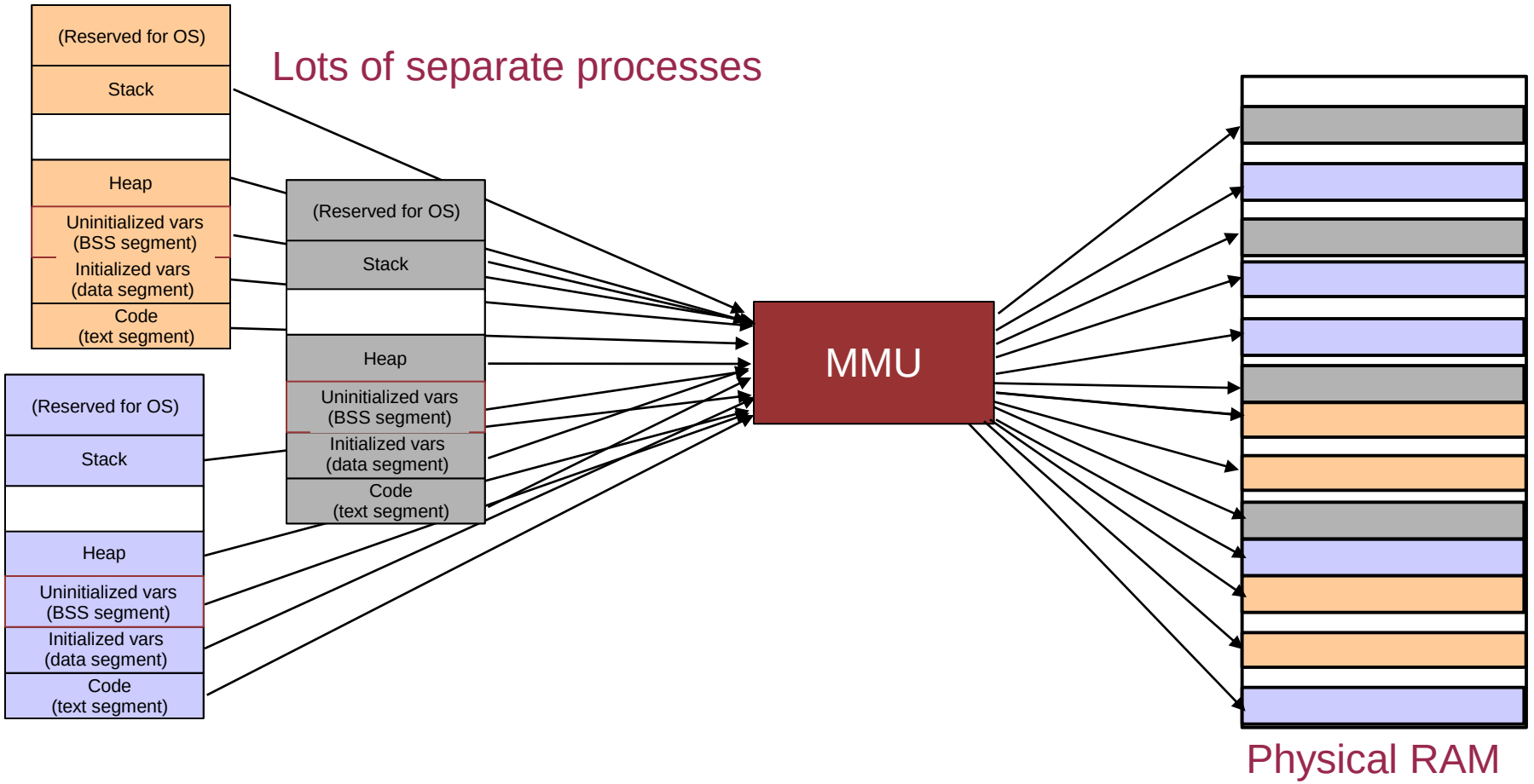
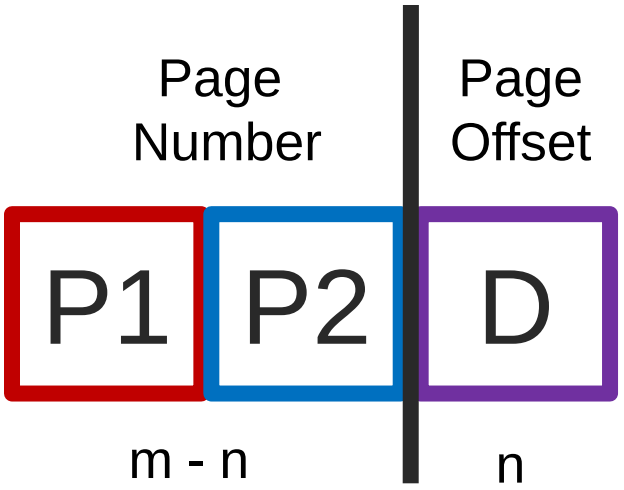
Memory

Application Perspective



Address Translation Scheme

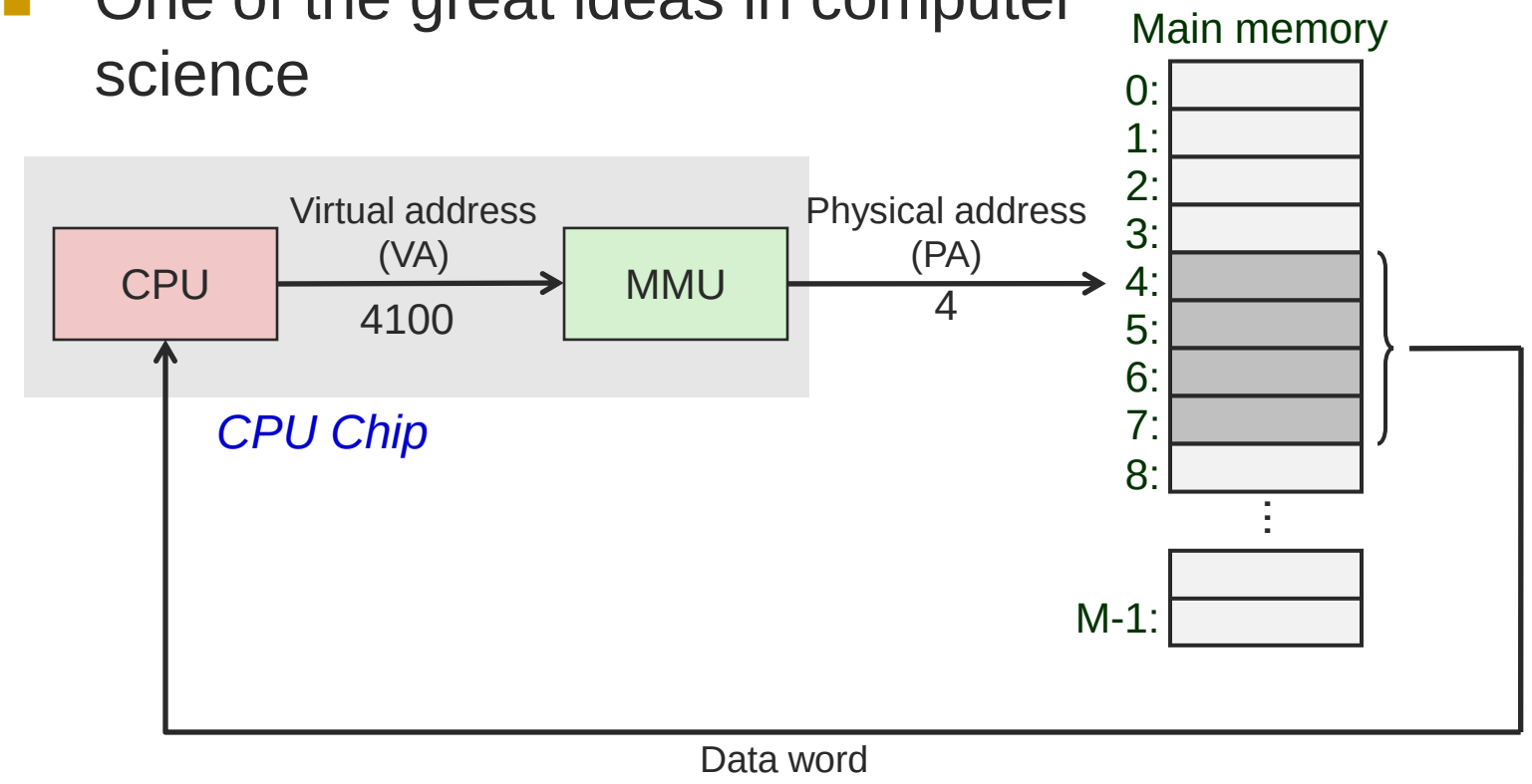
- Address generated by CPU is divided into
  - Page number (p)
    - An index into a page table
    - Contains base address of each page in physical memory
  - Page offset (d)
    - Combined with base address
    - Defines the physical memory address that is sent to the memory unit



For given logical address space 2m and page size 2n

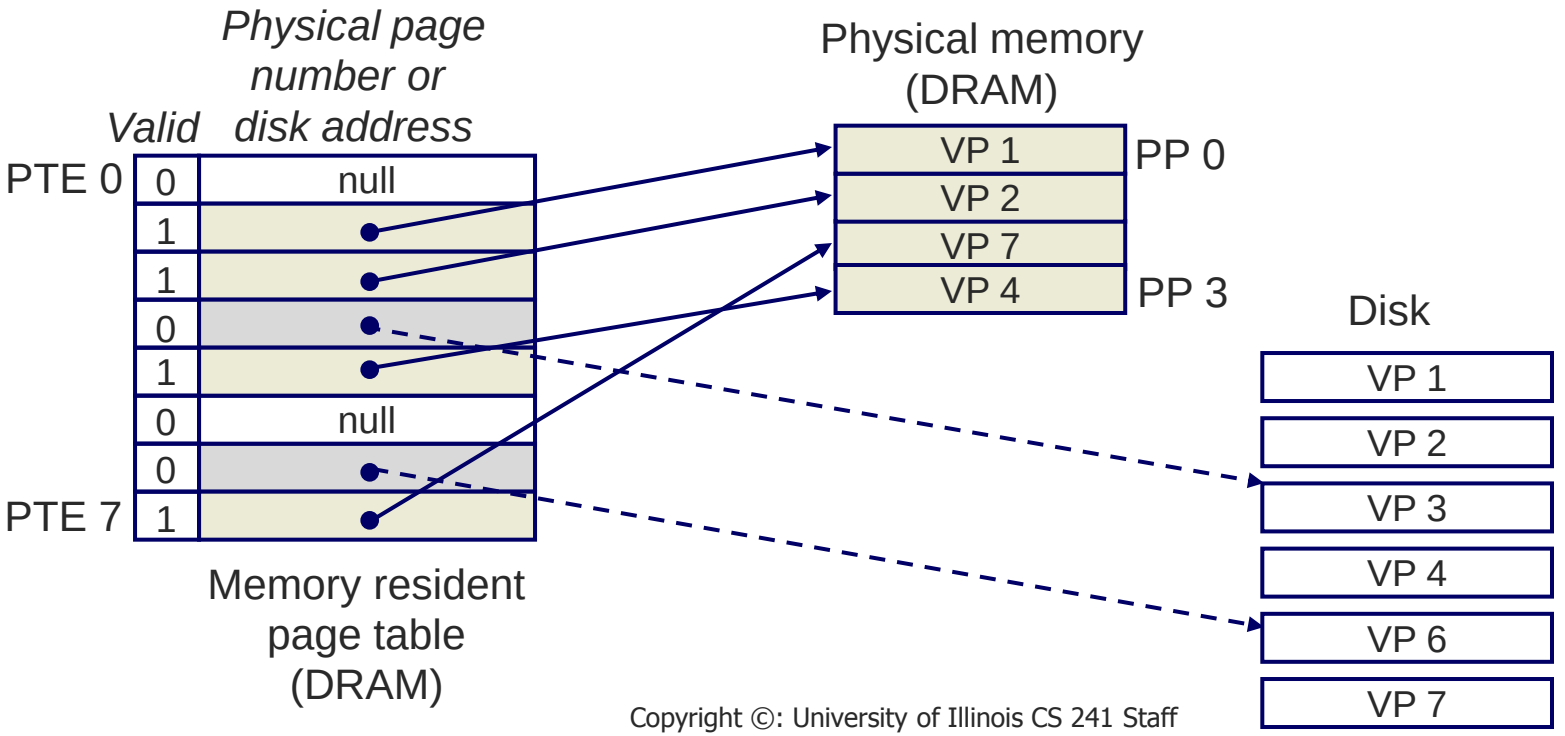
Virtual addressing

- Used in all modern servers, desktops, and laptops
- One of the great ideas in computer science



