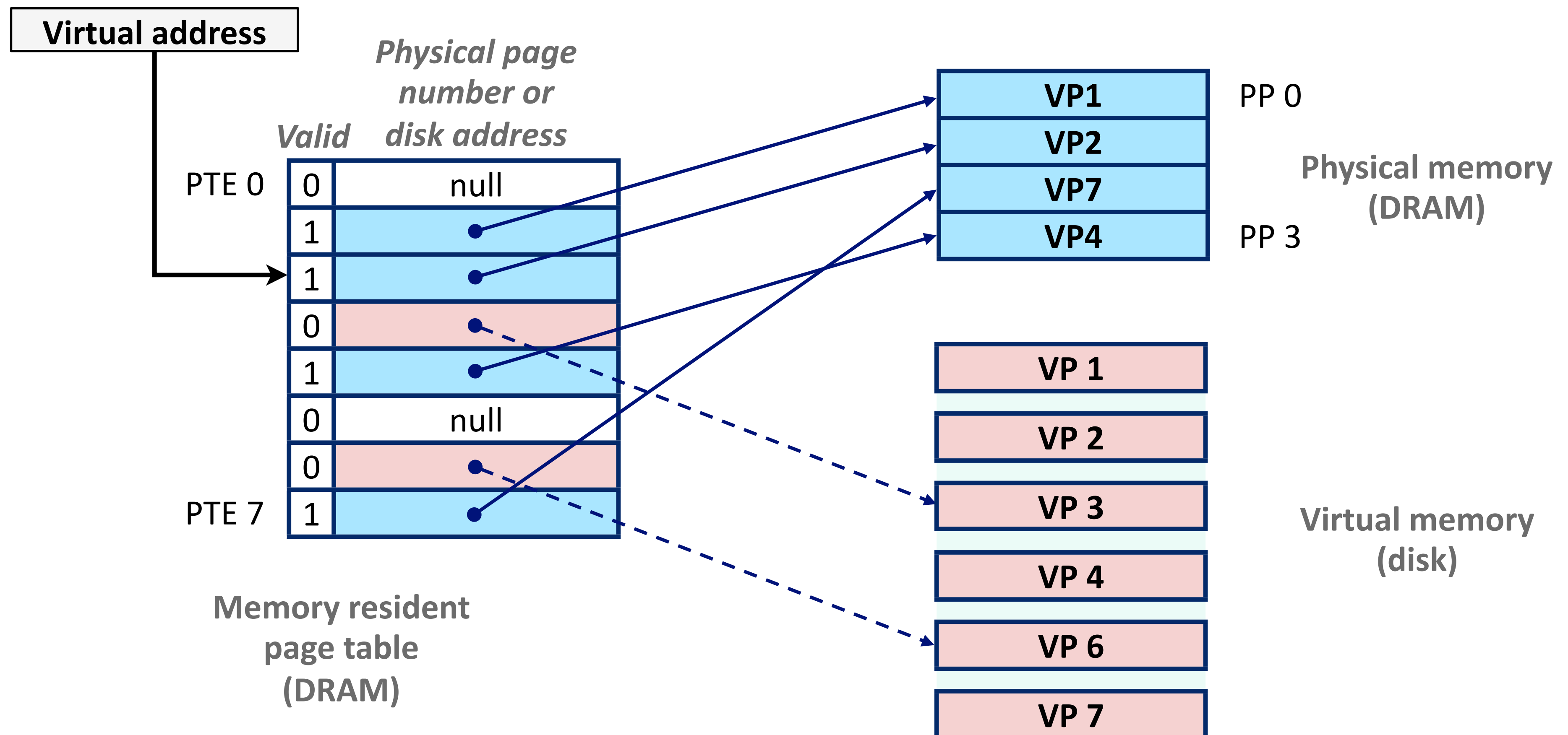




Address Translation With a Page Table

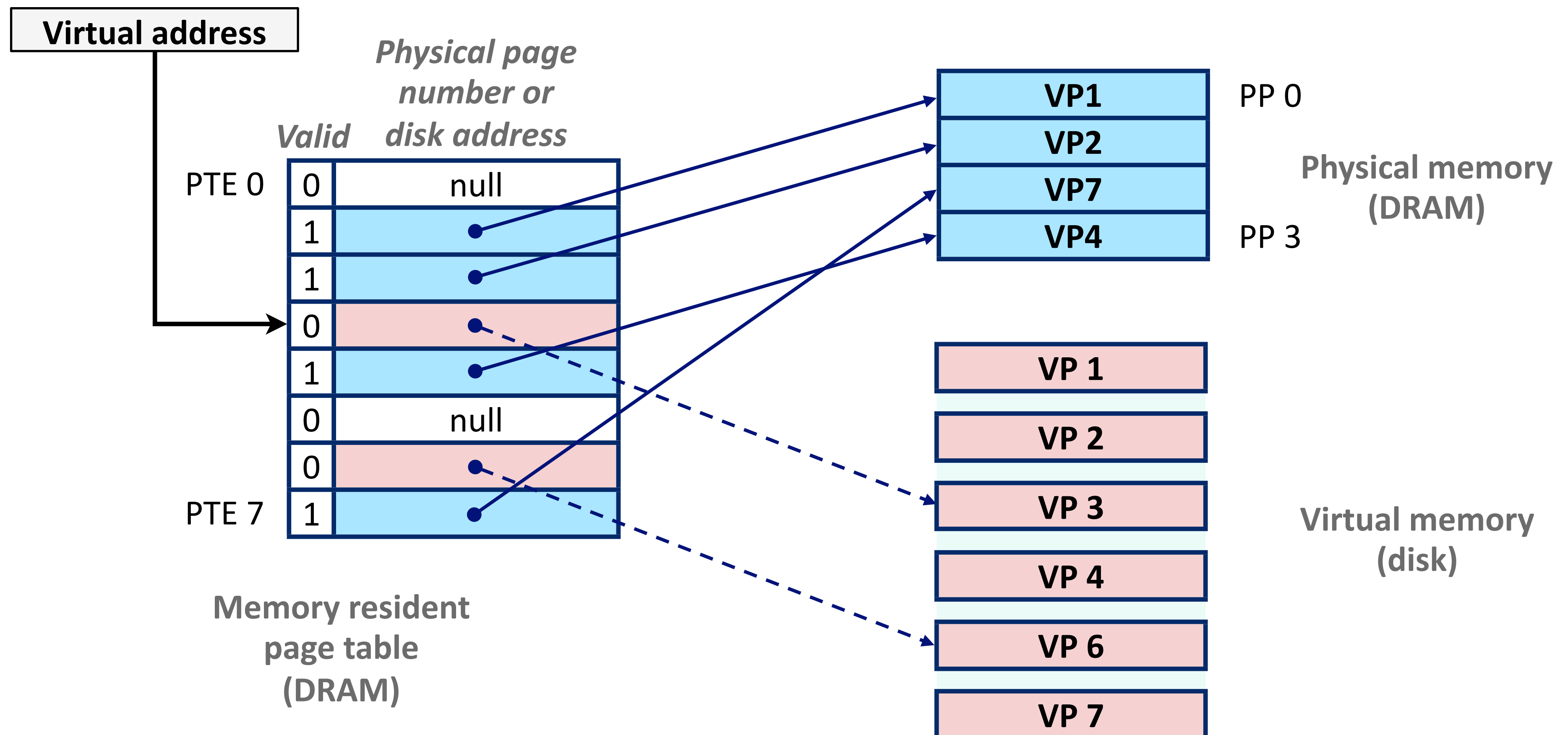
Page Hit

- ***Page hit:*** **reference