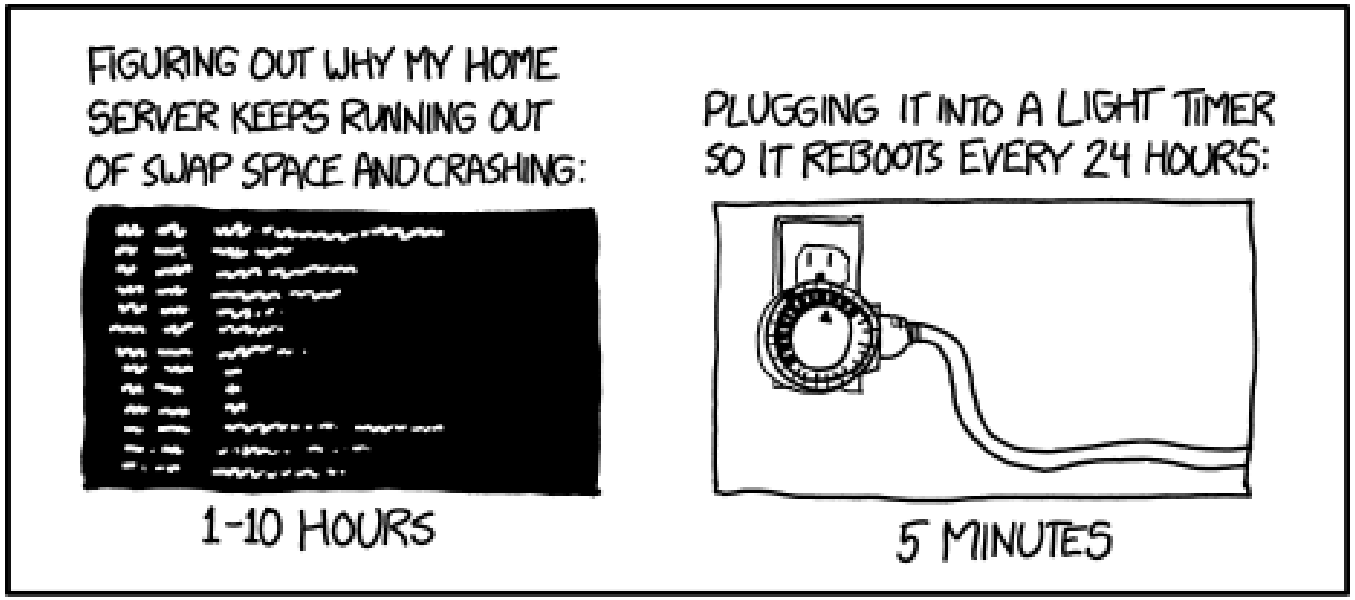
The Hardware/Software Interface

Virtual Memory II

Instructors:
Justin Hsia, Amber Hu

Teaching Assistants:

Anthony Mangus	Divya Ramu
Grace Zhou	Jessie Sun
Jiuyang Lyu	Kanishka Singh
Kurt Gu	Liander Rainbolt
Mendel Carroll	Ming Yan
Naama Amiel	Pollux Chen
Rose Maresh	Soham Bhosale
Violet Monserate	



https://xkcd.com/1495/

Relevant Course Information

- HW23 due tonight, HW24 due Wednesday, HW25 due *Monday* (12/1)
- No readings in Week 11 – "normal" lectures
- Today is the last day to submit Lab 4
- Lab 5 due next Thursday (12/4)
  - The most significant amount of C programming you will do in this class – combines lots of topics from this class: pointers, bit manipulation, structs, examining memory
  - Understanding the concepts *first* and efficient *debugging* will save you lots of time
  - Light style grading
- Final exam: Wednesday, 12/10 @ 12:30 pm
  - Final review section on 12/4, final review session on 12/5

Lecture Outline (1/3)

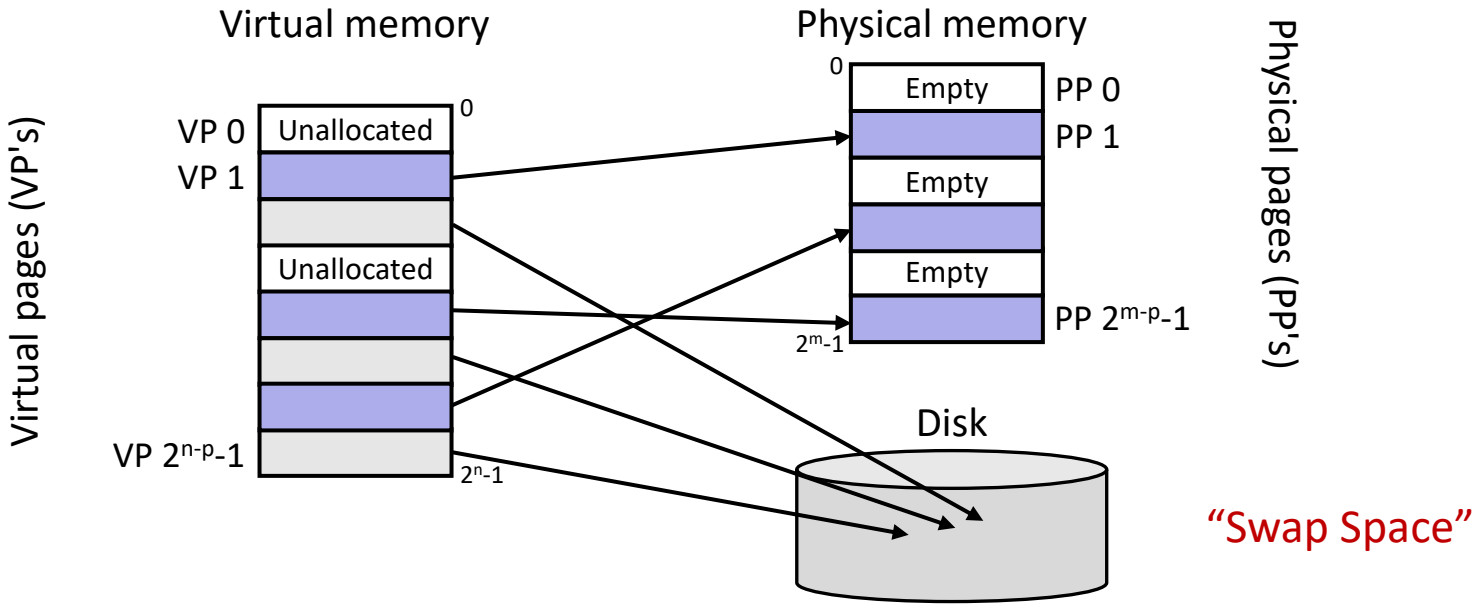
- **Paging, VM, and the Memory Hierarchy**
- Address Translation
- Memory Management

Why Virtual Memory (VM)?

- Simplifies memory management for programmers
  - Each process "gets" the same full, private linear address space
- 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
- Isolates address spaces (*i.e.*, provides protection)
  - A process can't interfere with another's memory – *different address spaces*
  - User process cannot access privileged information – implements memory permissions (*i.e.*, different memory layout segments)

VM in the Memory Hierarchy (Review)

- Memory (virtual or physical) as an array of bytes, now split into *pages*
  - Pages are another unit of aligned memory (size is $\mathrm{P} = 2^p$ bytes)
  - Each virtual page can be stored in *any* physical page (no fragmentation!)
- Pages of virtual memory usually stored in physical memory, but sometimes spill to disk ("*page out* to disk" and "*page in* to memory")



