Paging: inside the OS

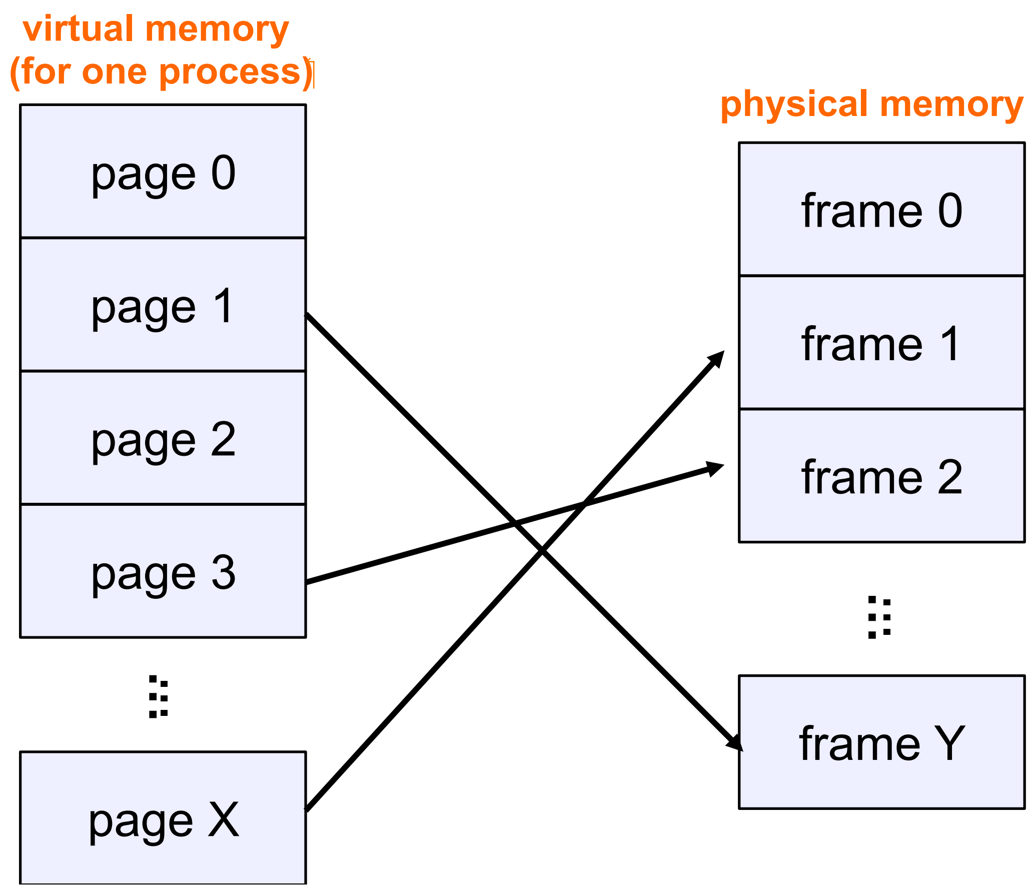
CS 241

February 8, 2012

Slides adapted in part from material by Matt Welsh, Harvard U. and material accompanying Bryant & O'Hallaron, "Computer Systems: A Programmer's Perspective", 2/E

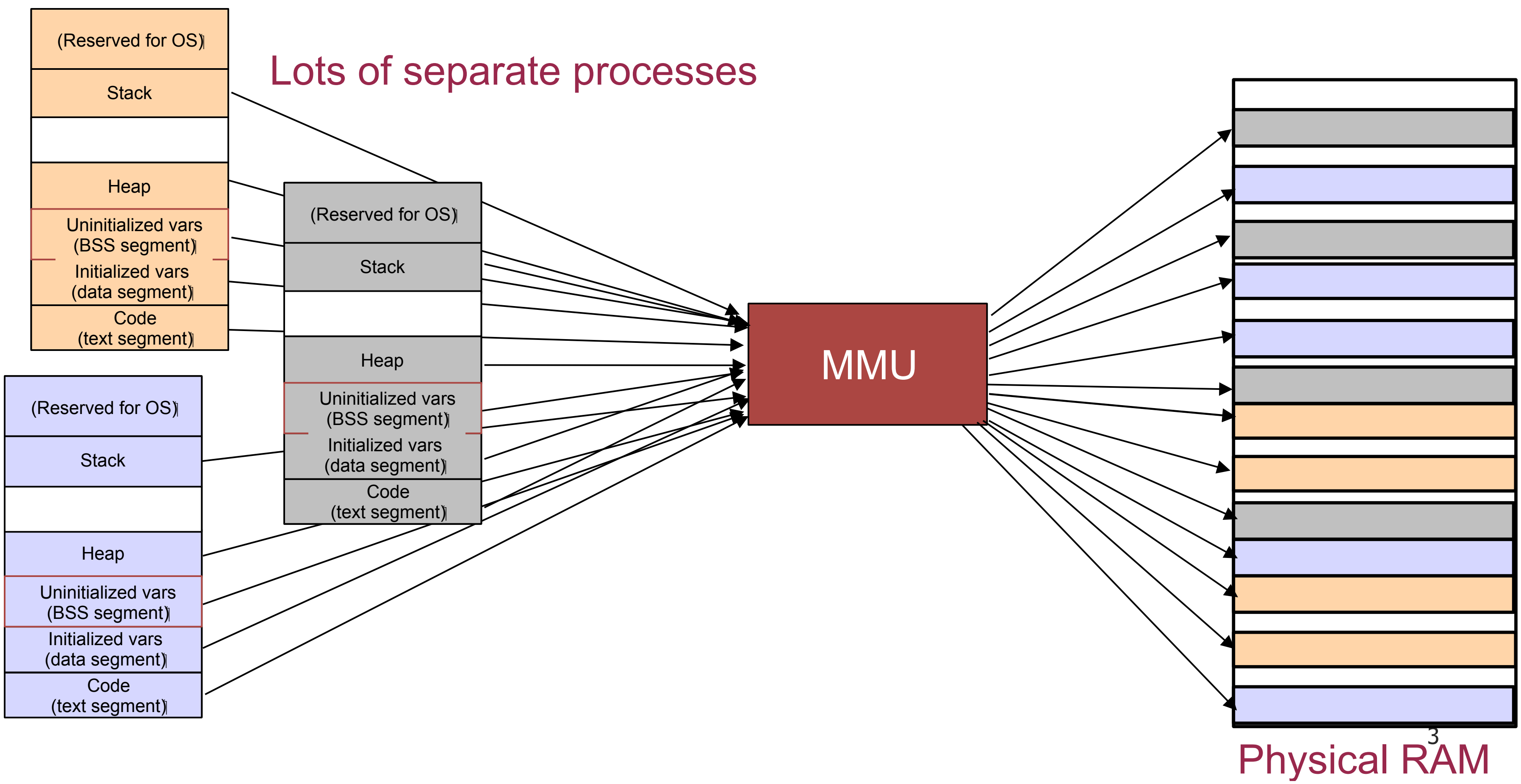
Paging

- Solve the external fragmentation problem by using **fixed-size chunks** of virtual and physical memory
  - Virtual memory unit called a **page**
  - Physical memory unit called a **frame** (or sometimes **page frame**)

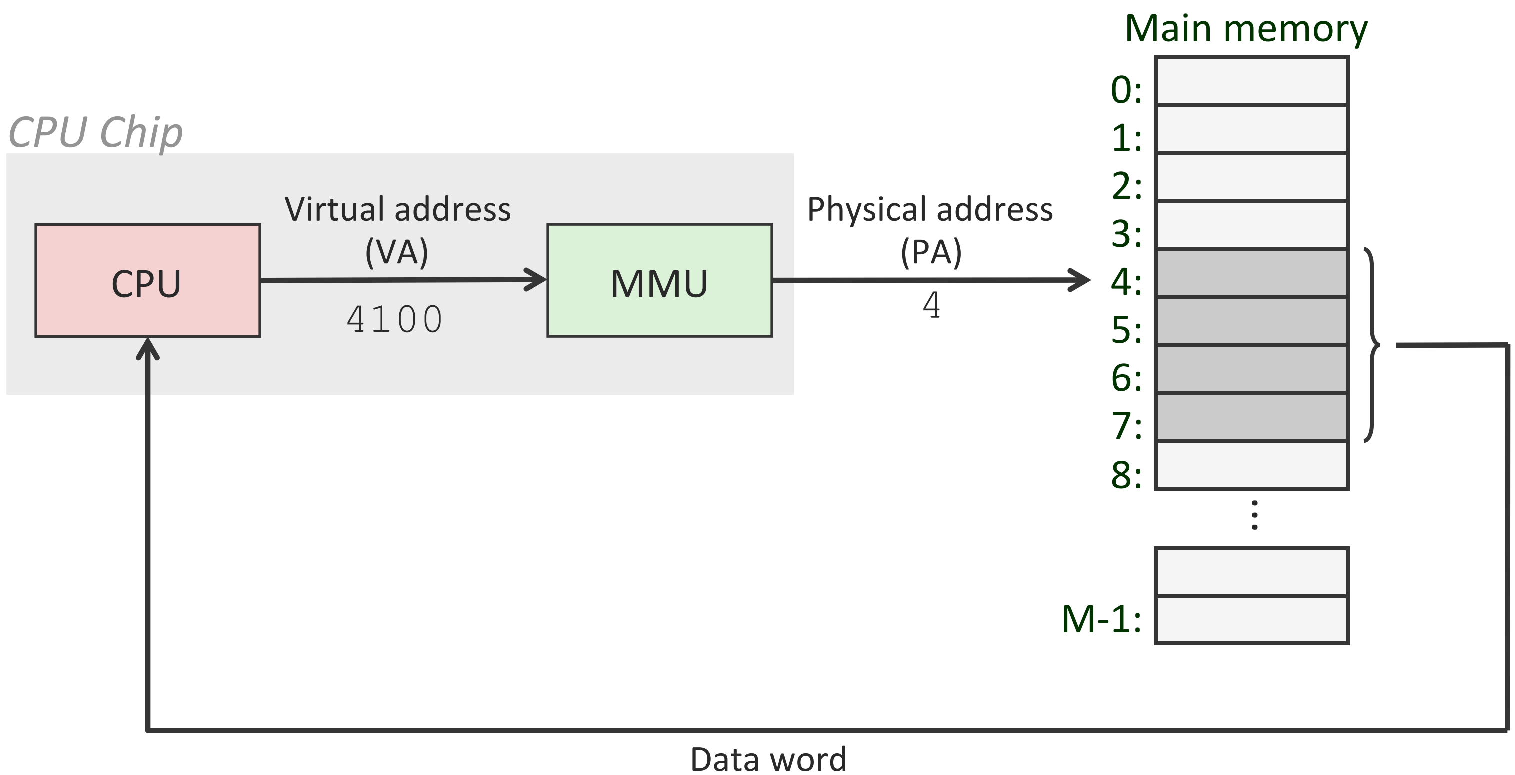


Application Perspective

- Application believes it has a single, contiguous address space ranging from 0 to 2P – 1 bytes
  - Where P is the number of bits in a pointer (e.g., 32 bits)
- In reality, virtual pages are scattered across physical memory
  - This mapping is invisible to the program, and not even under it's control!