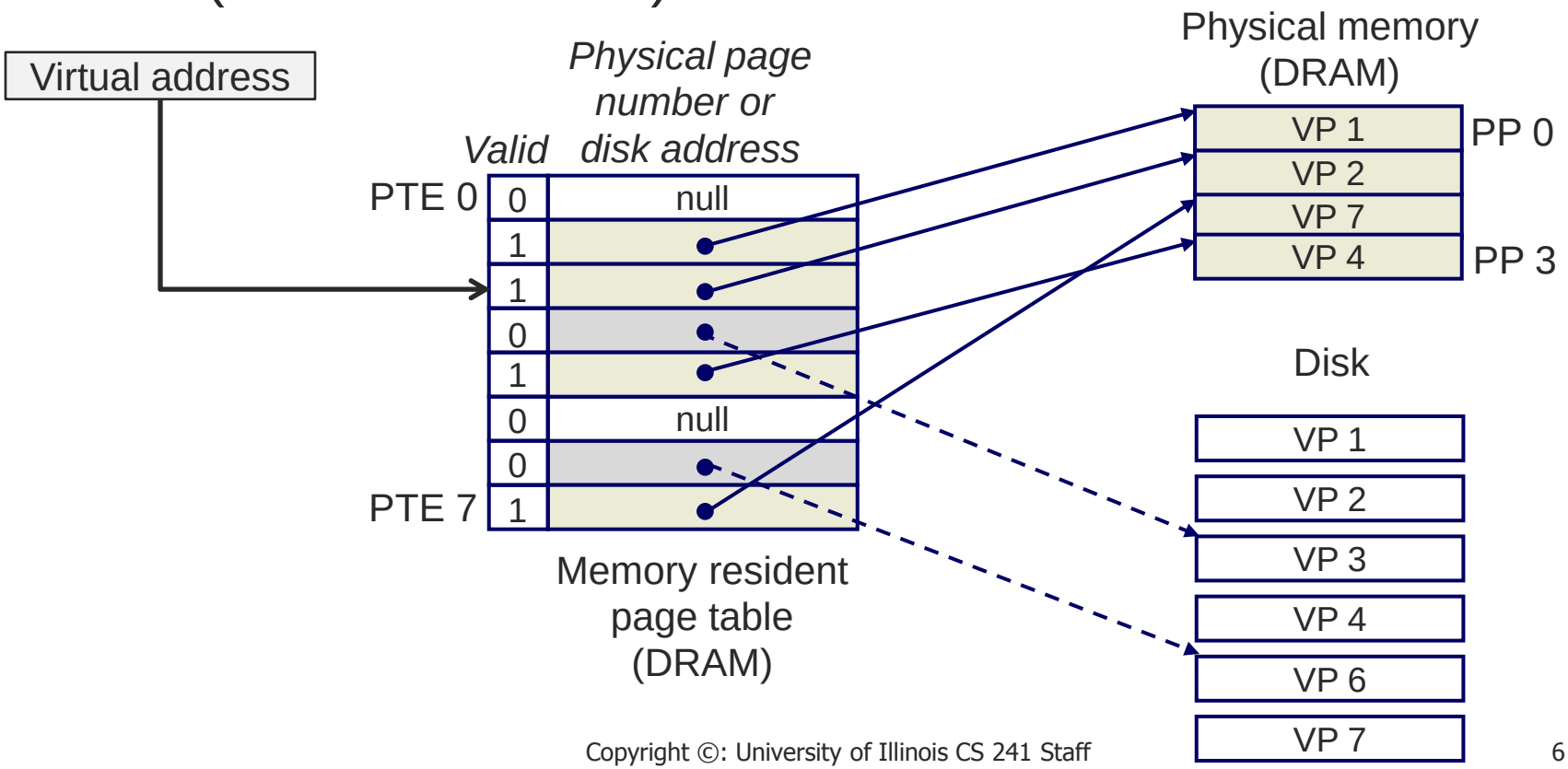
Data Structure: Page Table

- 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



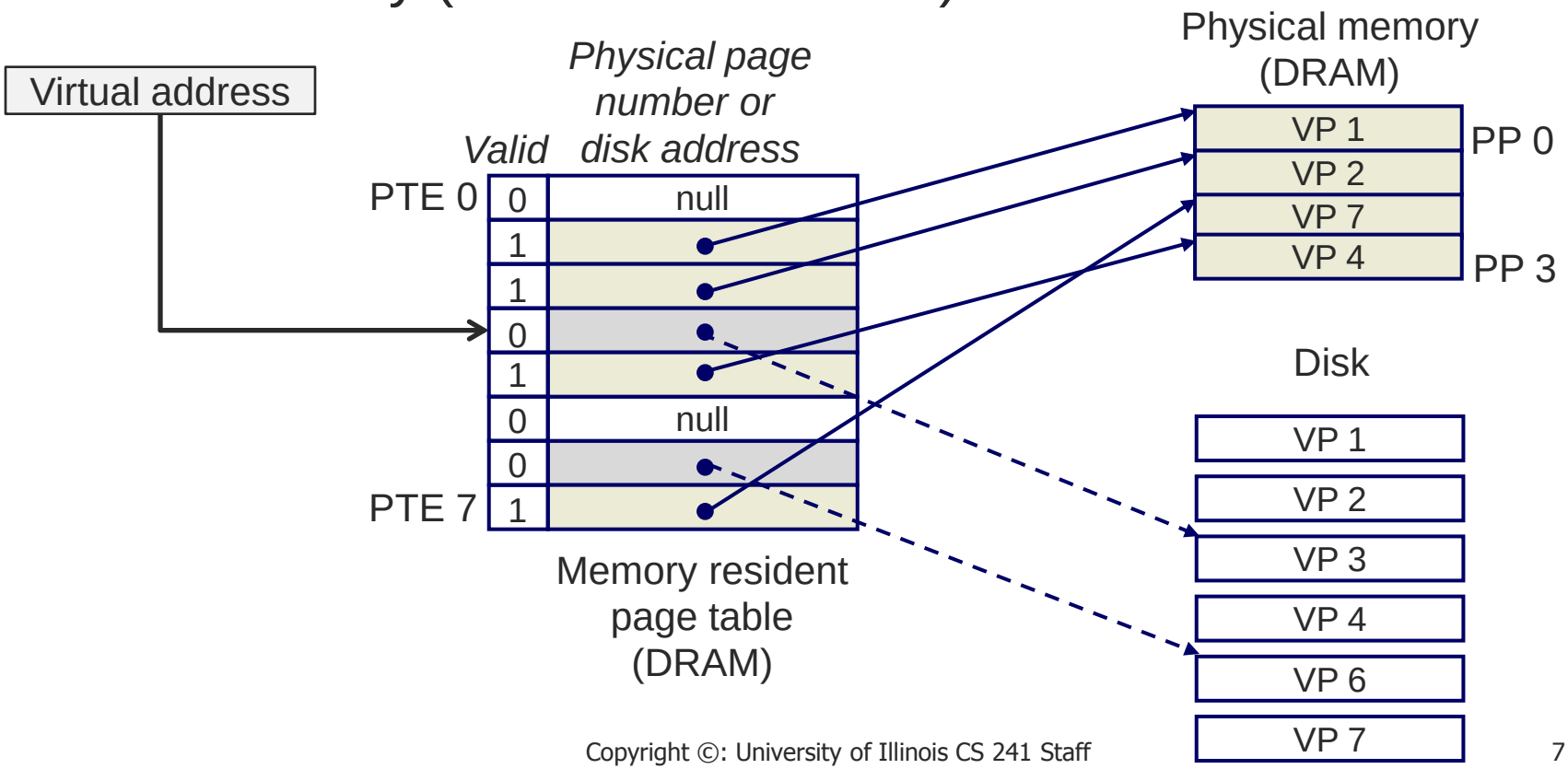
Page Hit

- Reference to VM word that is in physical memory (DRAM cache hit)



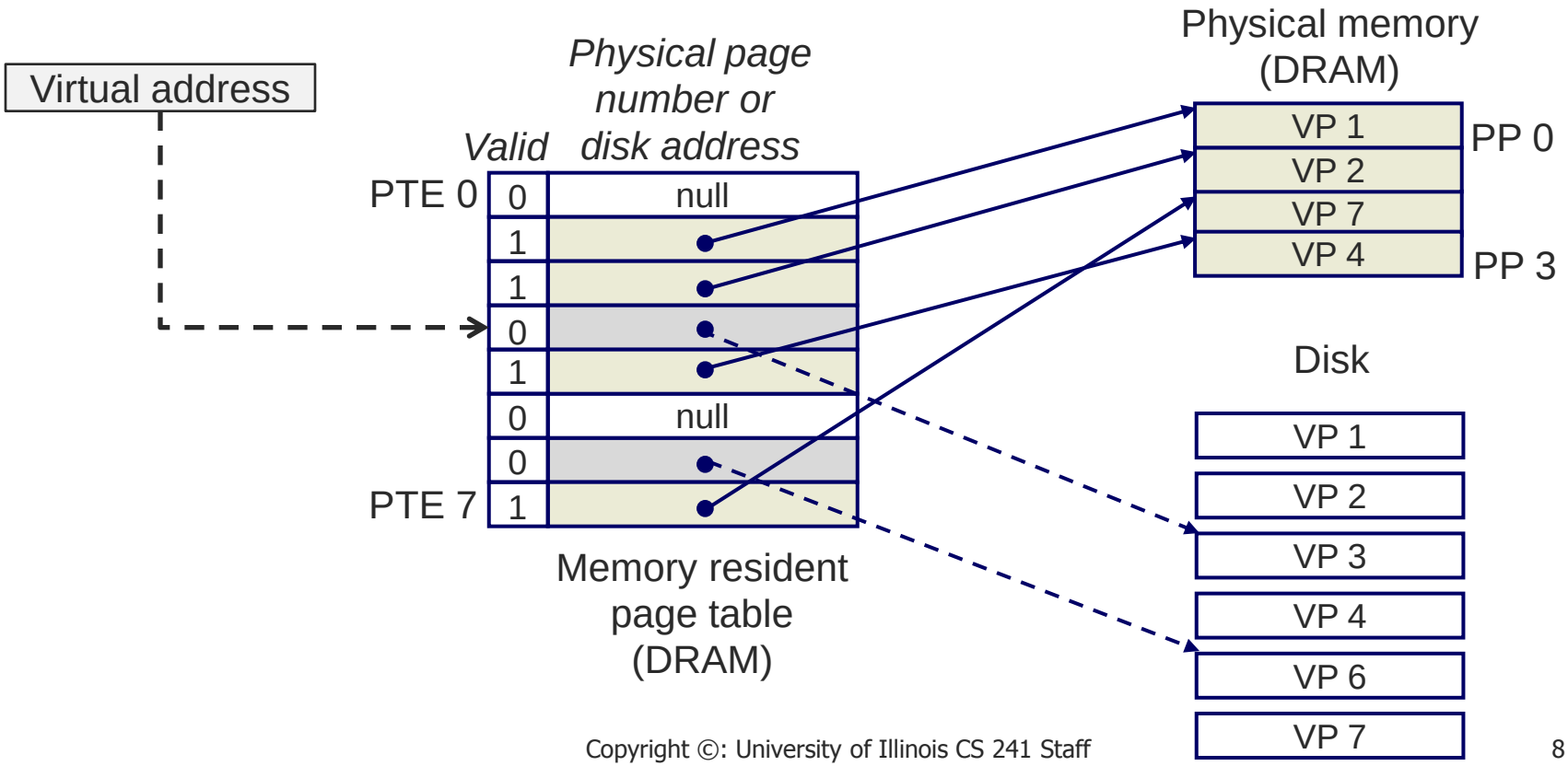
Page Fault

- Reference to VM word that is not in physical memory (DRAM cache miss)



