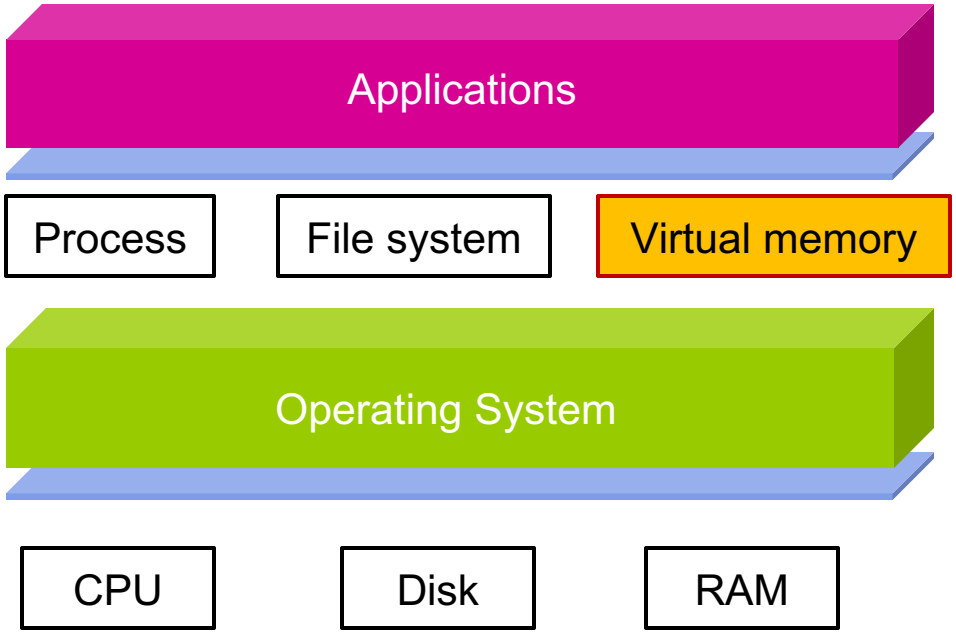
Virtual Memory

CS 202: Advanced Operating Systems

OS Abstractions



Problems with physical memory

- Size?
- Holes in address space?
- Isolation?

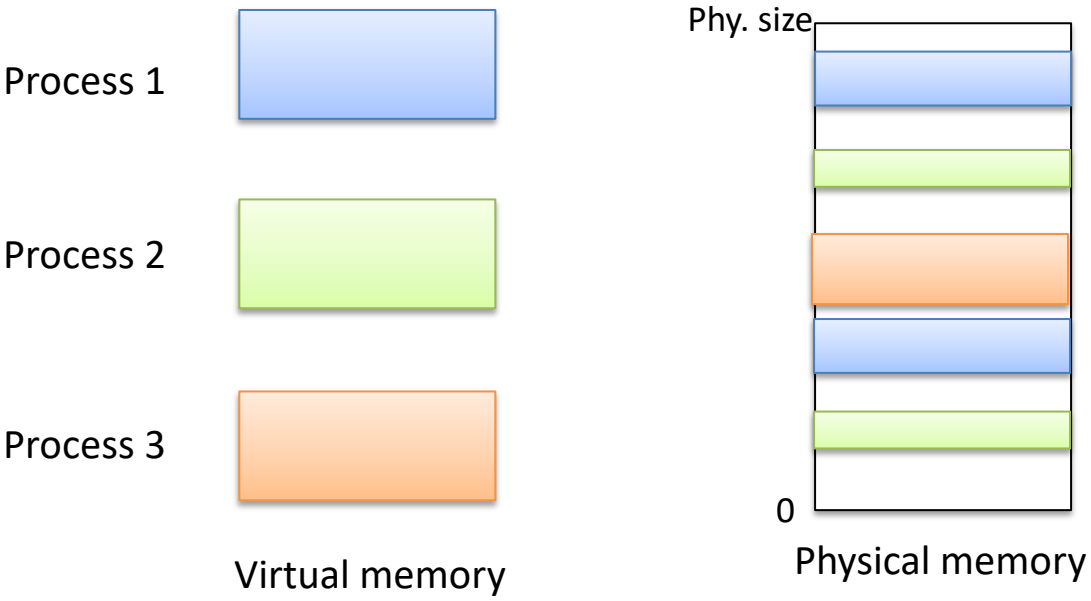
Problems with physical memory

- Crash if trying to access more than what we have
- Fragmentation
- Multiple programs may access the same address

- Motivation behind virtual memory
  - Key: flexibility in how we use the physical memory

Virtual vs. Physical Memory

- Physical memory: the actual memory installed in the computer
- Virtual memory: logical memory space owned by each process
  - Virtual memory space can be much larger than the physical memory space
  - Only part of the program needs to be in physical memory for execution



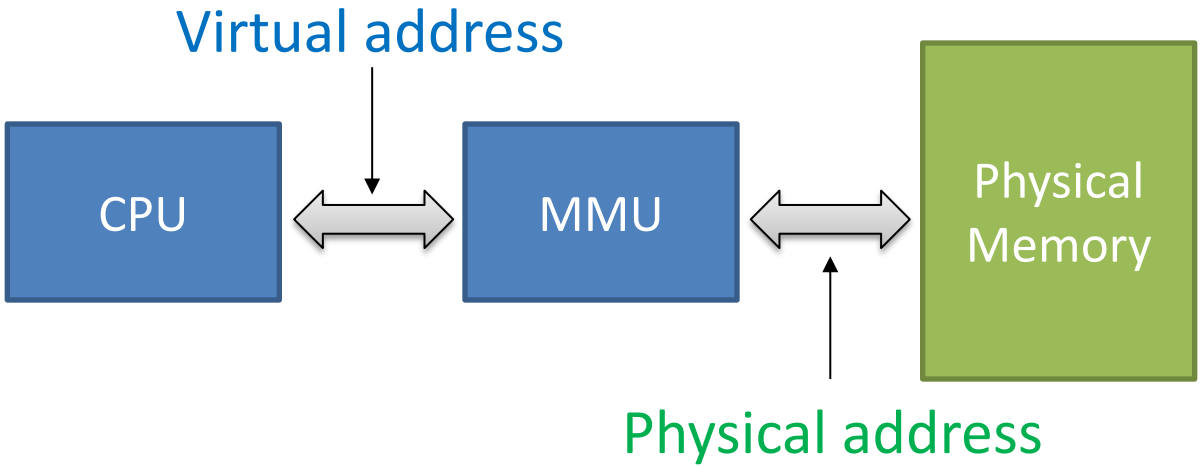
Virtual Address

- Processes use virtual (logical) addresses
  - Make it easier to manage memory of multiple processes
  - Virtual addresses are independent of the actual physical location of data referenced
  - Instructions executed by the CPU issue virtual addresses
  - Virtual address are translated by hardware into physical addresses (with help from OS)

- Mechanisms for virtual ↔ physical address translation
  - Hardware and OS support: MMU, Paging, and Page tables

Virtual Address

*MMU: Memory Management Unit



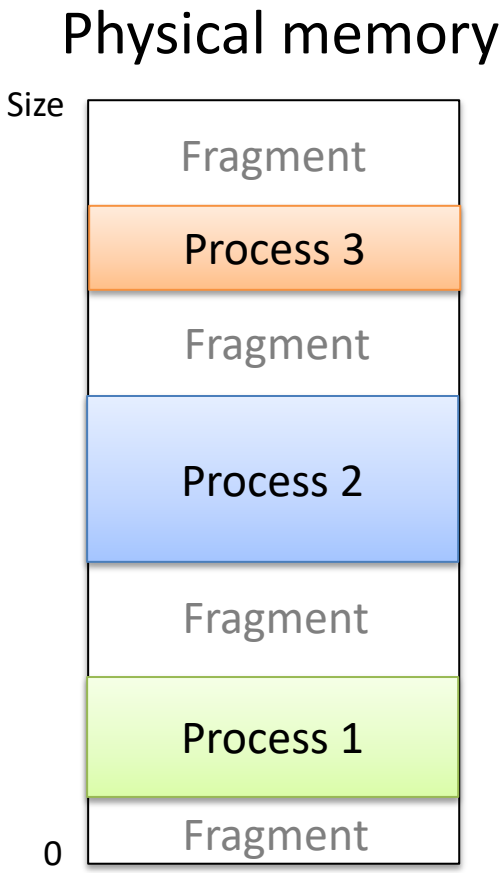
- MMU: Translates virtual address to physical address dynamically at every reference
  - Many ways to do this translation…
  - Need hardware support and OS management algorithms
- Requirements
  - Protection – restrict which addresses processes can use
  - Fast translation – lookups need to be fast
  - Fast change – updating memory hardware on context switch

External Fragmentation

New process

Process 4

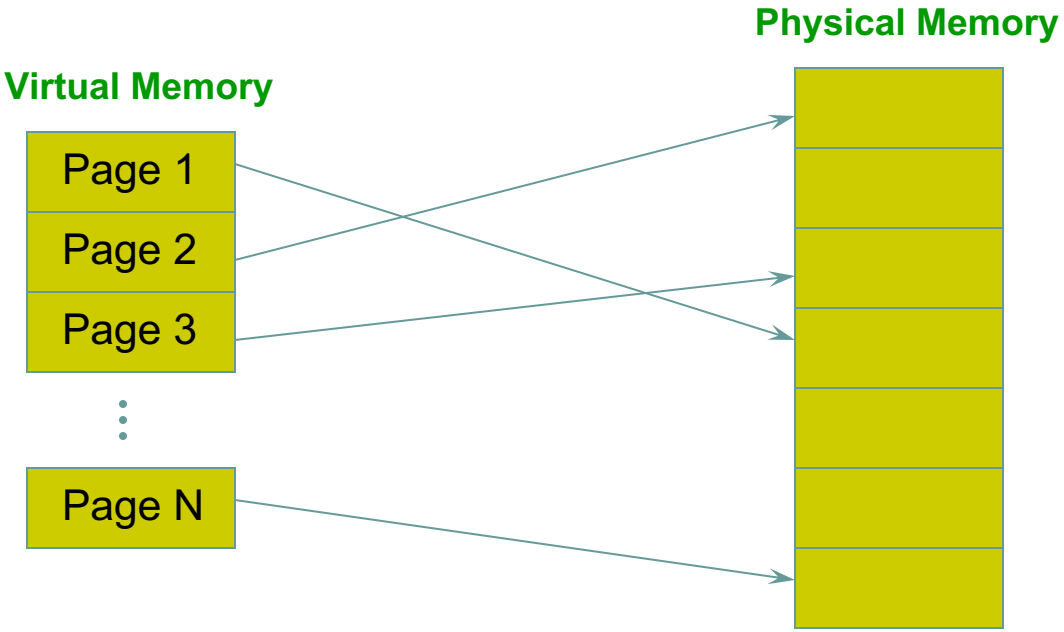
Not enough contiguous physical space



Paging

- Main Idea: Split address space into equal sized units called pages
  - Each can go anywhere!

e.g., page size of 4KB



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

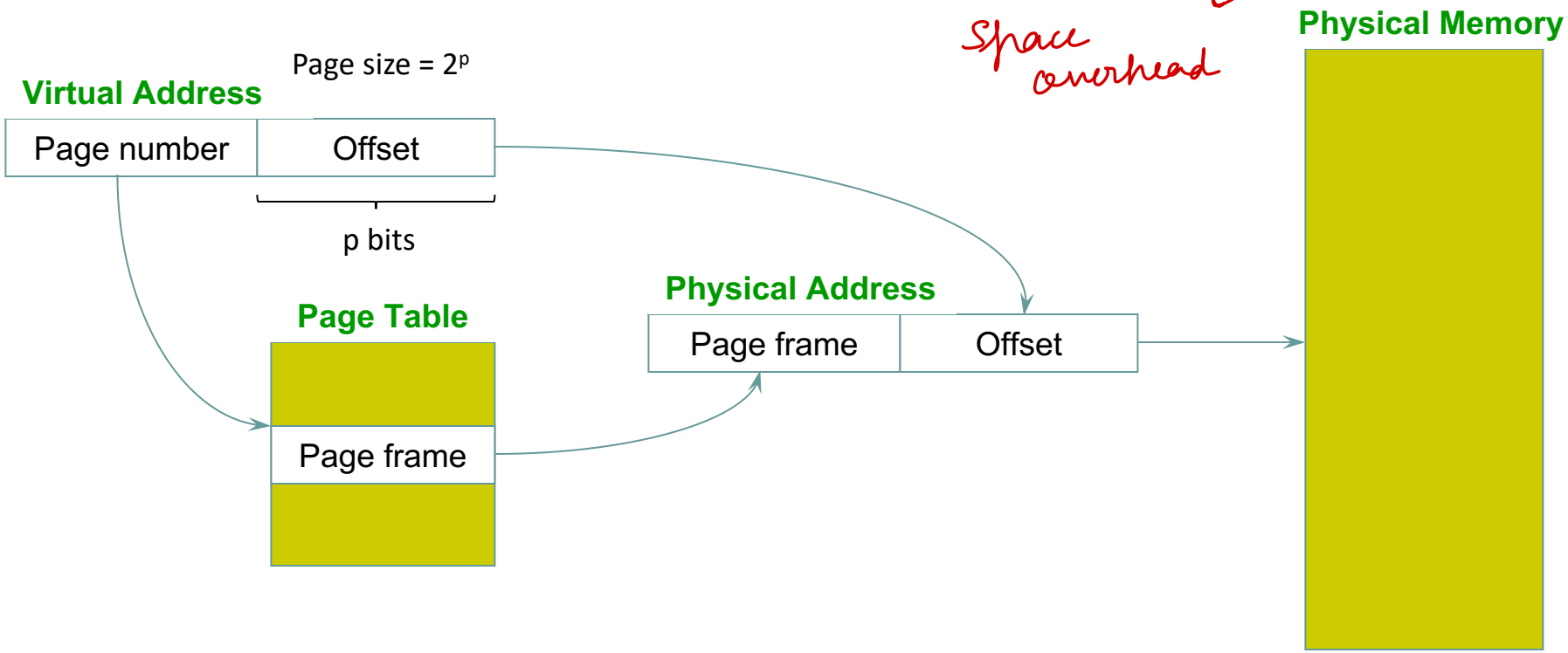
But need to keep track of where things are!

Page Lookups

- *Page table:* an array of page table entries (PTEs) that maps virtual pages to physical pages
  - Per-process kernel data structure (part of process's state) (why?)