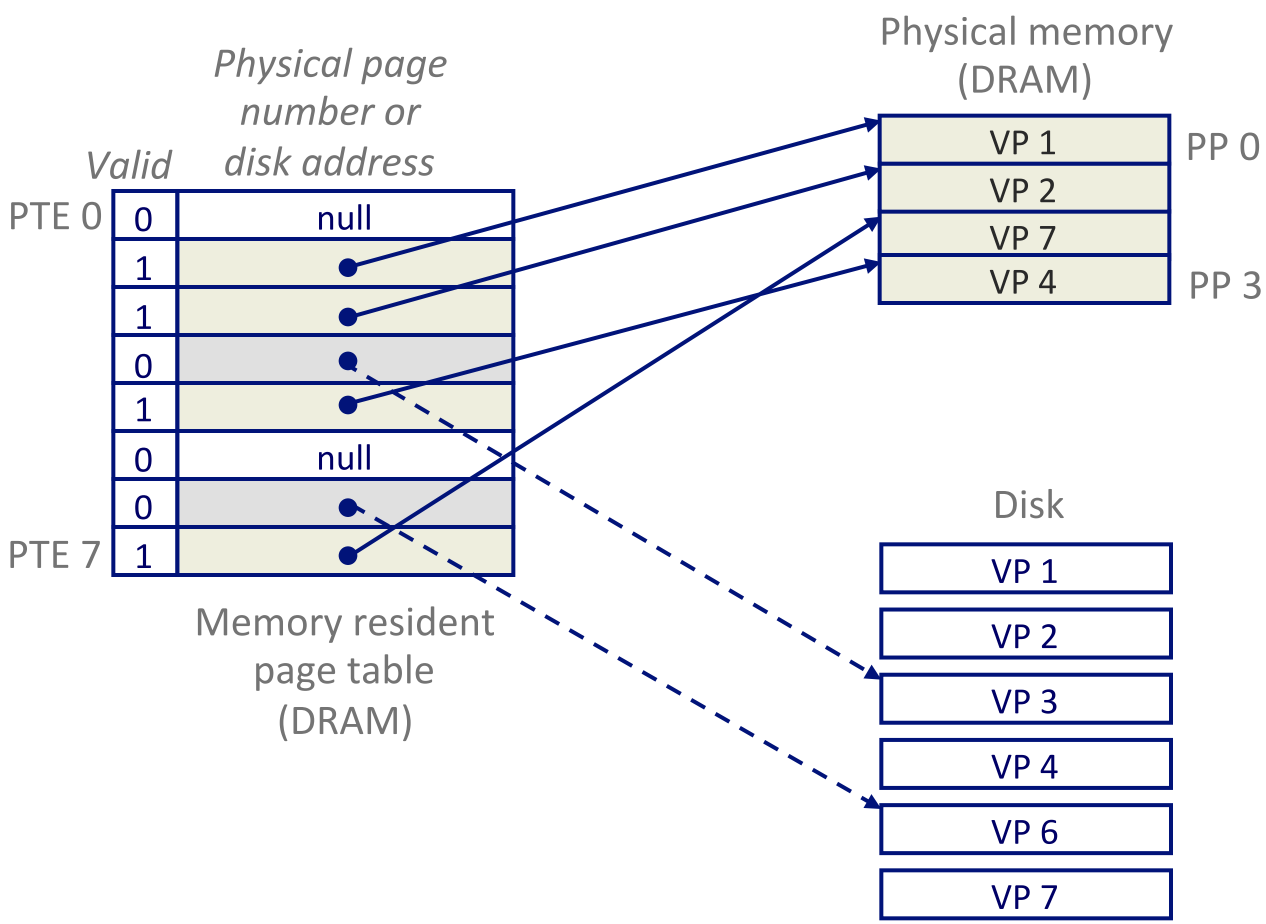
Virtual addressing



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

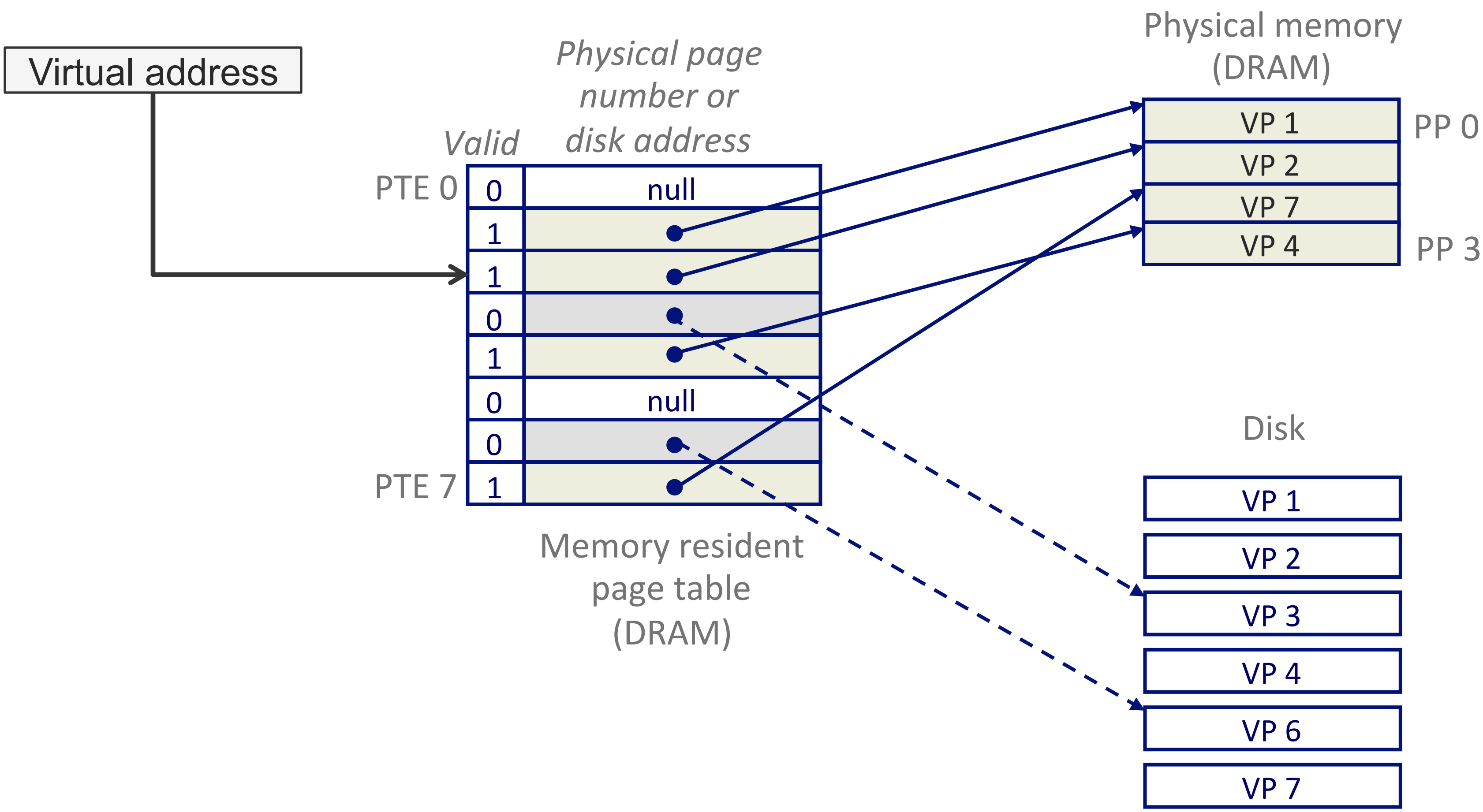
Enabling data structure

- 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



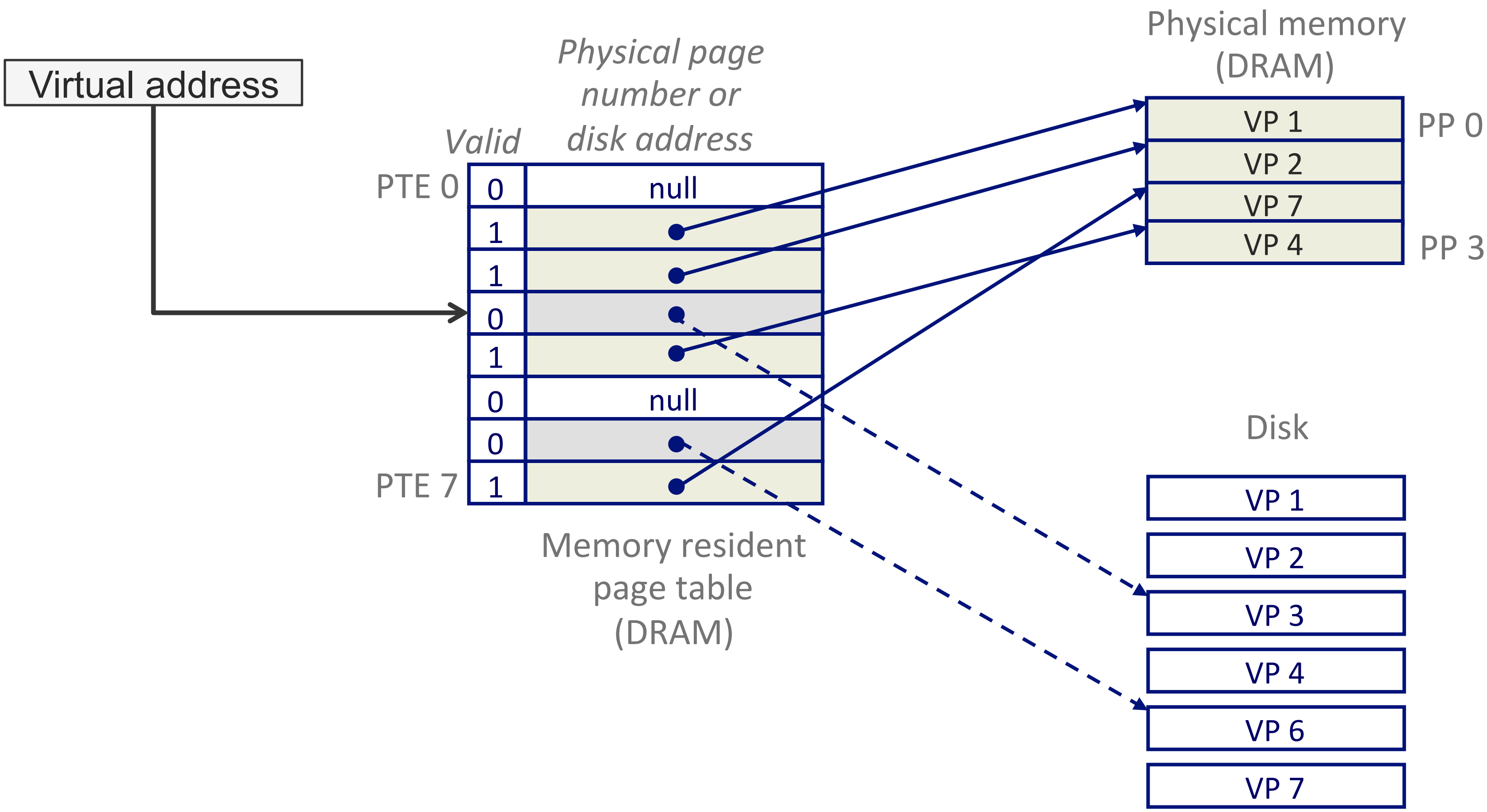
Page hit

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



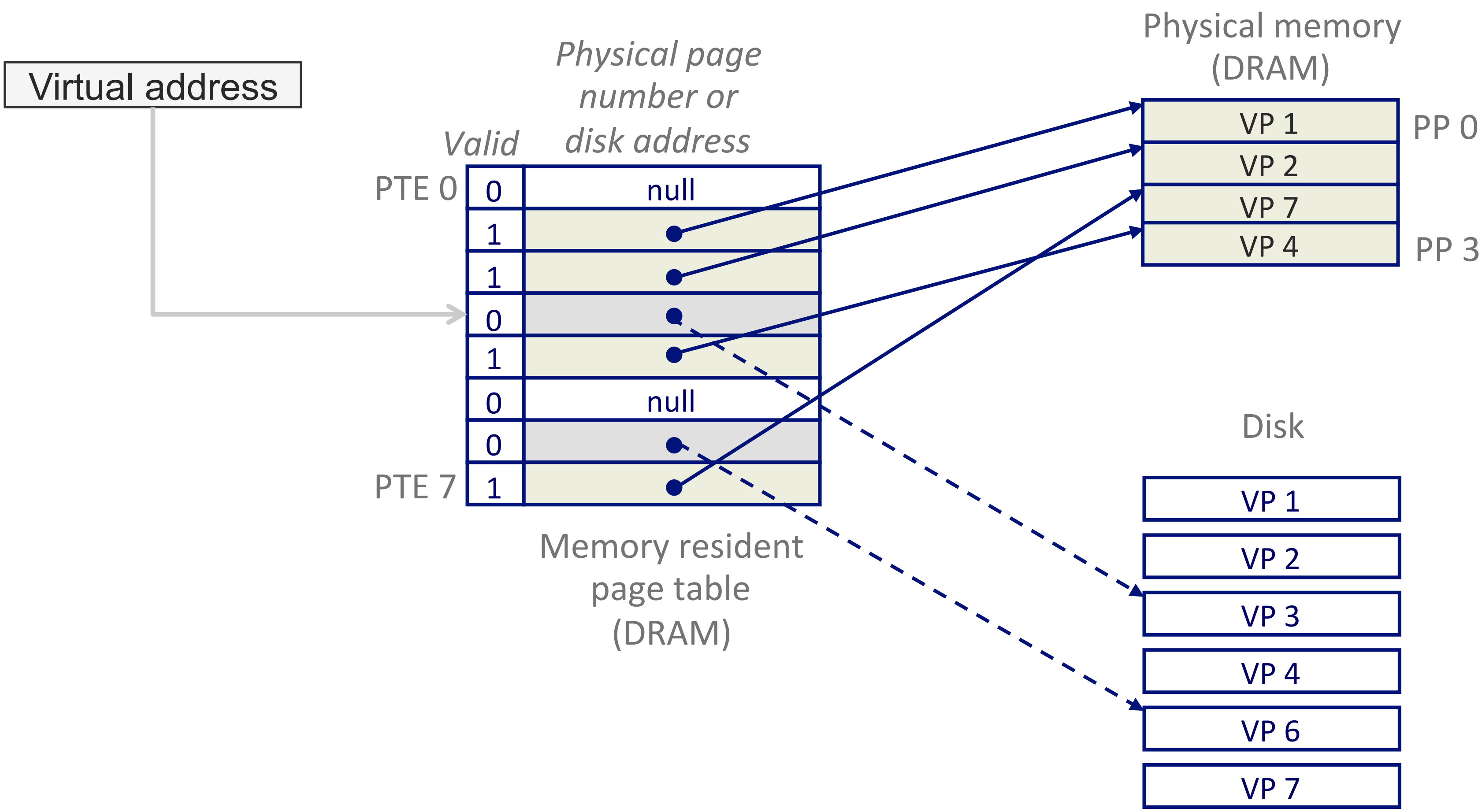
Page fault

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



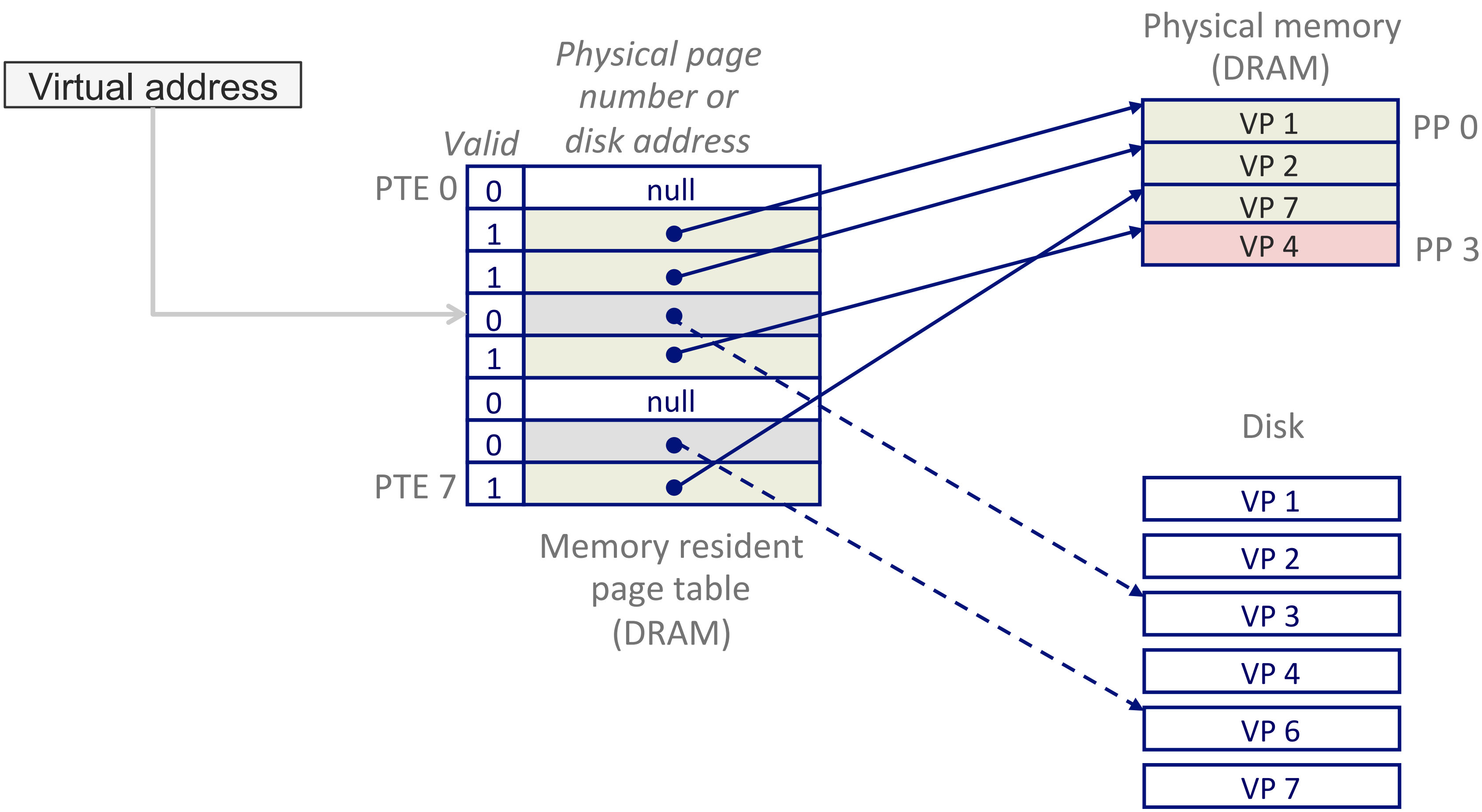