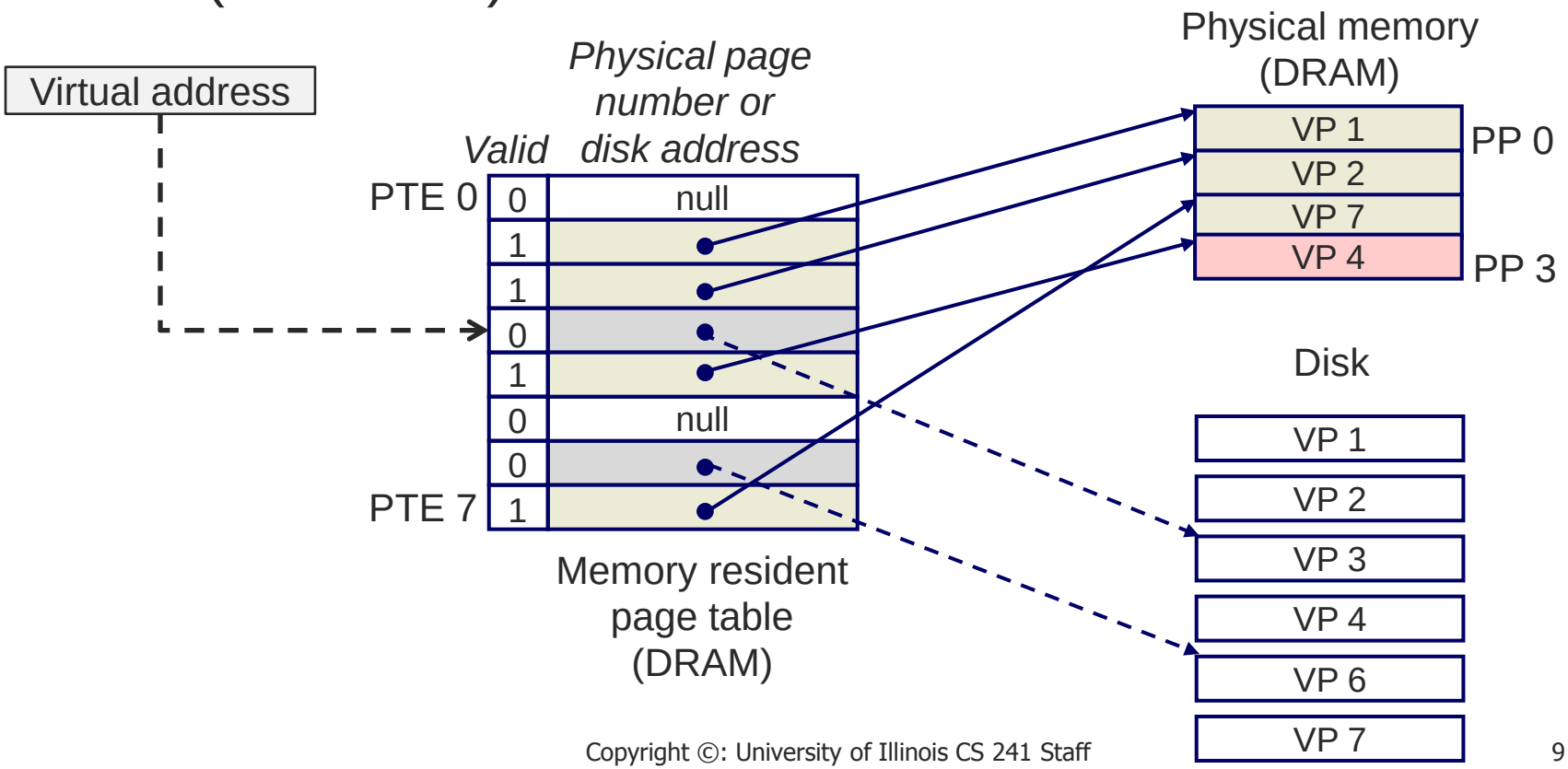
Handling Page Faults

- Page miss causes page fault (an exception)



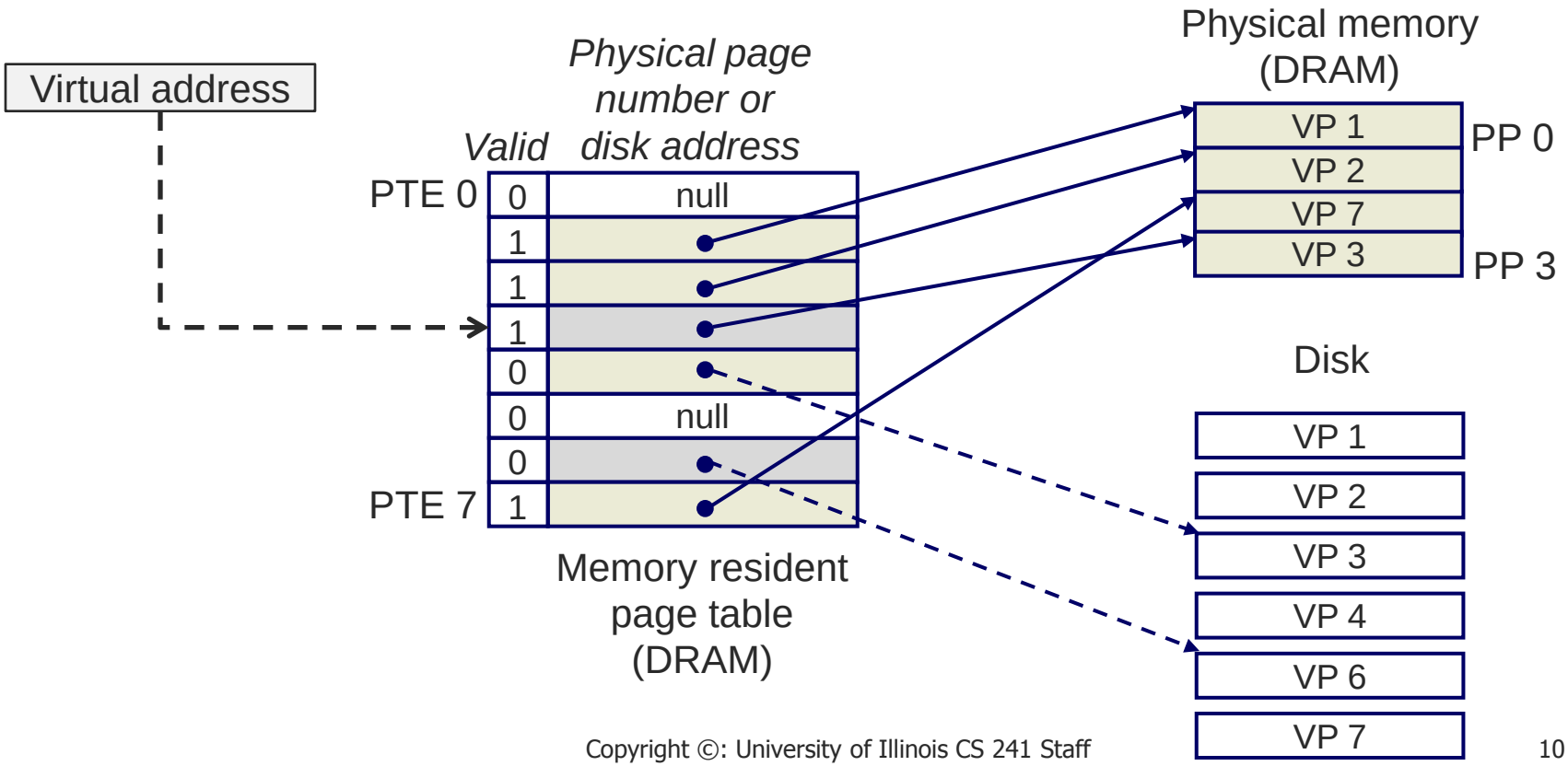
Handling Page Faults

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



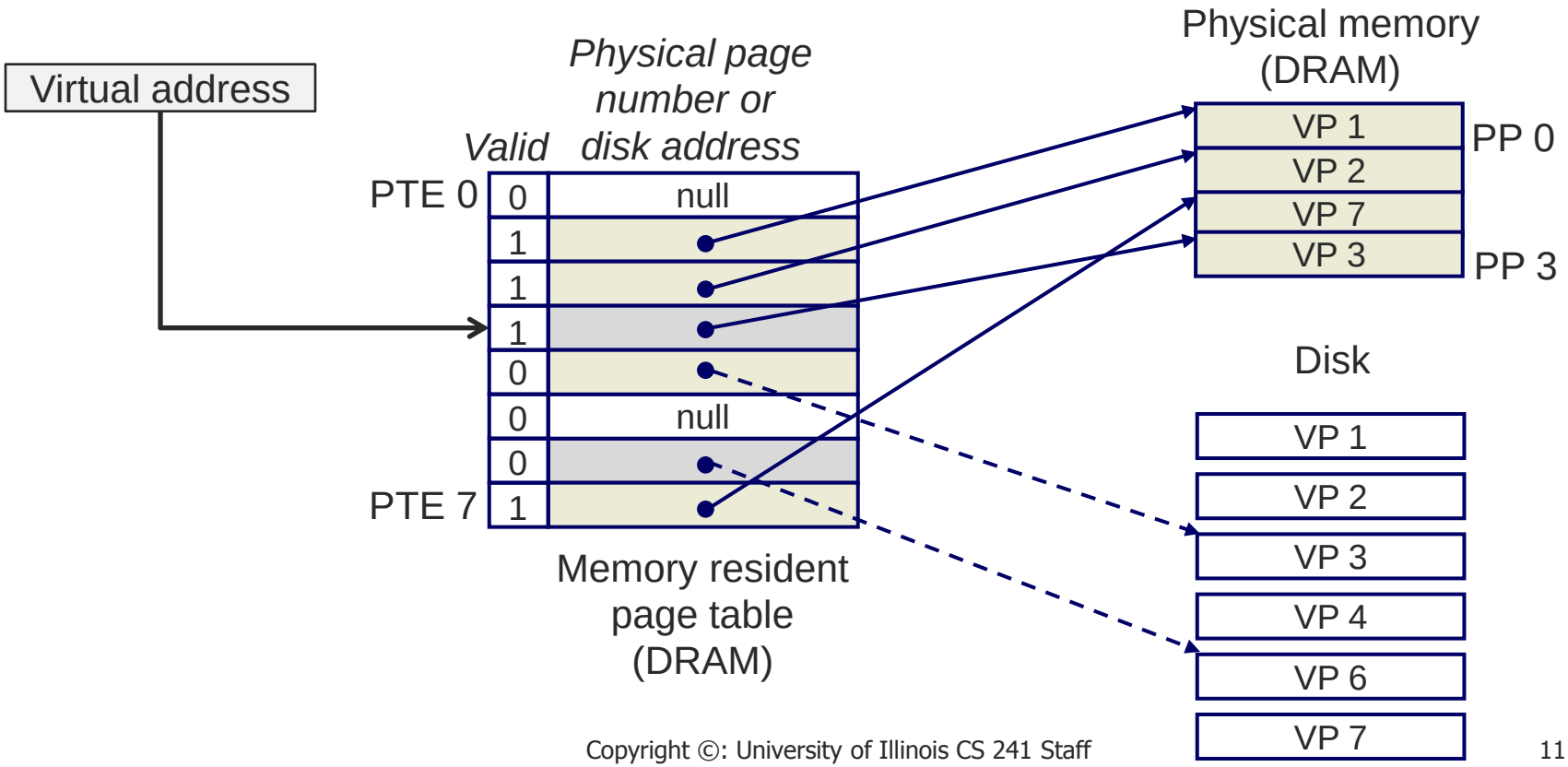
Handling Page Faults

- Loads new frame into freed slot



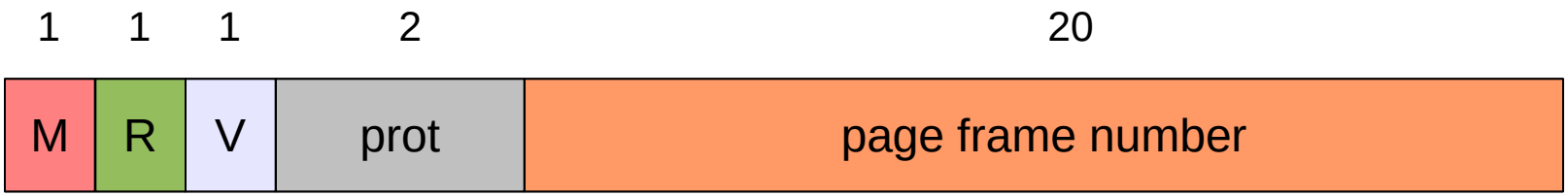
Handling Page Faults

- Offending instruction is restarted: page hit!



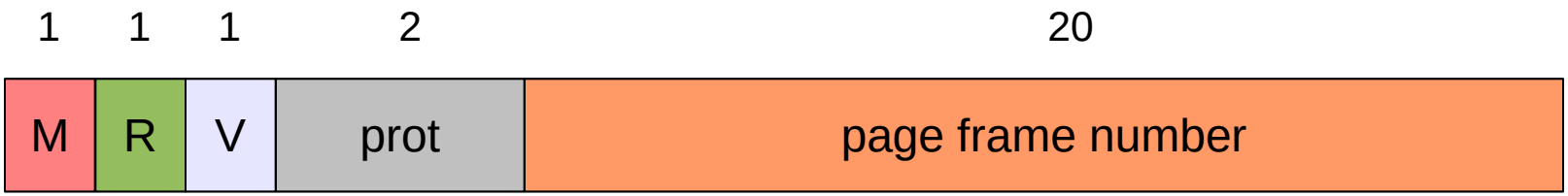
Page Table Entry

- Why is the page number 20 bits wide?