For isolation

Space overhead

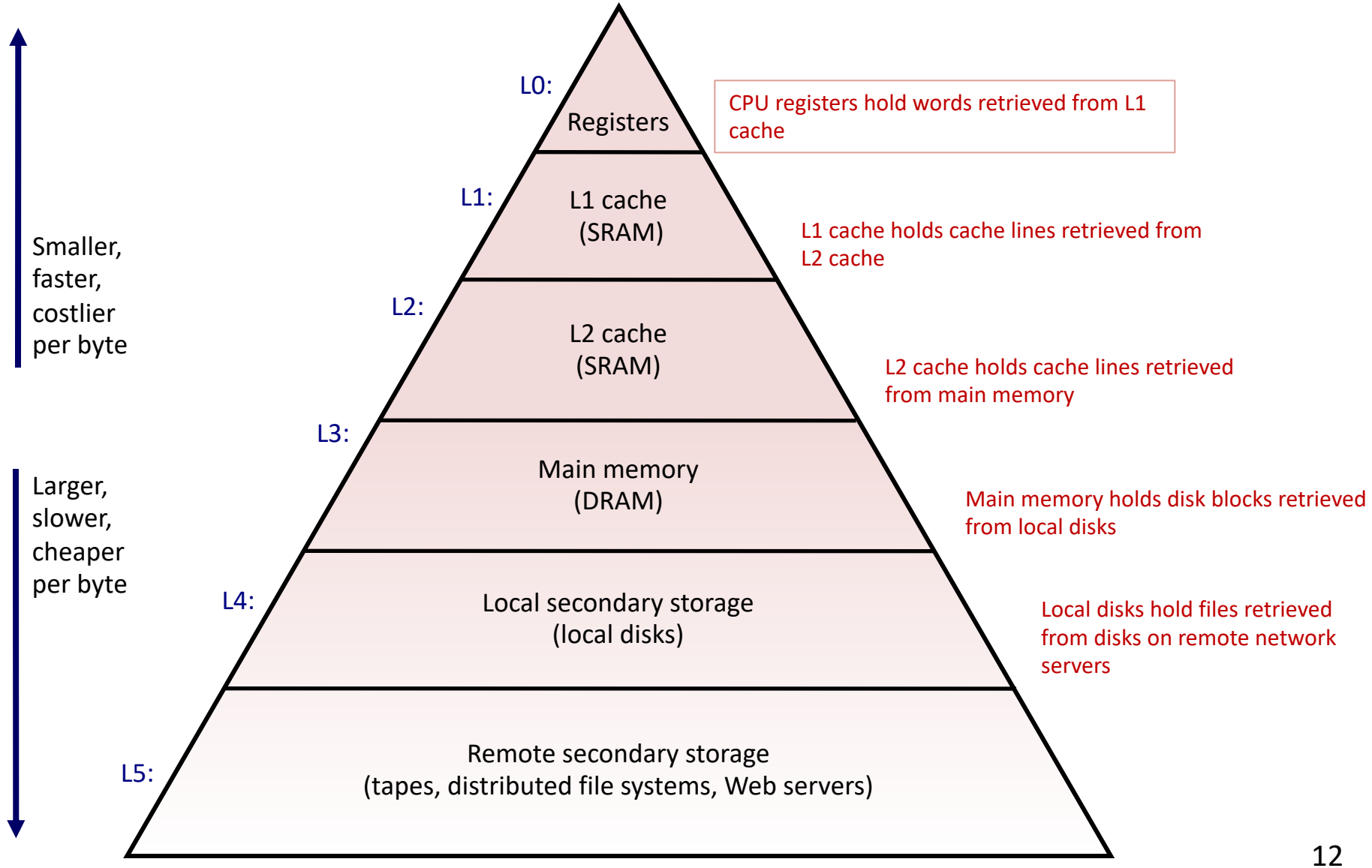


Why Virtual Memory (VM)?

- Simplifies memory management
  - Each process gets the same uniform linear address space

- With VM, we can:
  - Use DRAM as a cache for the parts of a virtual address space
  - Protect against illegal memory access
  - Share data among processes efficiently and safely

Example Memory Hierarchy



Memory hierarchy

- Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

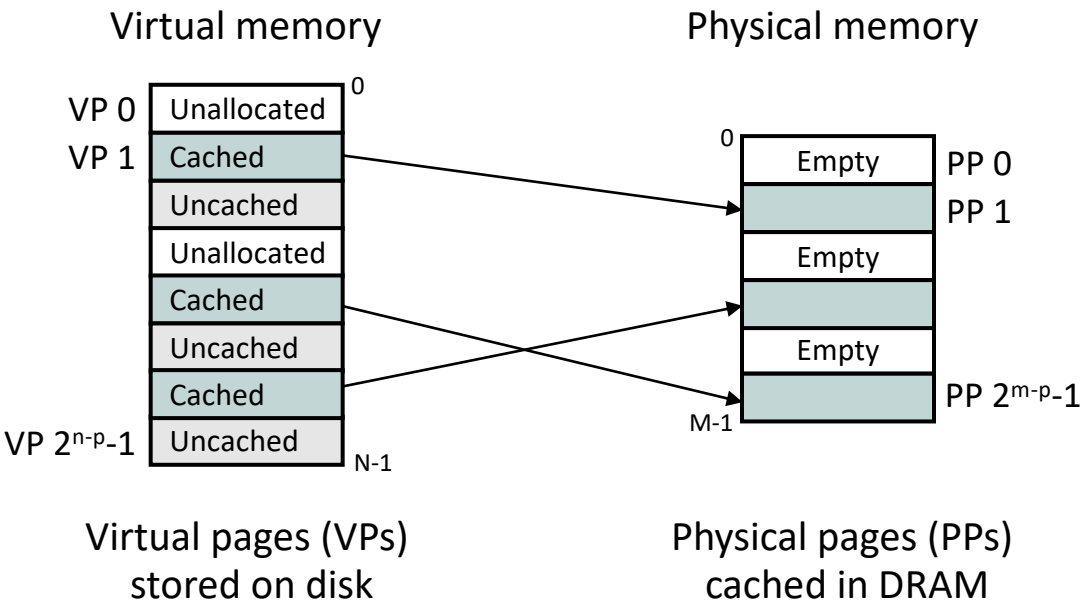
- Fundamental idea of a memory hierarchy:
  - For each layer, a faster, smaller device caches a larger, slower device

- Why do memory hierarchies work?
  - Because of locality!
    - Hit fast memory much more frequently even though its smaller
  - Thus, the storage at level k+1 can be slower (but larger and cheaper!)

- *Big Idea:* The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

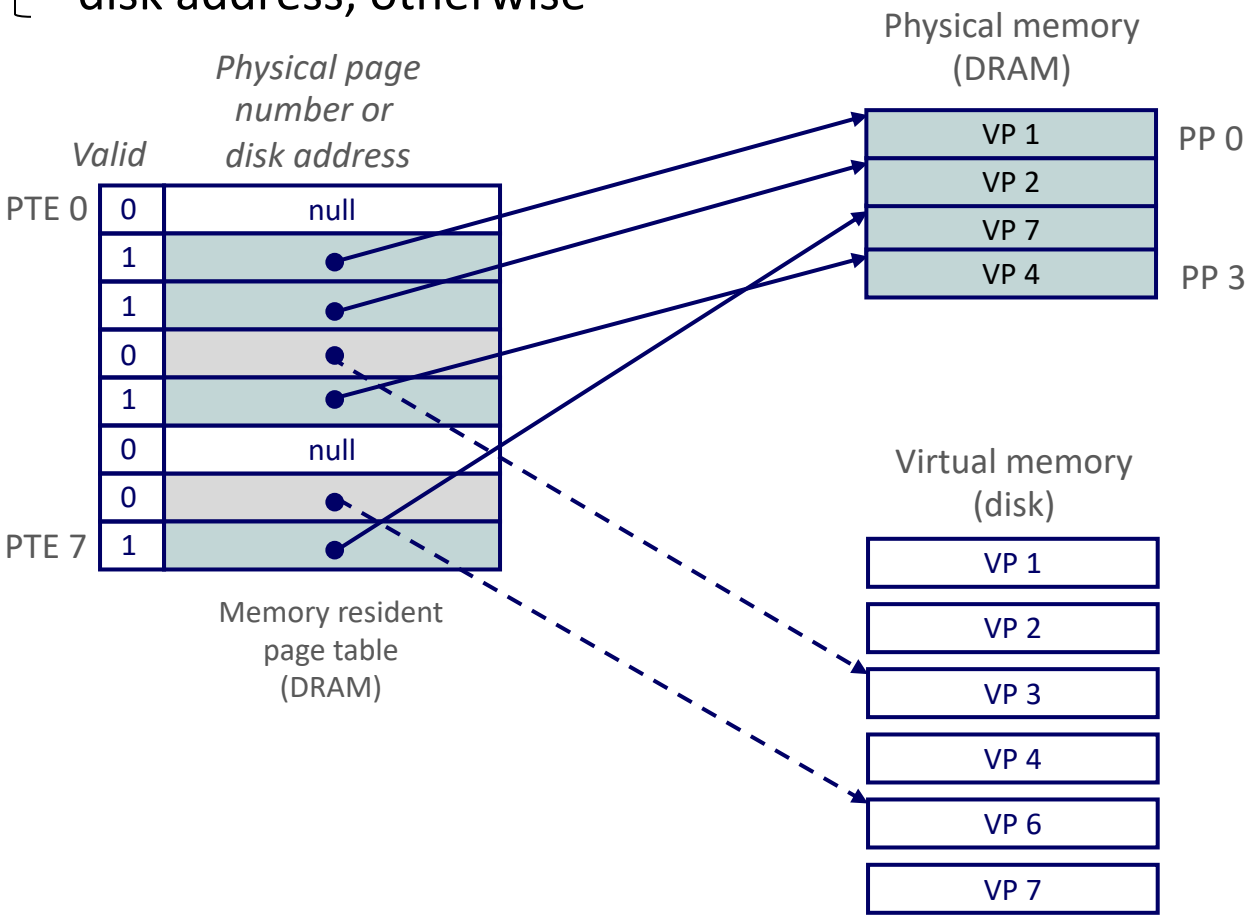
VM as a tool for caching

- *Virtual memory* mapped to *N* contiguous bytes stored on disk
- The contents of the array on disk are cached in *physical memory*
  - *DRAM as a cache for disk*
  - These cache blocks are called *pages* (size is P = 2^p bytes)



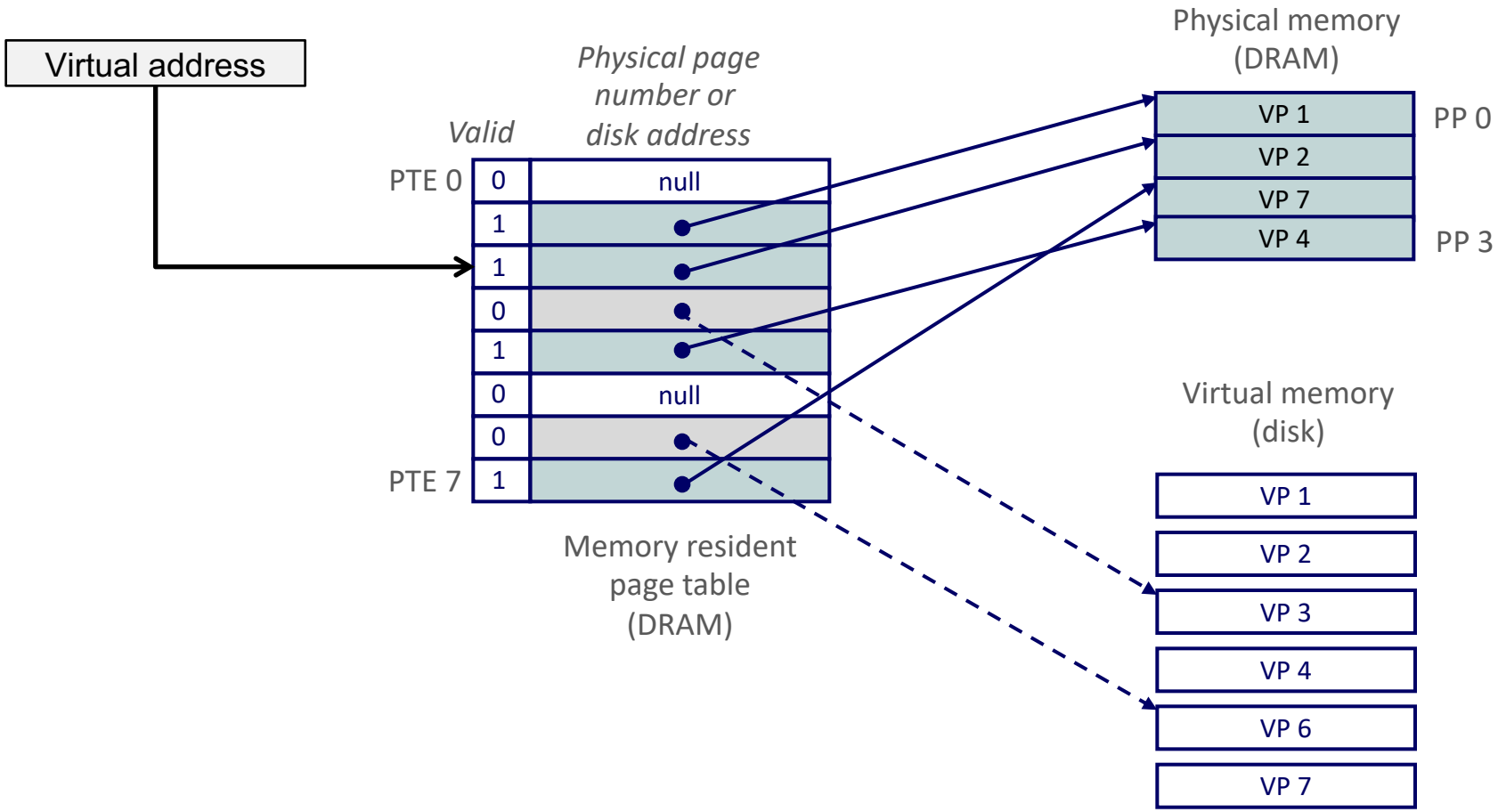
Page Tables

- *Page table:* page table entries (PTEs) map virtual to physical pages
  - PTE: - physical page number, if page is cached (resident) in DRAM
    - disk address, otherwise