Handling page fault

- Page miss causes page fault (an exception)



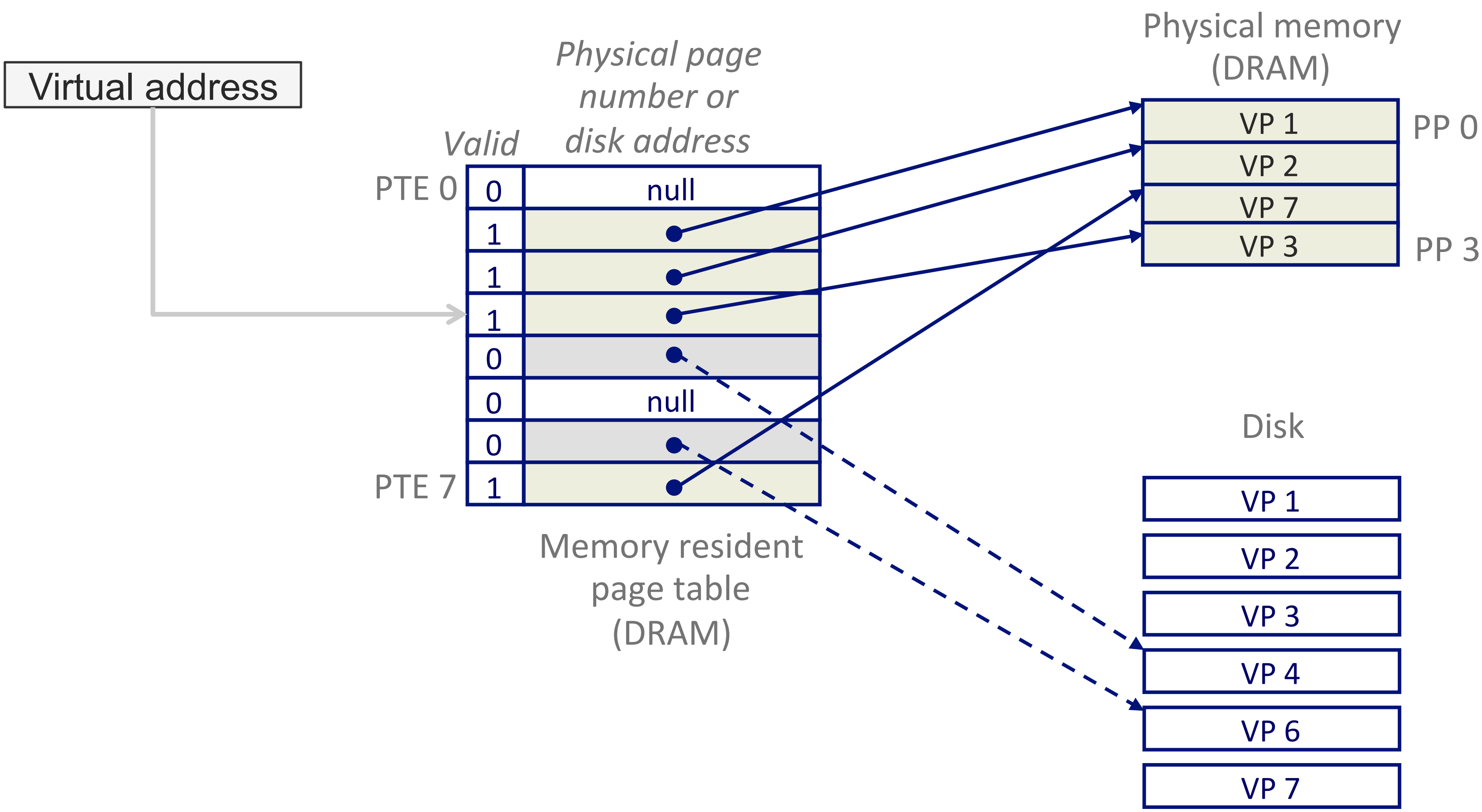
Handling page fault

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



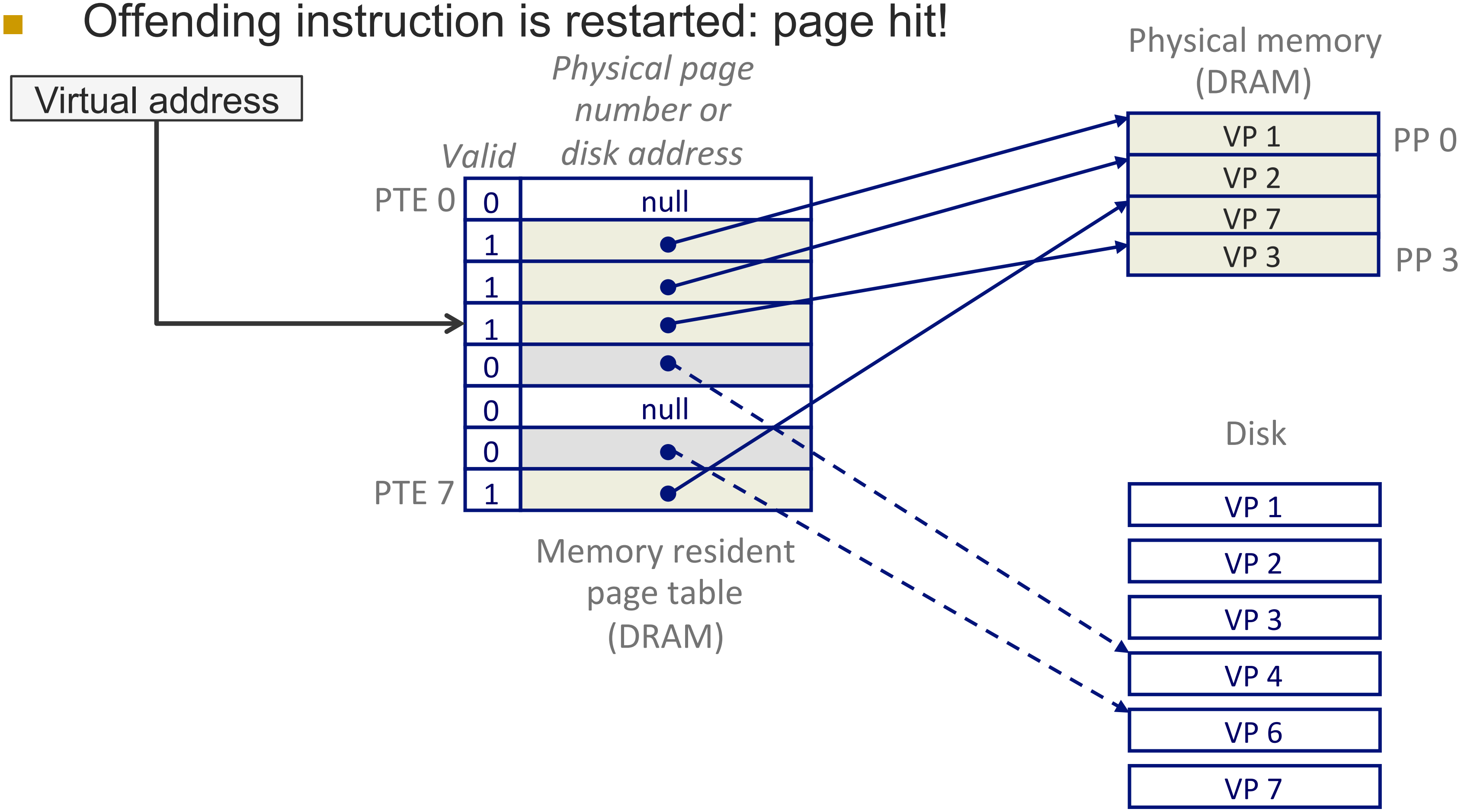
Handling page fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Loads new frame into freed slot



Handling page fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Loads new frame into freed slot
- Offending instruction is restarted: page hit!