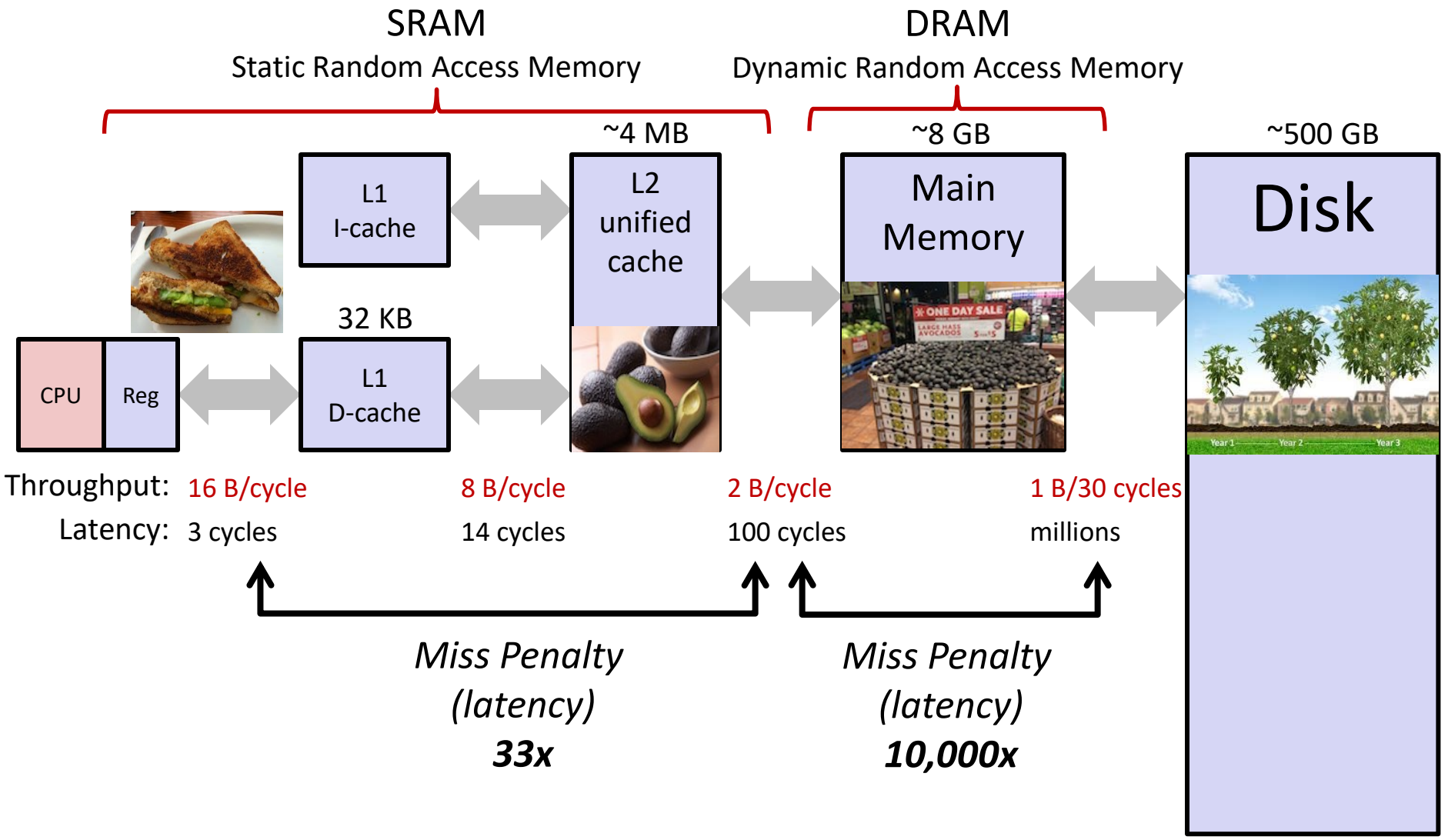
Small Paging Example

- Suppose we have a computer with the following parameters:
  - 1 KiB pages
  - 14-bit virtual addresses
  - 4 KiB of physical memory

- Let's visualize:
  - How big is the virtual address space?
  - How many virtual pages are there?
  - How many physical pages are there?
  - How wide is a physical address?

Memory Hierarchy: Core 2 Duo

Not drawn to scale



Why does VM work on RAM/disk?

- Avoids disk accesses because of *locality*
  - Same reason that L1 / L2 / L3 caches work

- The set of virtual pages that a program is "actively" accessing at any point in time is called its *working set*
  - If (*working set of one process* **≤** *physical memory*):
    - Good performance for one process (after compulsory misses)
  - If (*working sets of all processes* **>** *physical memory*):
    - *Thrashing*: Performance meltdown where pages are swapped between memory and disk continuously (CPU always waiting or paging)
    - This is why your computer can feel faster when you add RAM

Virtual Memory Design Consequences

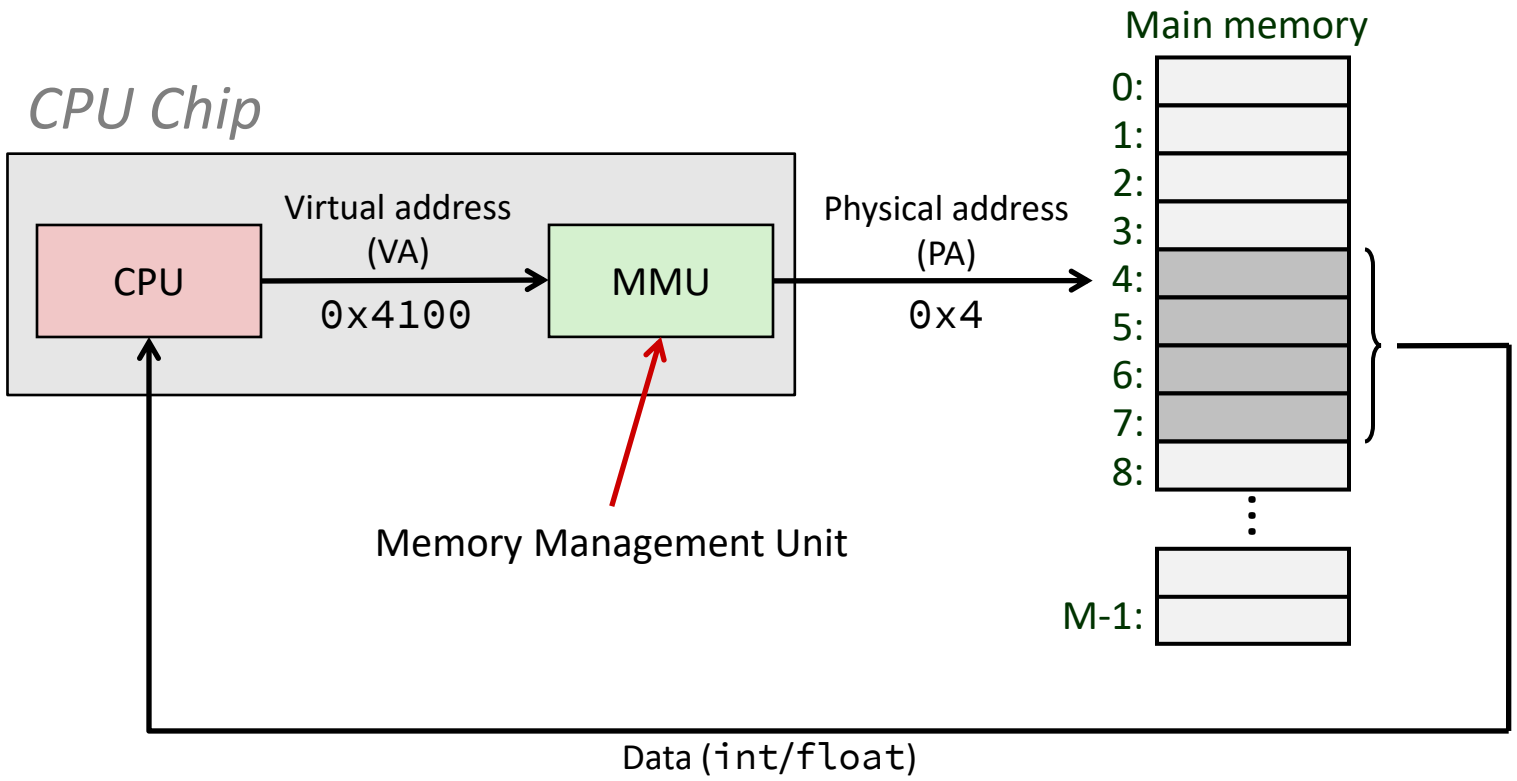
- Large page size – typically 4-8 KiB or 2-4 MiB (*can* be up to 1 GiB)
  - Compared with 64-byte cache blocks
- Fully associative
  - Any virtual page can be placed in any physical page
  - Requires a "large" mapping function – different from CPU caches
- Highly sophisticated, expensive replacement algorithms in OS
  - Too complicated and open-ended to be implemented in hardware
- *Write-back* rather than *write-through*
  - *Really* don't want to write to disk every time we modify memory
  - Some things may never end up on disk (*e.g.*, stack for short-lived process)

Lecture Outline (2/3)

- Paging, VM, and the Memory Hierarchy
- **Address Translation**
- Memory Management

Address Translation

- How do we perform the virtual → physical address translation?



Address Translation: Page Tables

- CPU-generated address can be split into:

  - Request is Virtual Address (VA), want Physical Address (PA)
  - Note that Physical Offset = Virtual Offset (page-aligned)