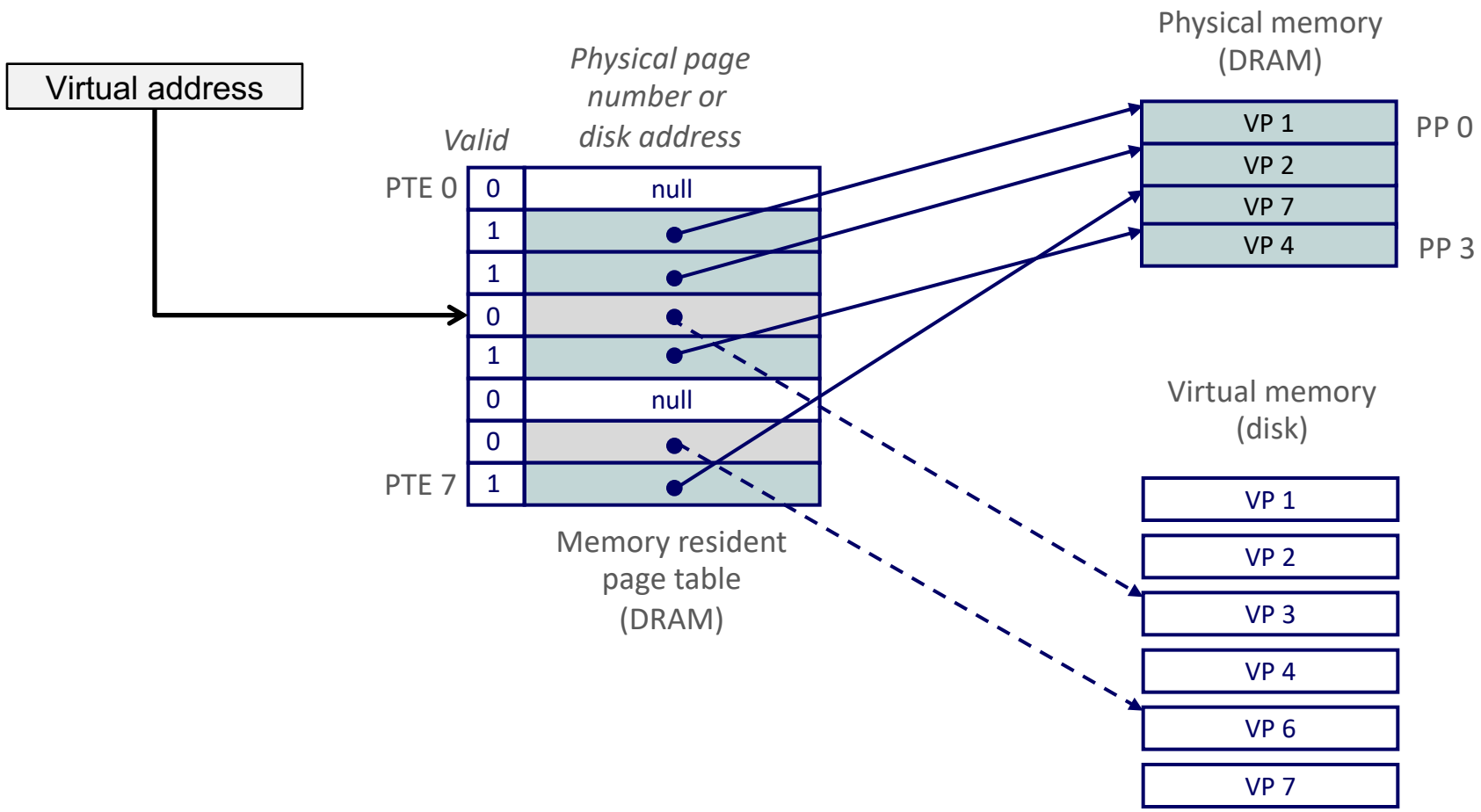
Page Hit

- *Page hit:* Access to a page that is in physical memory



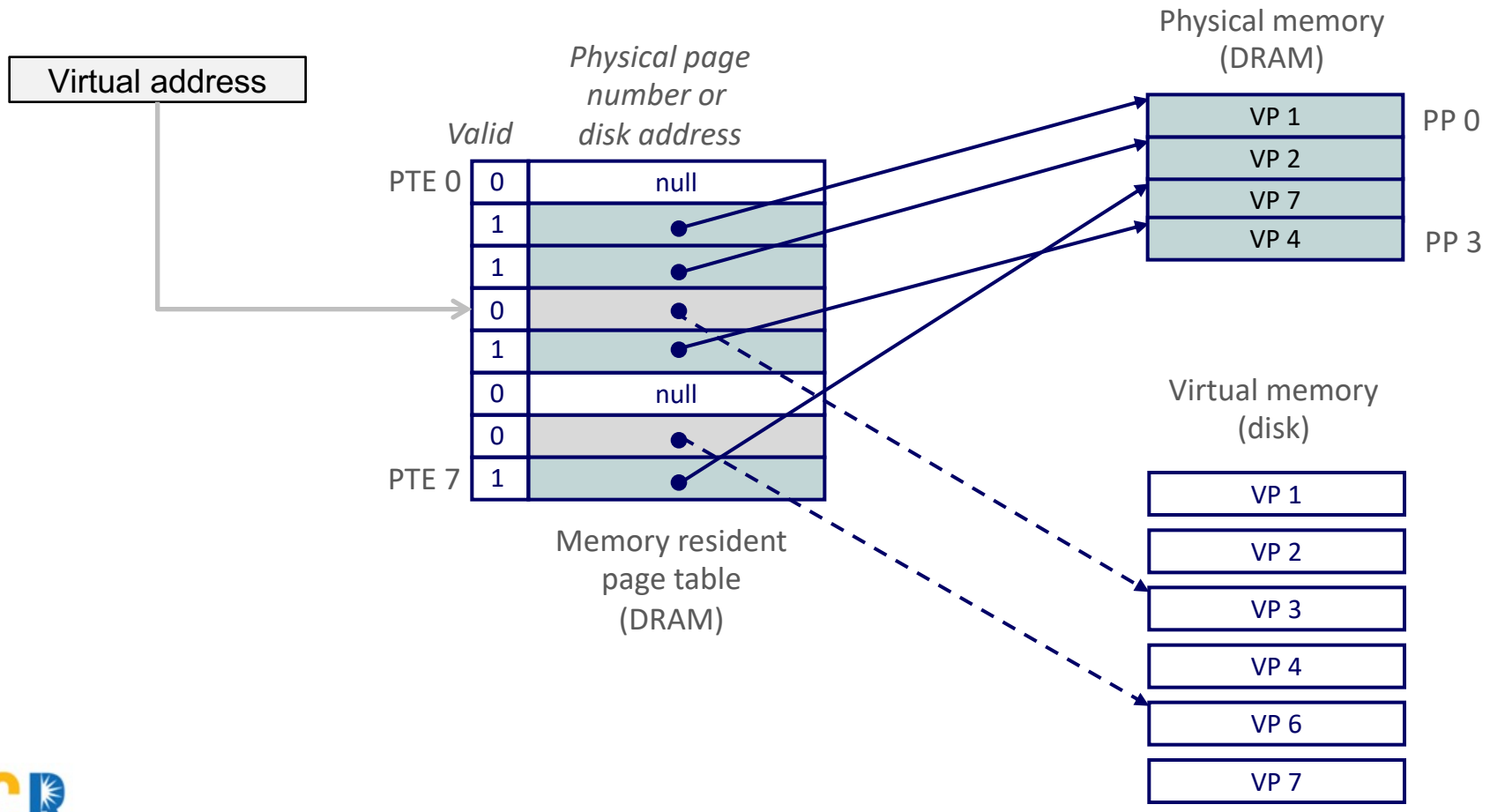
Page Fault

- *Page fault:* Access to a page that is *not* in physical memory



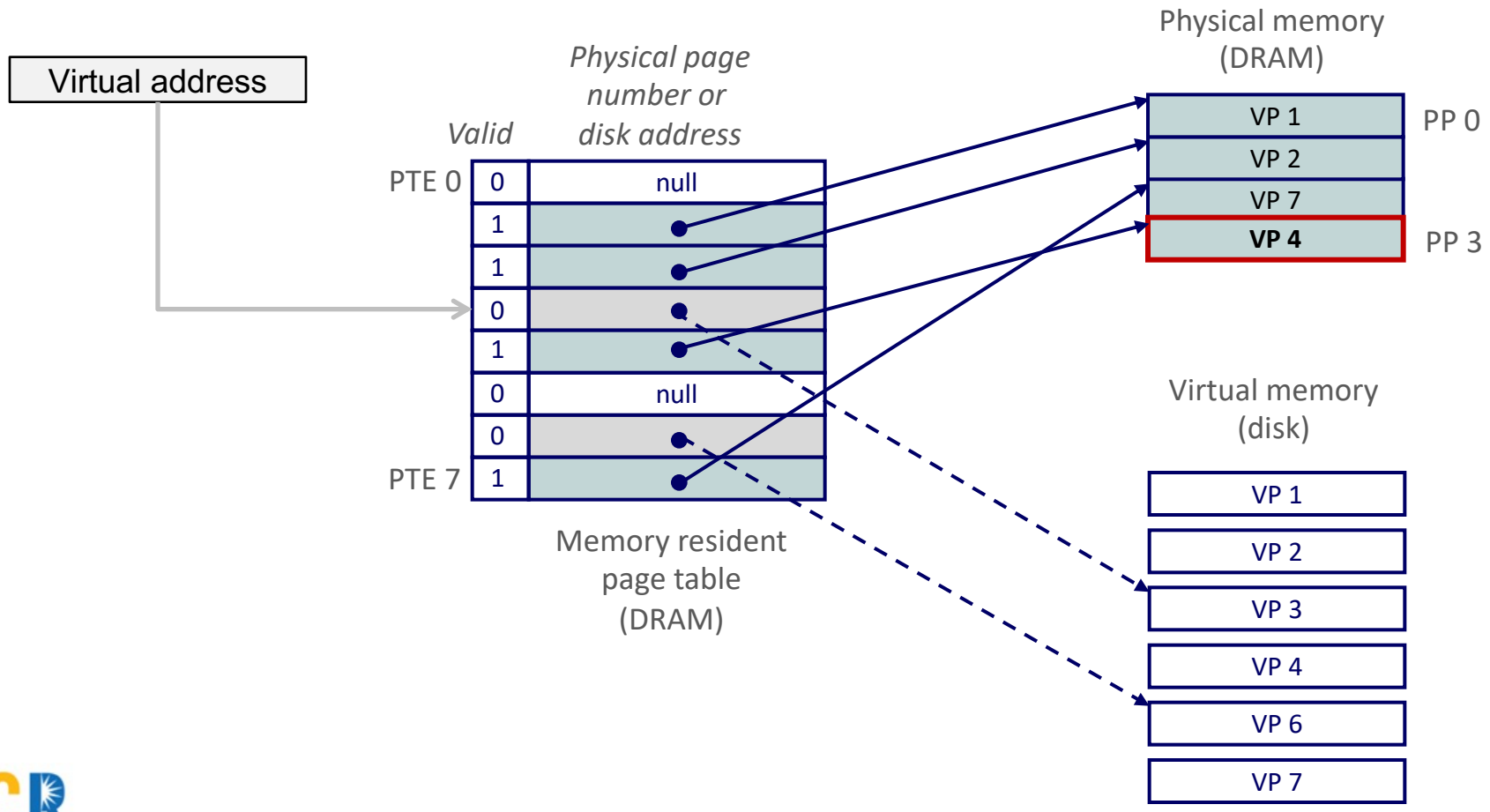
Handling Page Fault

- Page miss causes page fault (an exception)