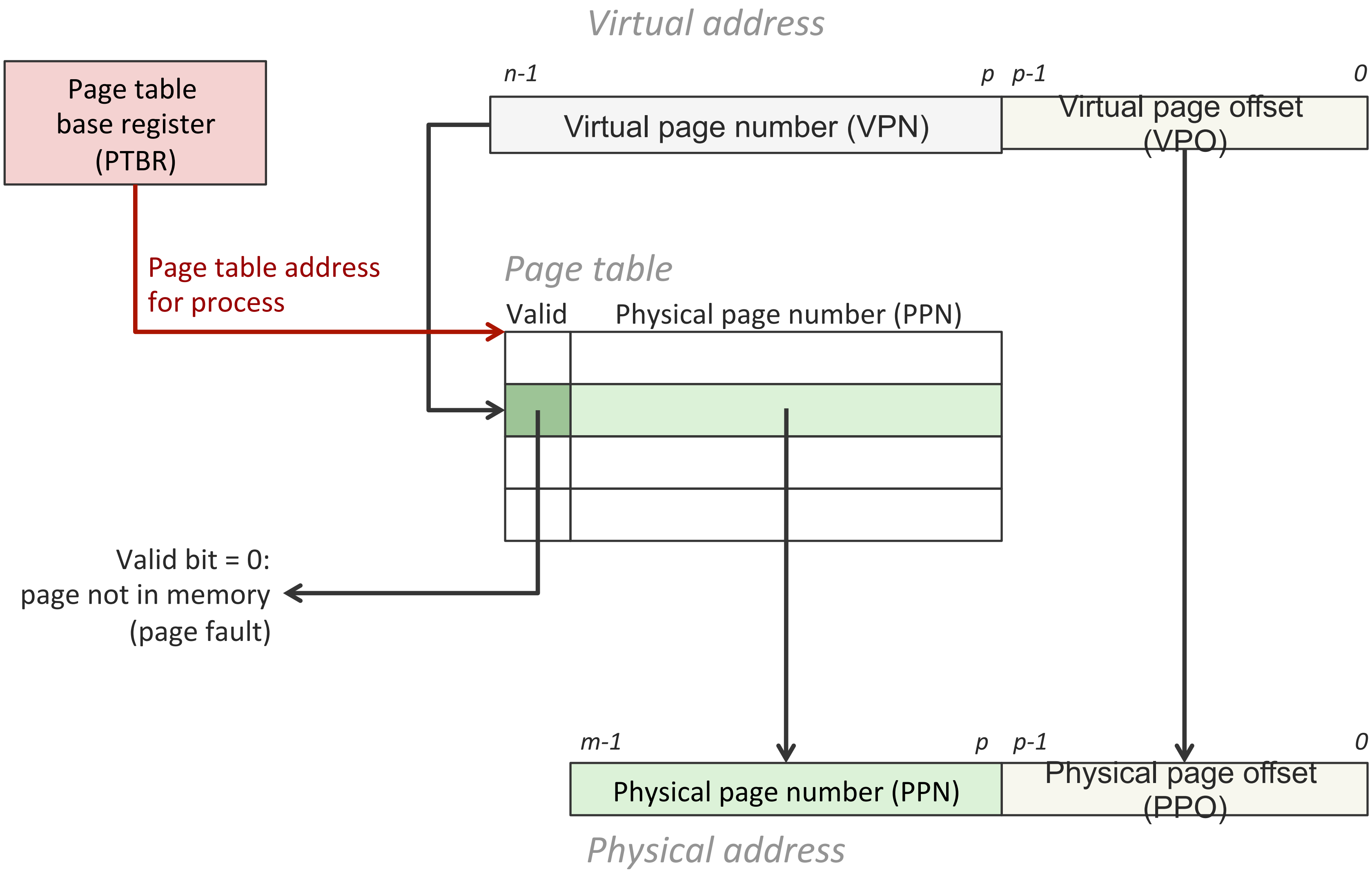
Page Table Entry

- Typical PTE format (depends on CPU architecture!)

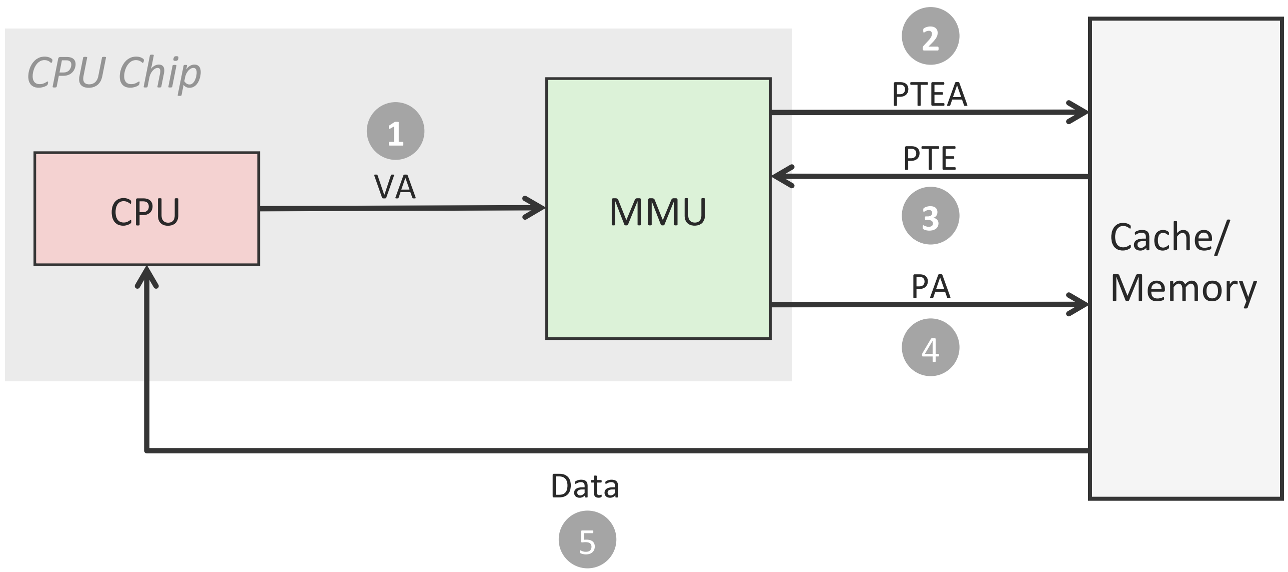
1	1	1	2	20
M	R	V	prot	physical page (frame) number

- Various bits accessed by MMU on each page access:
  - Modify bit: Indicates whether a page is "dirty" (modified)
  - Reference bit: Indicates whether a page has been accessed (read or written)
  - Valid bit: Whether the PTE represents a real memory mapping
  - Protection bits: Specify if page is readable, writable, or executable
  - Physical page number: Physical location of page in RAM
    - Why is this 20 bits wide in the above example?

Address translation with a P.T.



Address translation: page hit



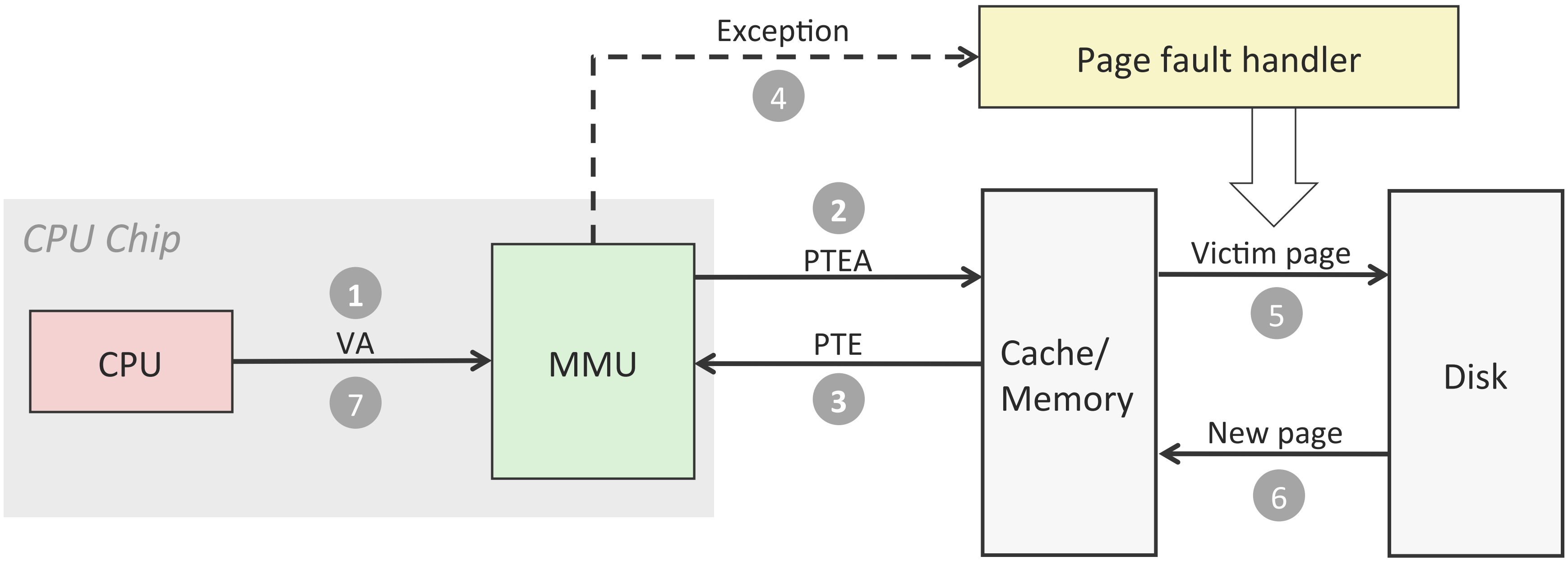
1) Processor sends virtual address to MMU

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

Question 1

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

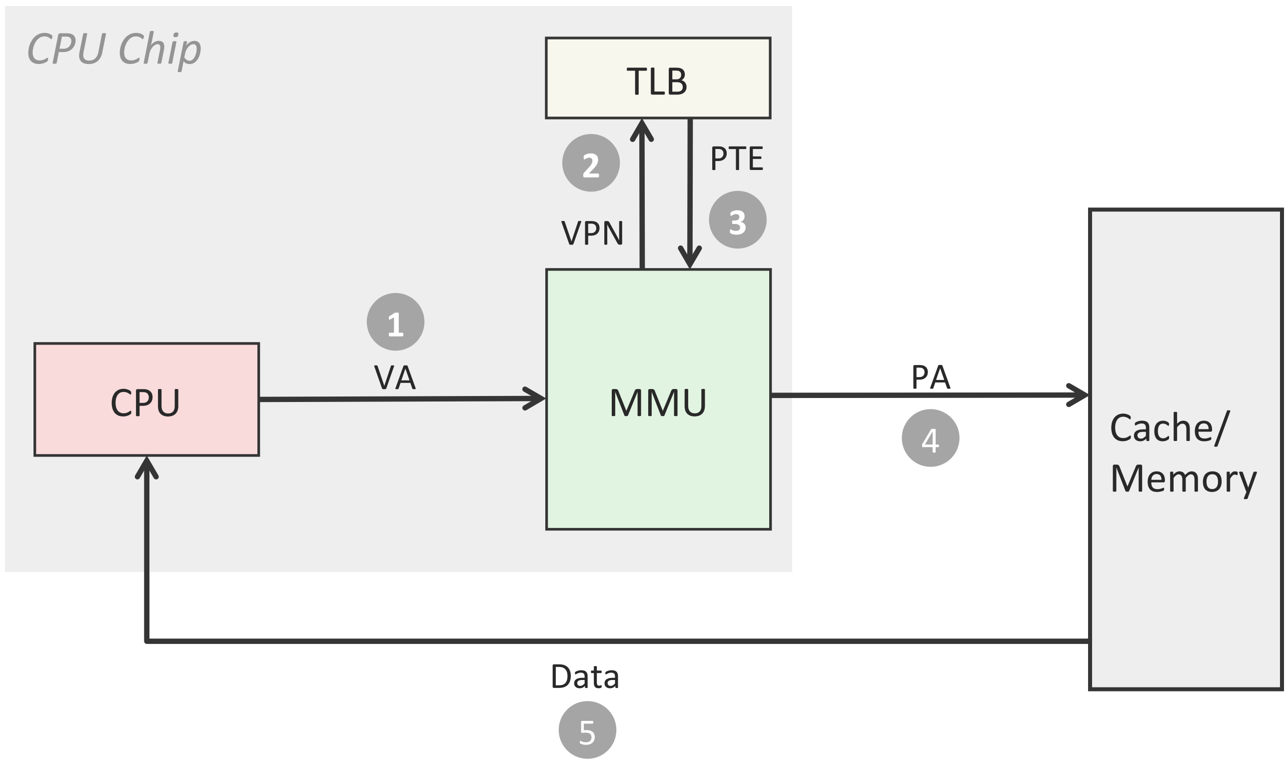
- Yes, it would be… so, real MMUs don’t

Speeding up translation with 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