to VM word that is in physical memory**



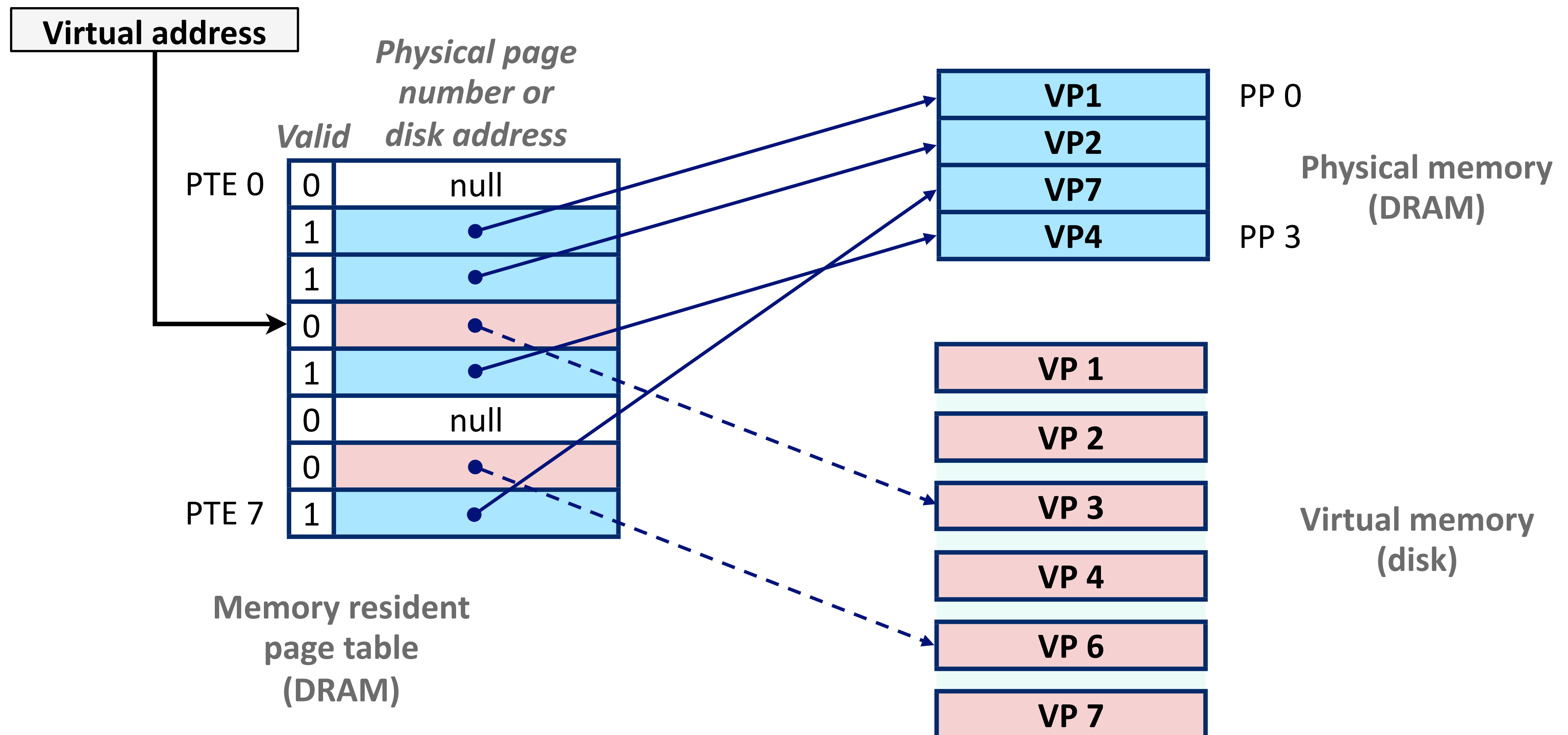
Page Miss

- *Page miss:* reference to VM word that is not in physical memory