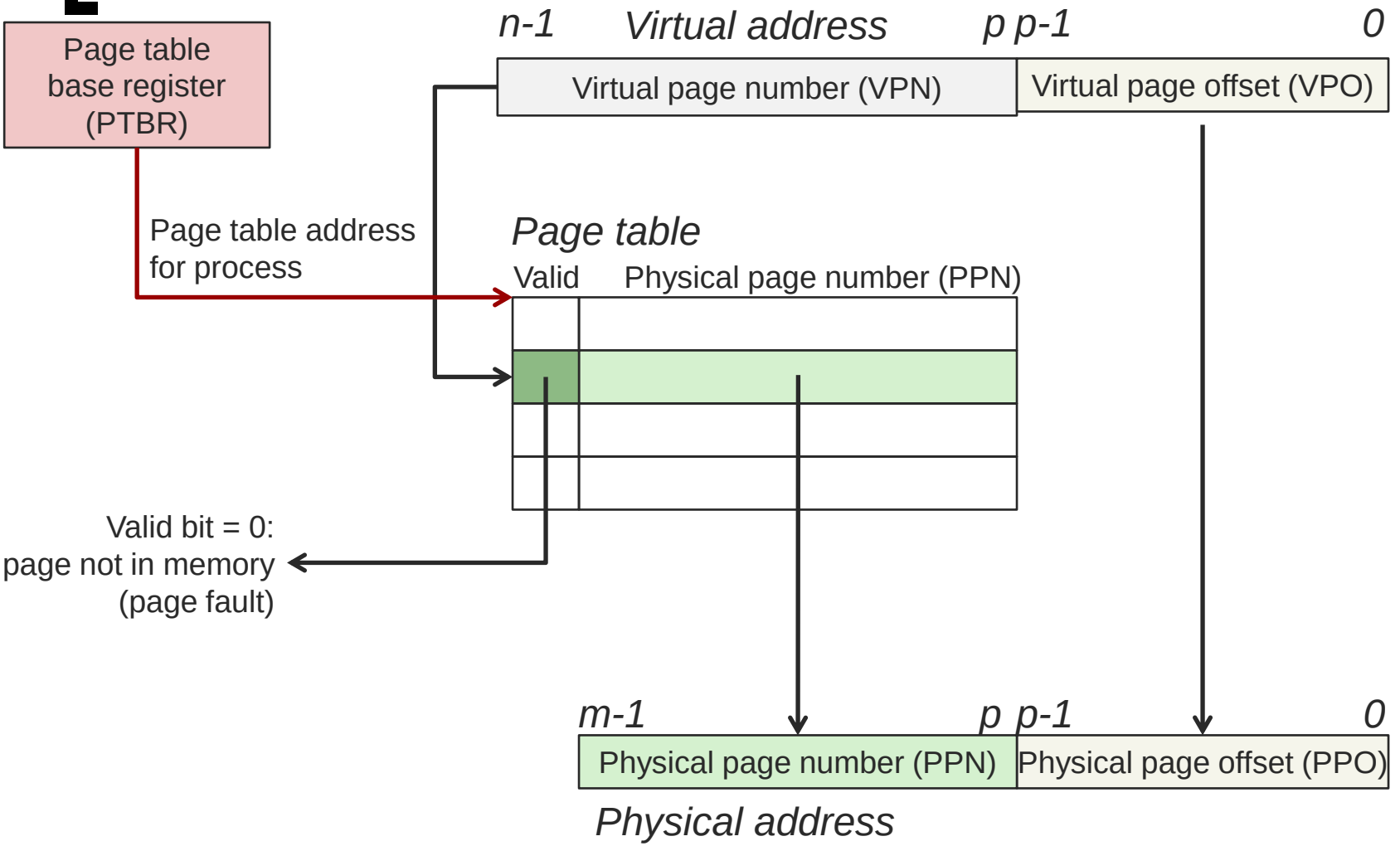
Page Table Entry

- Why is the page number 20 bits wide?

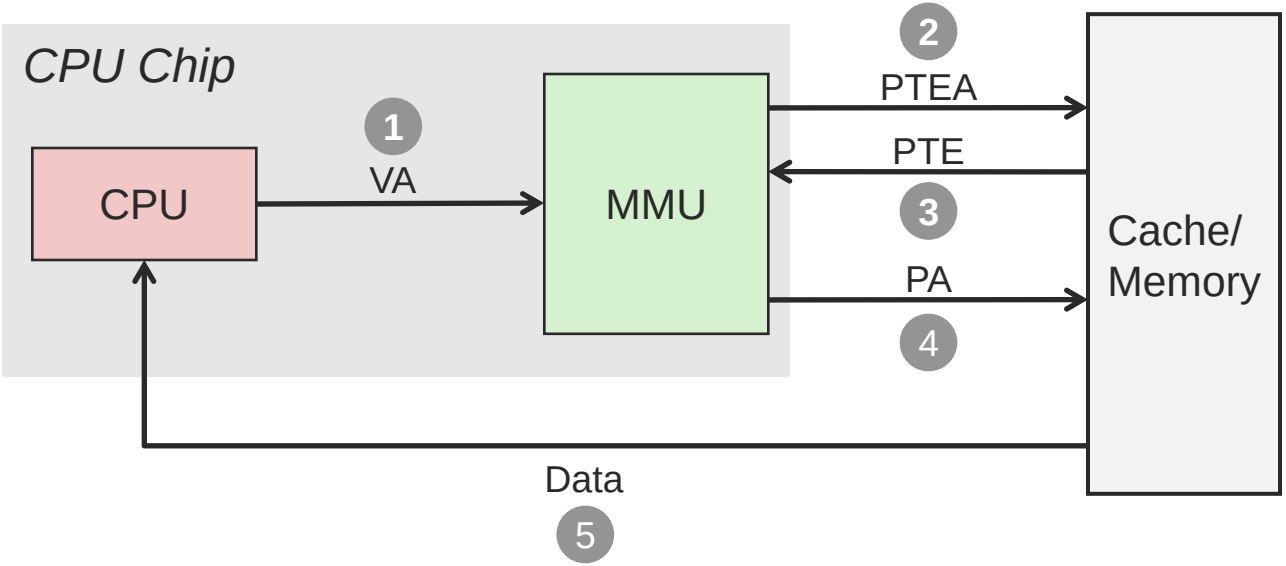


- Because that's how many bits you use to refer to pages!,
  - Each page is 4KB
  - Need 12 bits to refer to offset within page

Address Translation with a Page Table



Address Translation: Page Hit



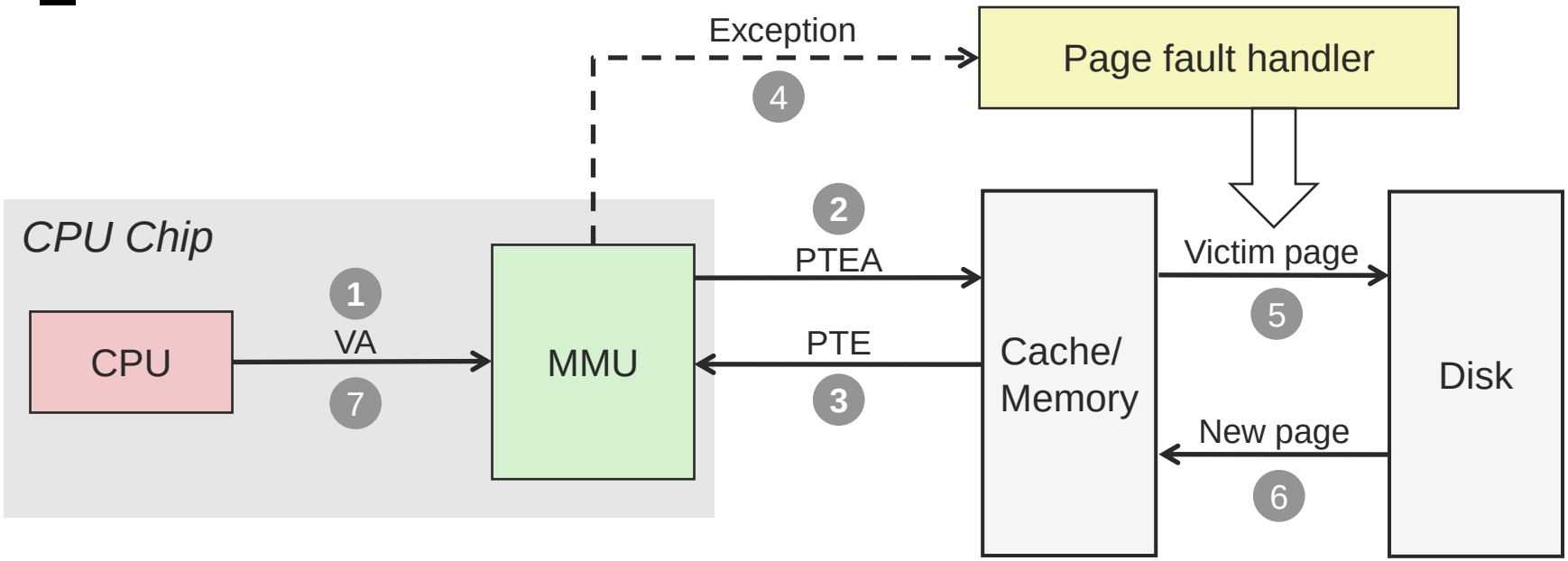
1) Processor sends virtual address to MMU

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



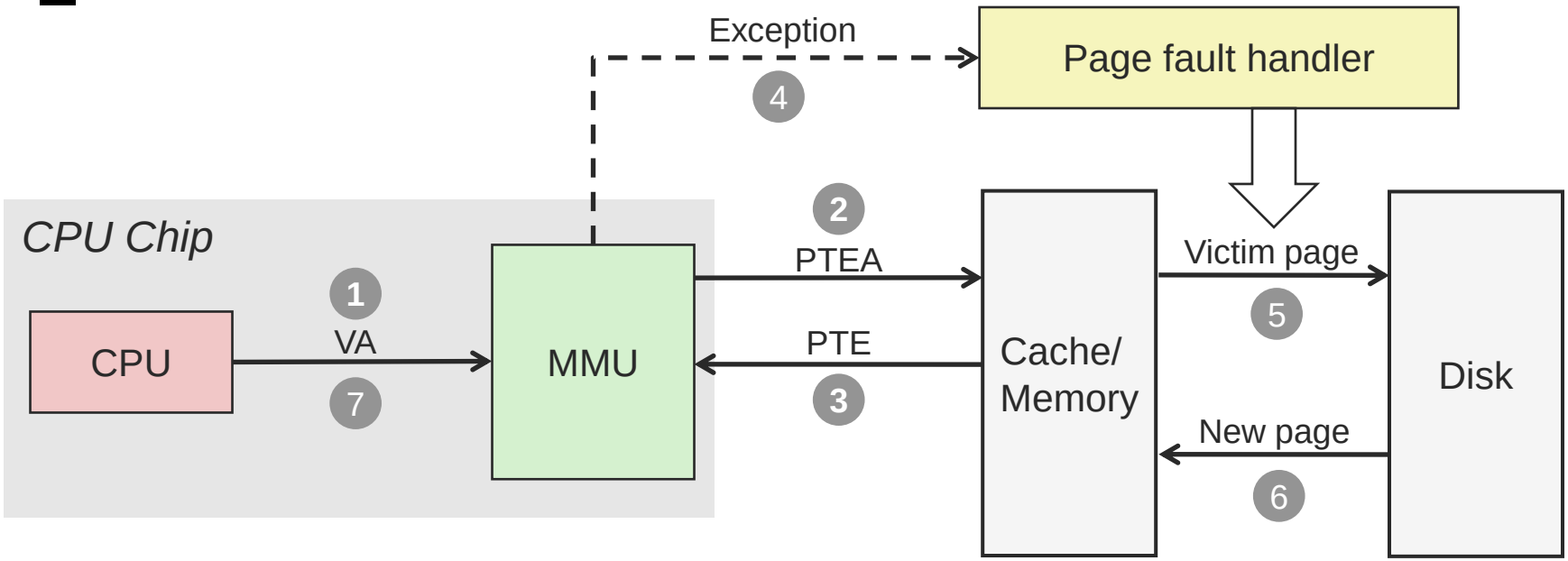
1) Processor sends virtual address to MMU

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)