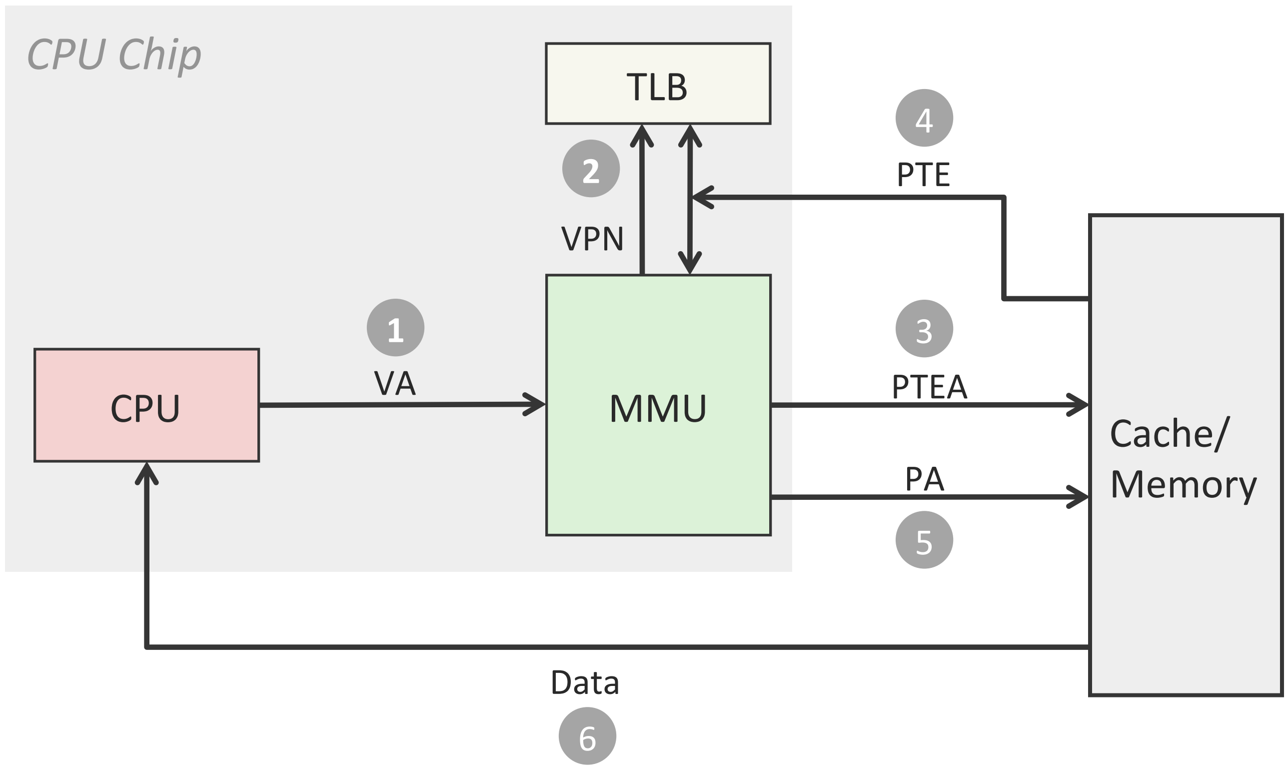
- Solution: *Translation Lookaside Buffer* (TLB)
  - Small, dedicated, super-fast hardware cache of PTEs in MMU
  - Contains complete page table entries for small number of pages

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?

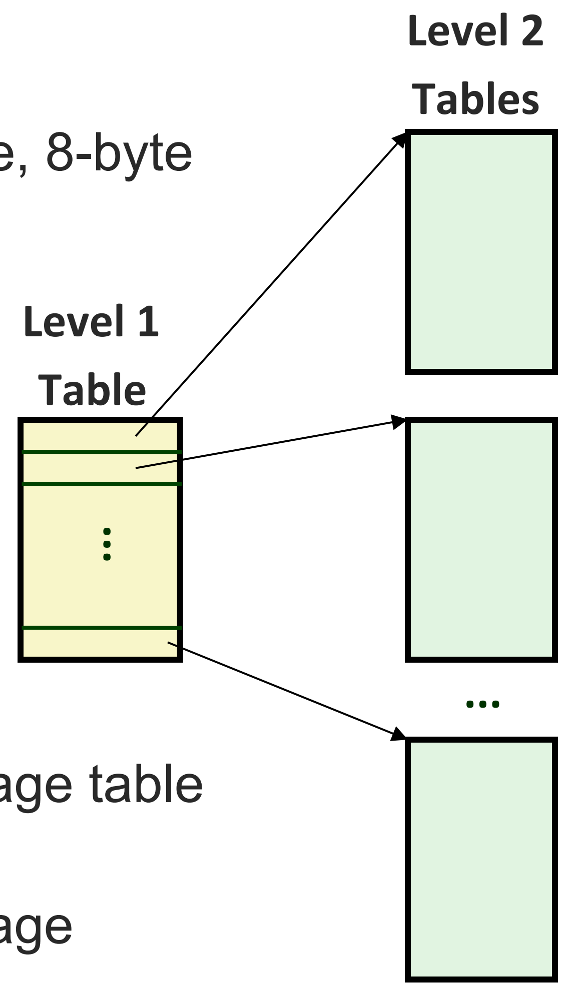
Question 2

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

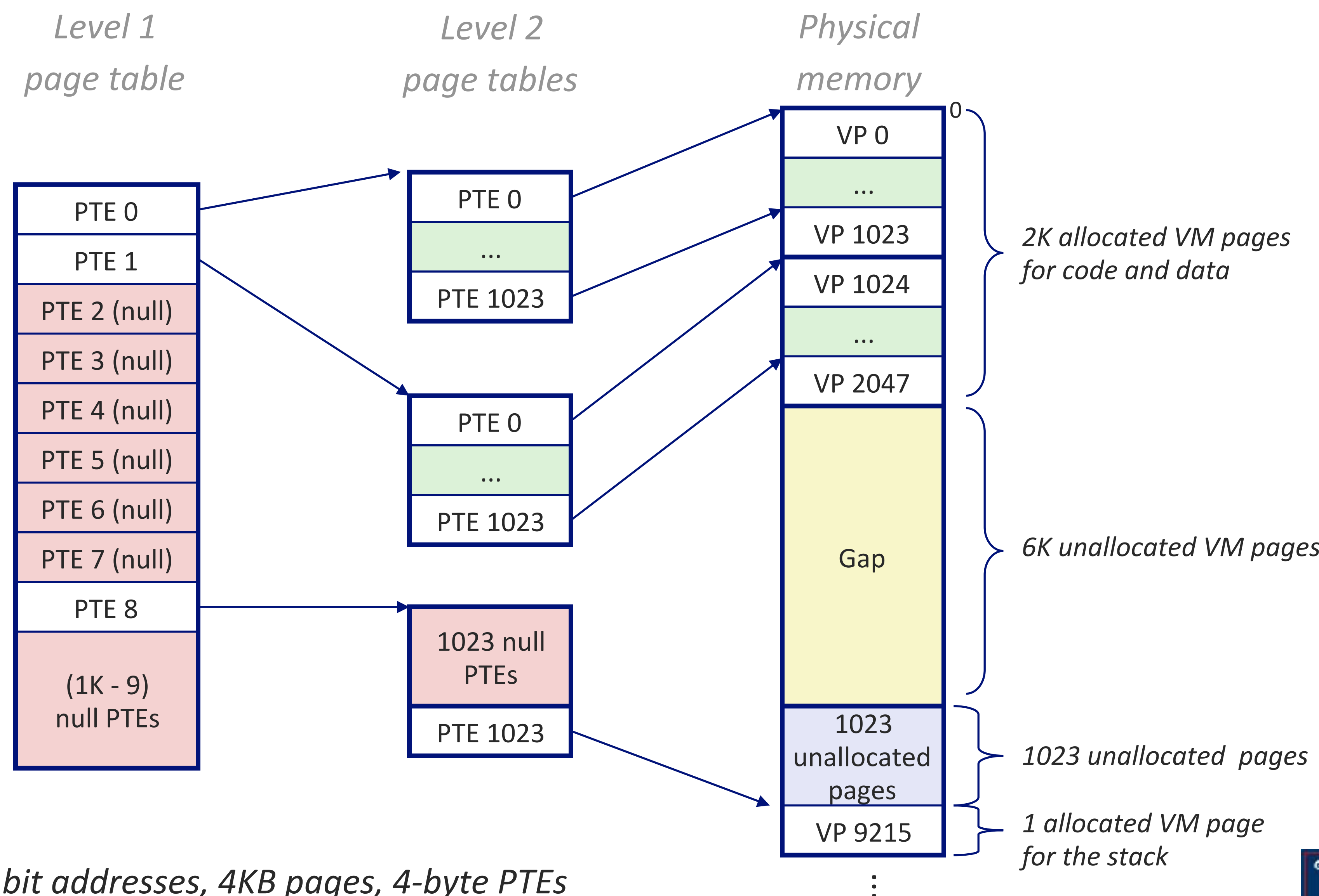
- Yes, it would be… so, real page tables aren't simple arrays

Multi-Level Page Tables

- Suppose:
  - 4KB (2^{12}) page size, 64-bit address space, 8-byte PTE
- Problem:
  - Would need a 32,000 TB page table!
  - $2^{64} * 2^{-12} * 2^{3} = 2^{55}$ bytes
- 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)

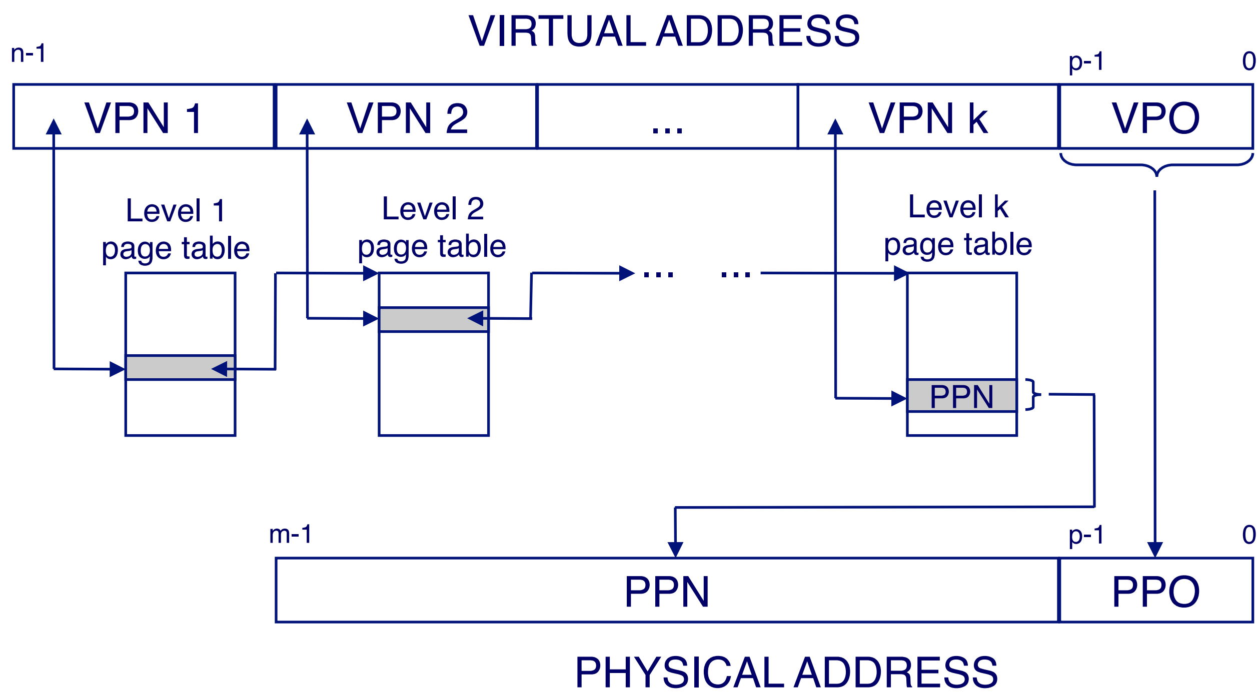


2-level page table hierarchy



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

Addr. translation with k-level PT



Multilevel Page Tables

- With two levels of page tables, how big is each table?
  - Say we allocate 10 bits to the primary page, 10 bits to the secondary page, 12 bits to the page offset
  - Primary page table is then 2^{10} * 4 bytes per PTE == 4 KB
  - Secondary page table is also 4 KB
  - Hey ... that's exactly the size of a page on most systems ... cool
- 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
- Issues
  - Page translation has very high overhead
    - Up to three memory accesses plus two disk I/Os!!
  - TLB usage is clearly very important

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?