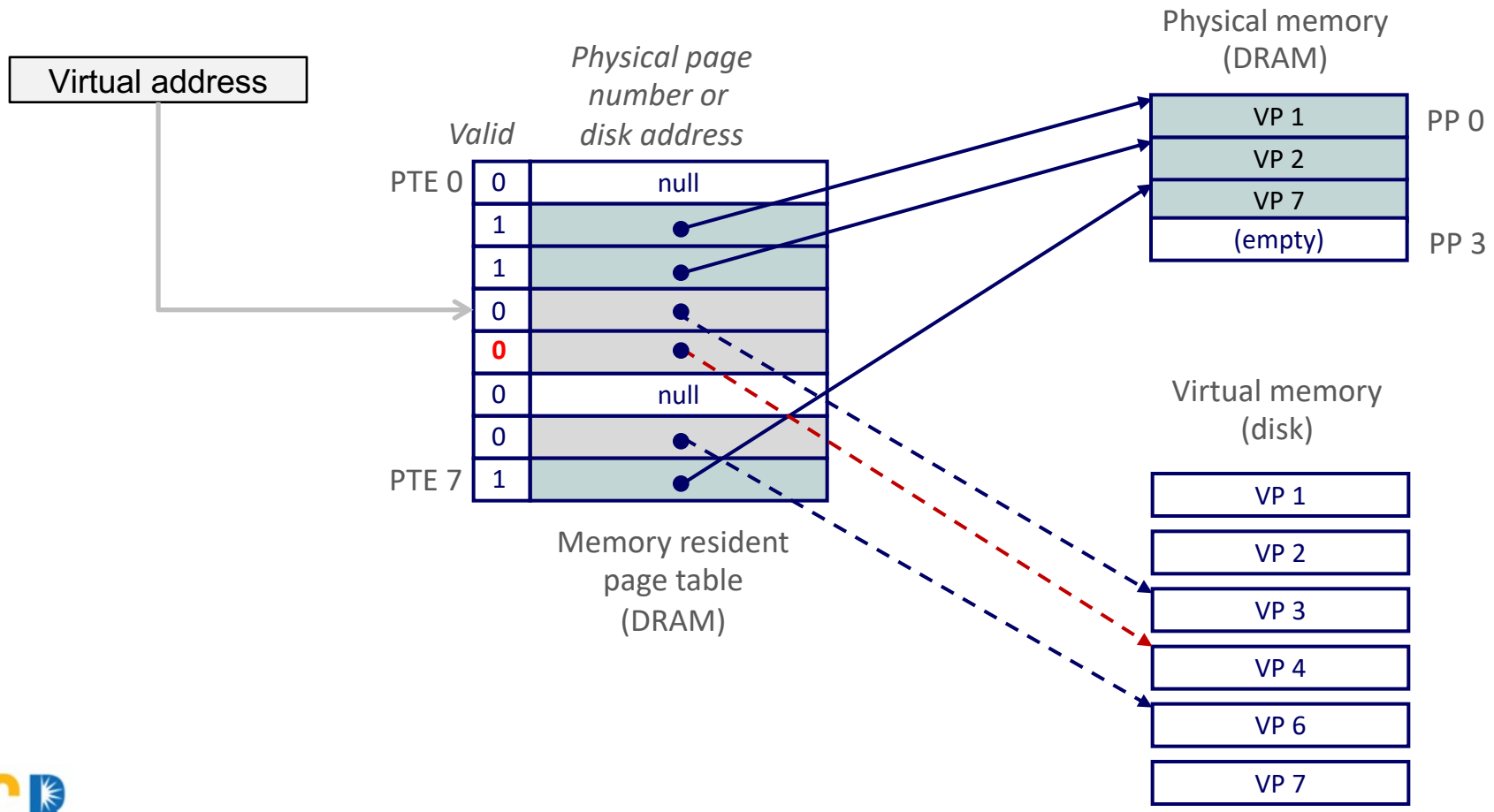
Handling Page Fault

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



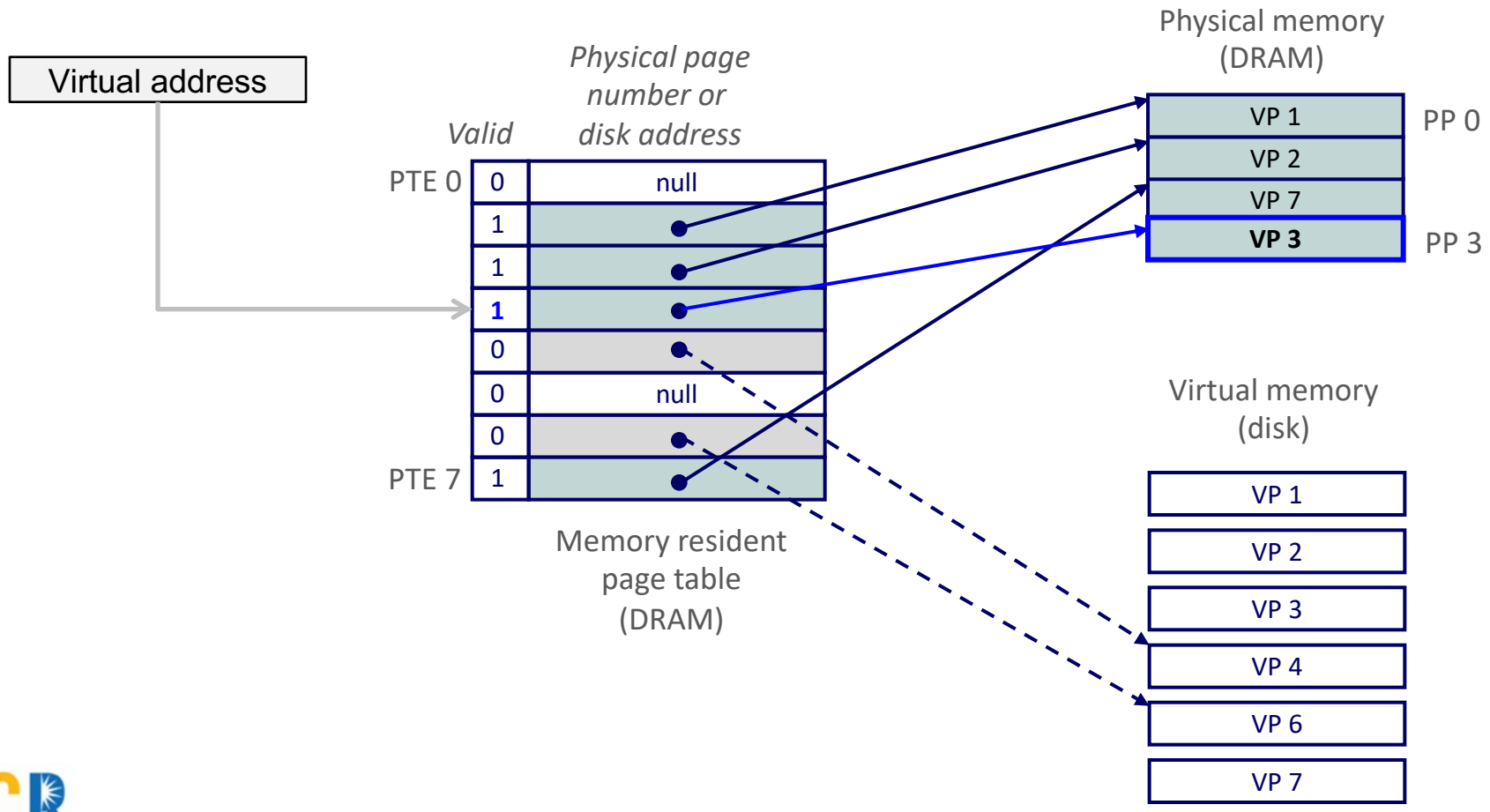
Handling Page Fault

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



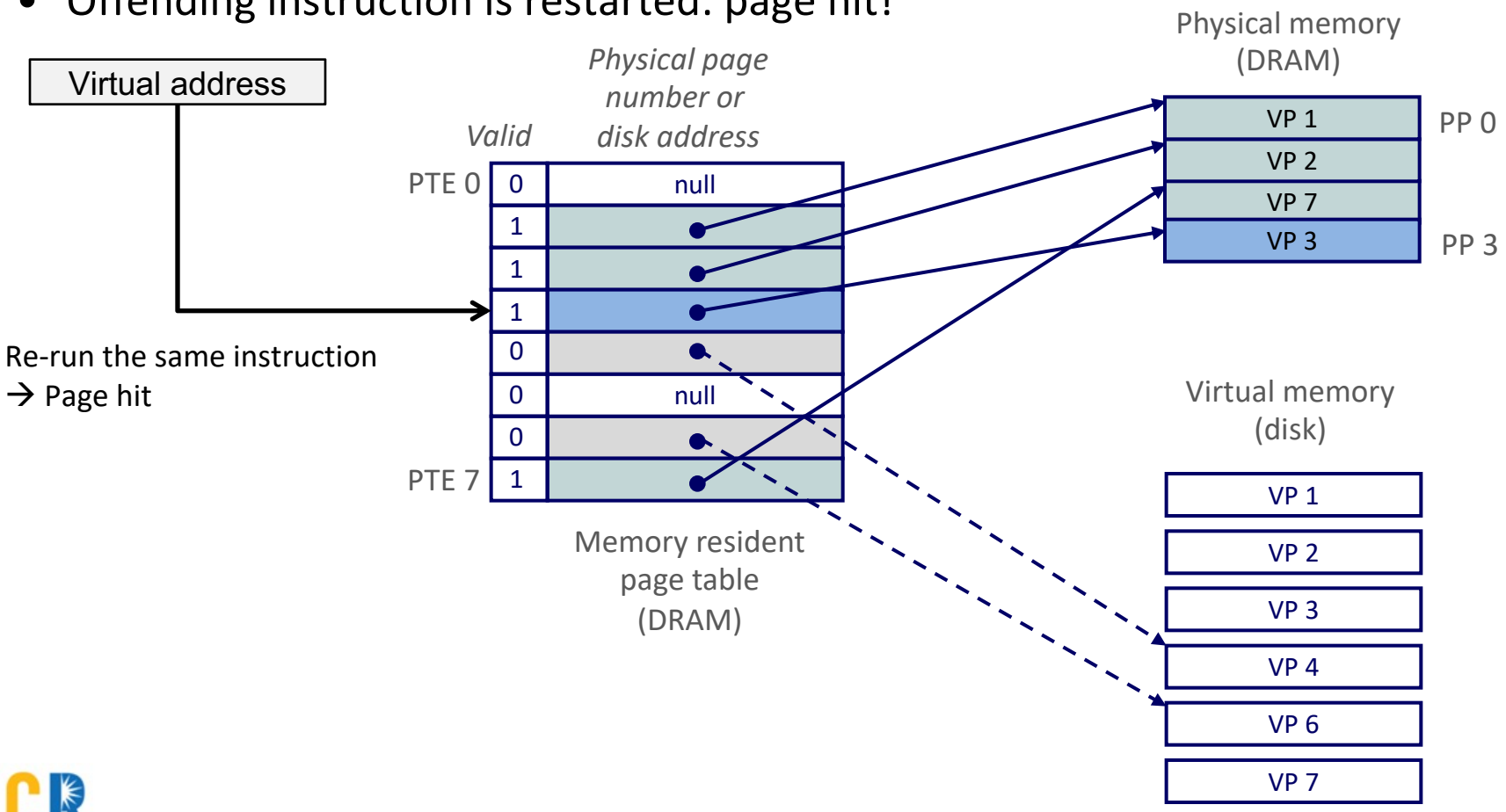
Handling Page Fault

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



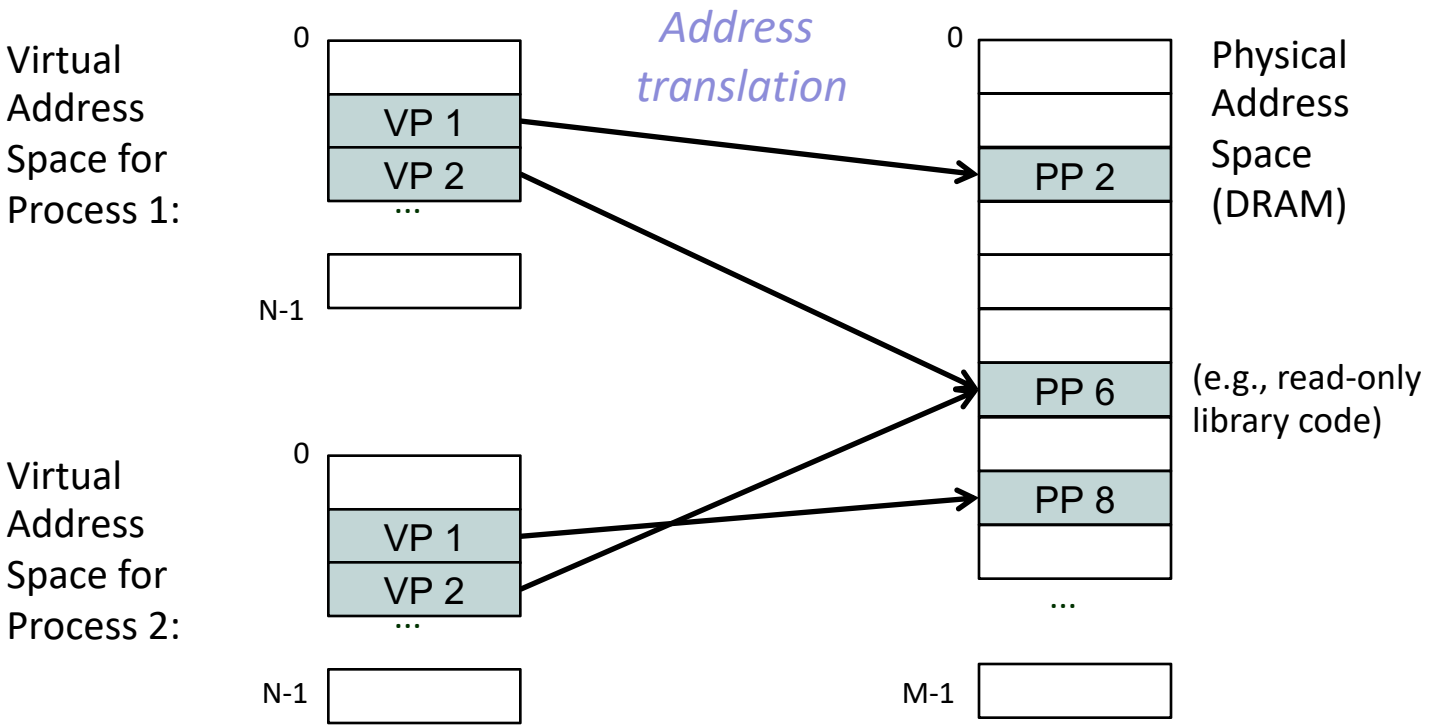
Handling Page Fault

- 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!



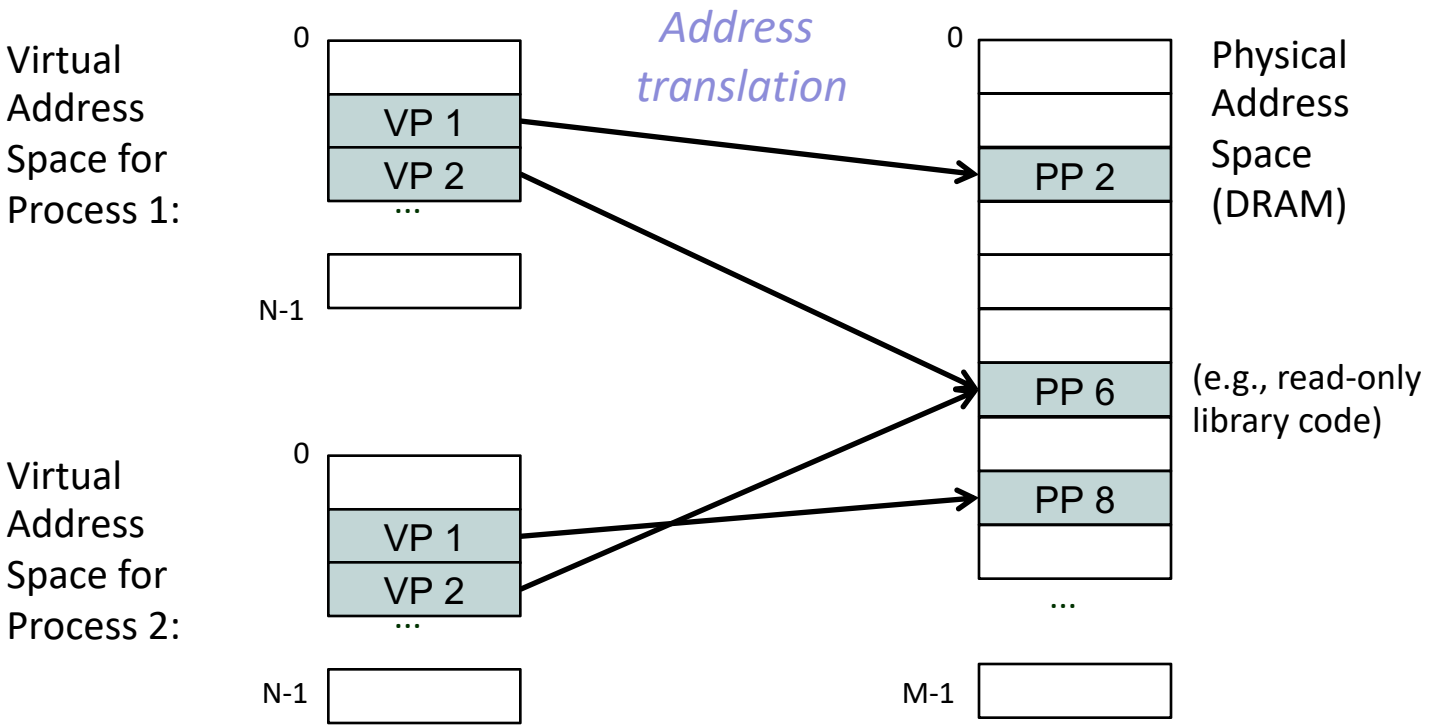
VM as a Tool for Mem Management

- 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 Mem Management

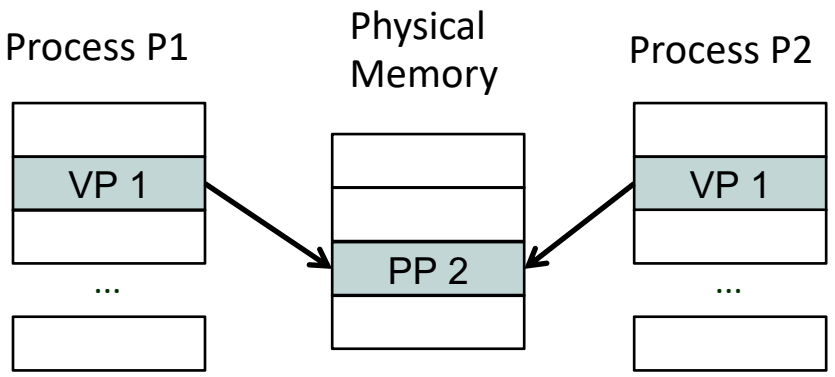
- Memory allocation
  - Each virtual page can be mapped to any physical page
  - A virtual page can be stored in different physical pages at different times
- Sharing code and data among processes
  - Map virtual pages to the same physical page (here: PP 6)