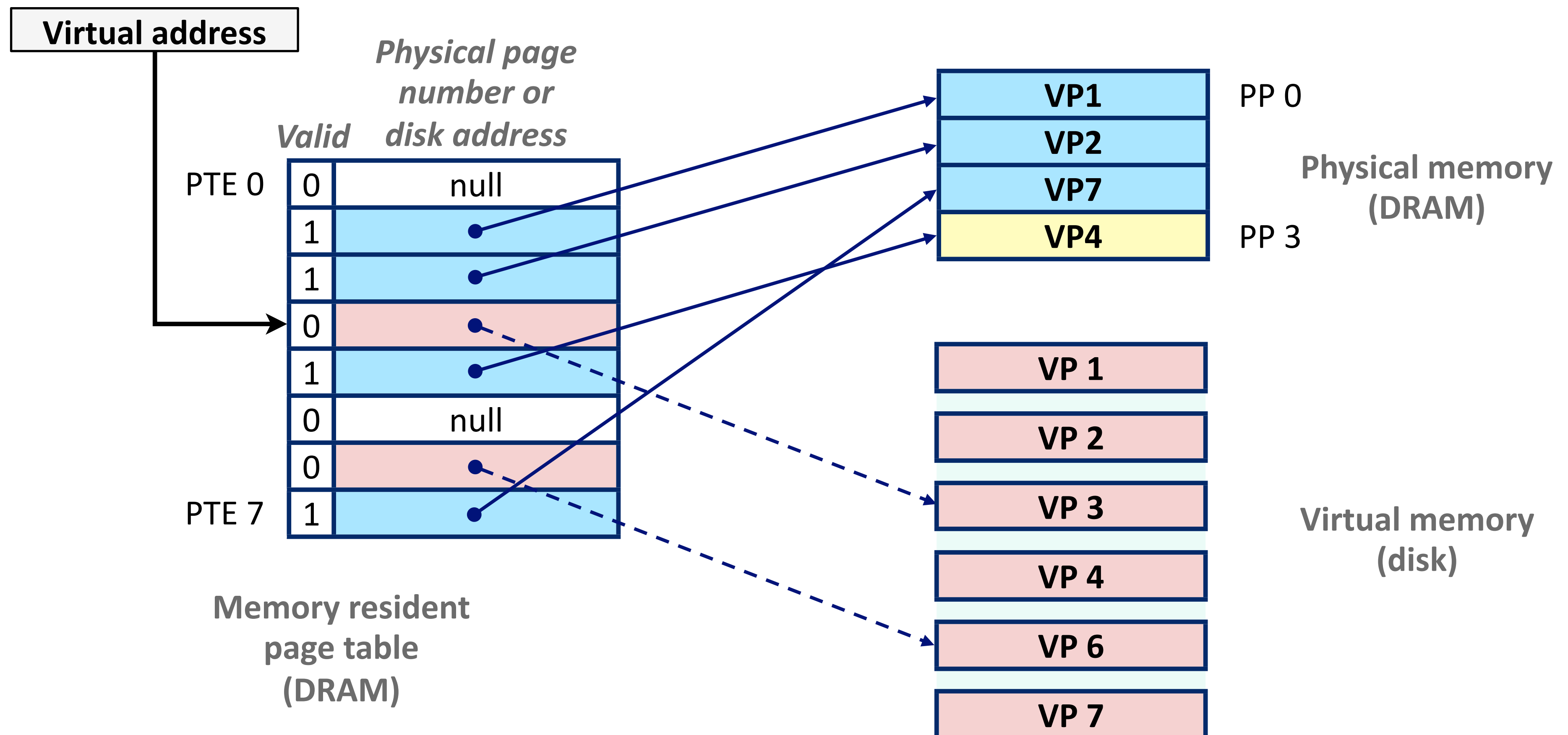
Handling Page Fault

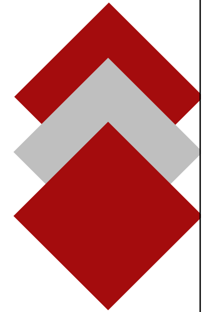
- **Page miss causes page fault (an exception)**