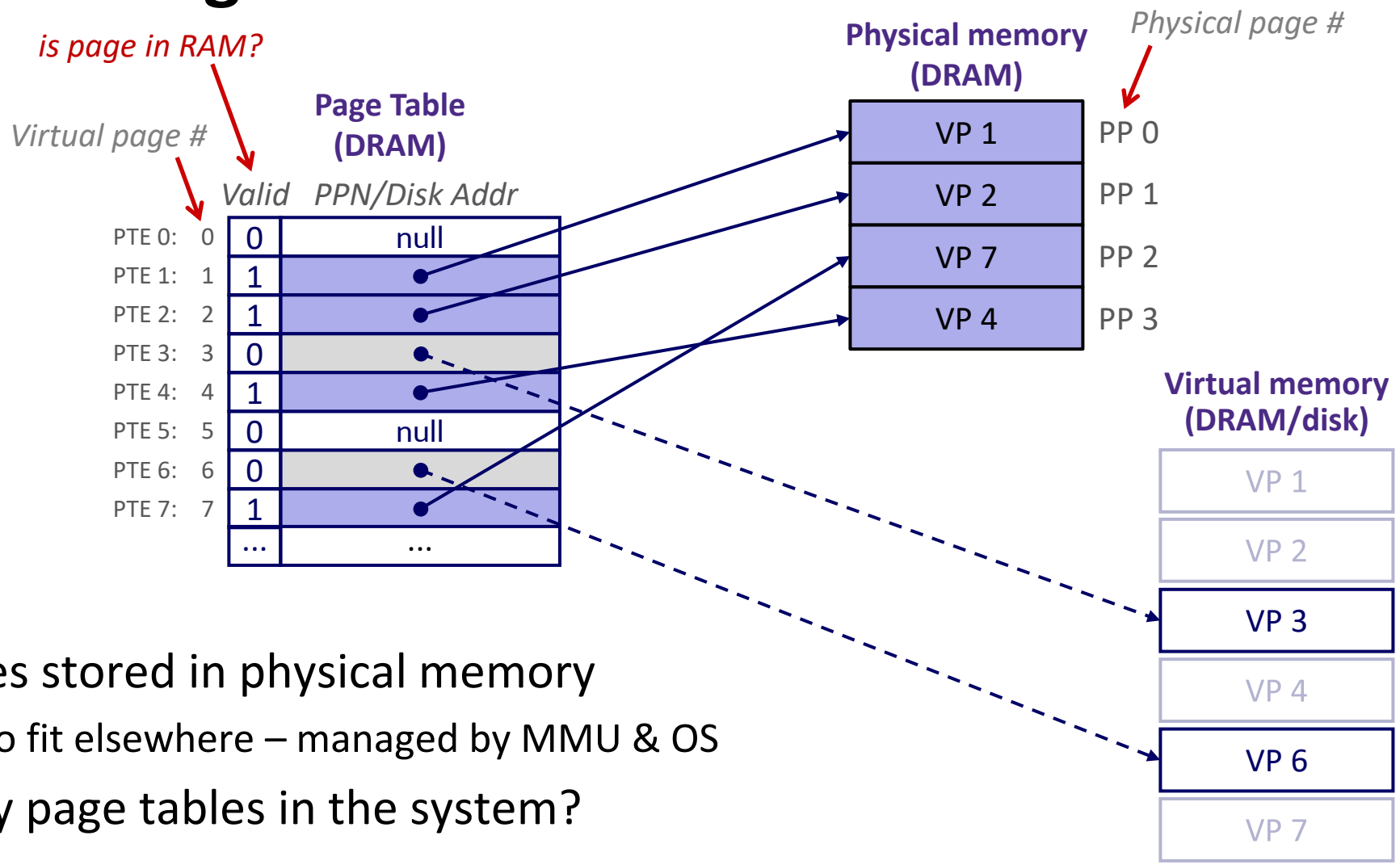
- Use lookup table that we call the *page table* (PT)
  - Replace Virtual Page Number (VPN) for Physical Page Number (PPN) to generate Physical Address
  - Index into the page table using the VPN: page table entry (PTE) stores the PPN plus management bits (*e.g.*, Valid, Dirty, access rights)
  - Has an entry for *every* virtual page

Polling Questions (1/2)

- Returning to the small paging example parameters:
  - 1 KiB pages
  - 14-bit virtual addresses
  - 4 KiB of physical memory

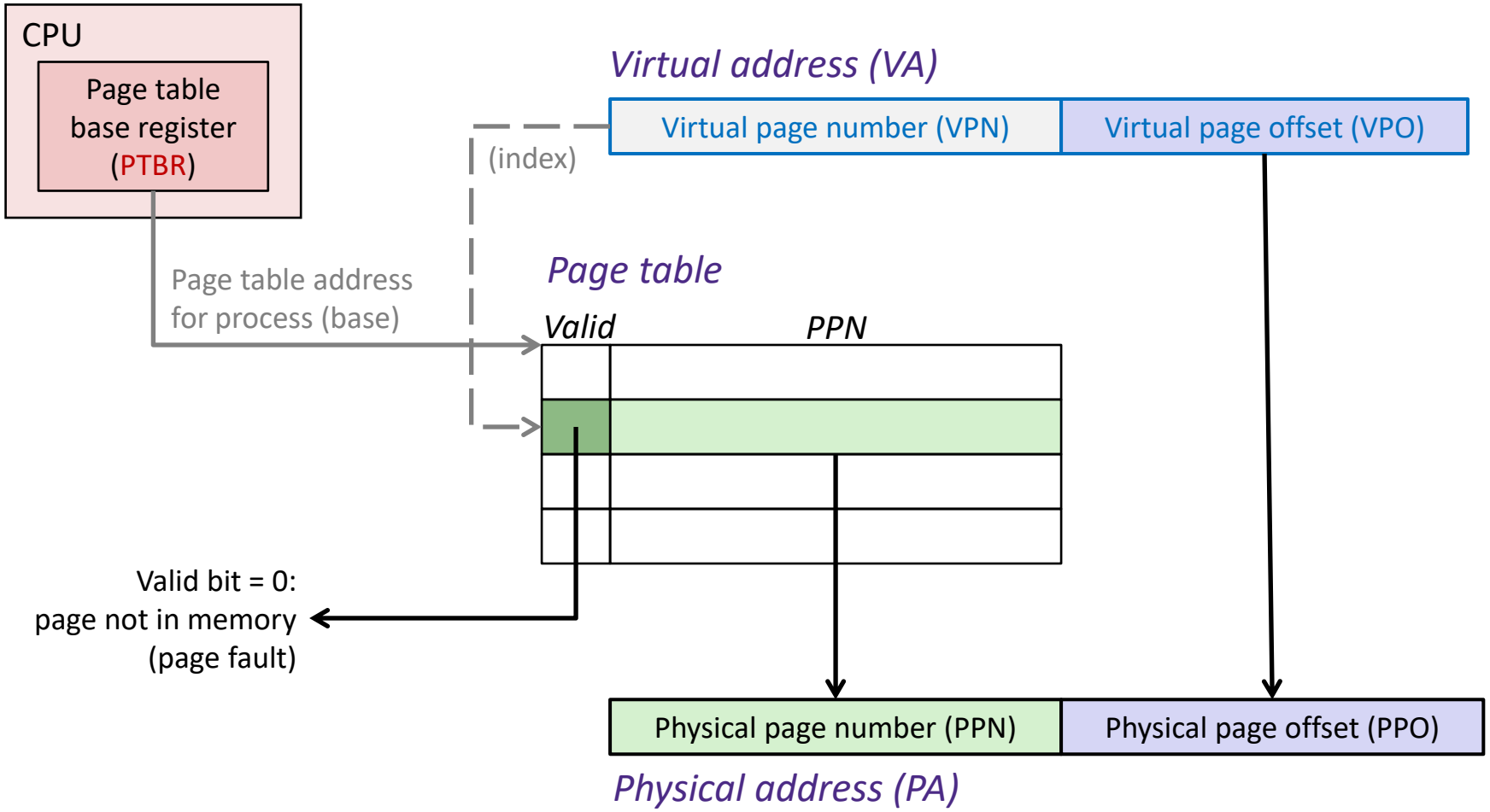
- How wide are the VPN and PPN fields?

Page Table Diagram



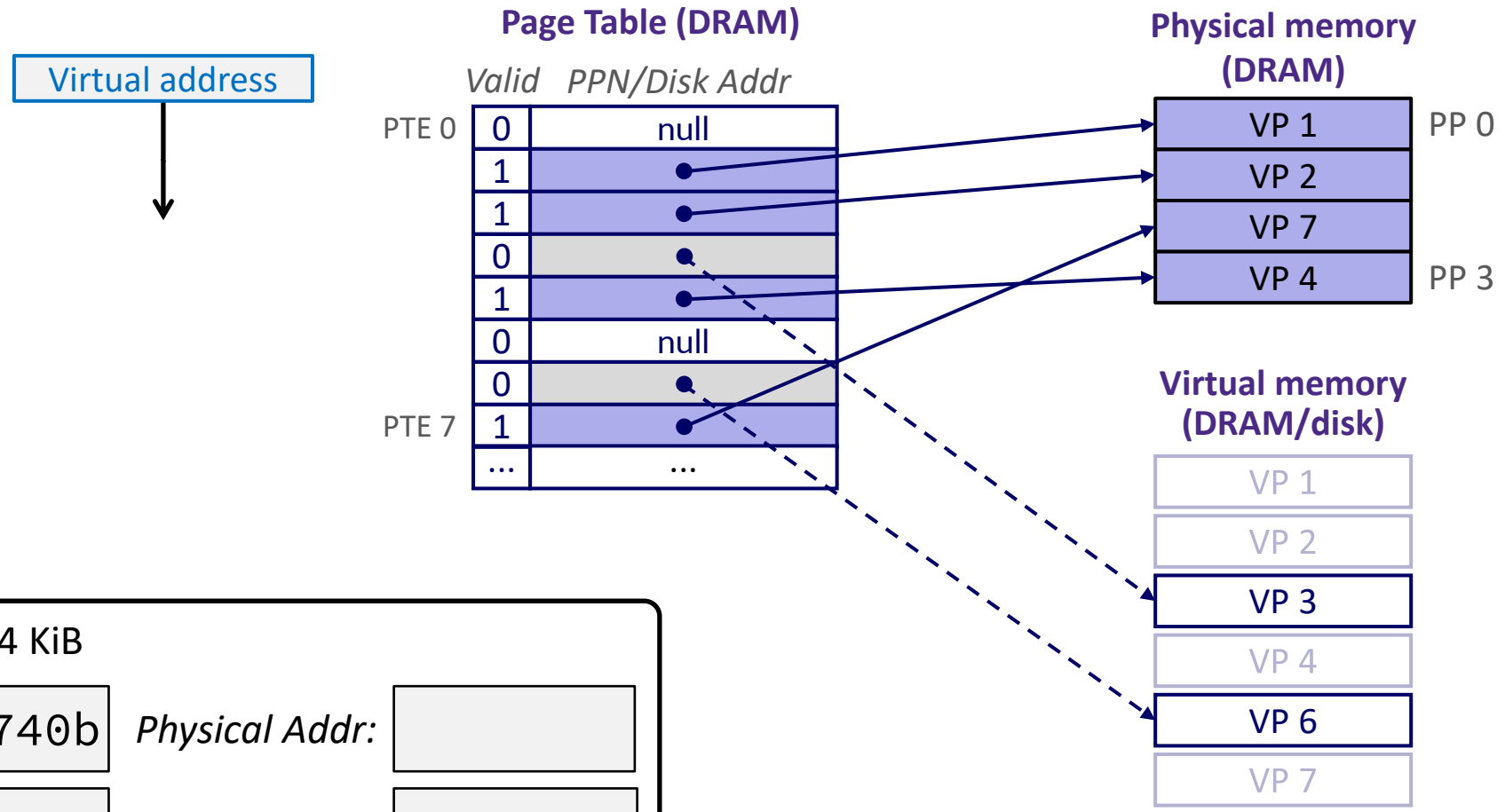
- Page tables stored in physical memory
  - Too big to fit elsewhere – managed by MMU & OS
- How many page tables in the system?
  - *One per process*