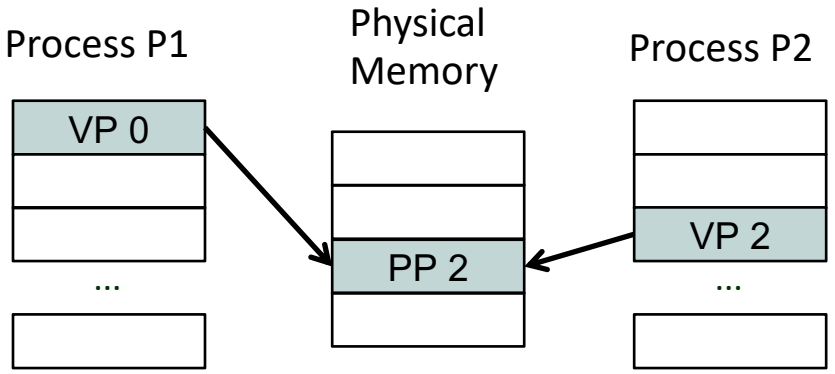
Sharing

- Can map shared physical memory at same or different virtual addresses in each process' address space

– Same VA:
  - Both P1 and P2 maps the 1st virtual page (VP1) to the shared physical page (PP2)



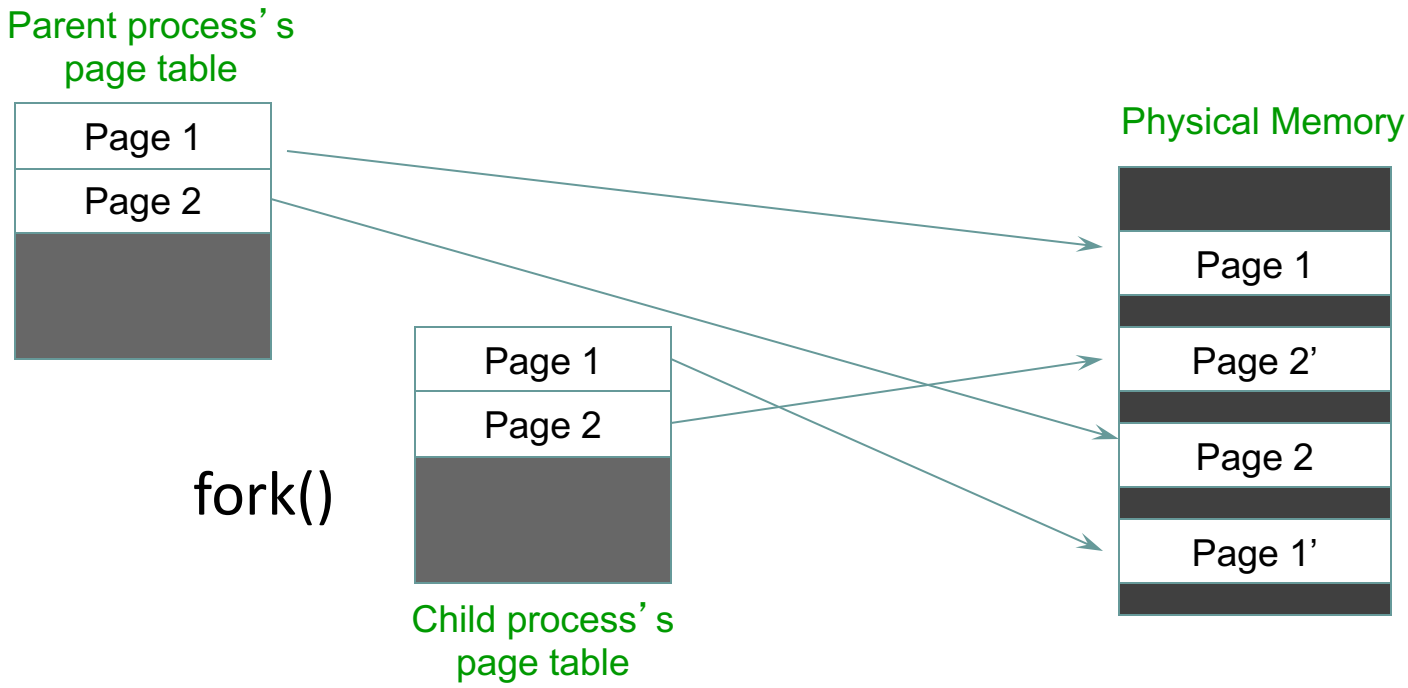
– Different VA:
  - P1 maps VP0 to PP2, but P2 maps VP2 to PP2



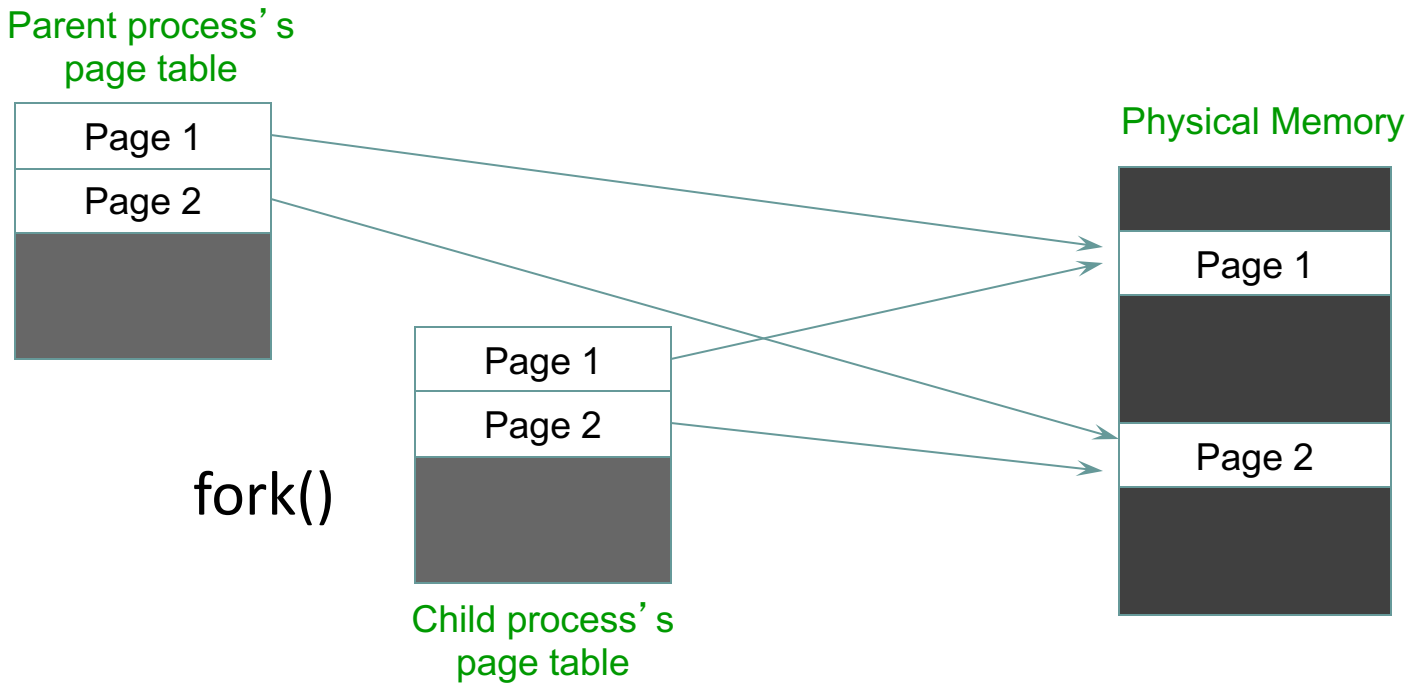
Copy on Write

- 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()?

fork() without Copy on Write

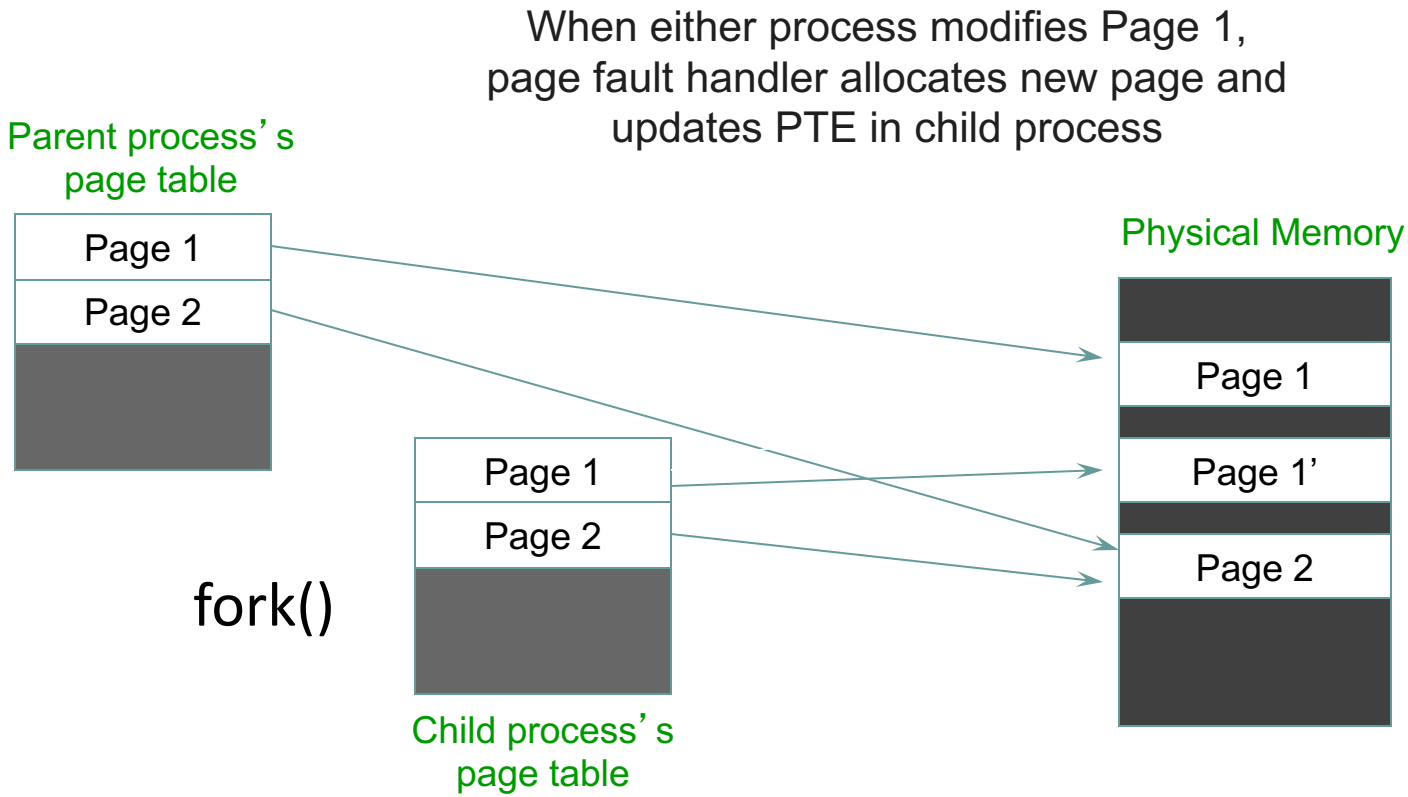


fork() with Copy on Write



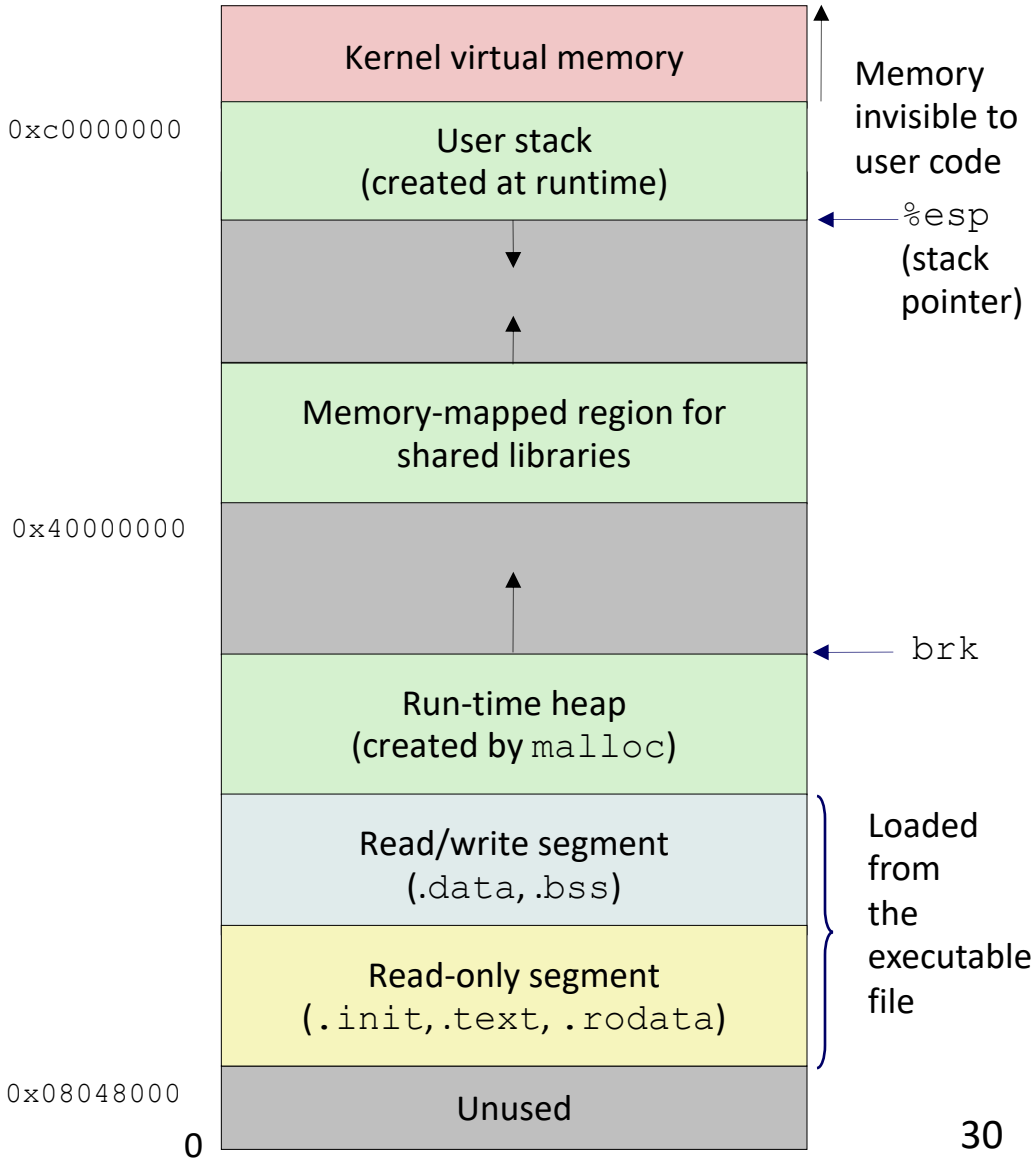
Protection bits set to prevent either process from writing to any page