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 is 12 bits
  - Page size 2^{12} bytes = 4KB
  - # Virtual pages = $(2^{32} / 2^{12}) = 2^{20}$
  - Note: driven by number of bits in offset
    - Independent of size of top and 2nd level

Question 3

- 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)

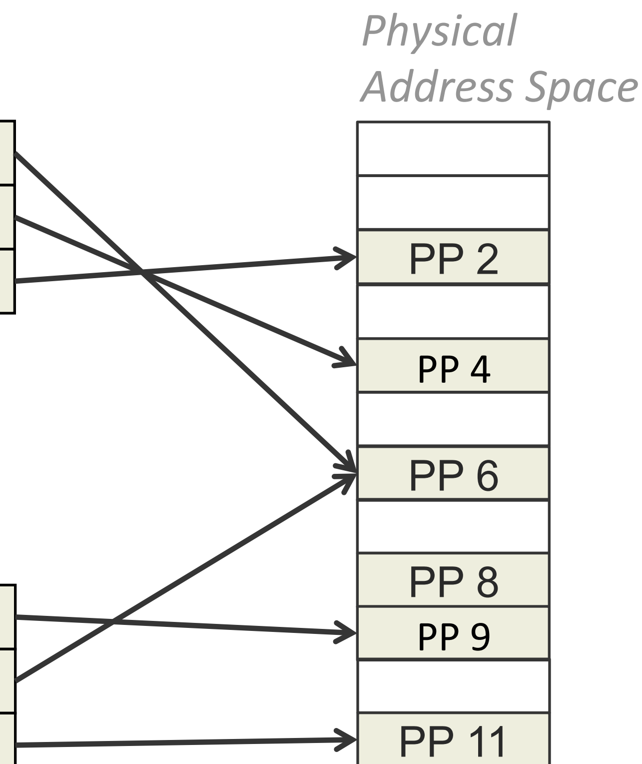
Process i:

	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 6
VP 1:	No	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	PP 2

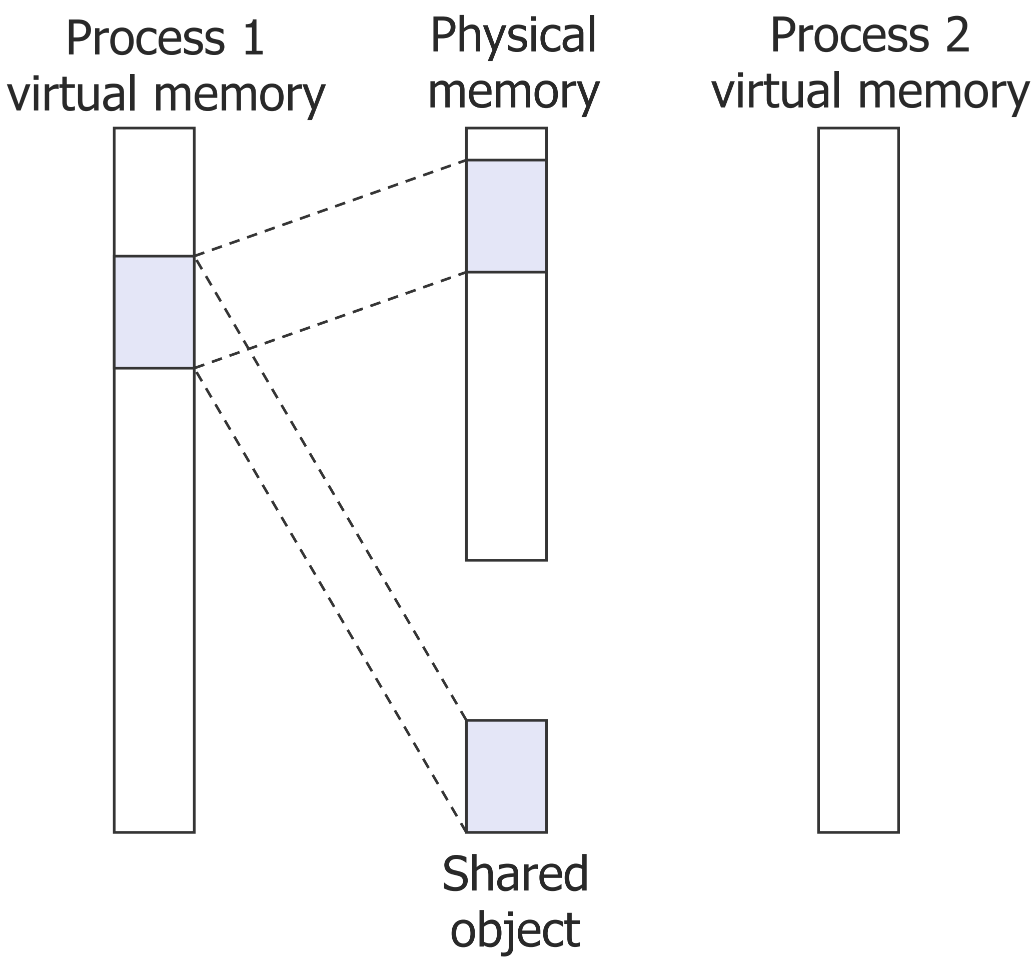
•
•
•

Process j:

	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 9
VP 1:	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	PP 11

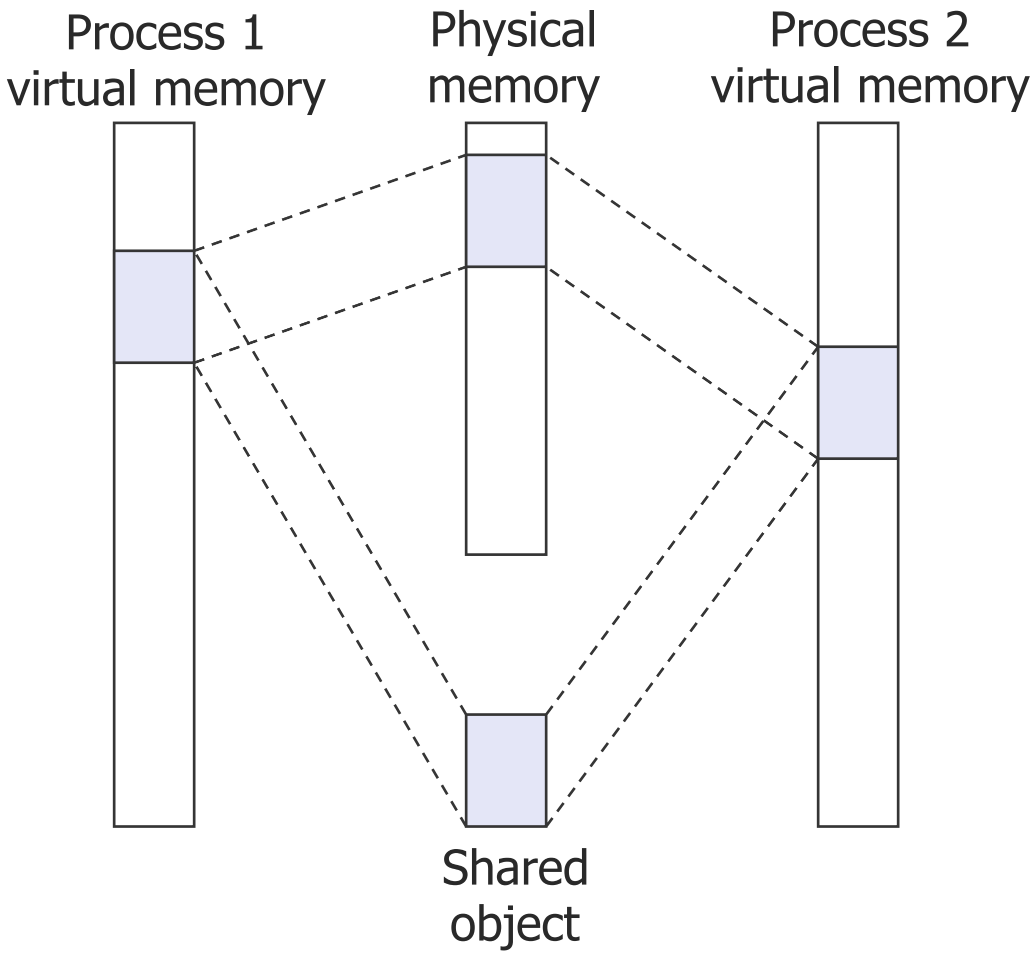


VM as a tool for sharing



- 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

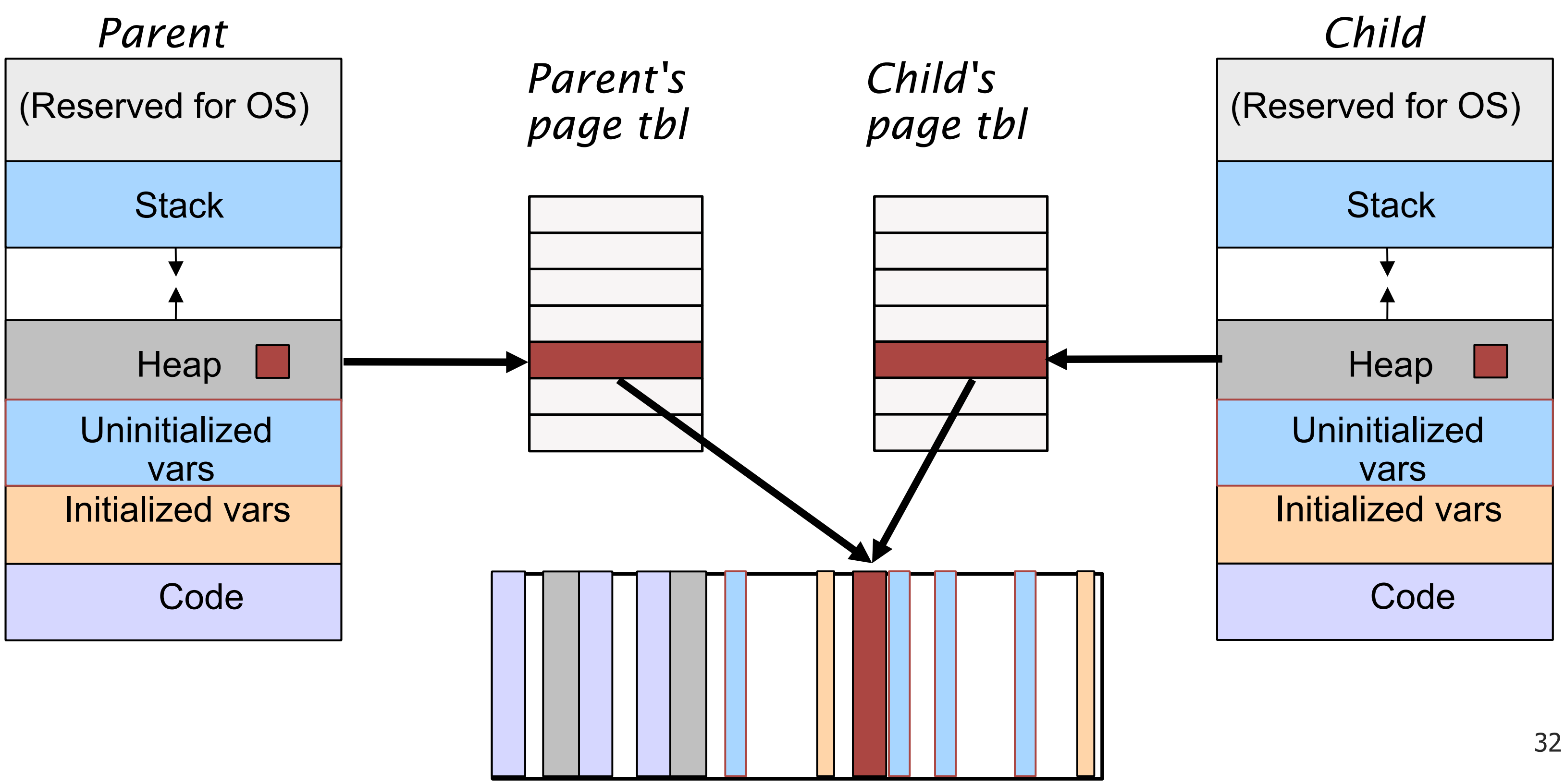
- `fork()` creates exact copy of a process
  - Lots more on this next week…

- When we fork a new process, does it make sense to make a copy of all of its memory?
  - Why or why not?