Address Translation: Page Fault



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

7) Handler returns to original process, restarting faulting instruction

Question 1: Memory Acceses

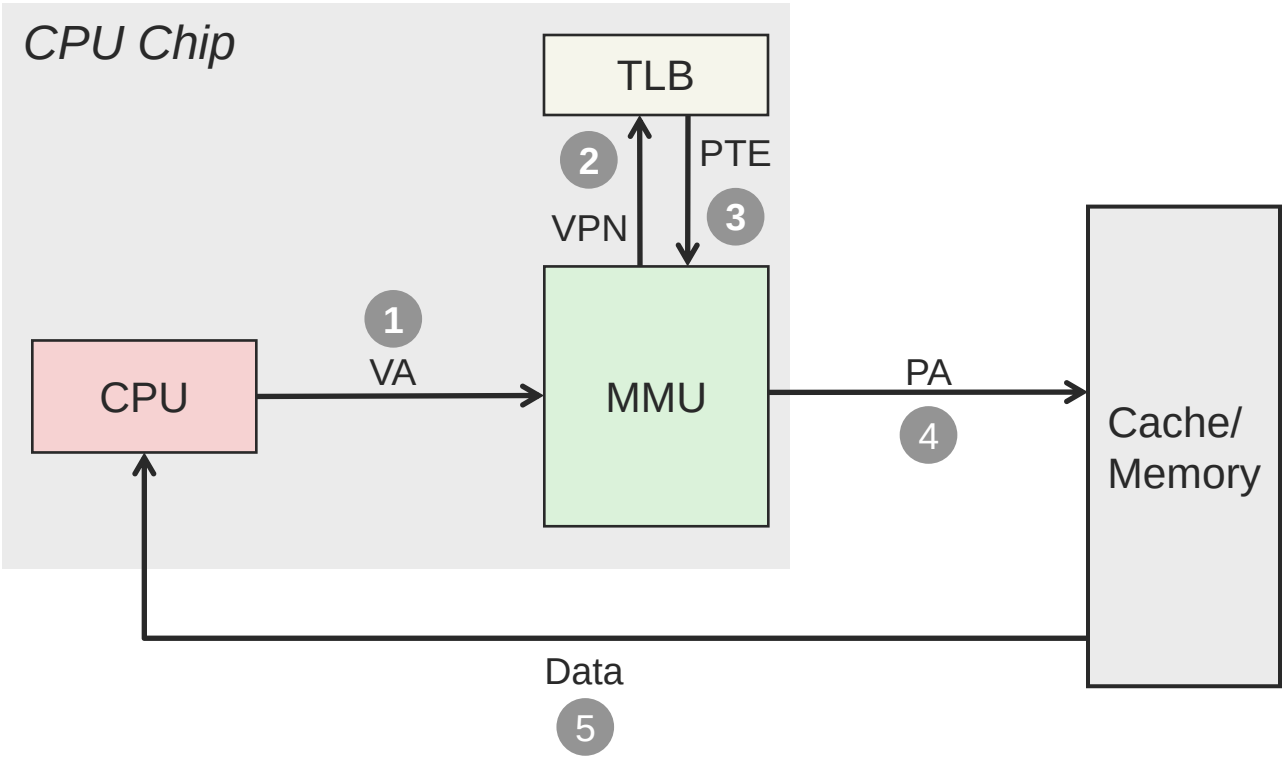
- Isn't it slow to have to go to memory twice every time?

- Yes, it would be… so, real MMUs don't

Speeding up Translation: TLB

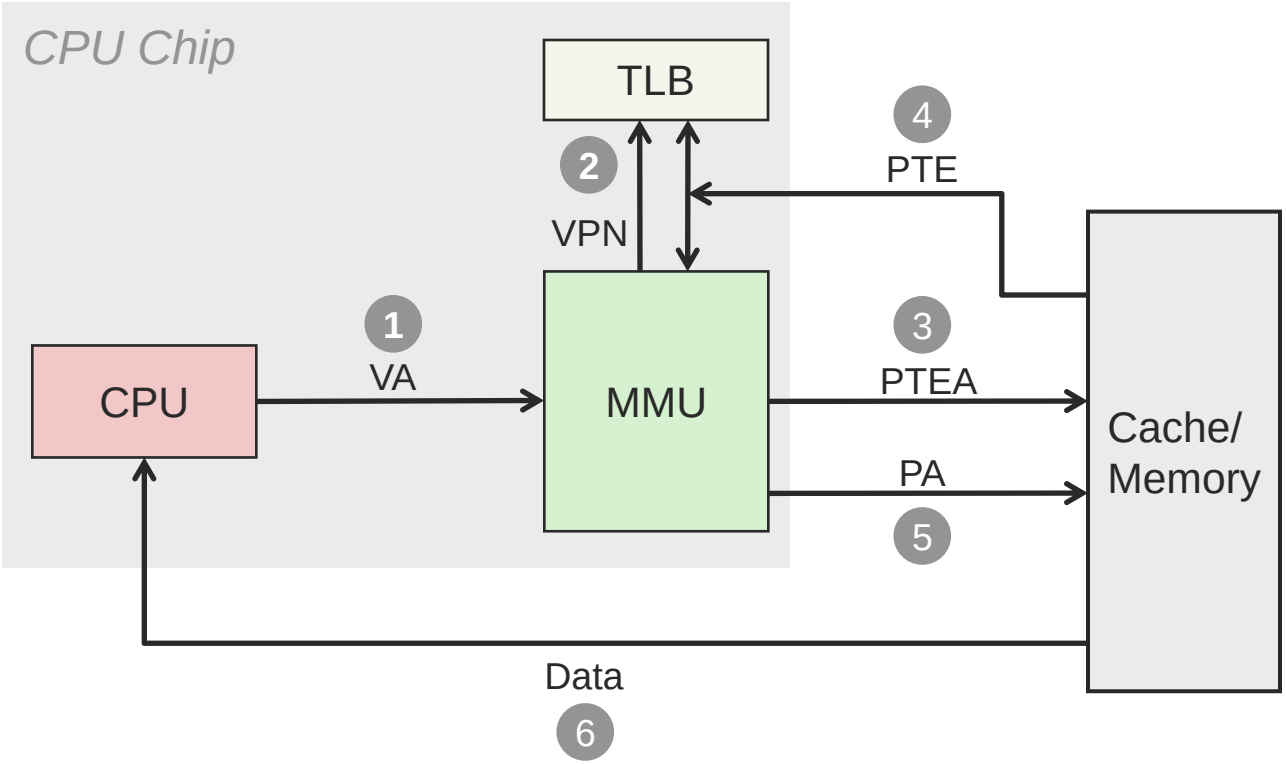
- Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay
- Solution: Translation Lookaside Buffer (TLB)
  - Small, dedicated, super-fast hardware cache of PTEs in MMU
  - Contains complete page 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?

Question 2: Size

- Isn't the page table huge? How can it be stored in RAM?

- Yes, it would be… so, real page tables aren't simple arrays

Page Table Size

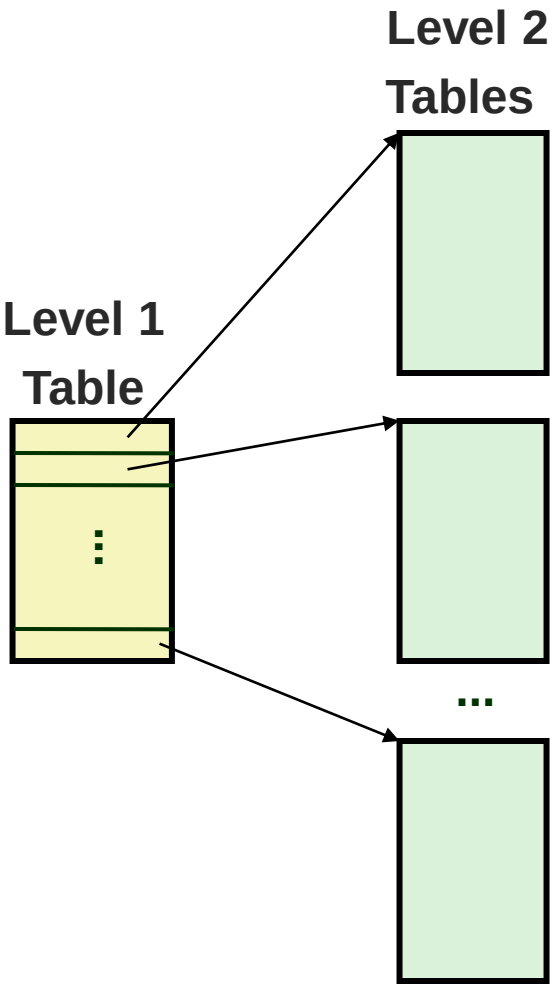
- Suppose
  - 4KB (212) page size, 64-bit address space, 8-byte PTE
- How big does the page table need to be?

Page Table Size

- Suppose
  - 4KB (212) page size, 64-bit address space, 8-byte PTE
- How big does the page table need to be?
  - 32,000 TB!
  - $2^{64} * 2^{-12} * 2^{3} = 2^{55}$ bytes

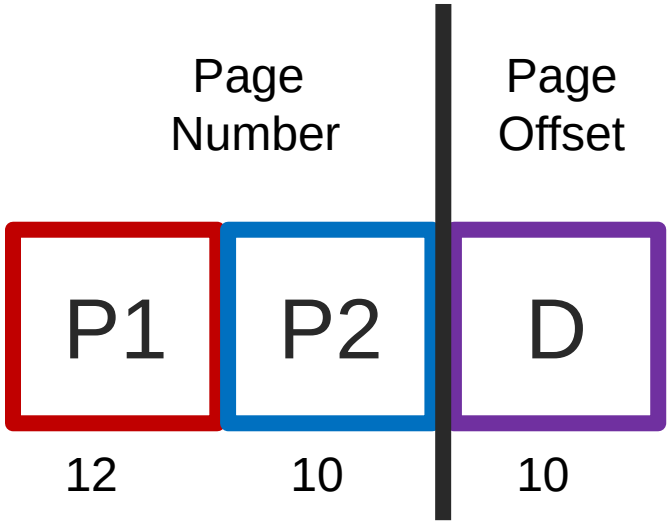
Multi-Level Page Tables

- 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)