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)

Virtual address

Physical page number or disk address

Valid

	Valid	Physical page number or disk address
PTE 0	0	null
	1	•
	1	•
	0/1	•
	0/1	•
	0	null
	0	•
PTE 7	1	•

Memory resident page table (DRAM)

VP1 — PP 0
VP2
VP7
VP3 — PP 3

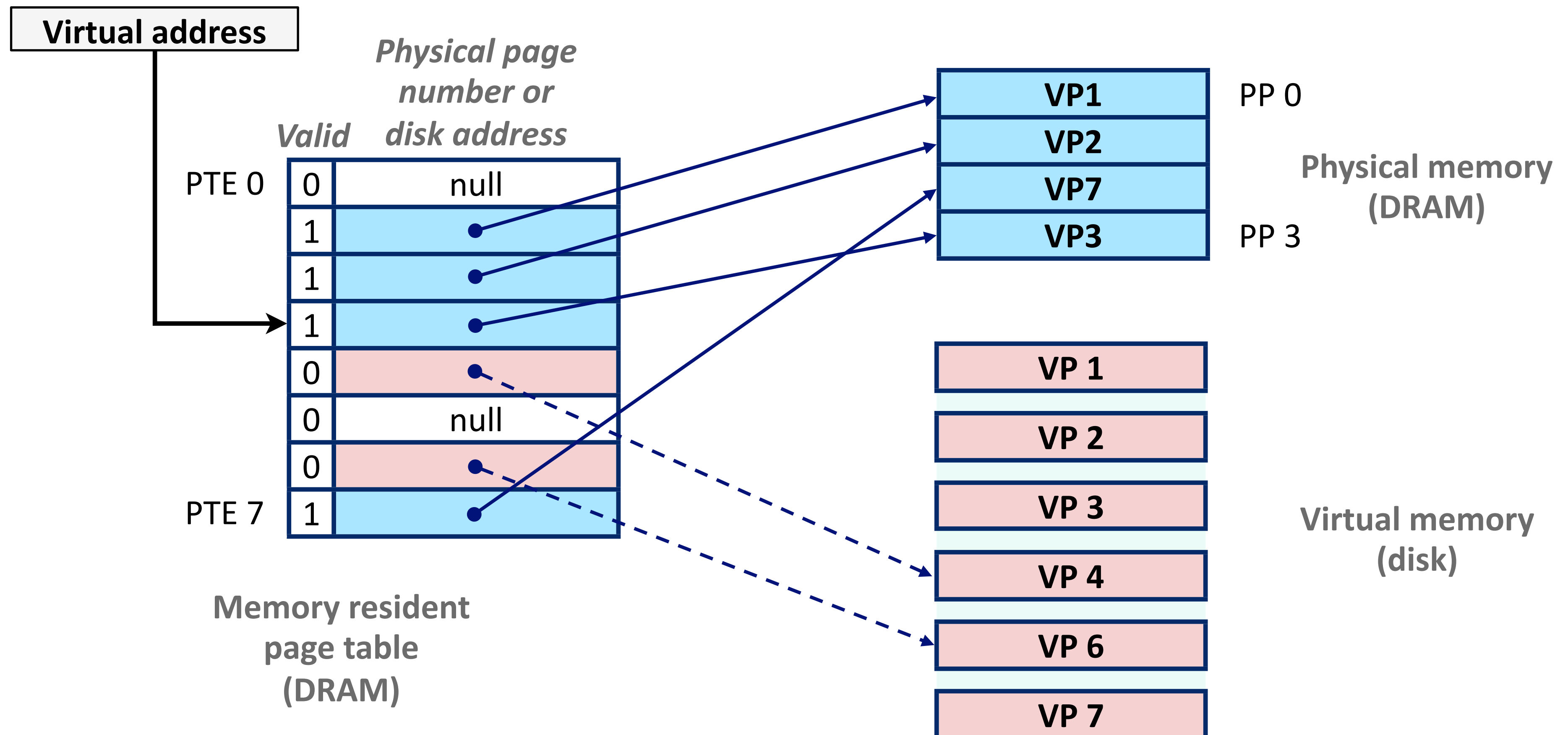
Physical memory (DRAM)

VP 1
VP 2
VP3
VP 4
VP 6
VP 7

Virtual memory (disk)

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!