- What if the child process doesn't end up touching most of the memory the parent was using?
  - `exec()` replaces a process with a new one
  - Extreme example *and common case*: What happens if a process does an `exec()` immediately after `fork()`?

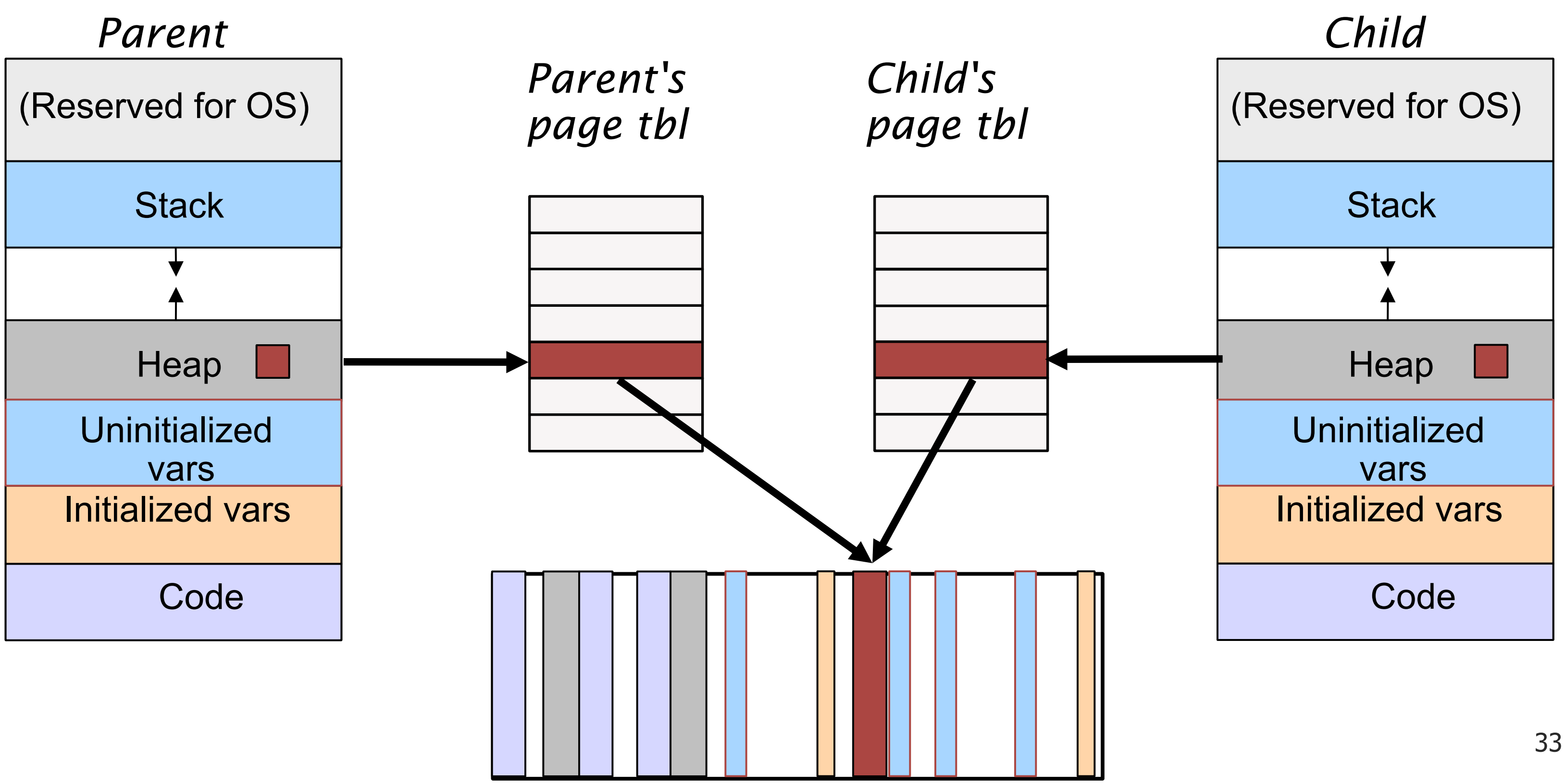
Copy-on-write

- Idea: Give the child process access to the same memory, but don't let it write to any of the pages directly!
  - 1) Parent forks a child process
  - 2) Child gets a copy of the parent's page tables
    - They point to the same physical frames!!!