fork() with Copy on Write



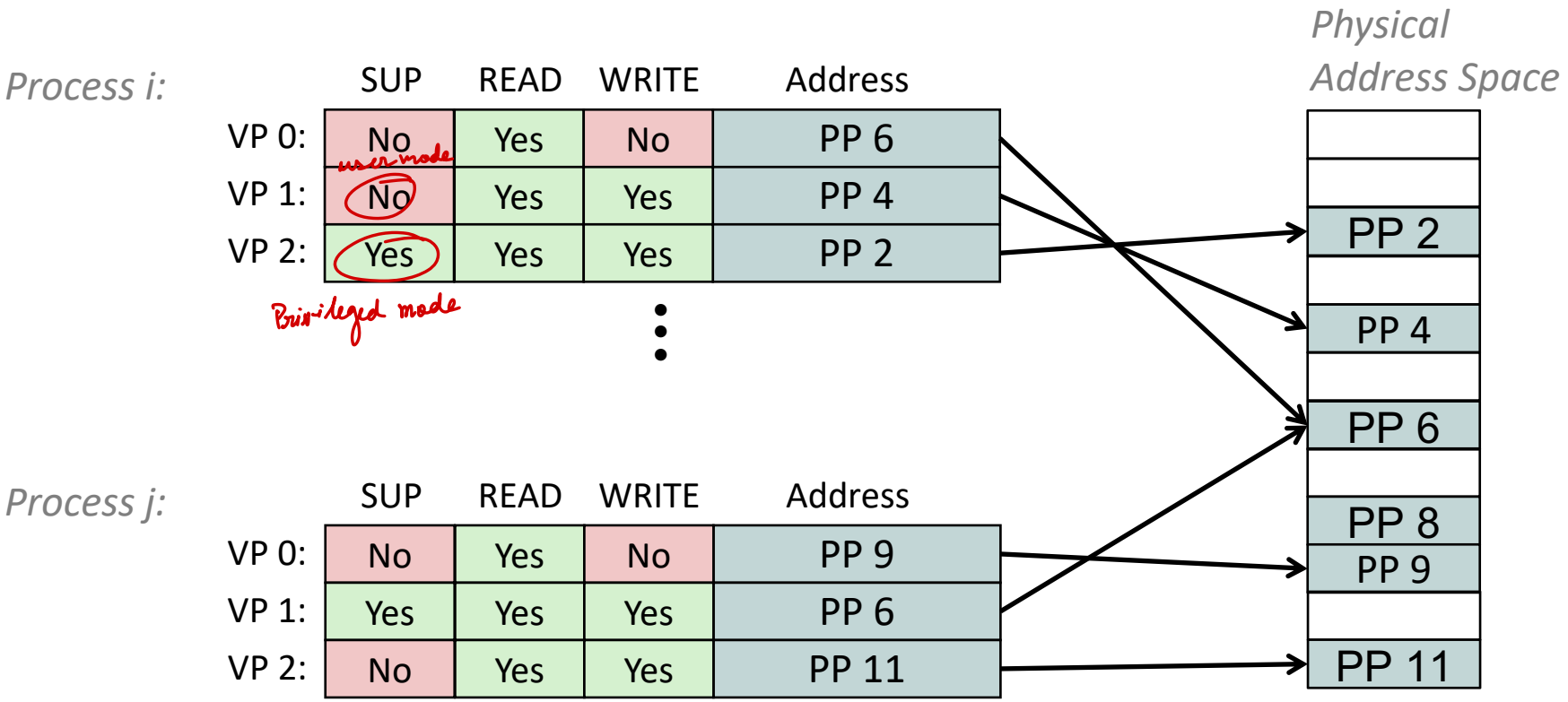
Simplifying Linking and Loading

- 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



VM as a Tool for Mem Protection

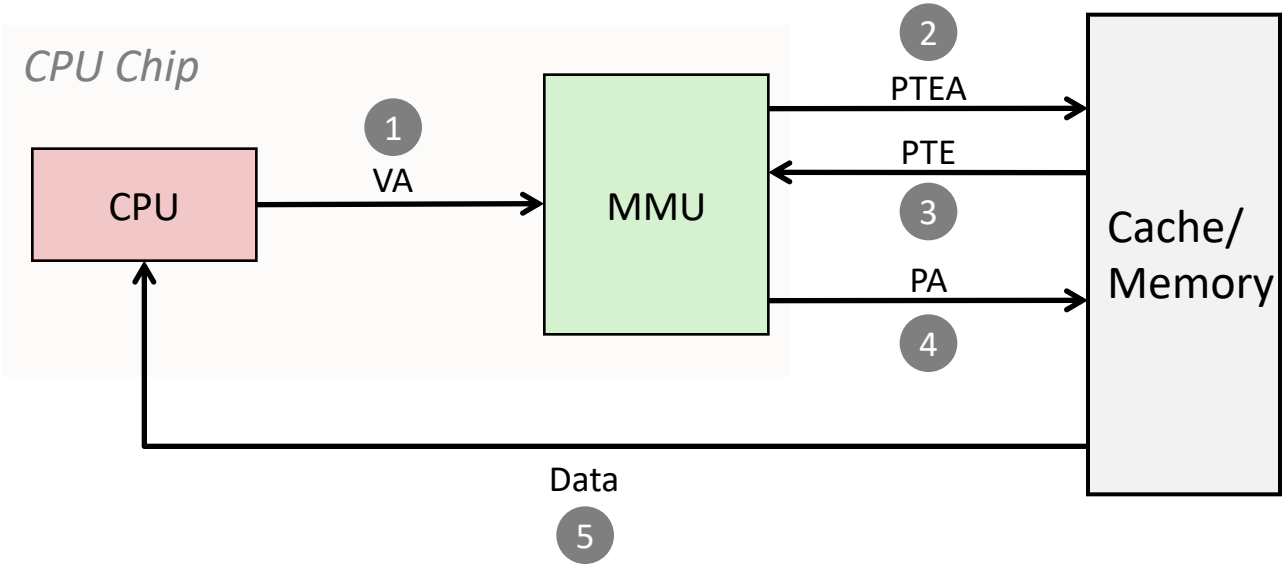
- Extend PTEs with permission bits
- Page fault handler checks these before remapping
  - If violated, send process SIGSEGV (segmentation fault)



* SUP = supervisor mode (kernel mode)

Address Translation: Page Hit

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

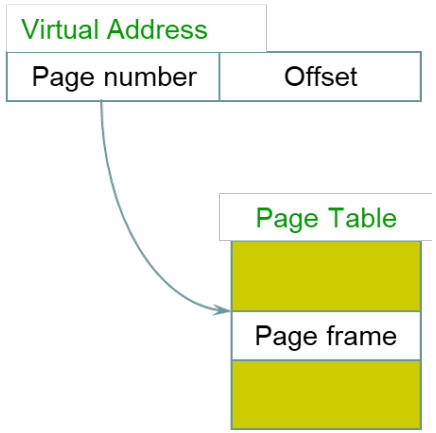


1) Processor sends virtual address to MMU

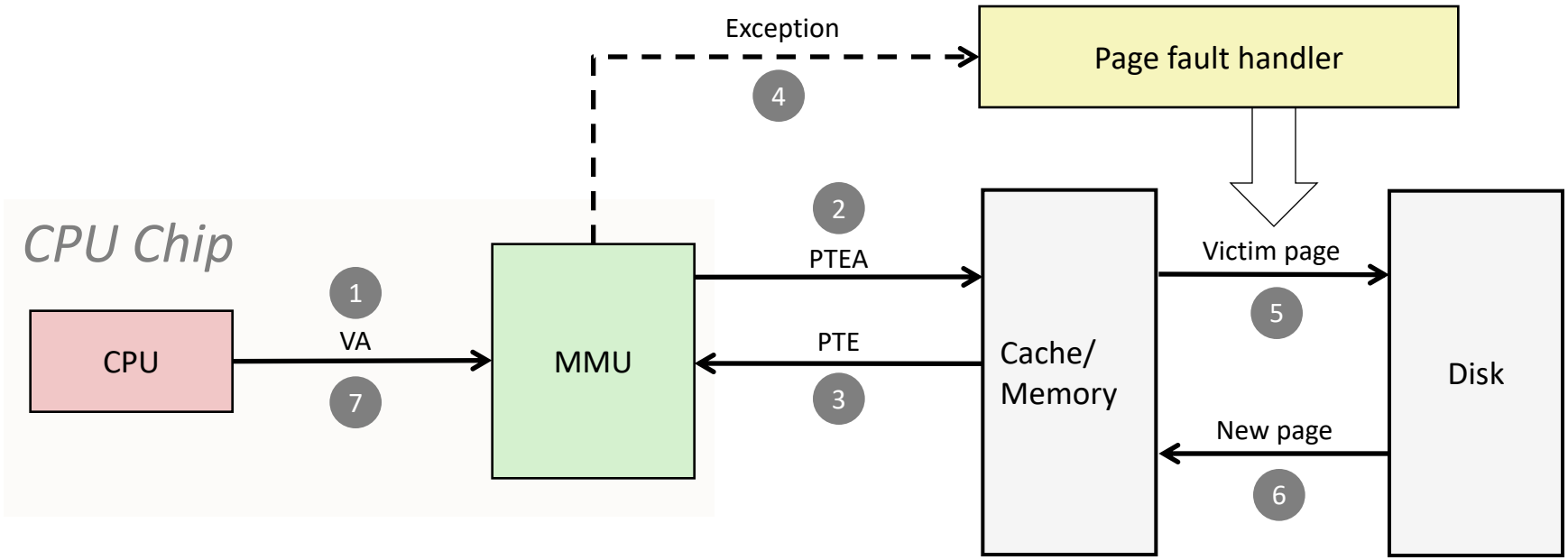
2-3) MMU fetches PTE from page table in memory

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)

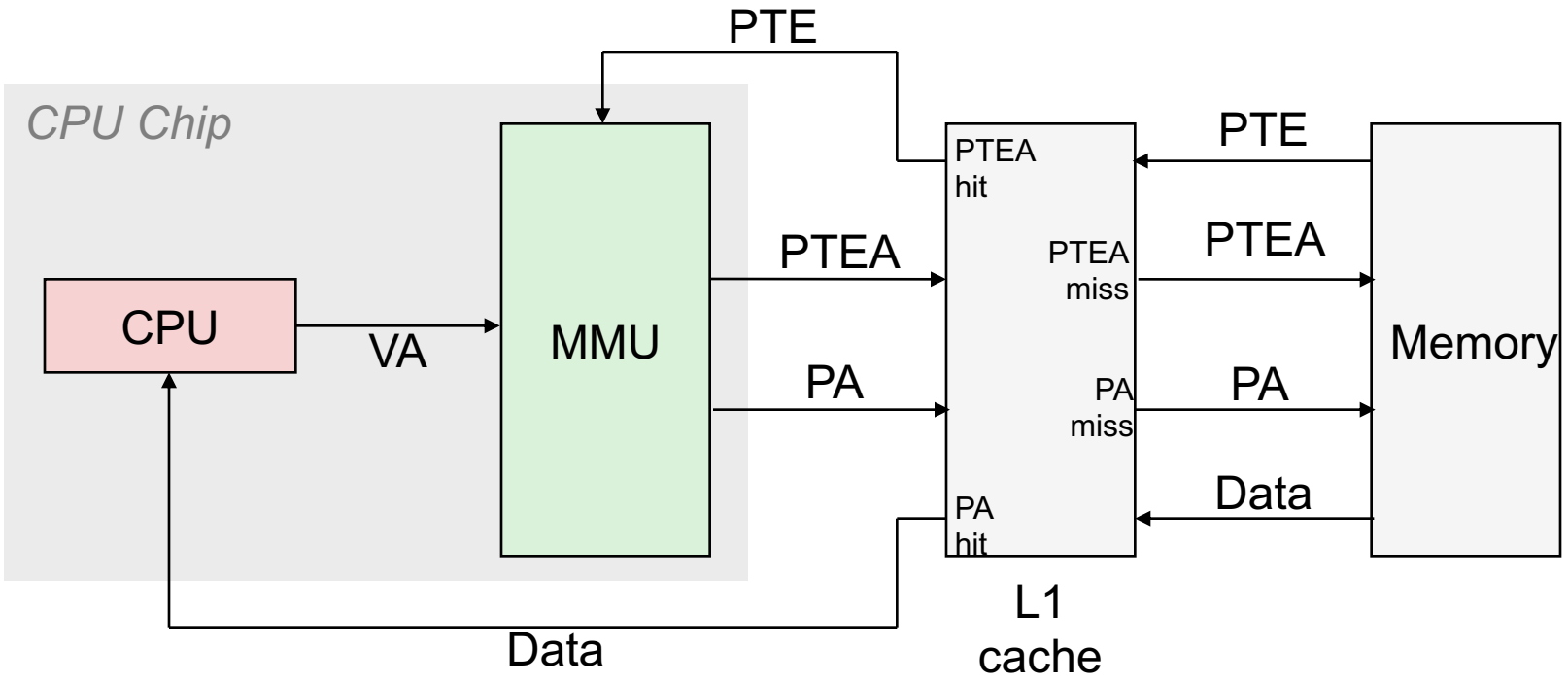
6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

Overhead due to a page fault

- PTE informs the page is on disk: 1 cycle
- CPU generates a page fault exception: 100 cycles
- OS page fault handler called
  - OS chooses a page to evict from DRAM and write to disk: 10k cycles
  - Write dirty page back to disk first: 40m cycles
  - OS then reads the page from disk and put it in DRAM: 40m cycles
  - OS changes the page table to map the new page: 1k cycles
- OS resumes the instruction that caused the page faul: 10k cycles
- Page faults are the **slowest** thing (except for human interactions)
- Interestingly, there are systems do not page:
  - iOS: kills the program if using too much memory