Why does it work? Locality

- **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

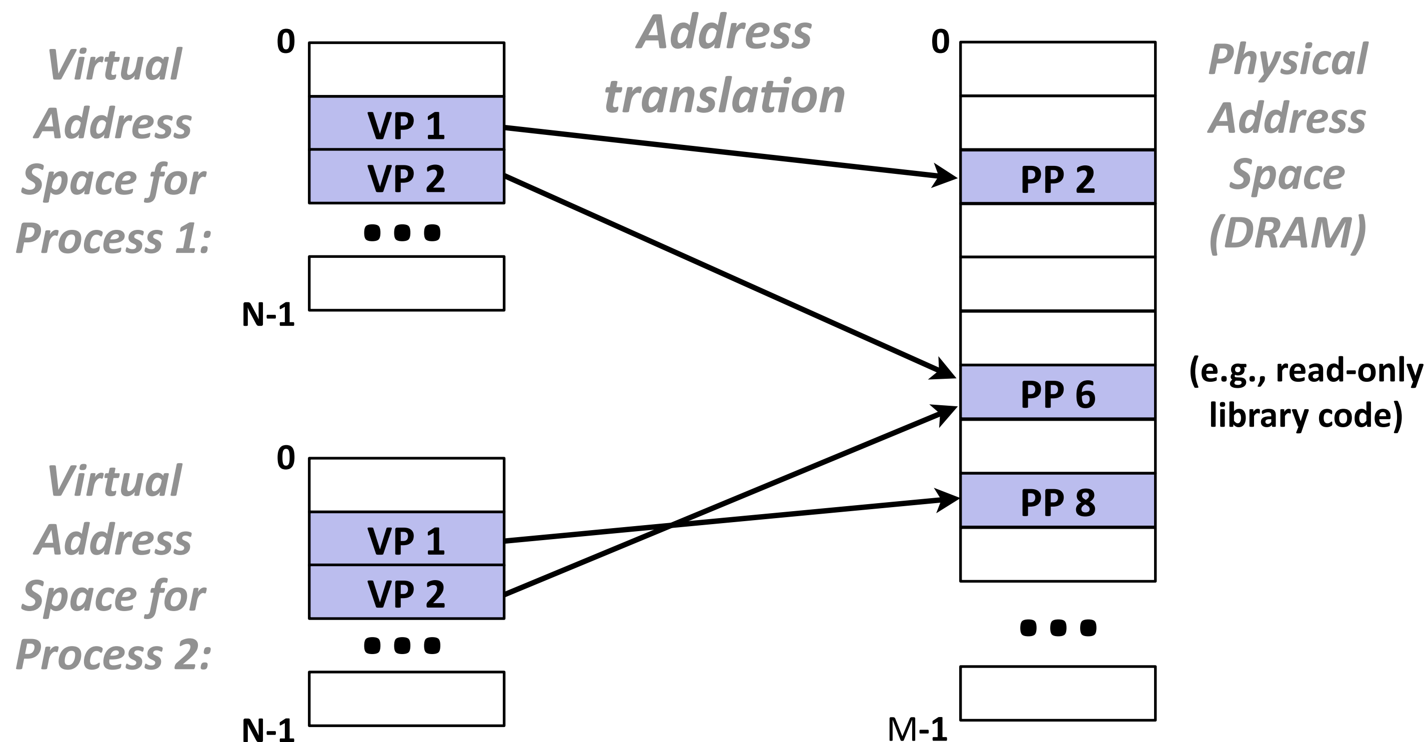
- **If (SUM(working set sizes) > main memory size)**
  - ***Thrashing:*** Performance meltdown where pages are swapped (copied) in and out continuously

Today

- Virtual Memory (VM) overview and motivation
- VM as tool for caching
- **VM as tool for memory management**
- VM as tool for memory protection
- Address translation