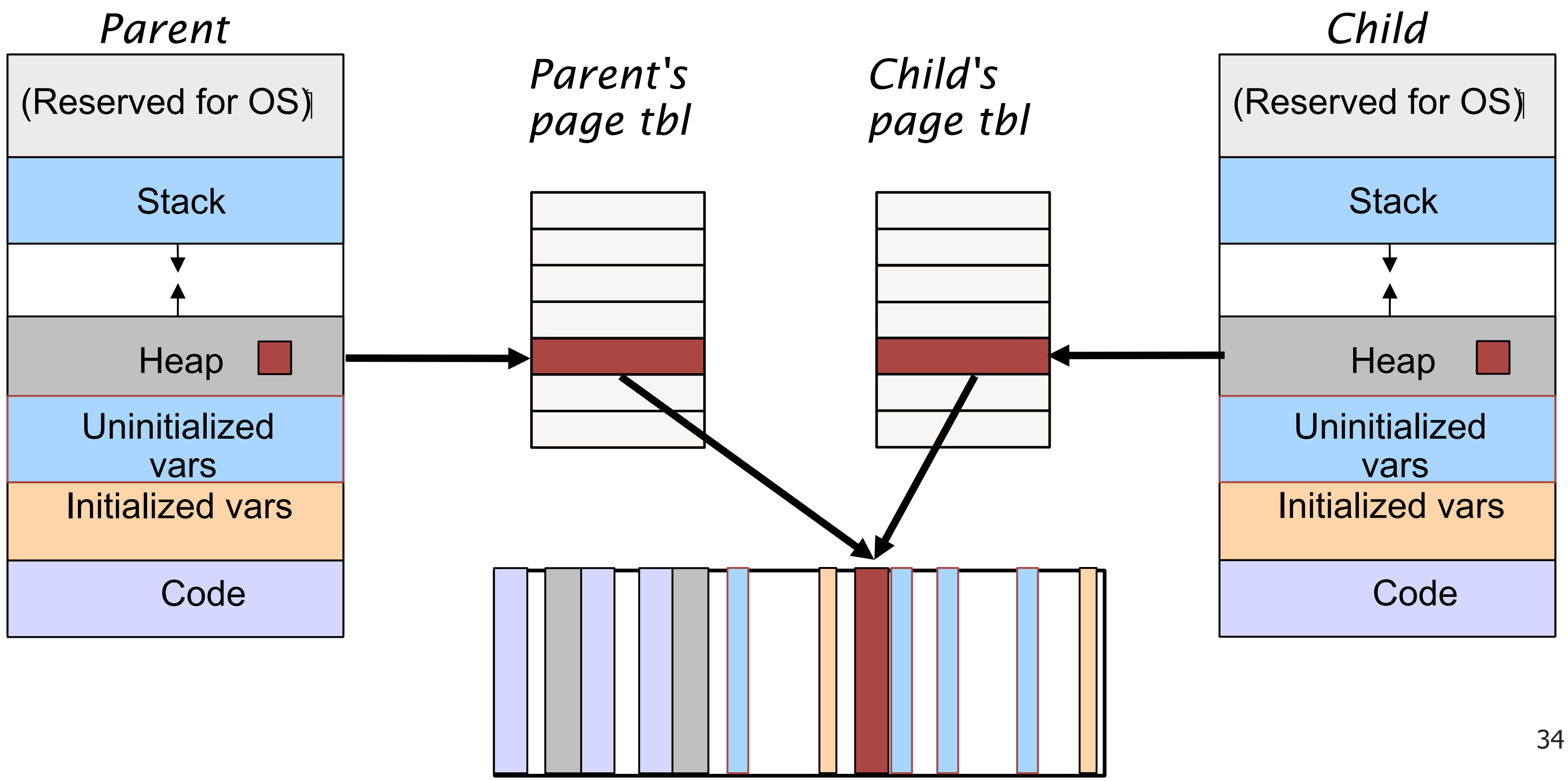
Copy-on-write

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



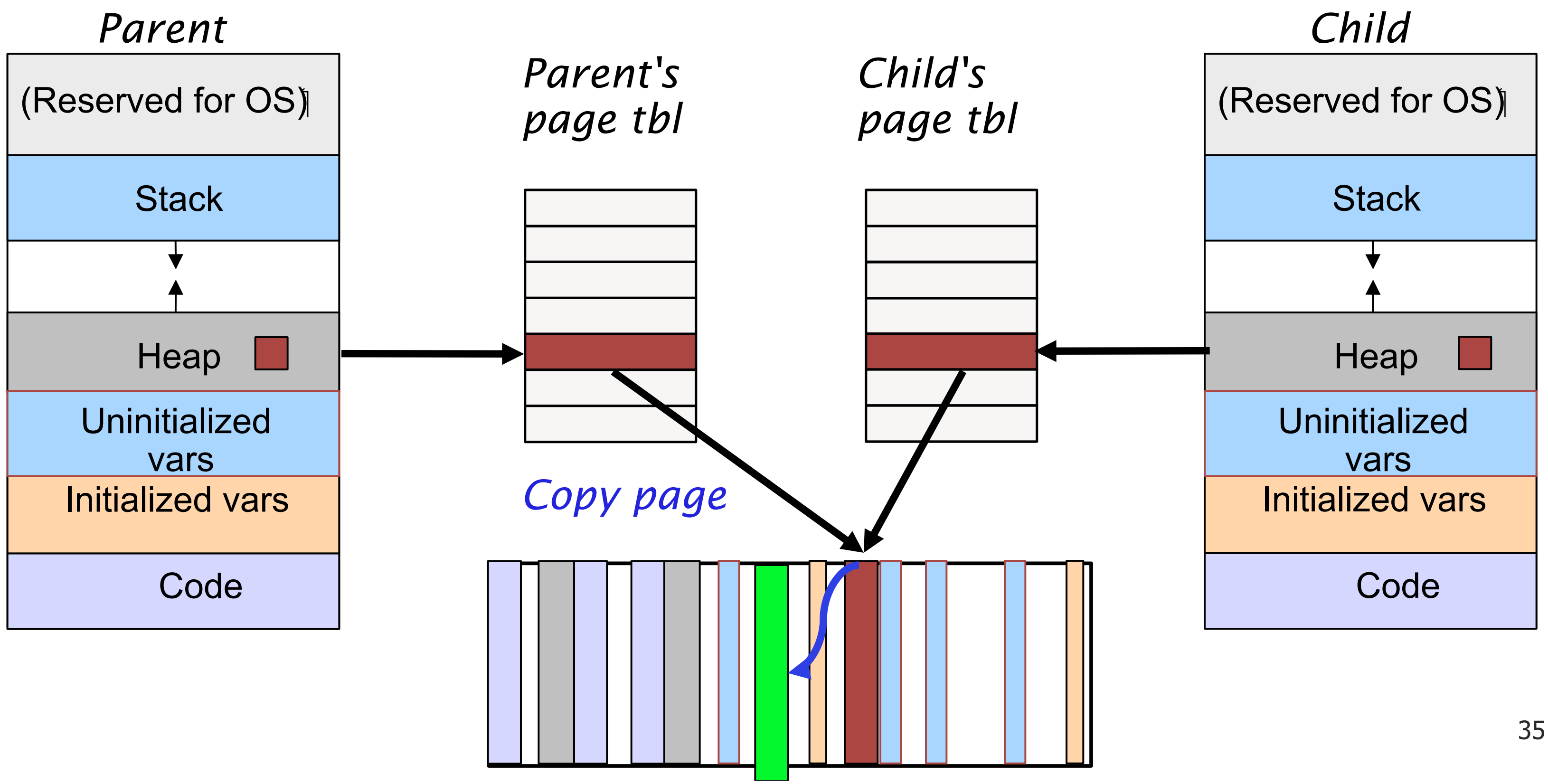
Copy-on-write

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



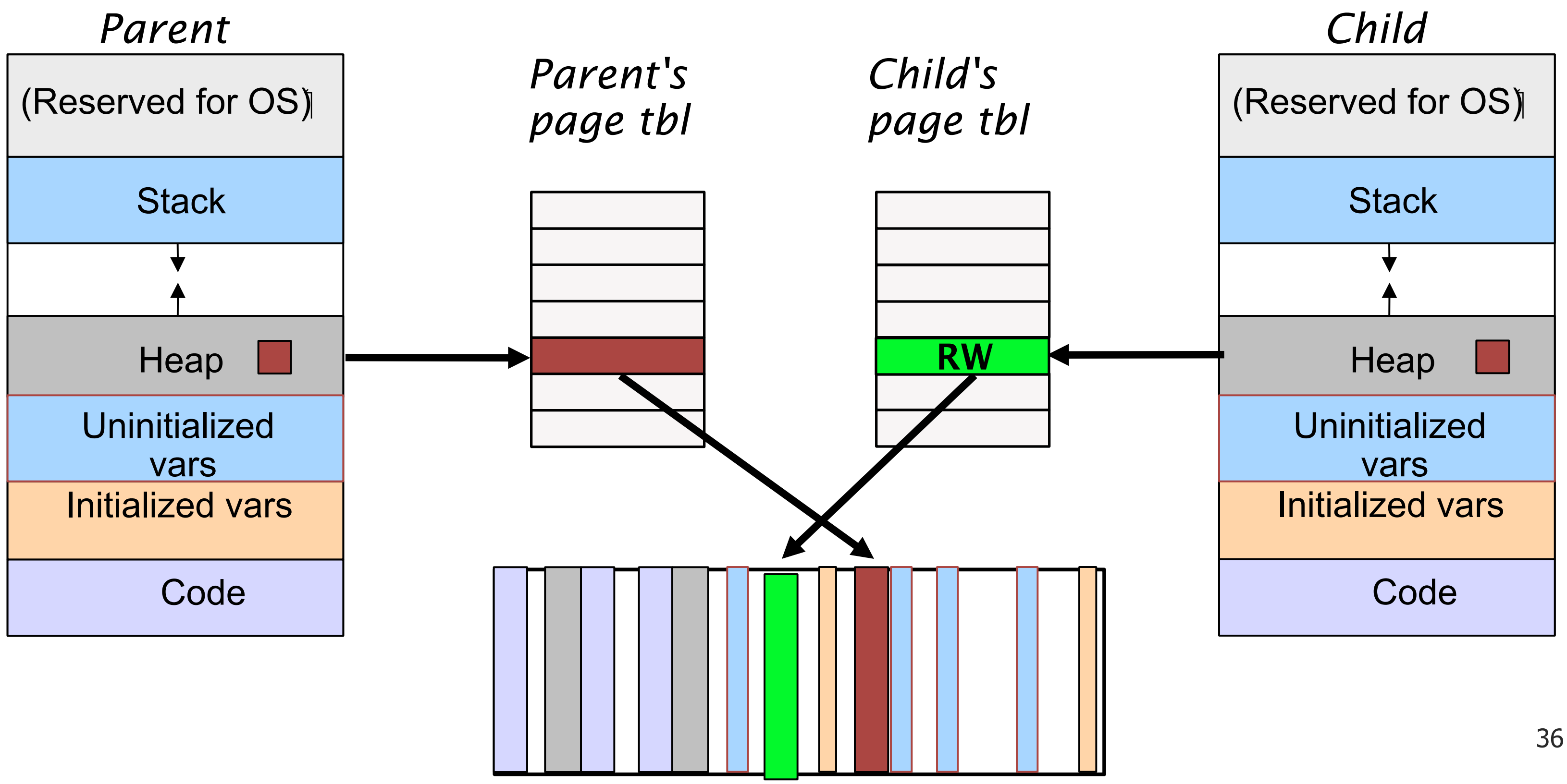
Copy-on-write

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



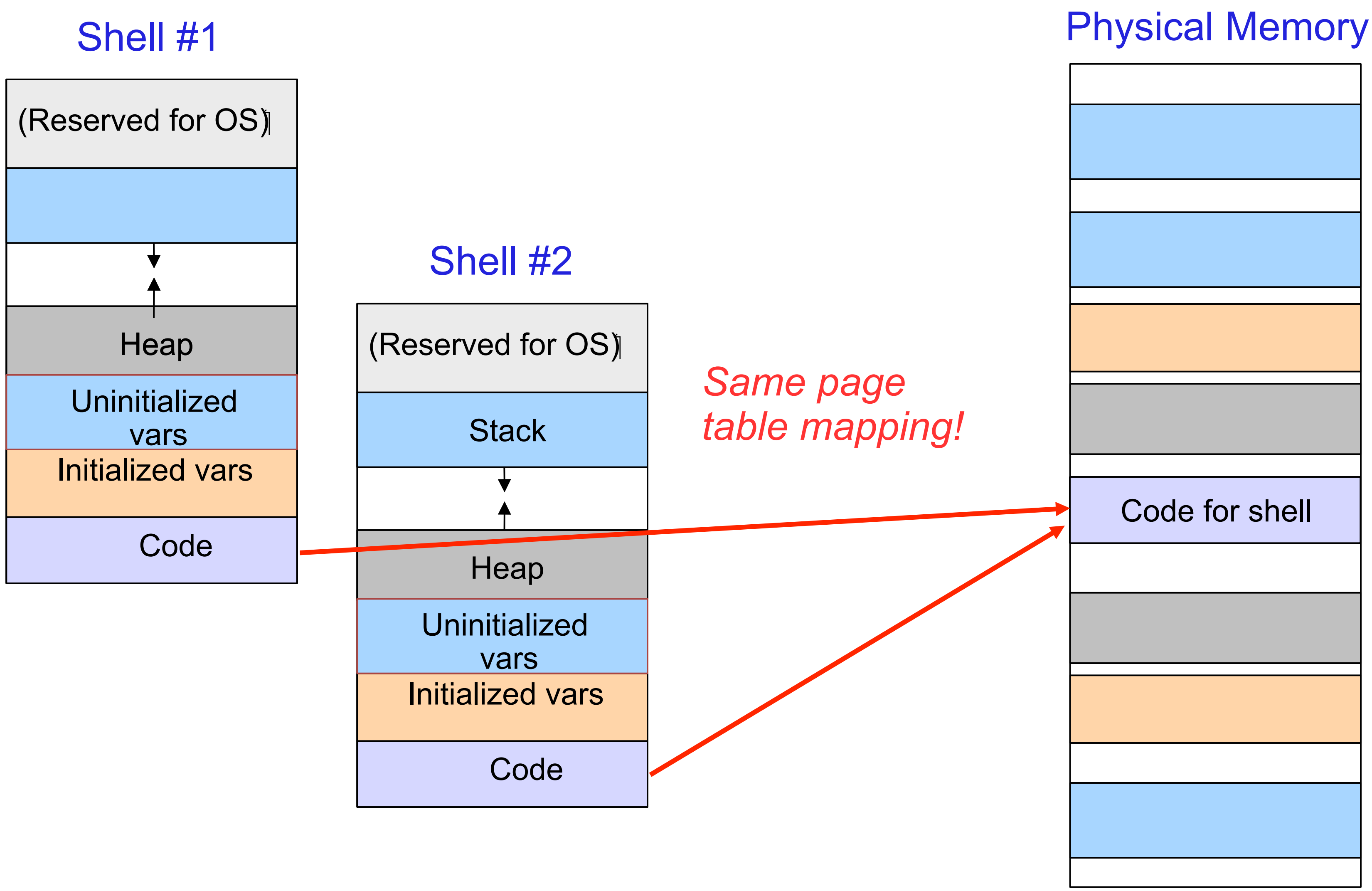
Copy-on-write

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



Another sharing example

- Can also share code segment



Benefits of sharing pages

- How much memory savings do we get from sharing pages across identical processes?
  - A lot! Use the "top" command...

Terminal — top — 88x26

```
Processes:  68 total, 2 running, 1 stuck, 65 sleeping... 246 threads   13:17:30
Load Avg:  0.75, 0.58, 0.52     CPU usage:  7.7% user, 17.9% sys, 74.4% idle
SharedLibs: num =  223, resident = 33.3M code, 4.61M data, 4.80M LinkEdit
MemRegions: num = 17413, resident =  208M + 11.0M private,  546M shared
PhysMem:   618M wired,  261M active,  130M inactive, 1010M used, 13.9M free
VM: 9.79G +  150M   635052(61) pageins, 455424(0) pageouts

  PID COMMAND      %CPU   TIME   #TH #PRTS #MREGS RPRVT  RSHRD  RSIZE  VSIZE
 3784 Grab         5.0%  0:00.51   3   126   159 2.23M+ 7.25M+ 16.8M+  216M+
 3781 less         0.0%  0:00.02   1    13    17  148K   304K   484K  26.7M
 3778 sh           0.0%  0:00.00   1     8    16 88.0K   608K   364K  27.1M
 3777 sh           0.0%  0:00.00   1    13    16 68.0K   608K   544K  27.1M
 3776 man          0.0%  0:00.01   1    13    16  184K   264K   460K  26.7M
 3752 bash         0.0%  0:00.01   1    14    16  228K   696K   816K  27.1M
 3751 login        0.0%  0:00.01   1    16    40  172K   380K   636K  26.9M
 3748 top         12.8%  0:23.16   1    25    20  704K   300K  1.14M  27.0M
 3725 bash         0.0%  0:00.02   1    14    16  228K   696K   812K  27.1M
 3724 login        0.0%  0:00.01   1    16    40  172K   380K   636K  26.9M
 3722 Terminal     0.2%  0:02.31   6    92   140 2.25M  11.1M  10.3M   218M
 3719 WinAppHelp   0.0%  0:00.05   1    57    95  716K  4.10M  3.00M   198M
 3713 mdimport     0.0%  0:00.90   4    68   119 1.59M  3.16M  4.64M  57.8M
 3675 iTunes       3.5%  6:51.76   9   193   370 7.12M  12.1M+ 10.2M   263M
 3670 Address Bo   0.0%  0:02.58   1    92   179 2.21M  5.56M  15.2M   216M
 3659 Mail         0.0%  0:59.65   8   172   415 25.3M  10.9M+ 27.2M   258M
 3084 Skype        0.7% 17:20.32  18   240   452 23.9M  8.65M+ 20.0M   304M
  655 vfstool      0.0%  0:00.07   2    14    29  120K   308K   256K  32.1M
```

Summary

- Paging implementation
  - Basics: get page off disk if necessary (*page fault*) and then map virtual to physical address
  - Problem: Mapping requires extra memory access (solution?)
  - Problem: Page table can get huge (solution?)

- Paging enables flexible use of memory
  - Protection
  - Sharing (e.g., copy-on-write defers writes as long as possible)
  - Caching
    - Q: How do I choose which page to evict when swapping?