Page Table Address Translation



Page Table Outcome: Page Hit

- ***Page hit***: VM reference is in physical memory



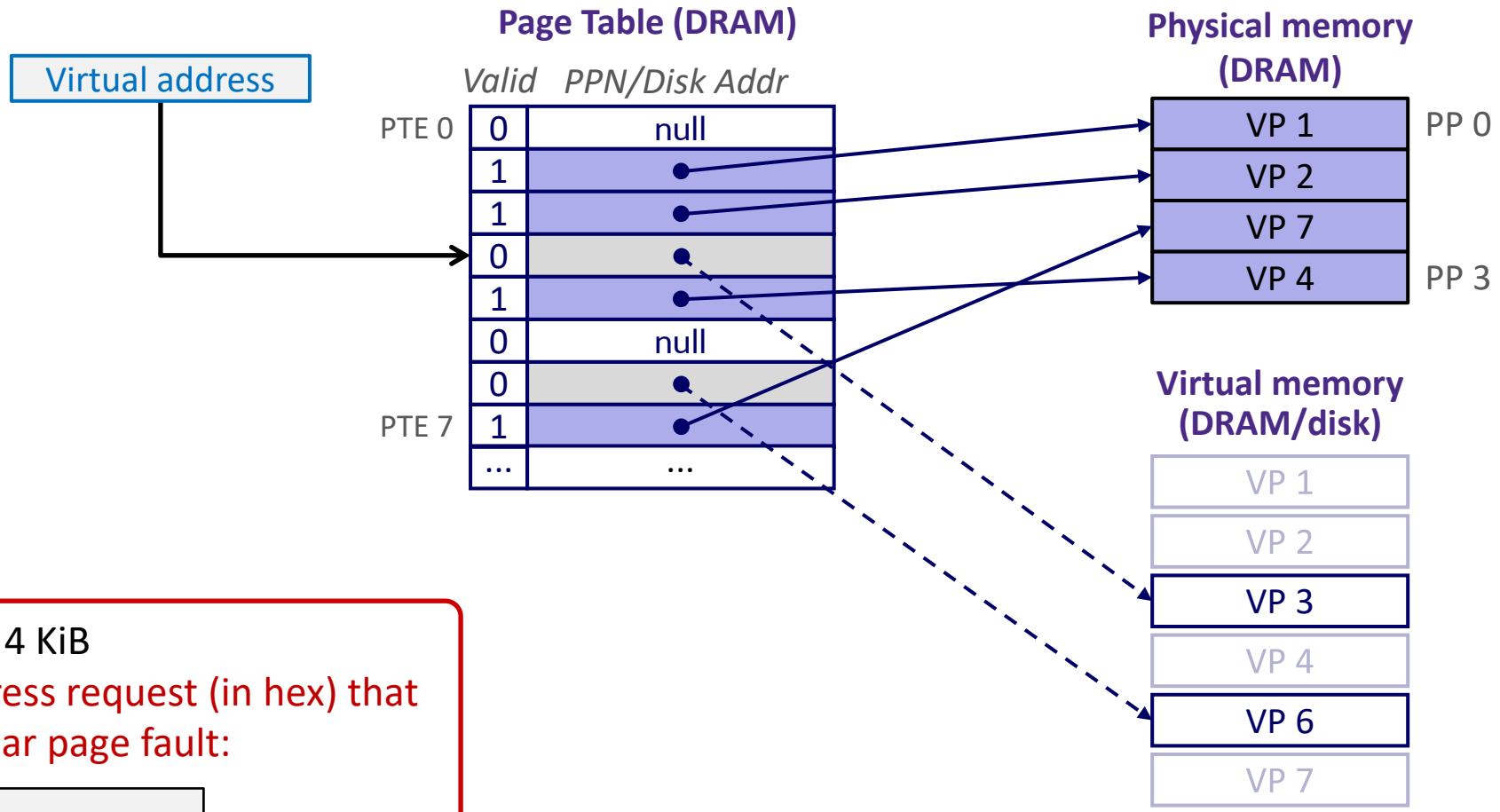
Example: Page size = 4 KiB

Virtual Addr: `0x00740b` *Physical Addr:*

VPN: *PPN:*

Page Table Outcome: Page Fault

- **Page fault**: VM reference is NOT in physical memory



Example: Page size = 4 KiB

Provide a virtual address request (in hex) that results in this particular page fault:

Virtual Addr:

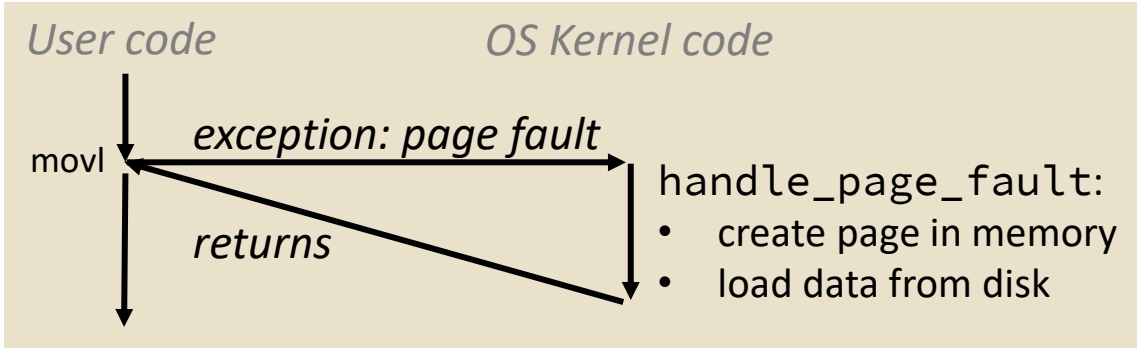
Reminder: Page Fault Exception

- User writes to memory location:

```
int a[1000];
int main () {
  a[500] = 13;
}
```