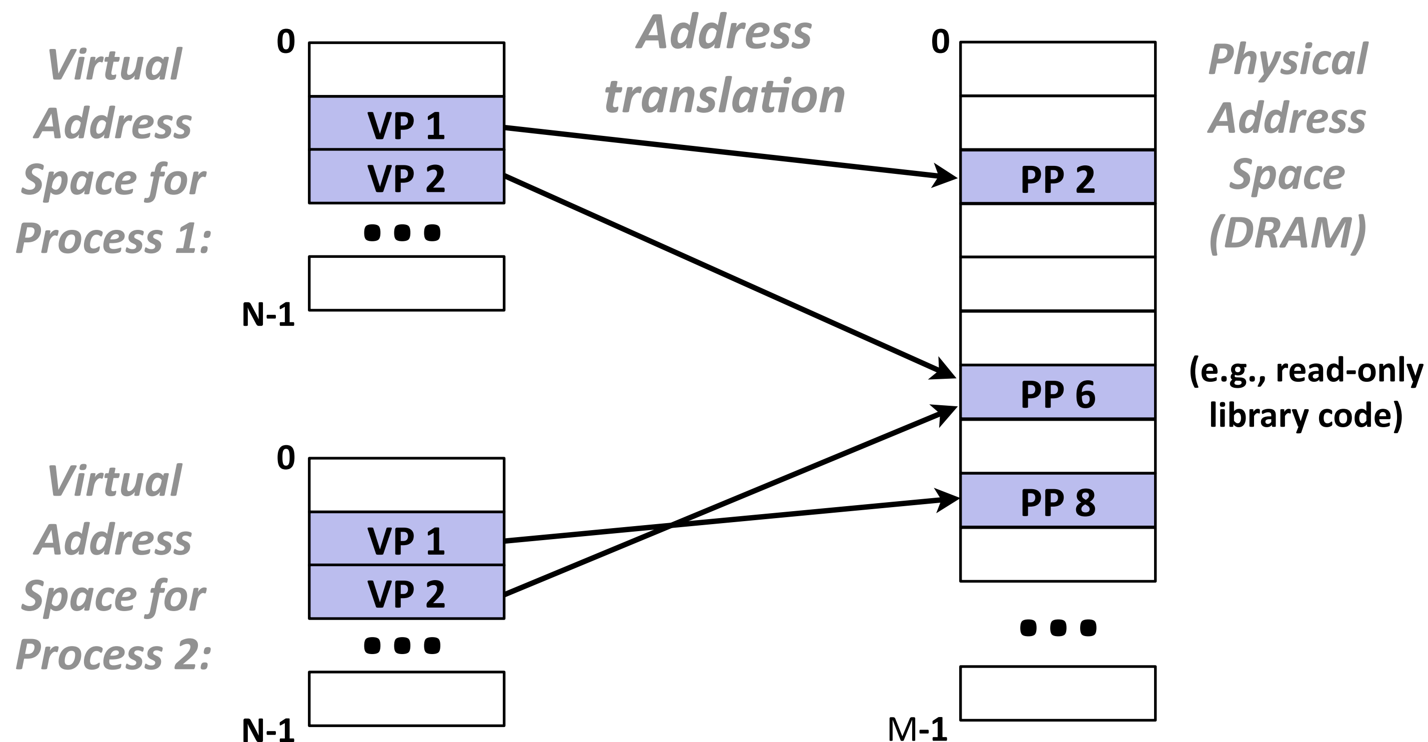
VM as a Tool for Memory Management

- **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



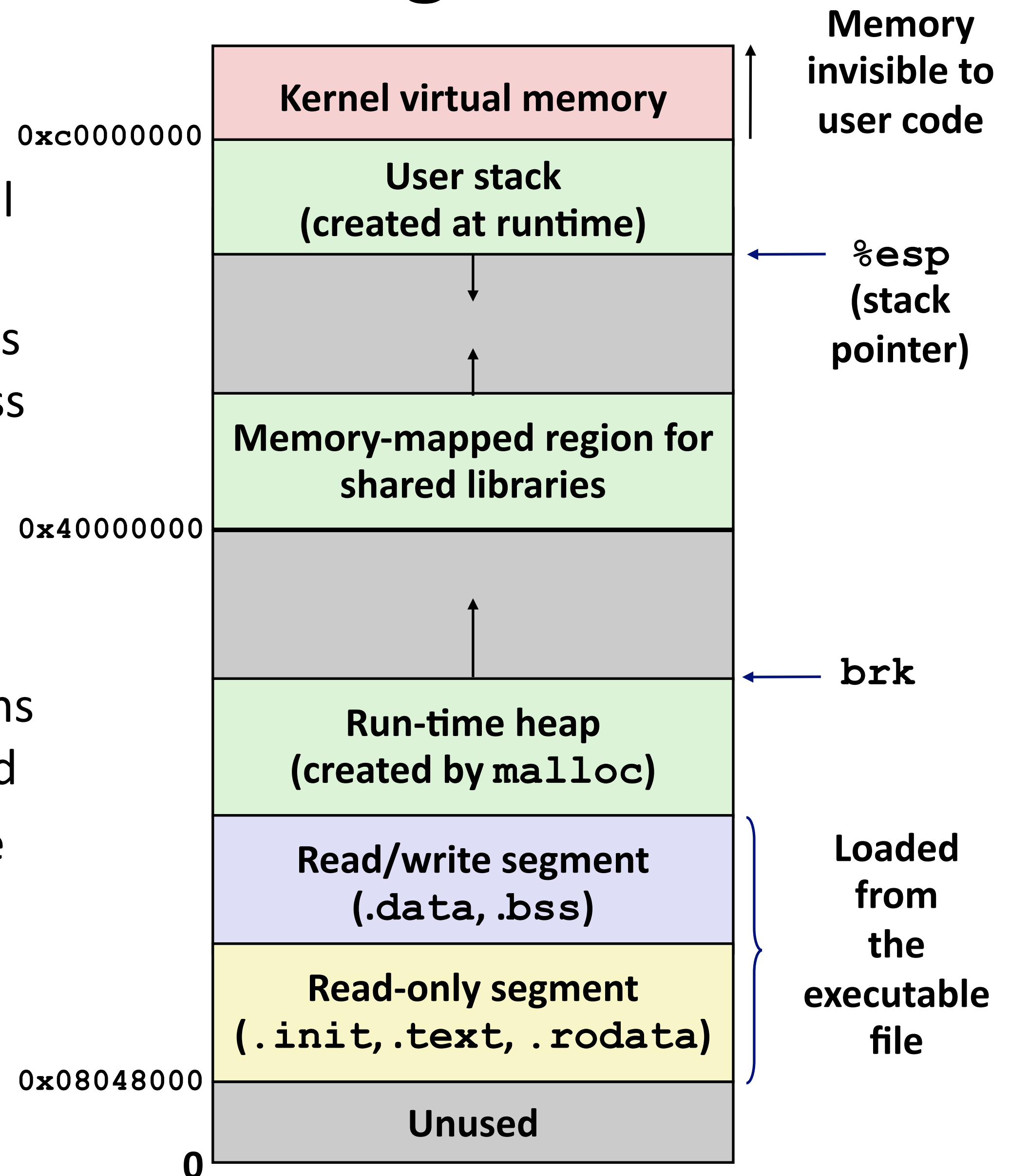
VM as a Tool for Memory Management

- **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