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
  - L1 cache is not effective
- **Problem 2: Page table can be gigantic!**
  - We need one for each process

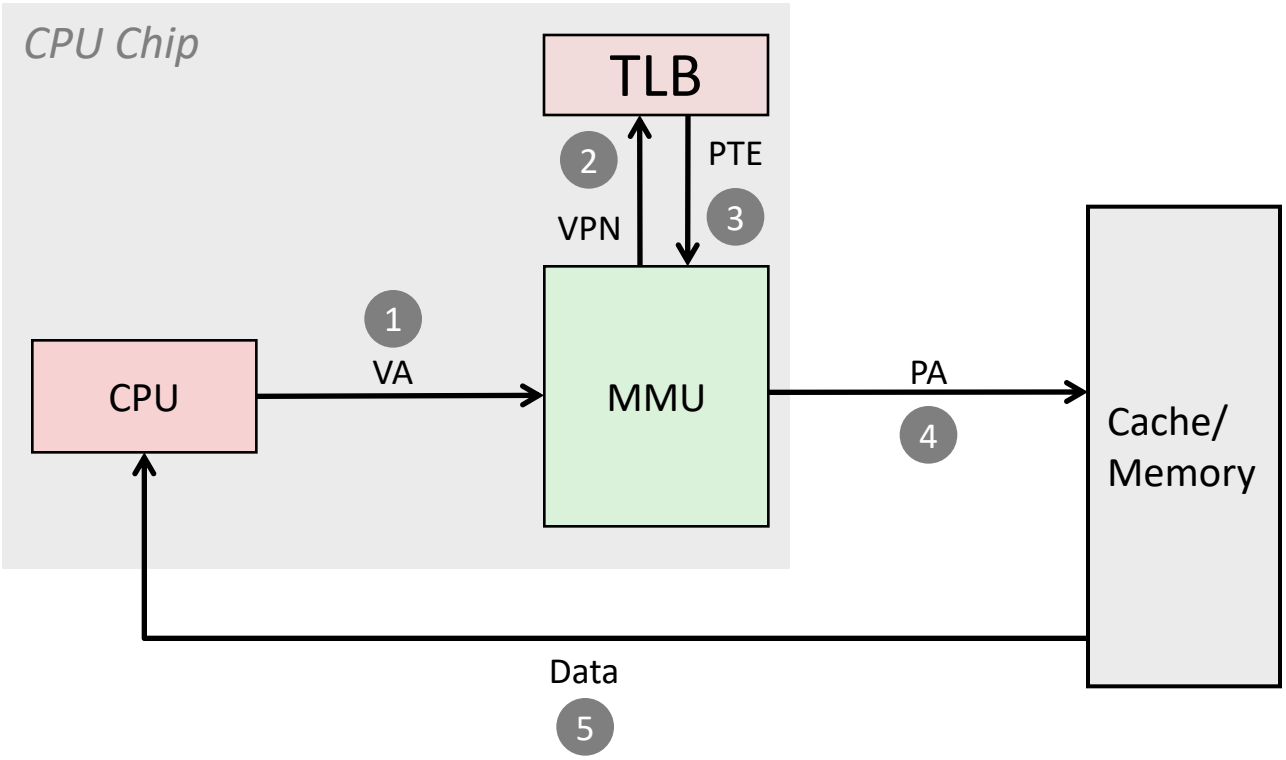
"Unfortunately, there's another elephant in the room."

Speeding up Translation with a TLB

- If page table entries (PTEs) are cached in L1 like any other data
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 access delay

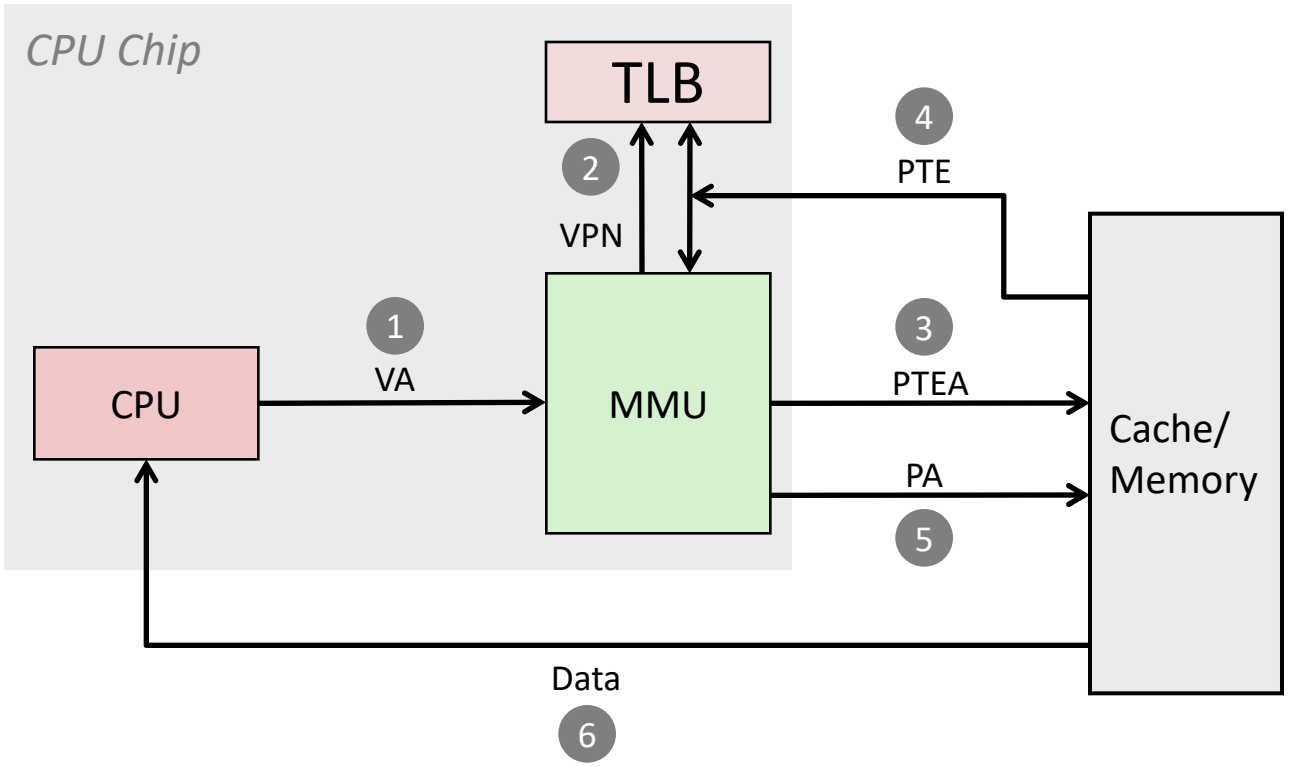
- Solution: *Translation Lookaside Buffer* (TLB)
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

TLB Hit



A TLB hit eliminates a memory access

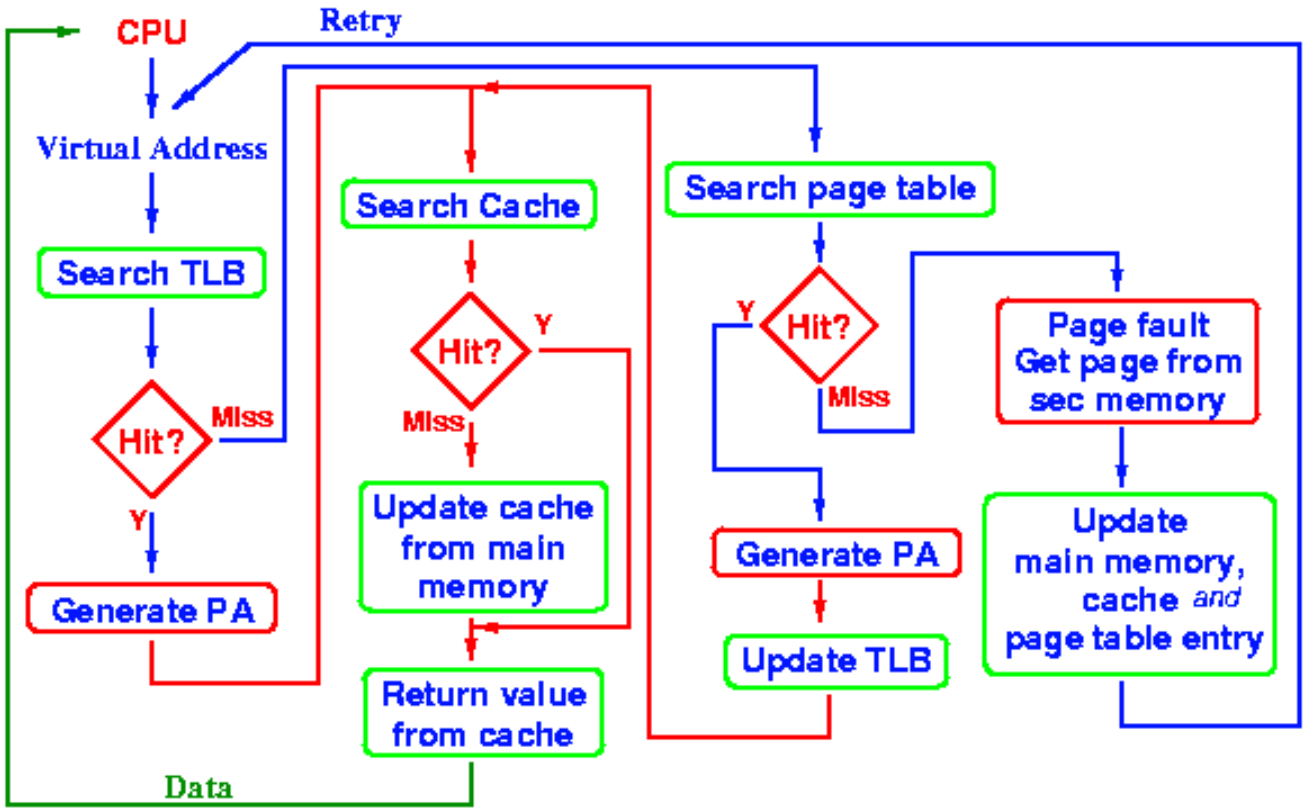
TLB Miss



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

Fortunately, TLB misses are rare. (Why?)

Integrating TLB and Cache



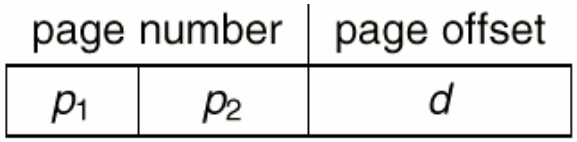
Multi-Level Paging

- Problem: Storage overhead of page tables
  - 4KB (2^{12}) page size
  - 48-bit address space
  - 8-byte PTE
  - Page table size = $2^{48} * 2^{-12} * 2^{3} = 2^{39}$ bytes
  - 512 GB page table per process!

- Solution: Multi-level paging
  - Breaks up virtual address space into multiple page tables in hierarchy
  - Maps a portion of the address space actually being used (tiny fraction of the entire address space)