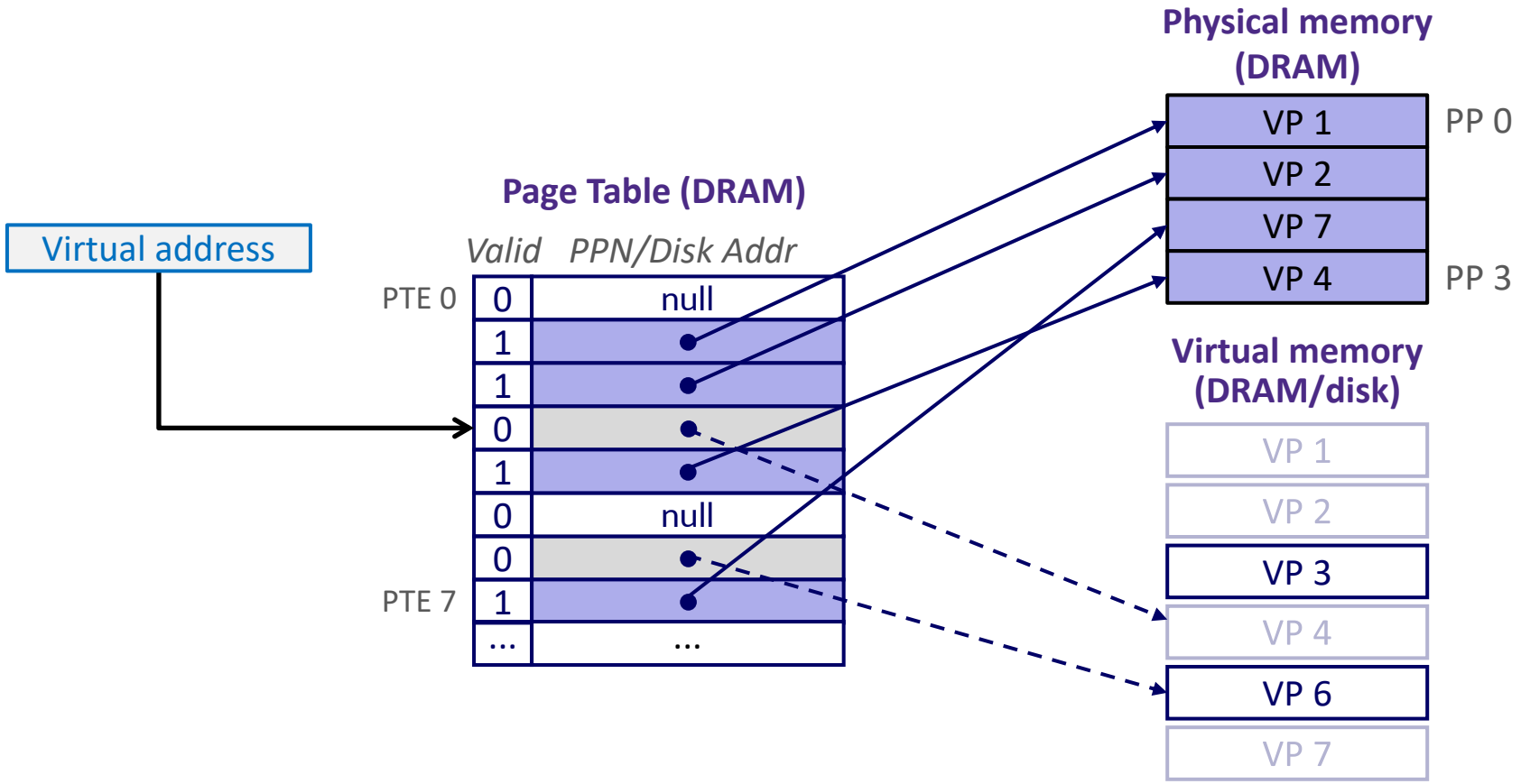
```
 80483b7:      c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```

  - That location happens to be currently on disk (not in memory)
    - Page fault handler must load page into physical memory
  - ***Execution returns to faulting instruction***: (`mov` is executed again)
    - Successful on second try



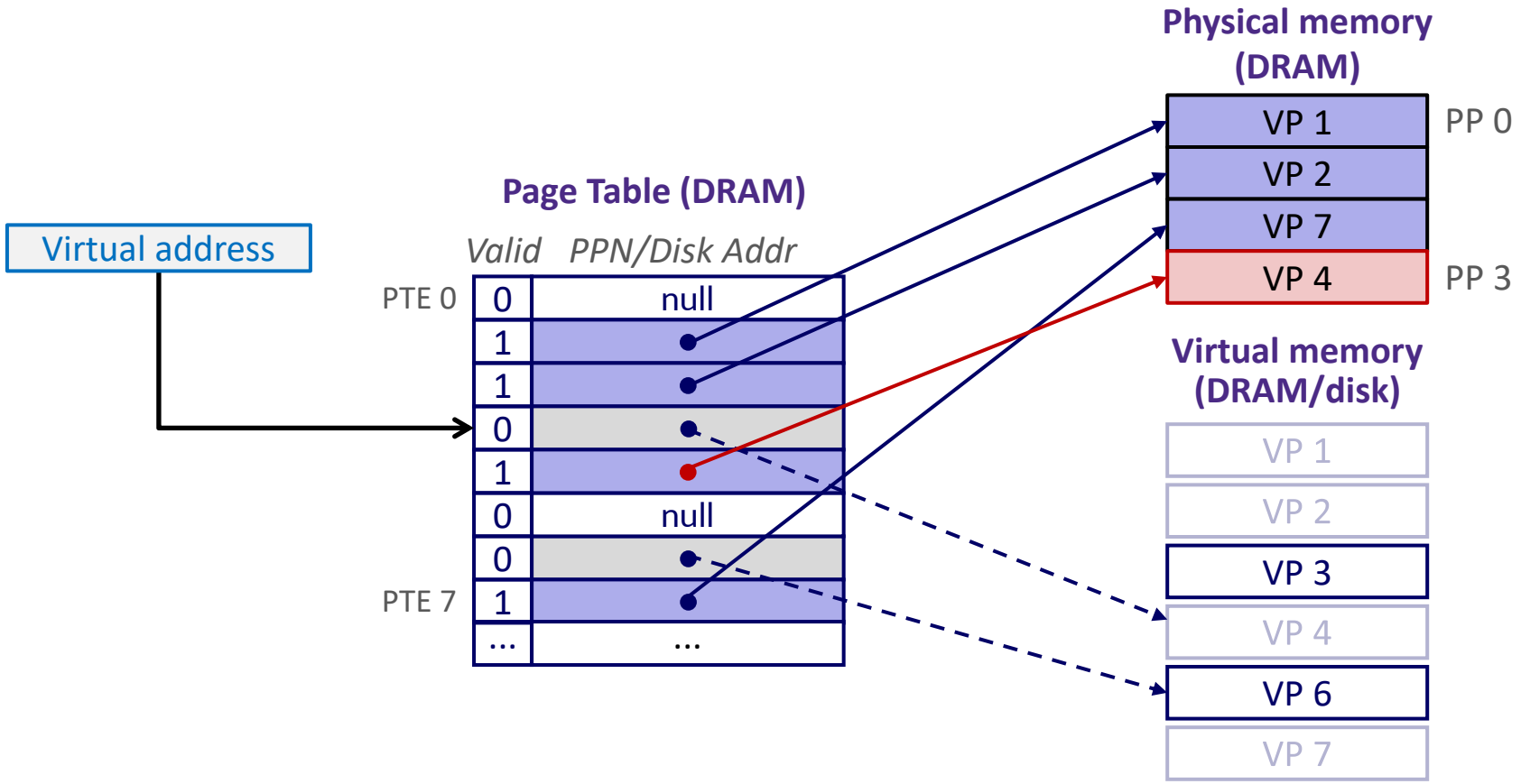
Handling a Page Fault (1/4)

1) Page miss causes page fault (an exception)



Handling a Page Fault (2/4)

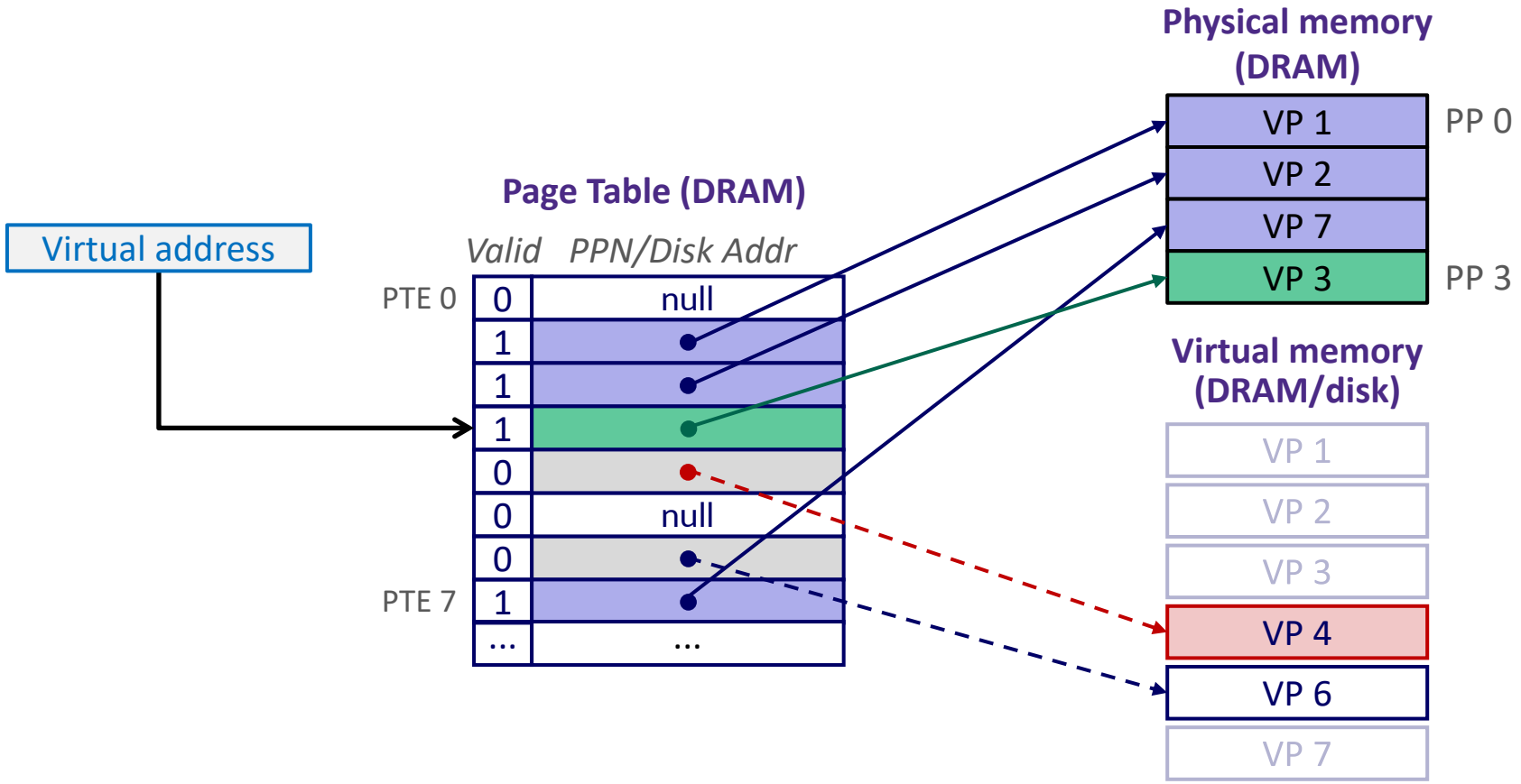
1) Page miss causes page fault (an exception)
2) **Page fault handler selects a *victim* to be evicted (here VP 4)**



Handling a Page Fault (3/4)

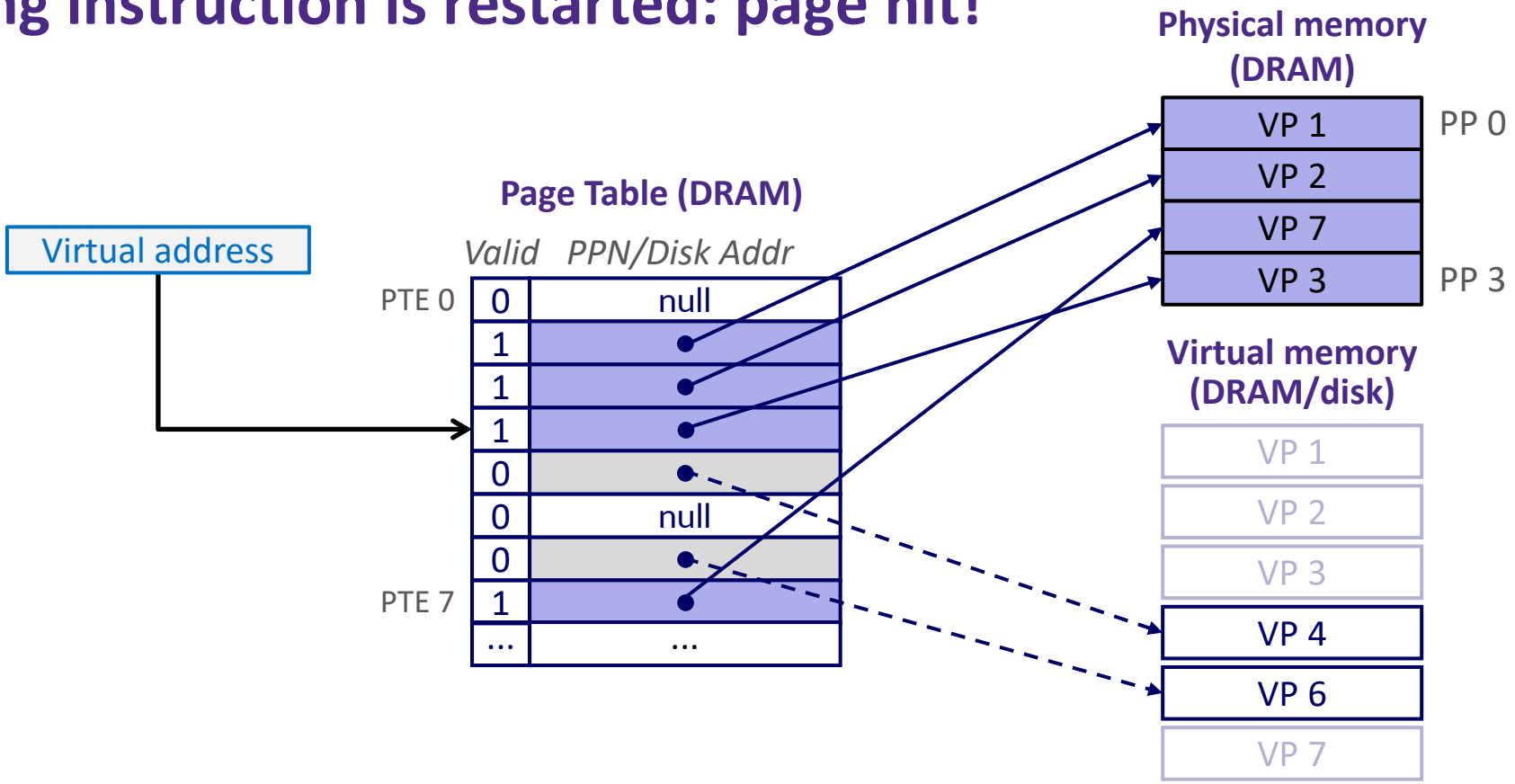
1) Page miss causes page fault (an exception)

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



Handling a Page Fault (4/4)

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

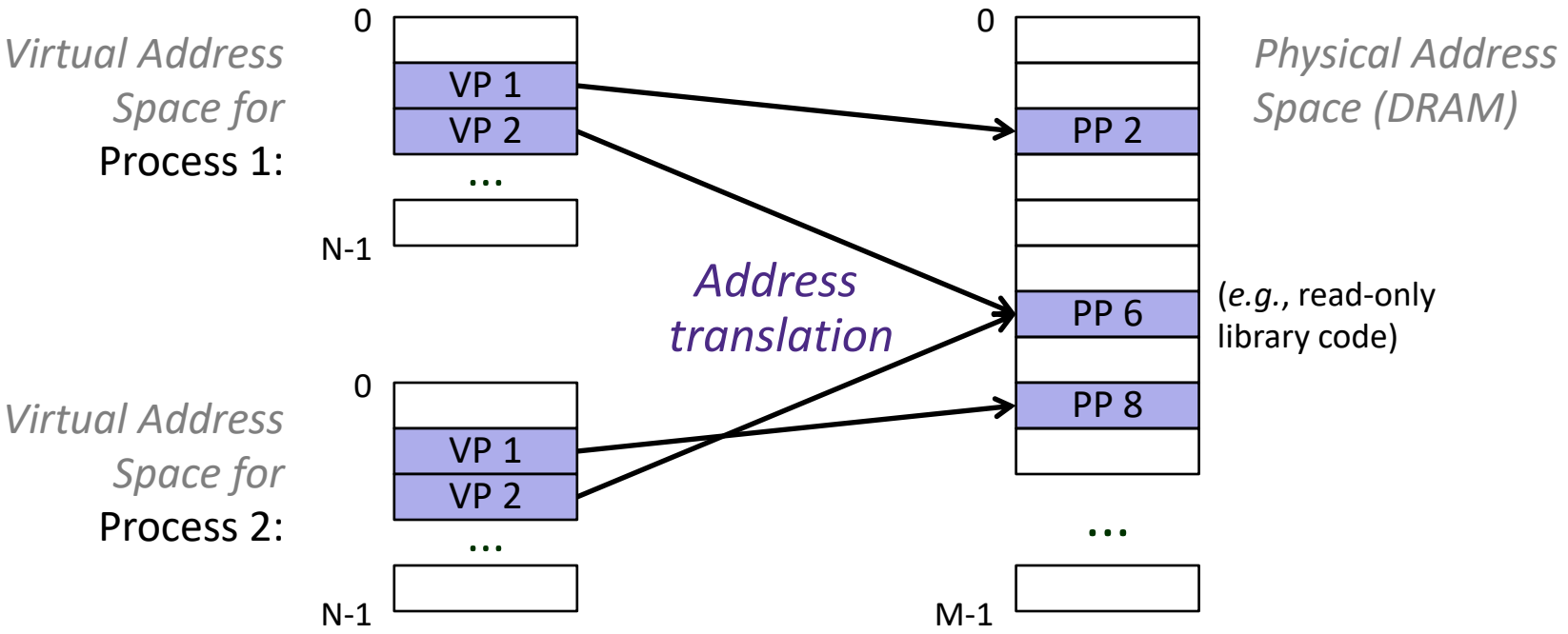


Lecture Outline (3/3)

- Paging, VM, and the Memory Hierarchy
- Address Translation
- **Memory Management**

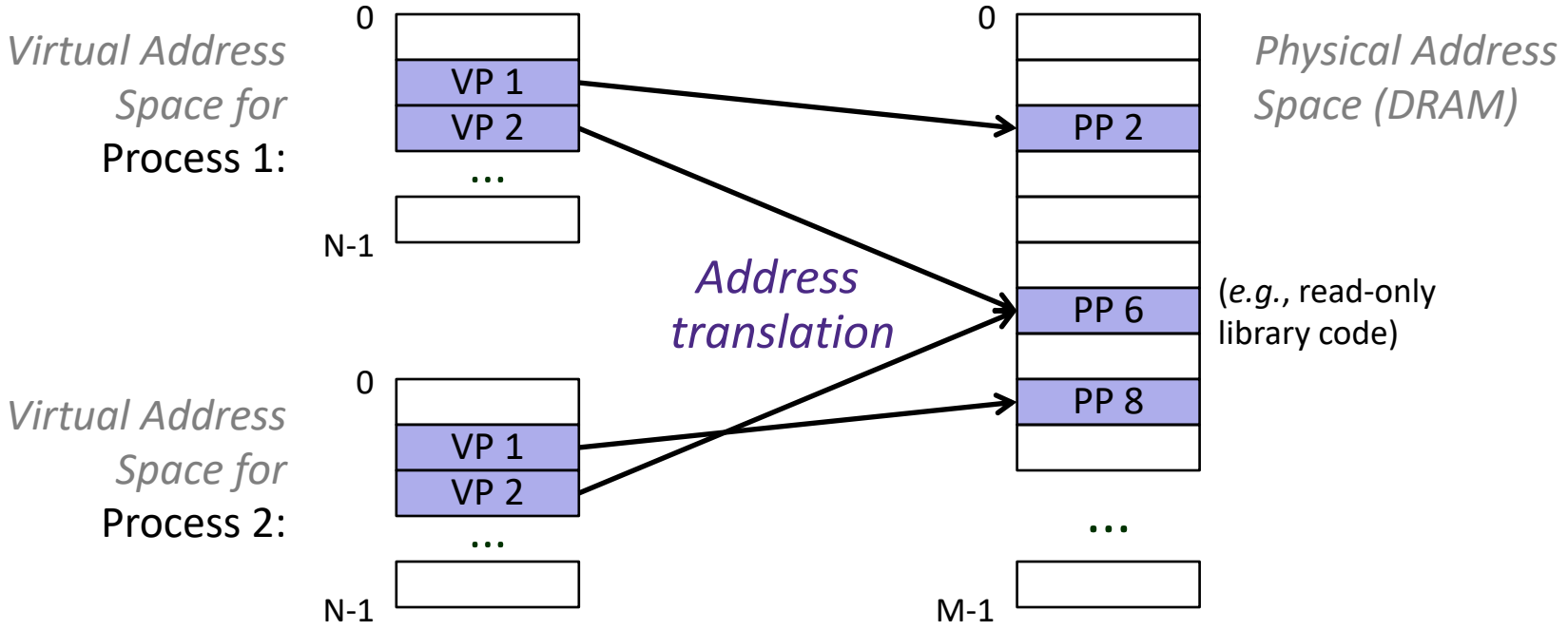
VM for Managing Multiple Processes