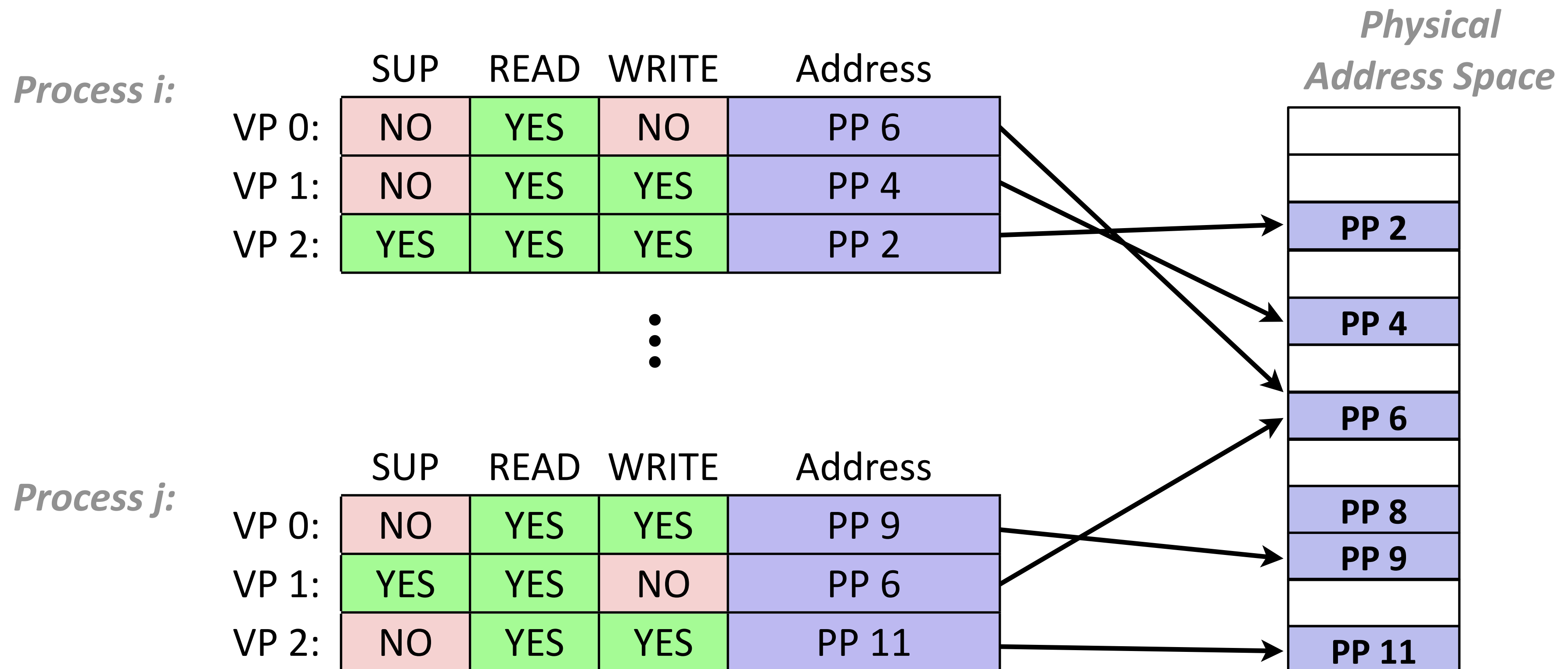


Today

- Virtual Memory (VM) overview and motivation
- VM as tool for caching
- VM as tool for memory management
- **VM as tool for memory protection**
- Address translation

