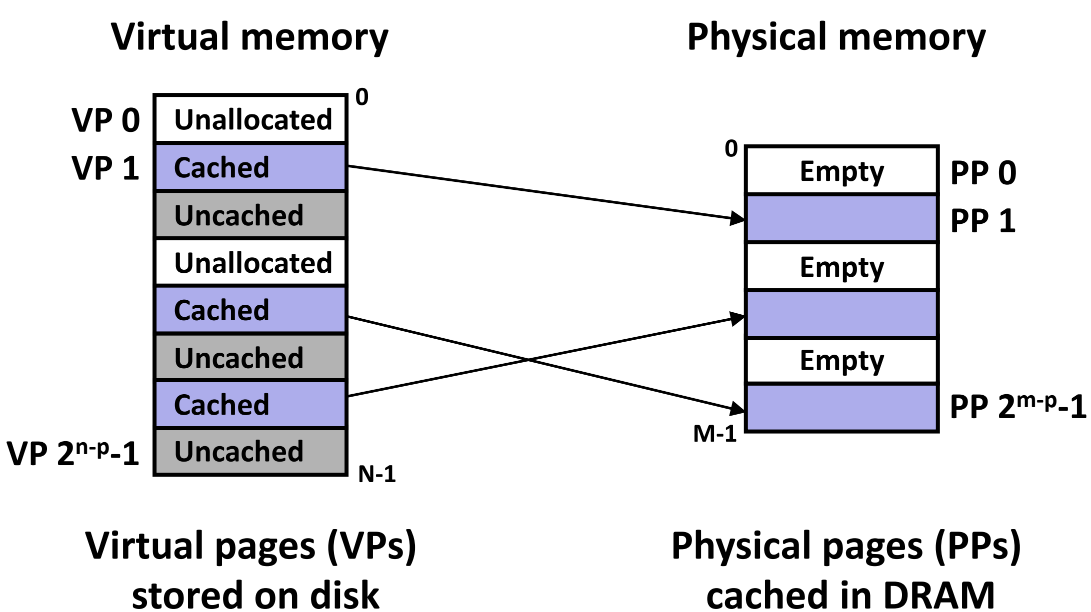
Section 9: Virtual Memory (VM)

- Overview and motivation
- Indirection
- VM as a tool for caching
- Memory management/protection and address translation
- Virtual memory example

VM and the Memory Hierarchy

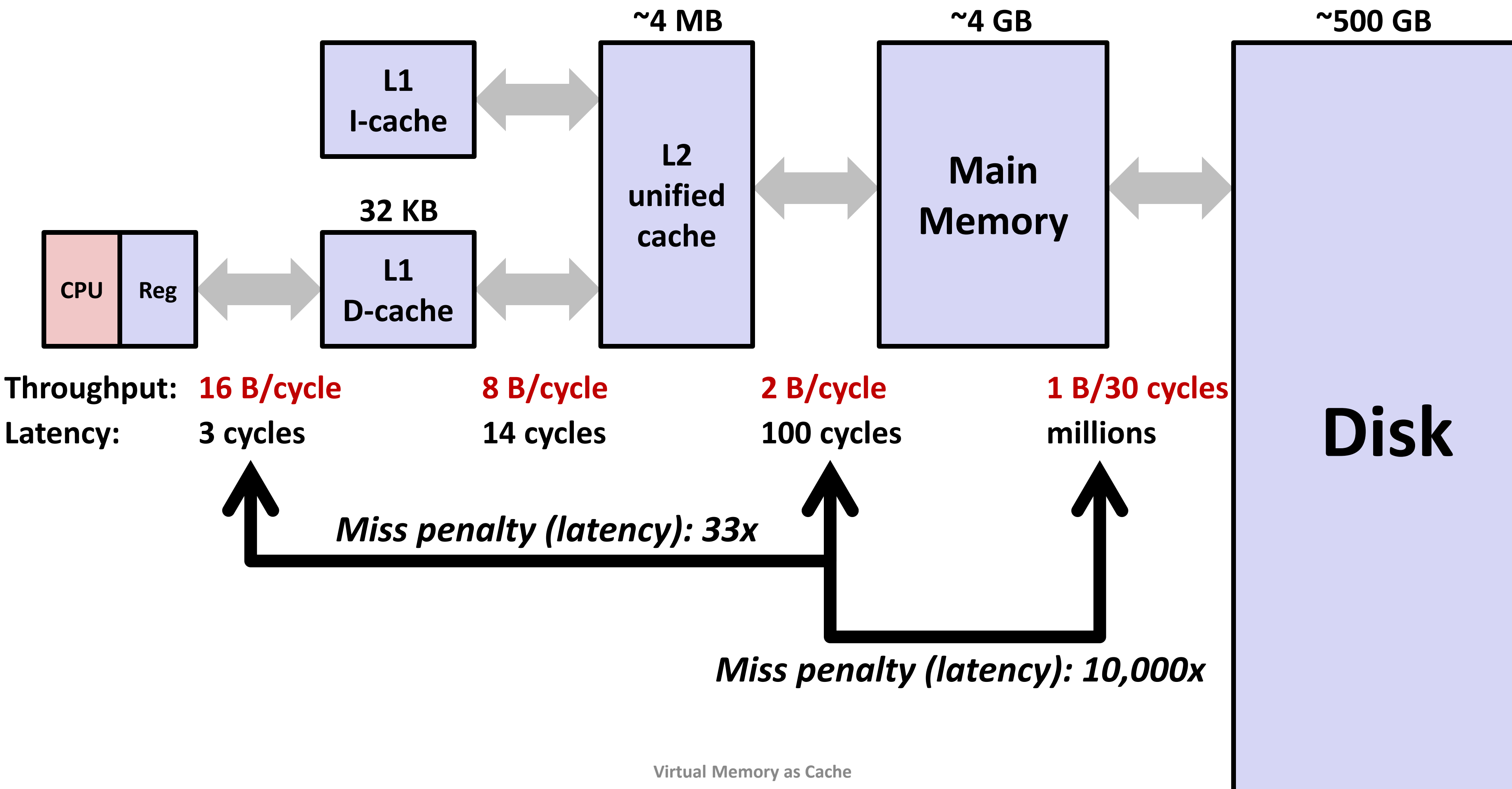
- Think of virtual memory as an array of $N = 2^n$ contiguous bytes stored *on a disk*
- Then physical main memory (DRAM) is used as a *cache* for the virtual memory array
  - The cache blocks are called *pages* (size is $P = 2^p$ bytes)