- Key abstraction: each process has its *own virtual address space*
  - It can view memory as a simple linear array
- Pages of this virtual address space are *not contiguous in physical memory*
  - Process needs another virtual page? Just map it to *any* open physical page!



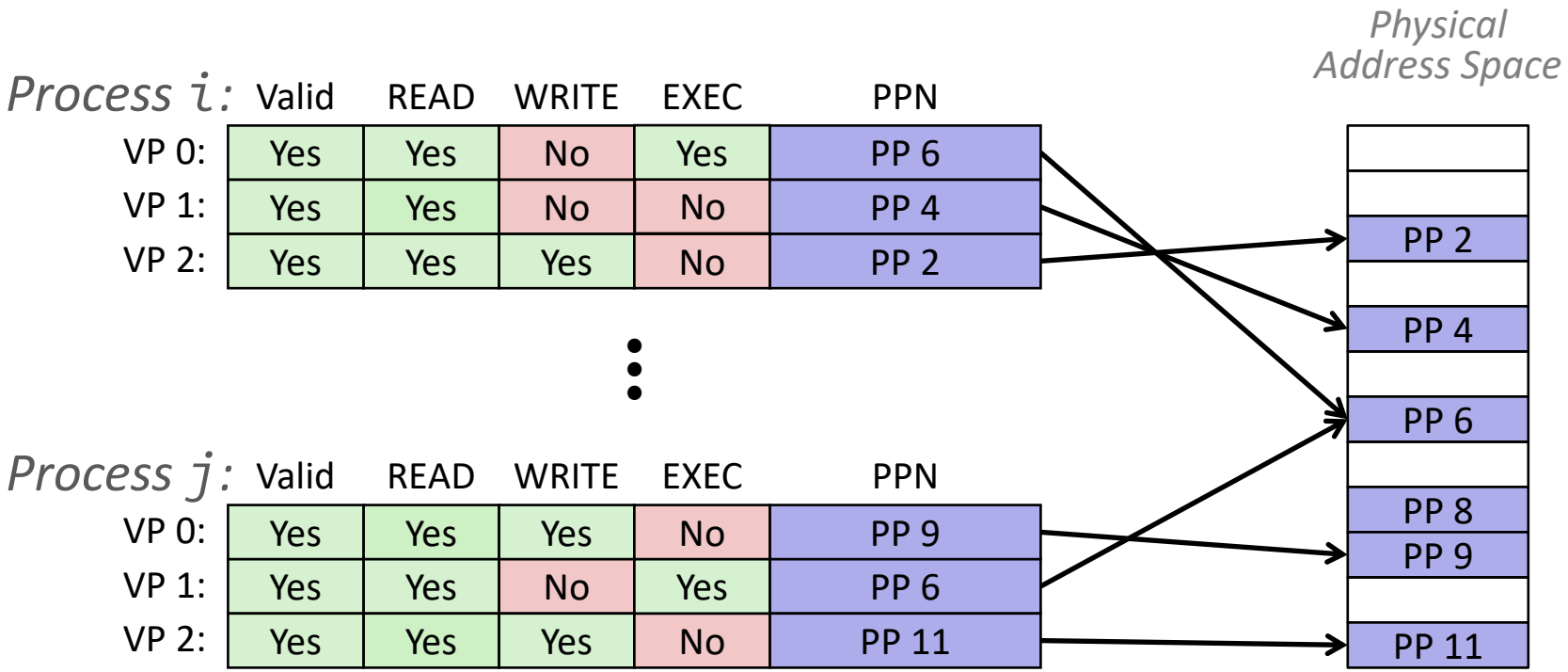
VM for Protection and Sharing

- Page mapping provides a simple mechanism to *protect* and *share* memory between processes
  - Sharing: Map VPs in separate address spaces to the same PP (here, PP 6)
  - Protection: Process can't access physical pages to which none of its VPs are mapped (here, Process 2 can't access PP 2)



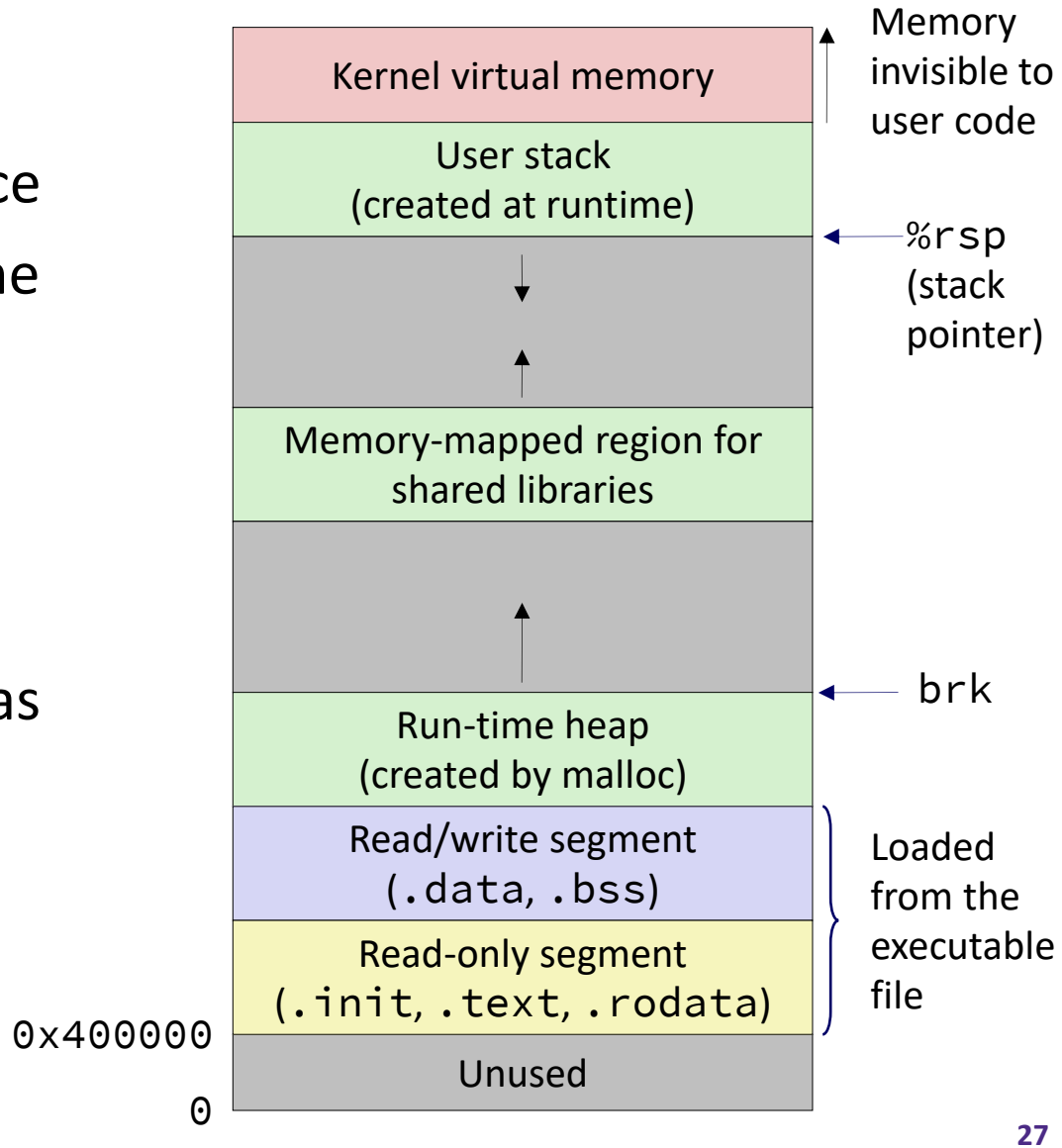
Memory Protection Within Process

- VM implements read/write/execute permissions
  - Extend page table entries with permission bits
  - MMU checks these permission bits on every memory access
    - If violated, raises exception and OS sends SIGSEGV signal to process (*i.e.*, segmentation fault)



Simplifying Linking and Loading

- Linking
  - Each program has similar virtual address space
  - Code, Data, and Heap always start at the same (virtual) addresses
- 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