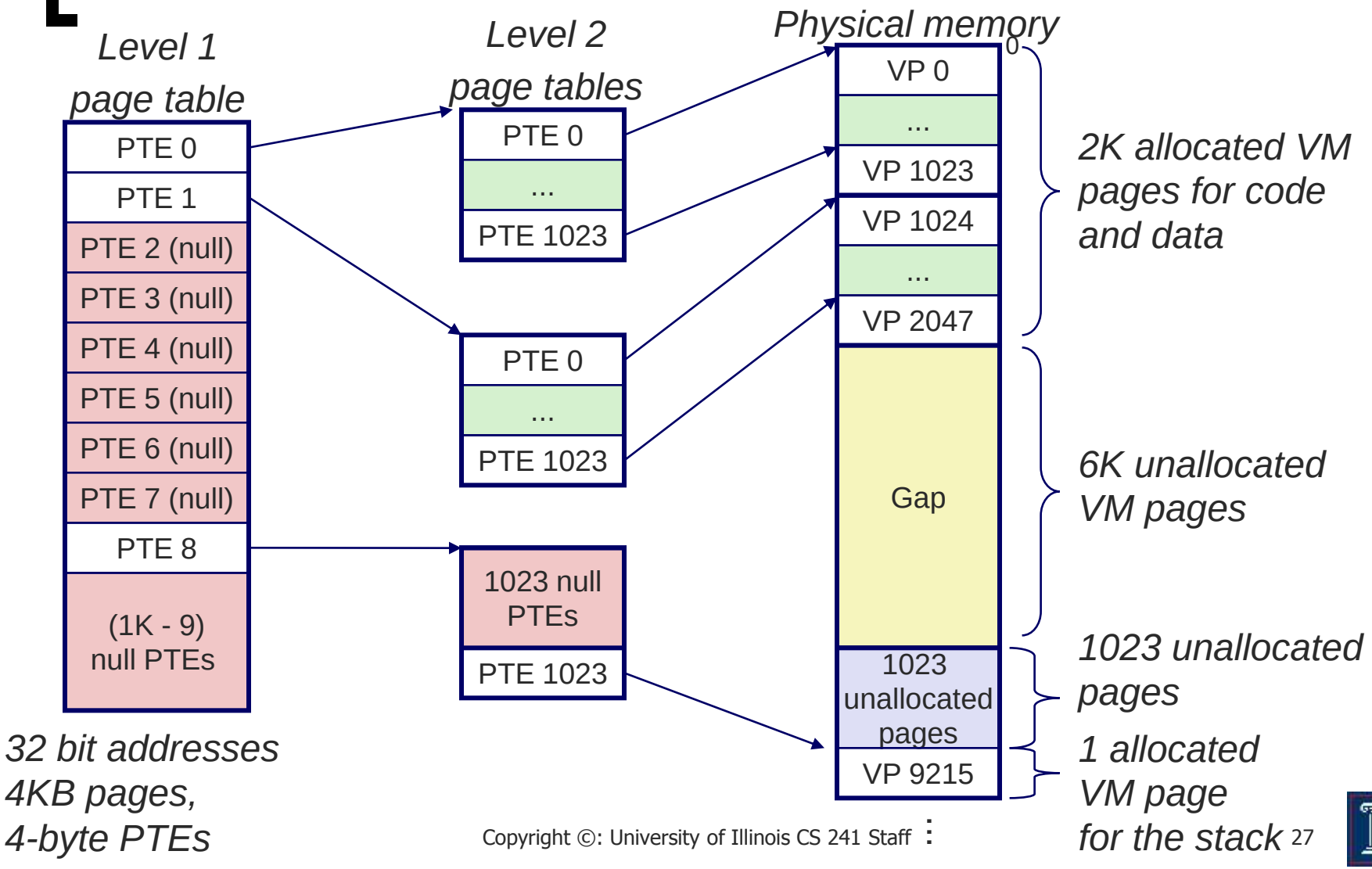


Addressing on Two-Level Page Table

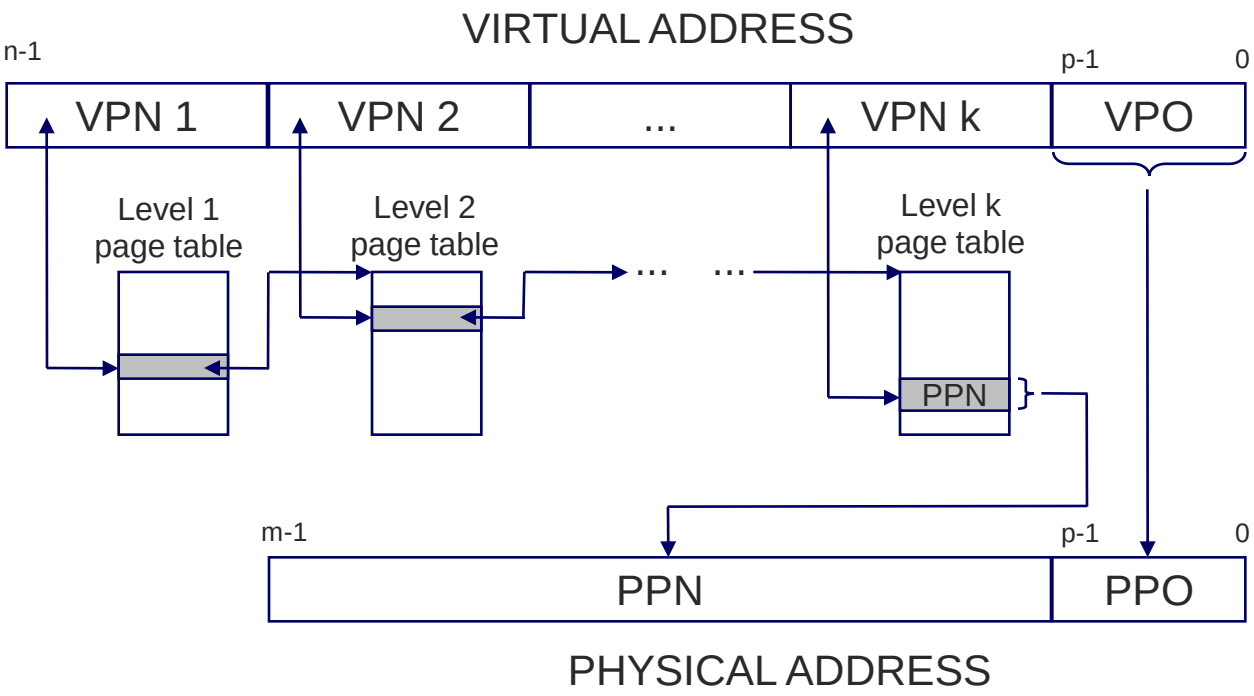
- 32-bit Architecture
  - 4096= 2^{12} B Page
- 4K Page of Logical Memory
  - 4096 addressable bytes
- Page the Page Table
  - 4K pages as well
  - 1024 addressable 4byte addresses



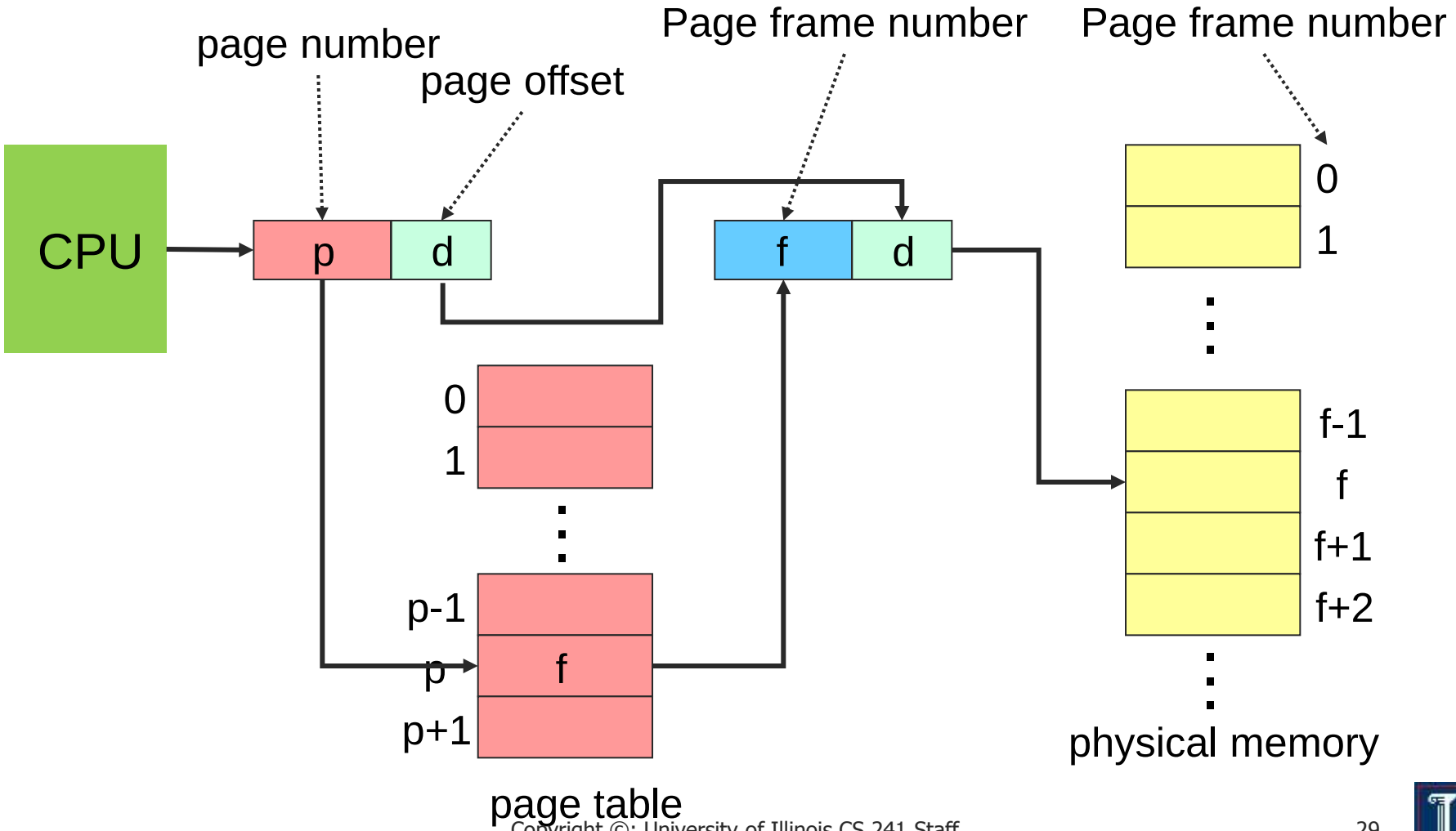
2-level page table hierarchy



Address Translation: k-level Page Table



Addressing on Two-Level Page Table



Multilevel Page Tables

- With two levels of page tables, how big is each table?
  - Allocate
    - 10 bits to the primary page
    - 10 bits to the secondary page
    - 12 bits to the page offset

Multilevel Page Tables

- What happens on a page fault?
  - MMU looks up index in primary page table to get secondary page table
  - MMU tries to access secondary page table
    - May result in another page fault to load the secondary table!
  - MMU looks up index in secondary page table to get physical frame #
  - CPU can then access physical memory address

Multilevel Page Tables

- Issues
  - Page translation has very high overhead
    - Up to three memory accesses plus two disk I/Os!!
  - TLB usage is clearly very important

Page Table Problem (from Tanenbaum)

- Suppose
  - 32-bit address
  - Two-level page table
  - Virtual addresses split into a 9-bit top-level page table field, an 11-bit second-level page table field, and an offset
- Question: How large are the pages and how many are there in the address space?
  - Offset
  - Page size
  - # virtual pages

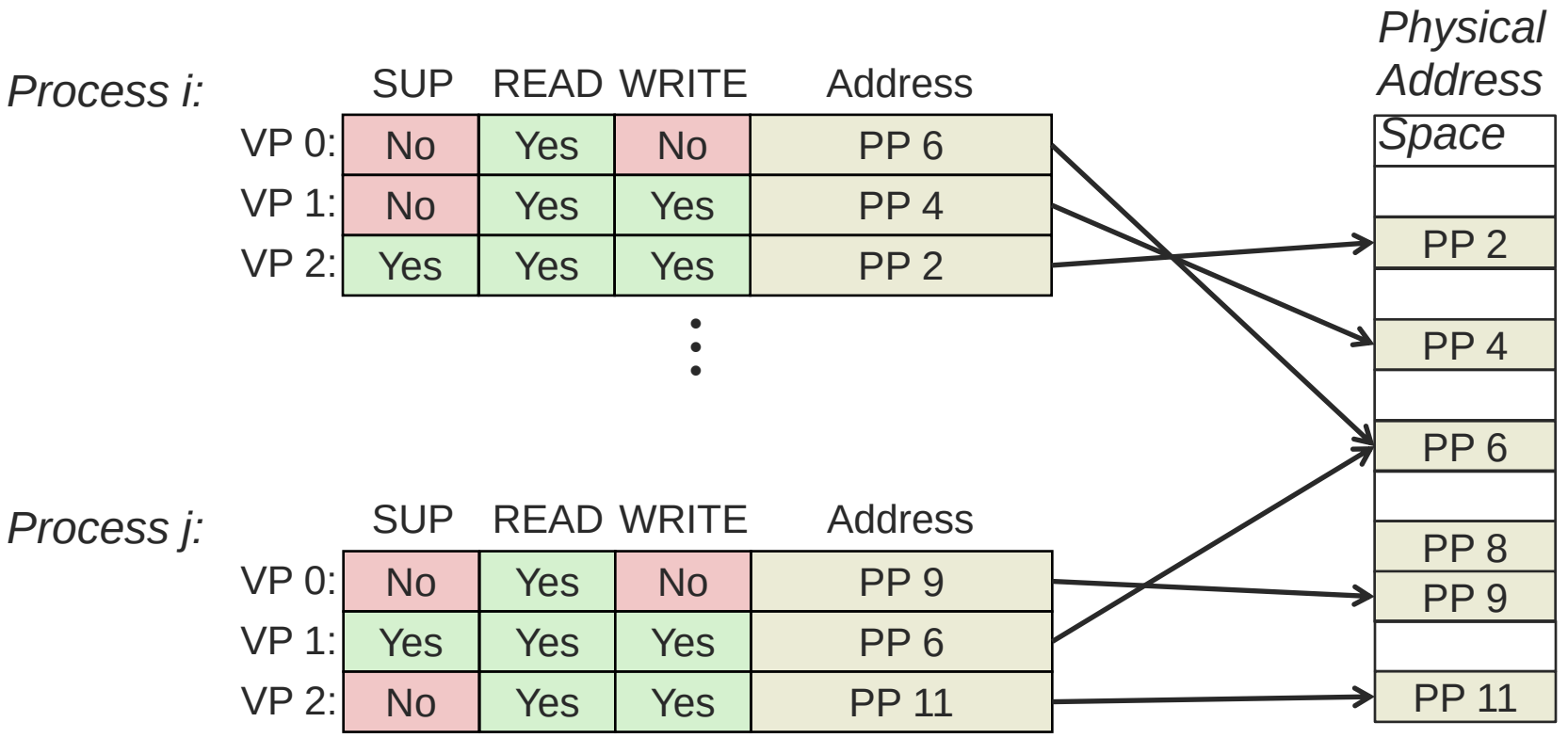
Question 3: So What?

- Is there any other super slick stuff can I do with page tables?