VM as a Tool for Memory Protection

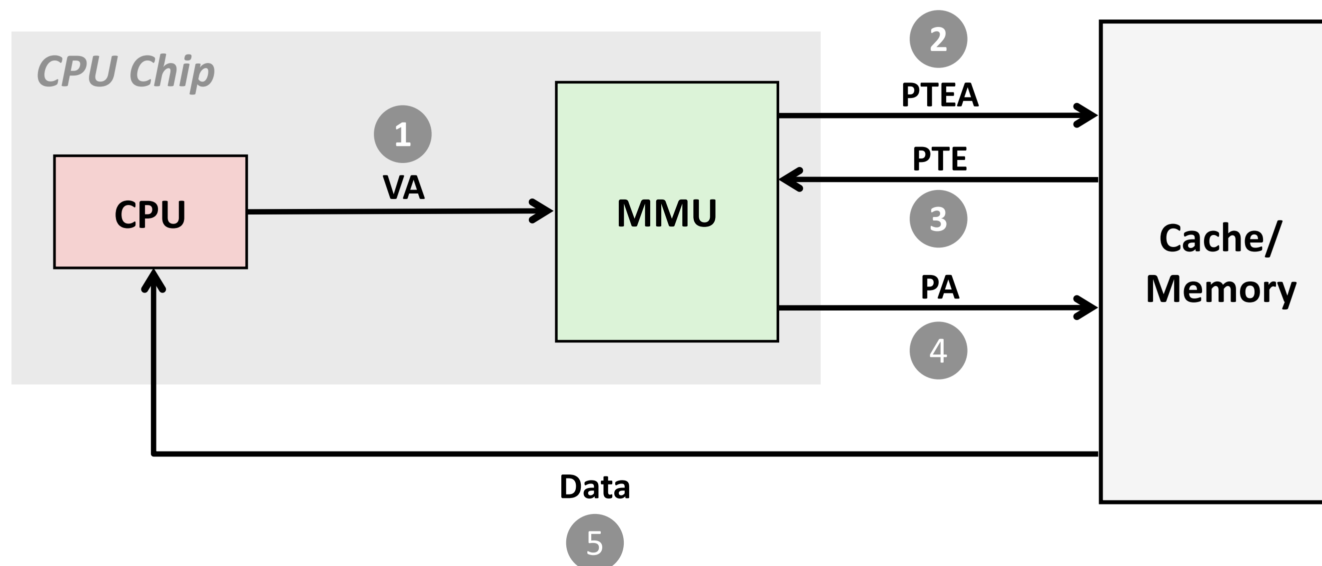
- **Extend PTEs with permission bits**
- **Page fault handler checks these before remapping**
  - If violated, send process SIGSEGV (segmentation fault)



Today

- Virtual Memory (VM) overview and motivation
- VM as tool for caching
- VM as tool for memory management
- VM as tool for memory protection
- **Address translation**

Address Translation: Page Hit



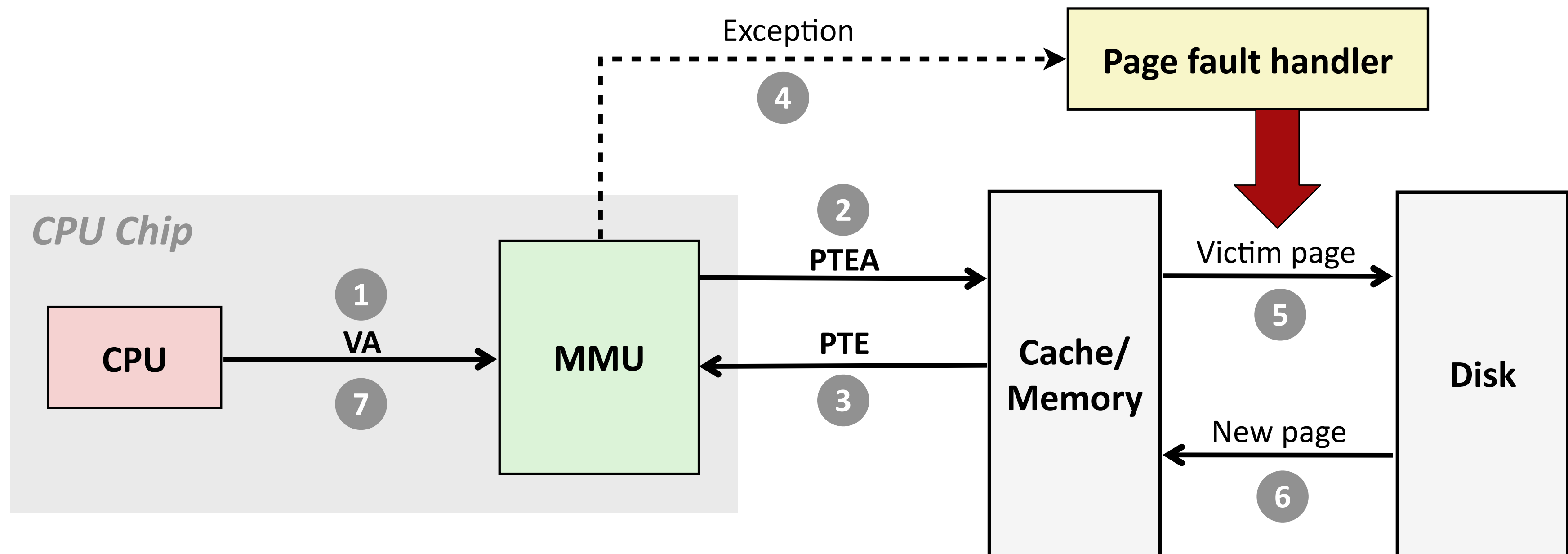
1) Processor sends virtual address to MMU

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

Summary

- **Virtual / Physical Addresses**
- **Memory Pages**
- **Page Tables**
- **Address Translation**

- **Next Time: Virtual Memory II**
  - Using TLB to speed address translation
  - Linux memory management