Virtual Memory Concept Questions

- Which terms from caching are most similar/analogous to the new virtual memory terms?
  - block #, block size, cache line, cache set, index width, management bits, offset width, tag width
  - page size
  - page offset width
  - virtual page number
  - physical page number
  - page table entry
  - access rights

VM Parameters Questions (if time)

- Our system has the following properties
  - 1 MiB of physical address space
  - 4 GiB of virtual address space
  - 32 KiB page size
  - 4-entry fully associative TLB with LRU replacement

a) Fill in the following blanks:

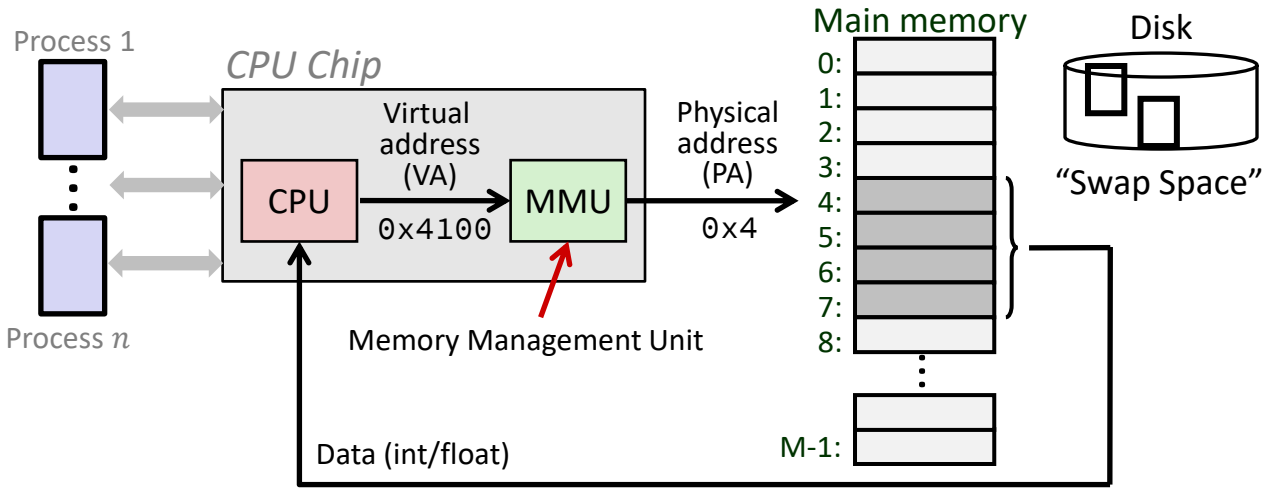
_____ Entries in a page table _____ Minimum bit-width of PTBR

_____ TLBT bits _____ Max # of valid entries in a page table

Lesson Summary (1/4)

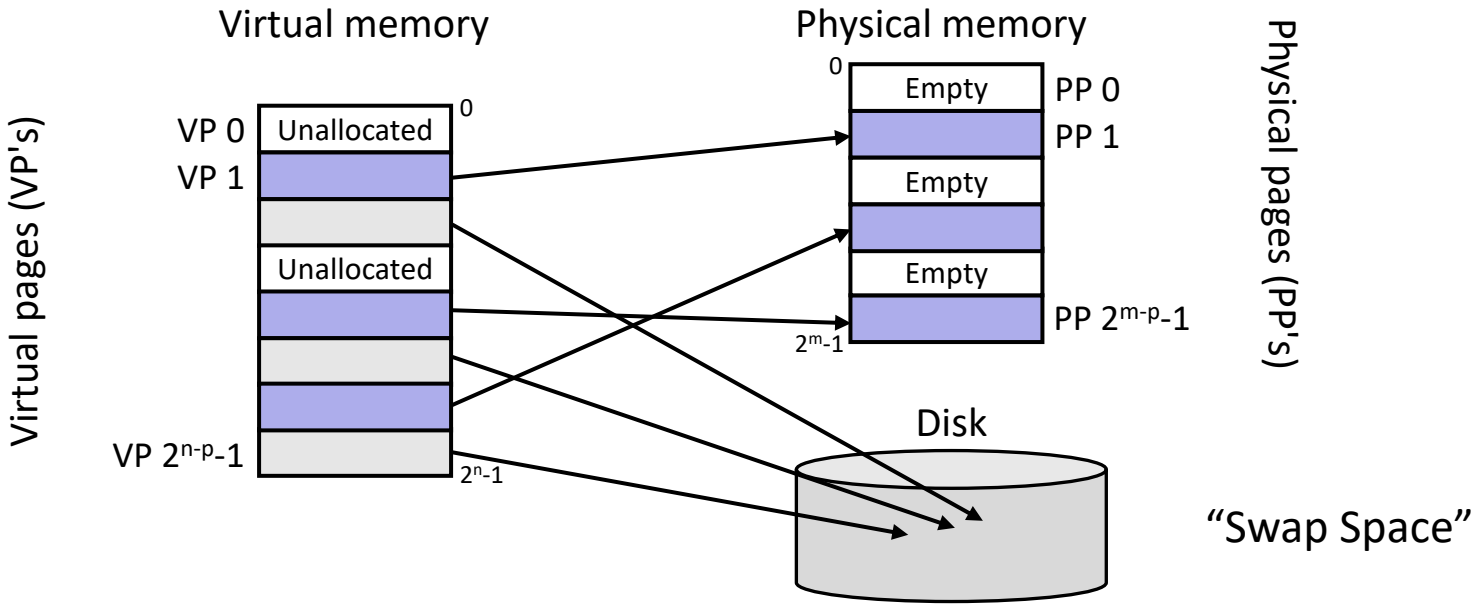
- ***Virtual memory*** is software's perspective (*e.g.*, memory layout), ***physical memory*** is hardware's perspective (*e.g.*, memory hierarchy)

- Virtual memory manages the memory for multiple concurrently running processes (implements *protection* and *sharing*)