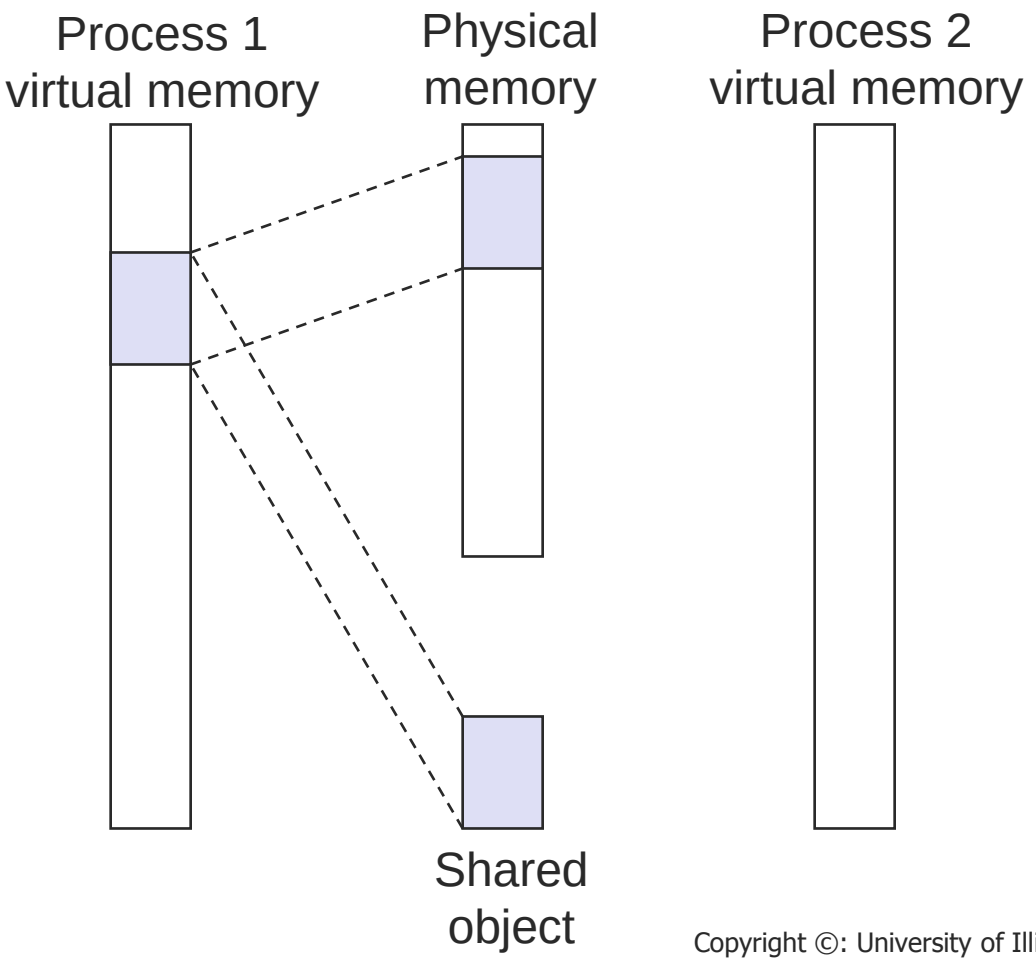
- Yes!

Paging as a tool for protection

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

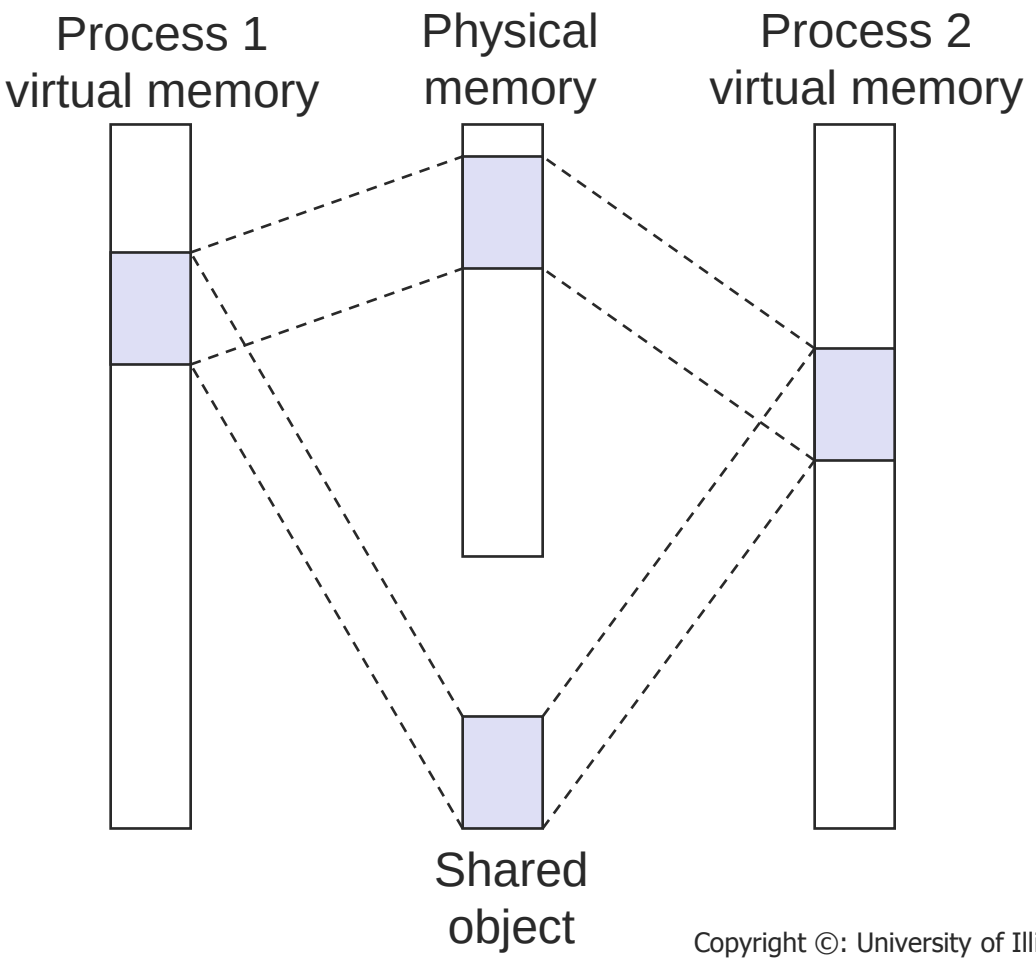


VM as a tool for sharing



- Process 1 maps the shared object

VM as a tool for sharing



- Process 2 maps the shared object
- Notice how the virtual addresses can be different

Protection + Sharing Example

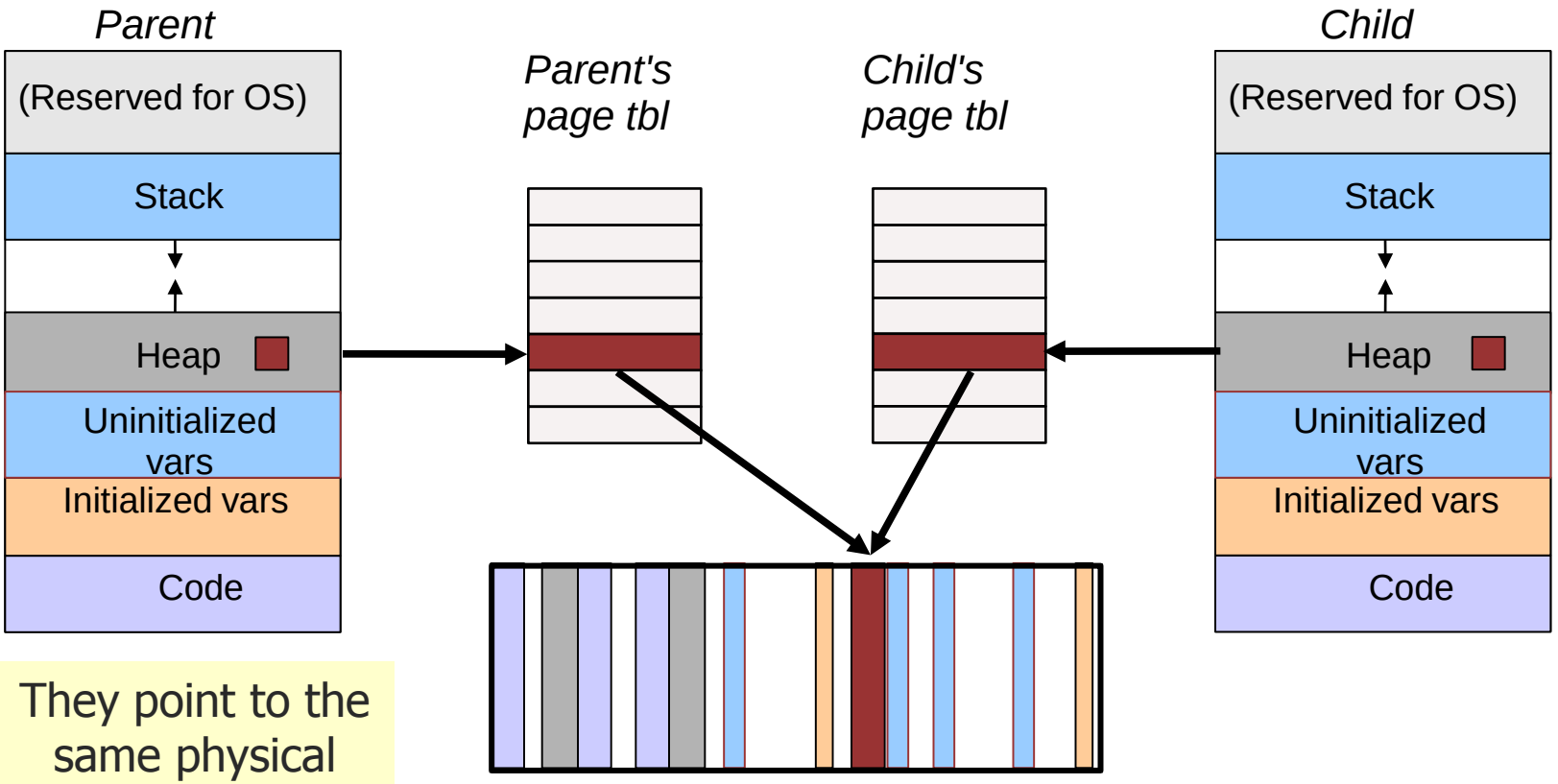
- `fork()` creates **exact** copy of a process
  - Lots more on this next week…
- When we fork a new process, all of the memory is duplicated for the child
  - Does it make sense to make a copy of **all** of its memory?
  - What if the child process doesn't end up touching most of the memory the parent was using?

Performance + Sharing

- Some processes may need to access the same memory
- Copy-on-Write (COW)
  - Allows parent and child processes to initially share the **same** pages in memory
  - Only copy page if one of the processes modifies the shared page
  - More efficient process creation

Copy-on-Write

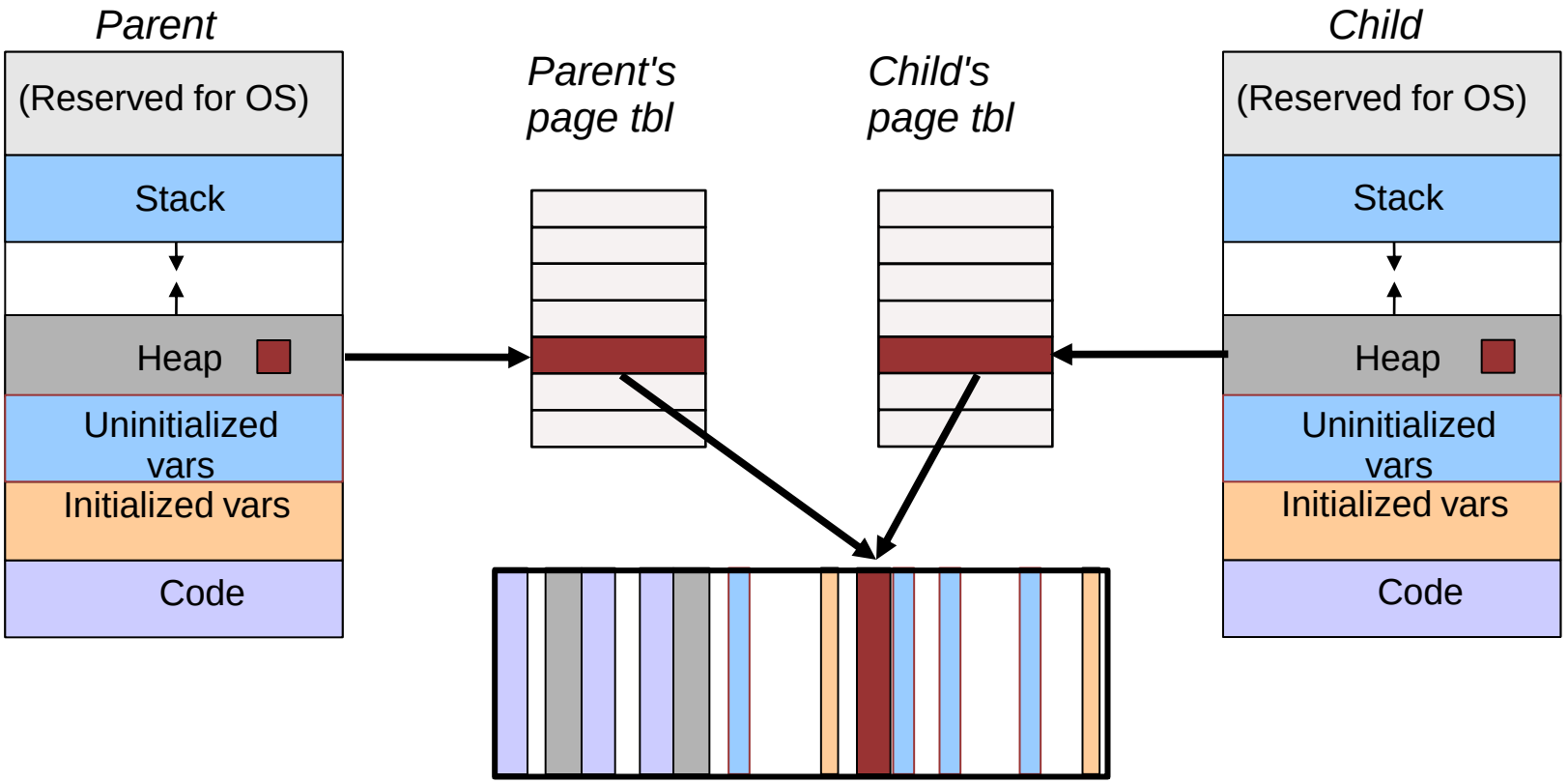
1. Parent forks a child process
2. Child gets a copy of the parent's page tables



They point to the same physical frames!!!