Multi-Level Page Tables

- Example: 2-level paging

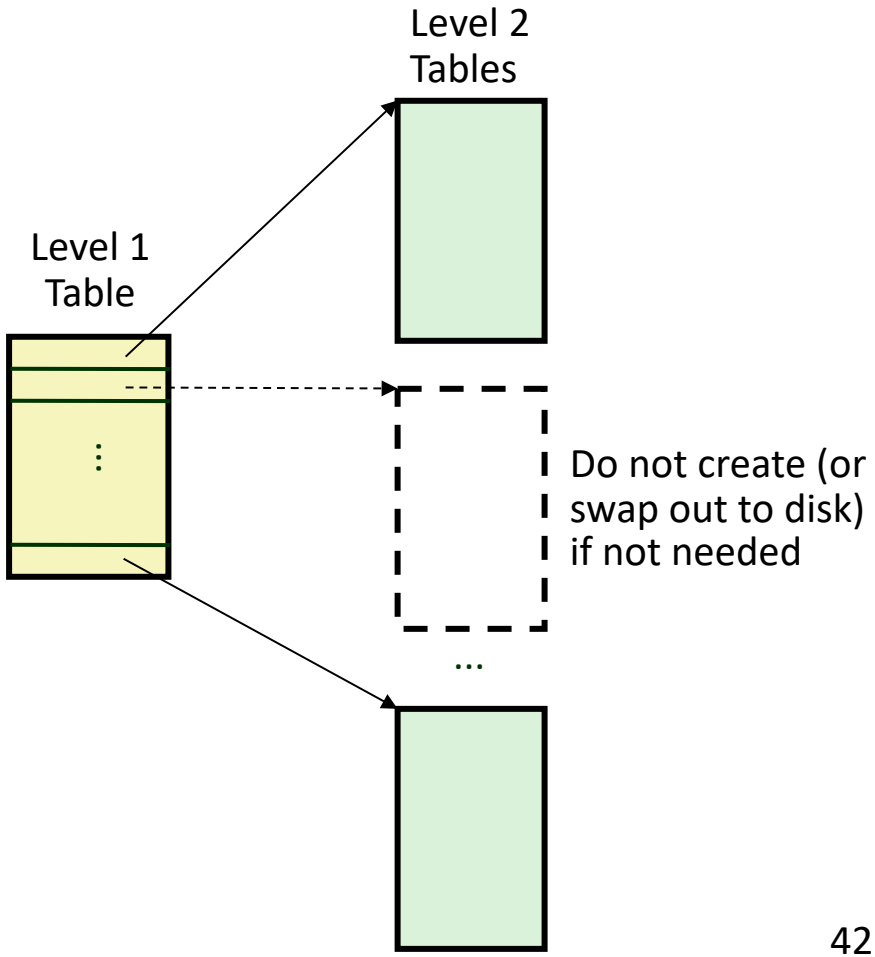


Level 1 table

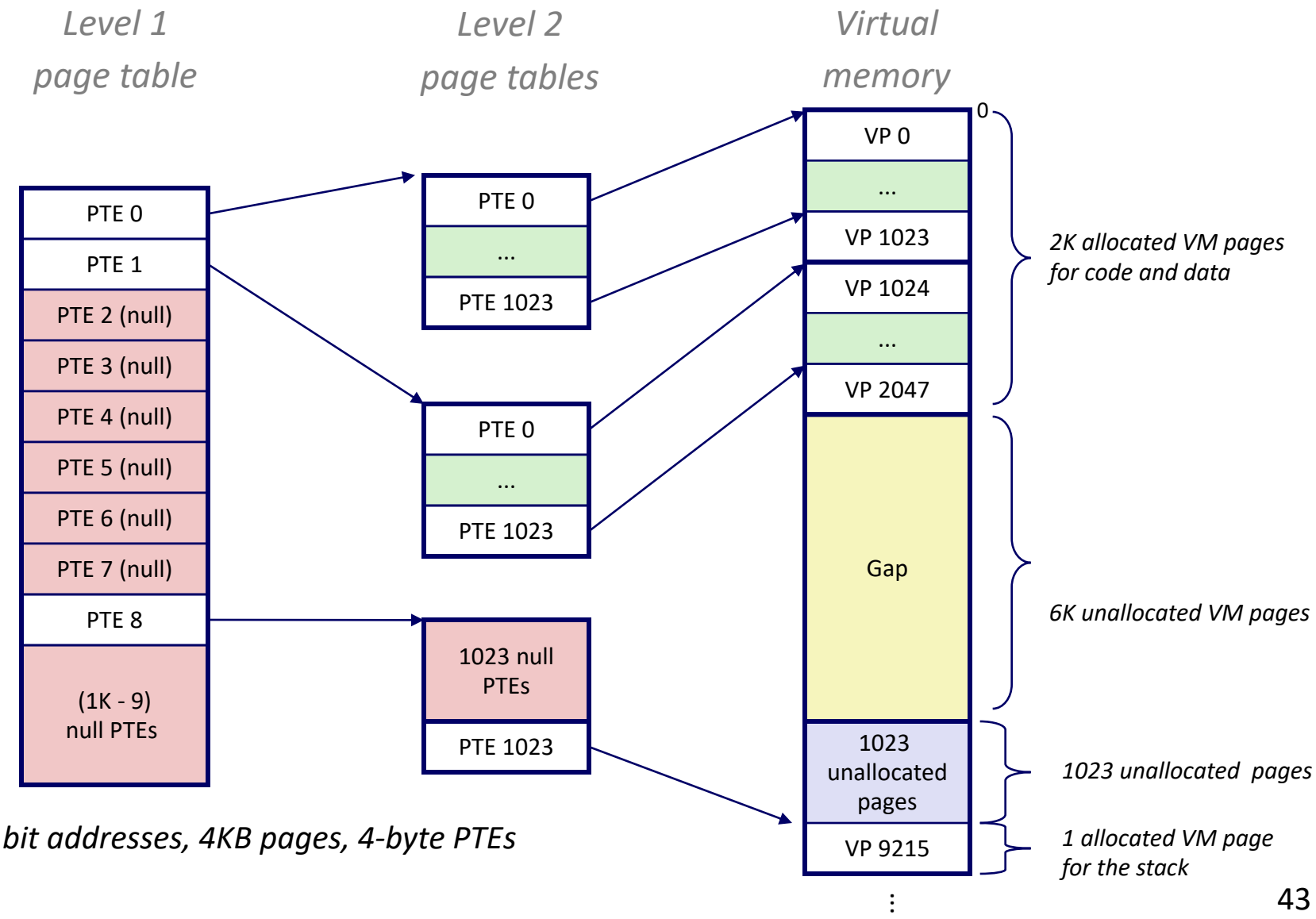
- Each PTE points to a L2 page table
- Always memory resident

Level 2 table

- Each PTE points to a page
- Paged in and out like any other data



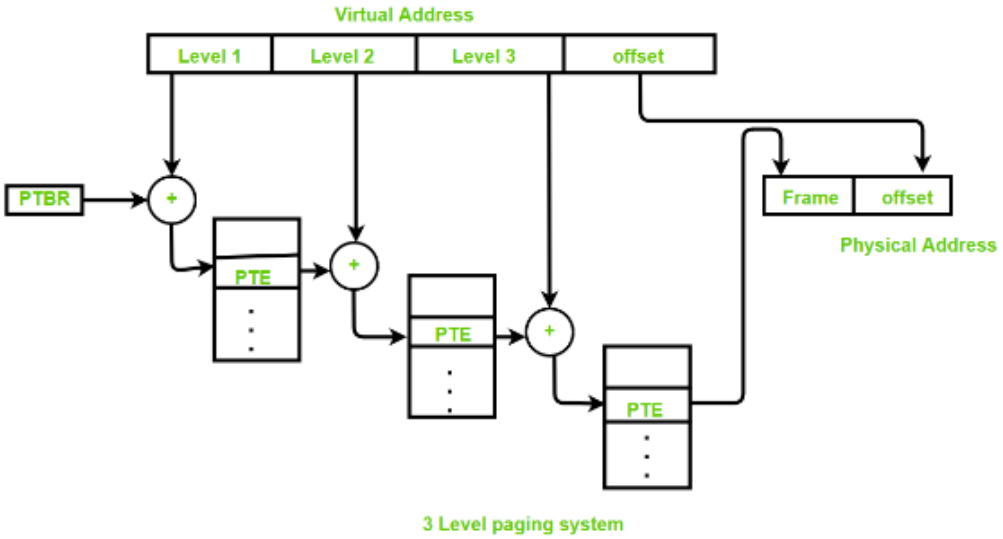
Two-Level Page Table Hierarchy



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

Multi-Level Paging

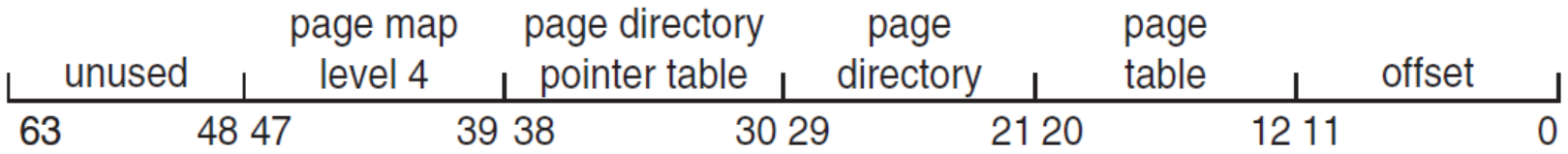
- Problem: Each (virtual) memory reference requires multiple memory references



- Trade off between spatial and temporal overhead
  - Memory access time, TLB hit ratio
  - Size of each page table

Intel x86-64 Paging

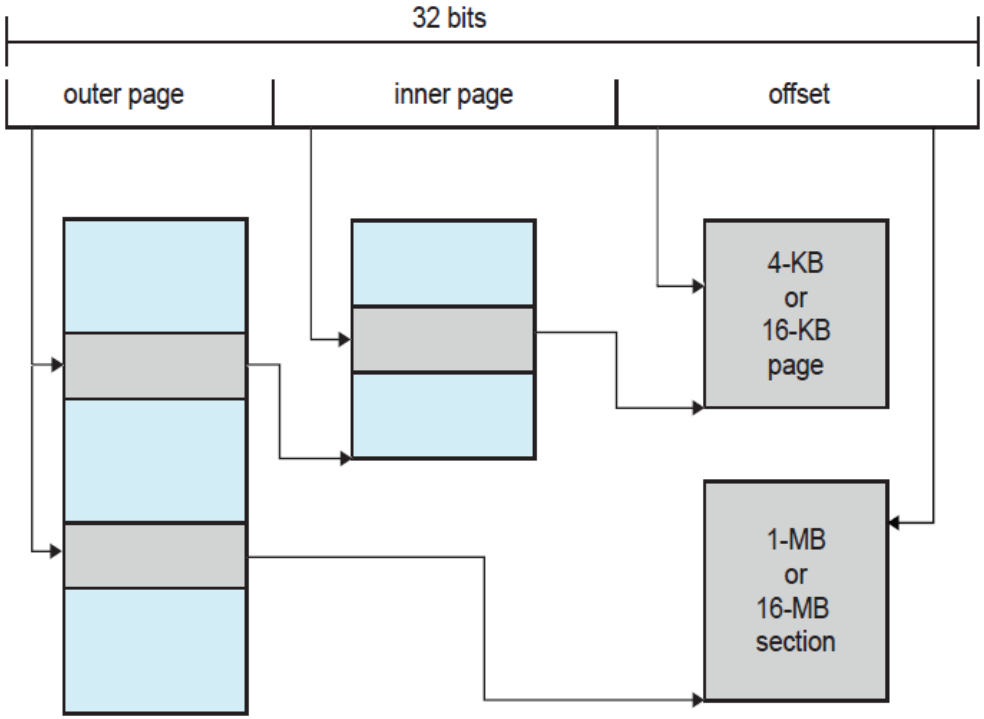
- Current generation Intel x86 CPUs
  - 64 bit architecture: maximum 2^{64} bytes address space
- In practice, only implement 48 bit addressing
  - Page sizes of 4 KB or 4 MB (optionally, 2 MB or 1 GB)
  - Four levels of paging hierarchy
    - Only use 48 bit addressing → 256 TiB of virtual address space



  - Recent processors have 5-level paging (e.g., Intel Ice Lake)
    - Use 57 bit addressing → 128 PiB of virtual memory space

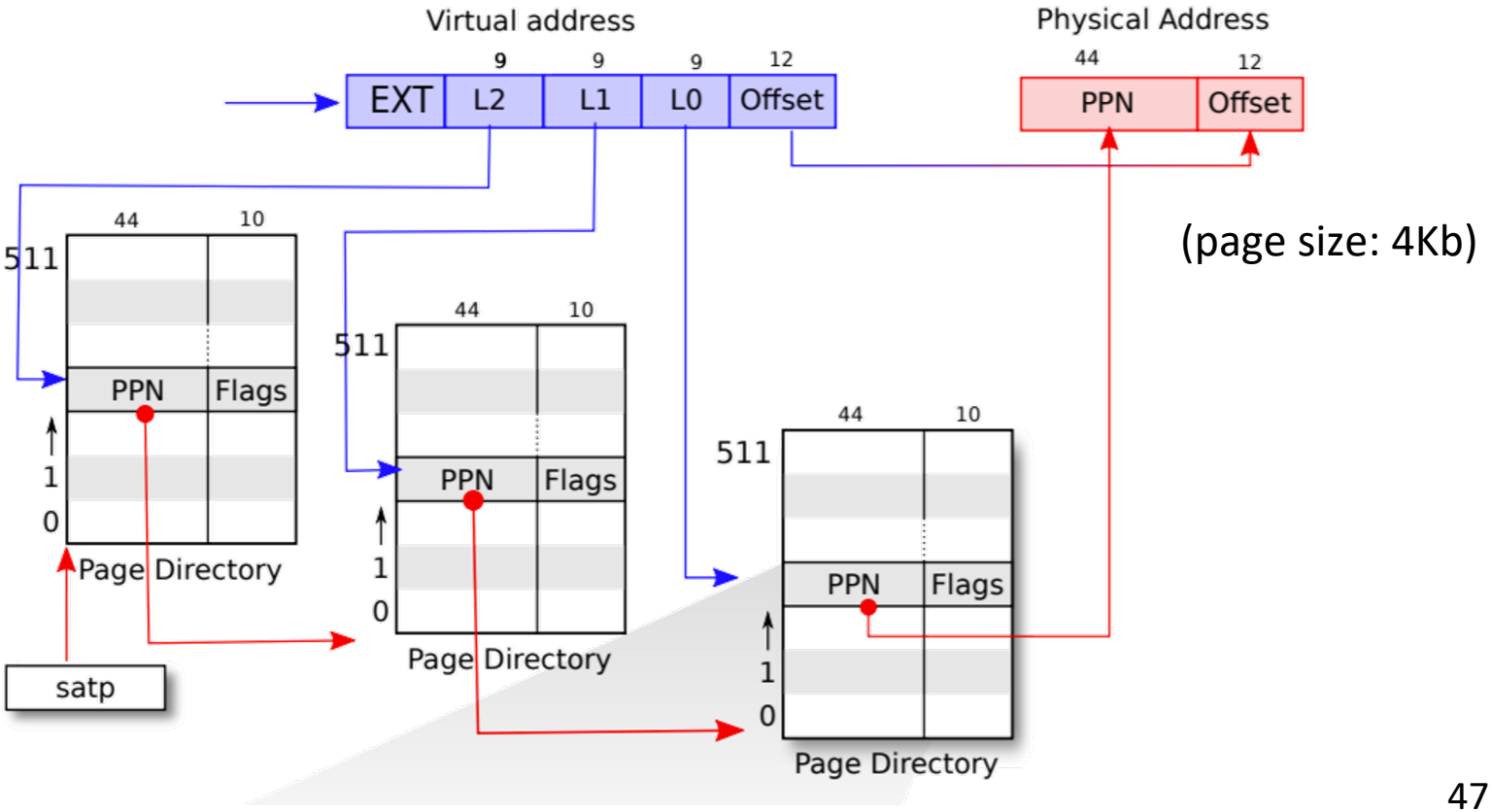
ARM Paging

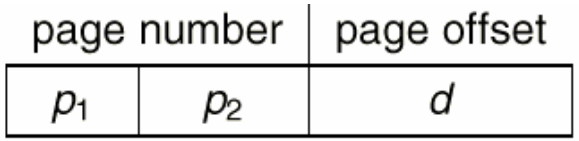
- 4 KB, 16 KB, 1 MB pages
- 32 bit architecture
  - Up to two-level paging
- 64 bit architecture
  - Up to three-level paging



RISC-V Paging (xv6 runs on Sv39 RISC-V)

- Sv39: Page-based 39-bit virtual addressing
  - Bottom 39 bits of a 64-bit virtual address; the top 25 bits are not used
- Three-level paging





- Consider two-level paging with a page size of 4 KB.
  - 32 bit address space
  - p1 is 10 bits
  - p2 is 10 bits
  - PTE size is 8 bytes

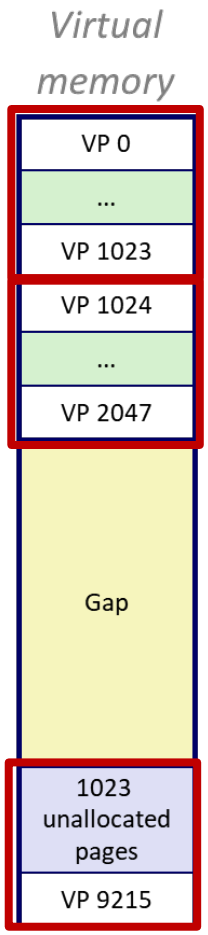
- How many PTEs in the Level-1 page table? 2^{10}

- How many Level-2 page tables can be created in the worst case? 2^{10}

How to make TLB larger without increasing size of it.

page number		page offset
p_1	p_2	d

- Consider two-level paging with a page size of 4 KB.
  - 32 bit address space
  - p1 is 10 bits
  - p2 is 10 bits
  - PTE size is 8 bytes



- Suppose there are 3 memory regions in a virtual address space, each region can be covered by one Level-2 table.
  How much space is required for all page tables?

4 x 8 KB