  - Each process has its own virtual address space that gets mapped into parts of the physical address space
  - When run out of physical address space, put least recently used data in disk



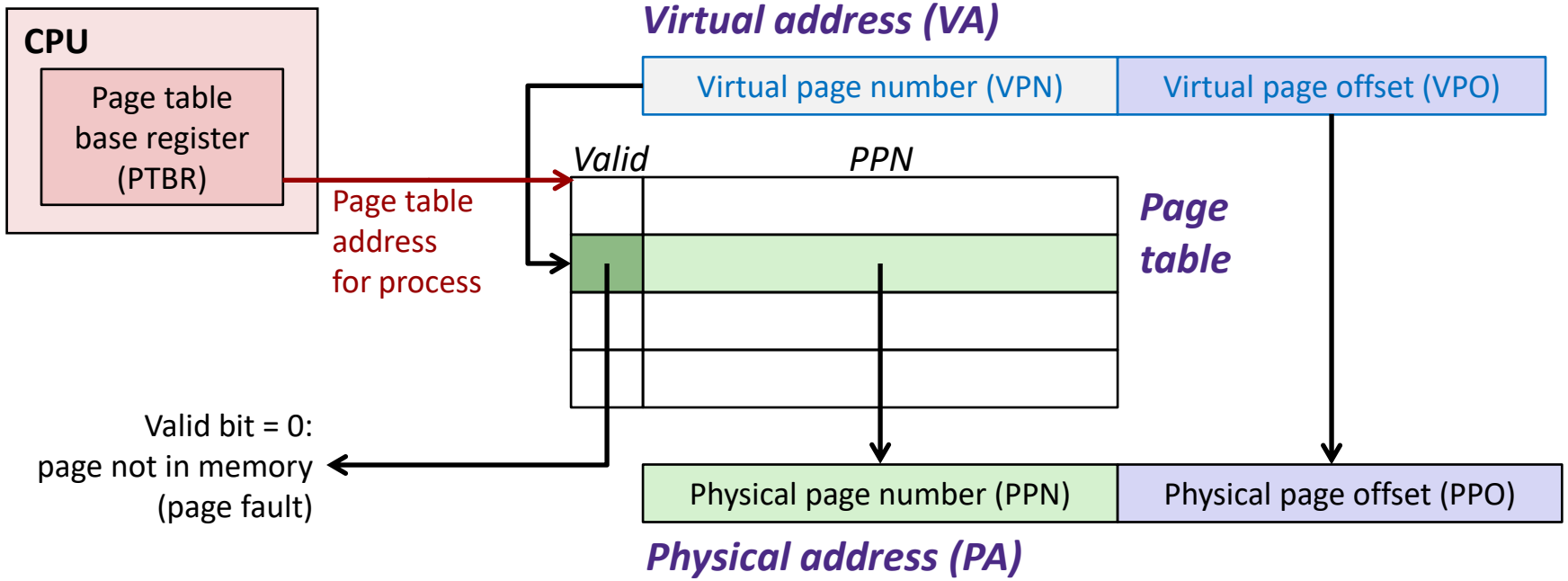
Lesson Summary (2/4)

- Can think of physical memory as a cache of virtual memory
  - Data is transferred between physical memory and swap space (disk) in ***pages***
  - Physical memory has caching parameters and properties
    - Large page size, fully associative, write-back, replacement policy
  - Caveats: virtual pages may not exist, data doesn't have to exist in both physical memory and disk



Lesson Summary (3/4)

- Address translation done via ***page tables***
  - Lookup tables (one per process) that map VPN → PPN
  - Uses management bits: valid bit, access rights (read, write, execute)
  - Stored in memory – page table for currently-running process is pointed to by ***page table base register*** (PTBR)



Lesson Summary (4/4)

- The address translation story (SO FAR) is check the page table in memory
  - Input: VPN, Output: PPN
  - *Page Fault*: Fetch page from disk to memory, update corresponding page table entry
  - *Page Table Hit*: Use existing page table entry