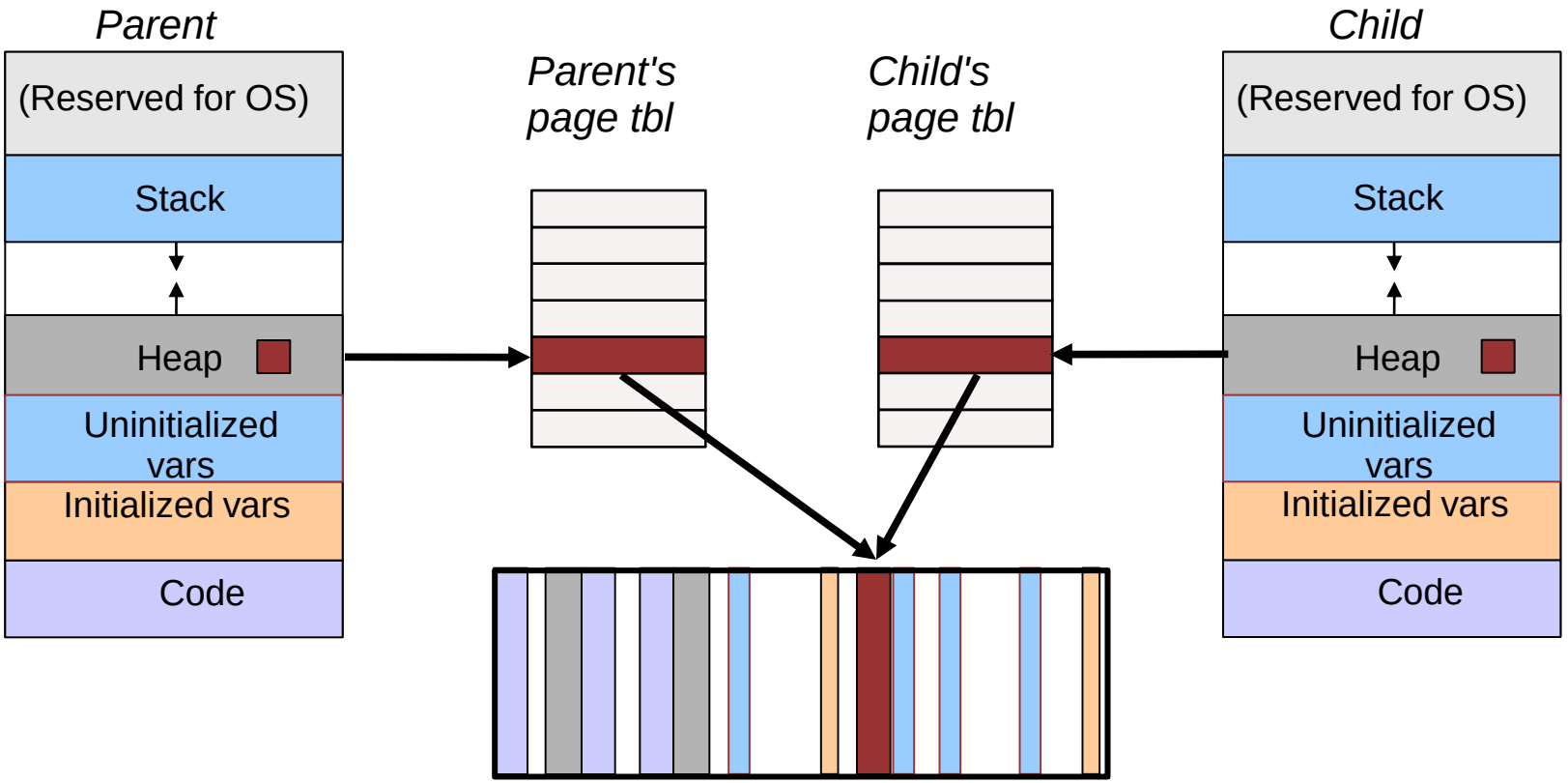
Copy-on-Write

- All pages (both parent and child) marked read-only
  - Why?



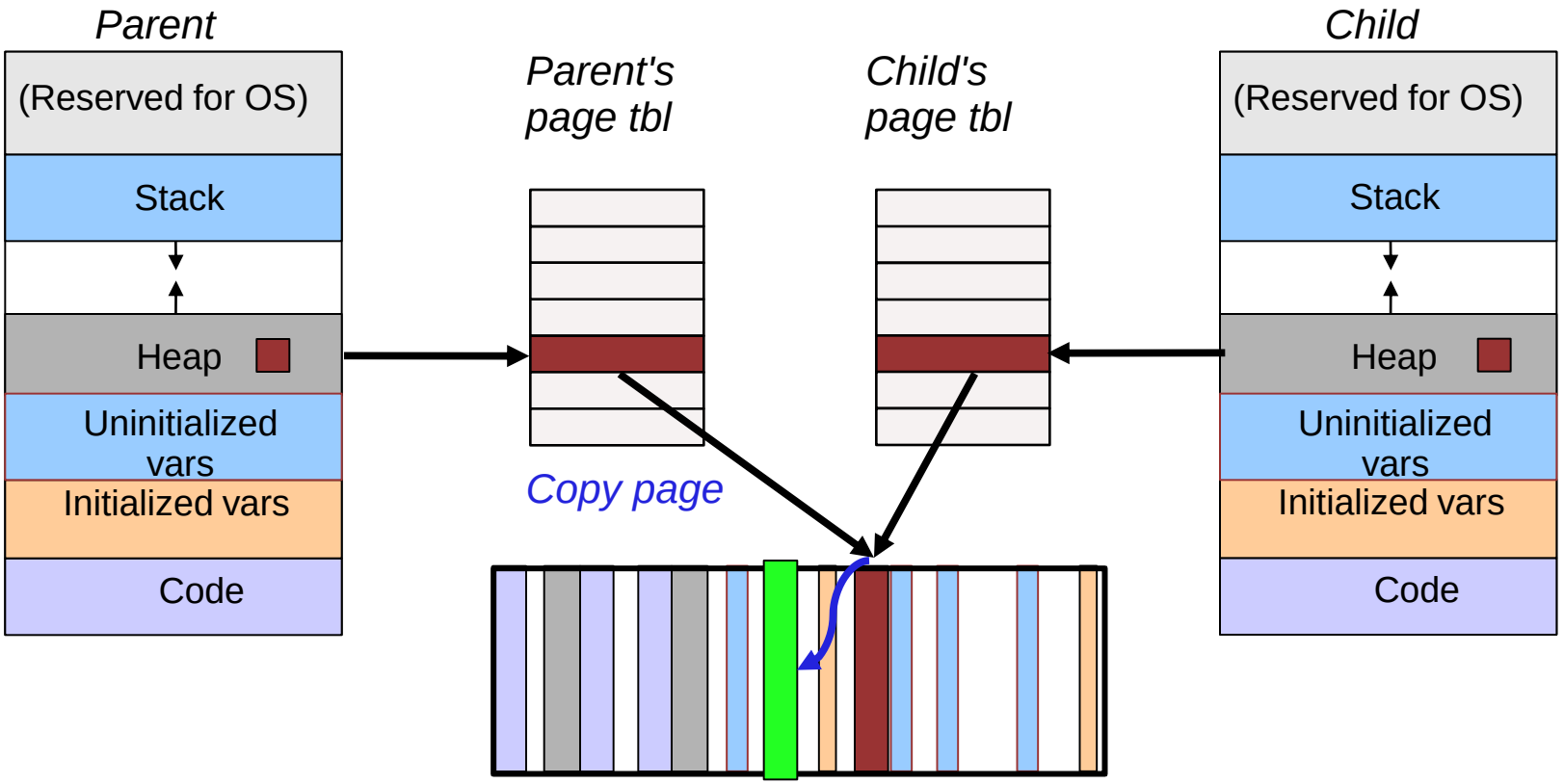
Copy-on-Write

- What happens when the child reads the page?
  - Just accesses same memory as parent .... Niiiiiice!



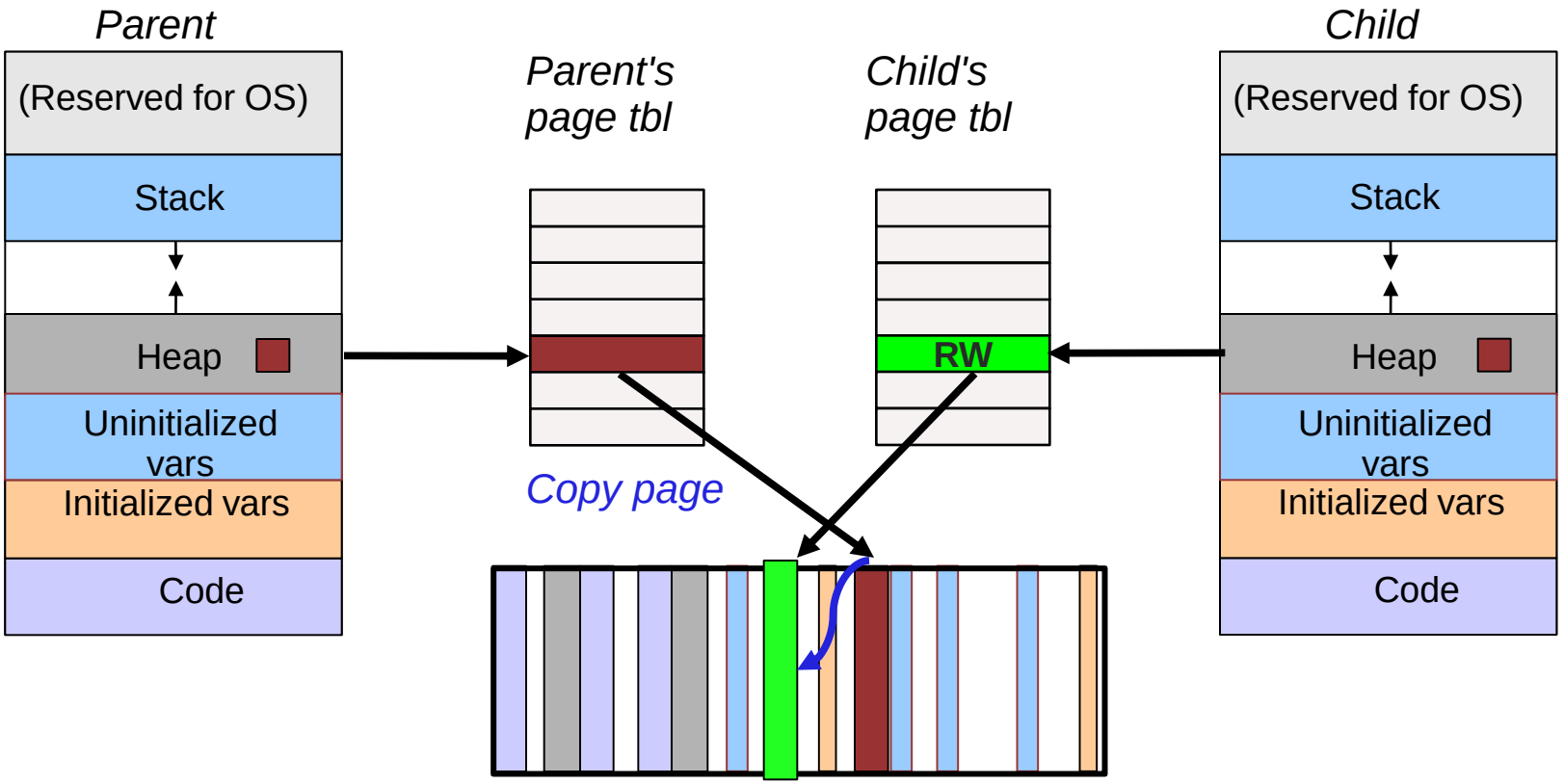
Copy-on-Write

- What happens when the child writes the page?
  - Protection fault occurs (page is read-only!)
  - OS copies the page and maps it R/W into the child's addr space



Copy-on-Write

- What happens when the child writes the page?
  - Protection fault occurs (page is read-only!)
  - OS copies the page and maps it R/W into the child's addr space



Sharing Code Segments