Memory Hierarchy: Core 2 Duo

Not drawn to scale

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)

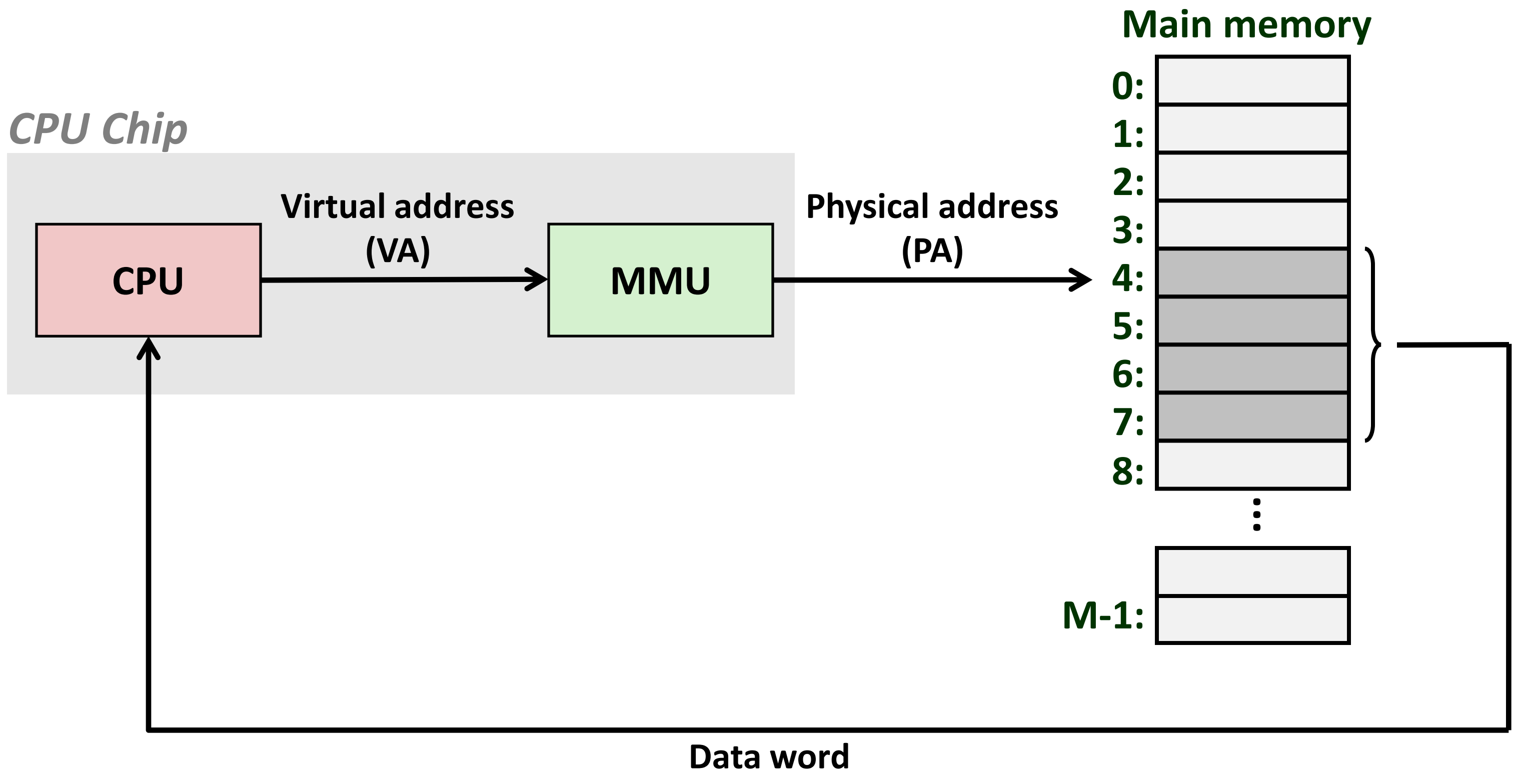
- **Consequences?**
  - Block size?
  - Associativity?
  - Write-through or write-back?

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

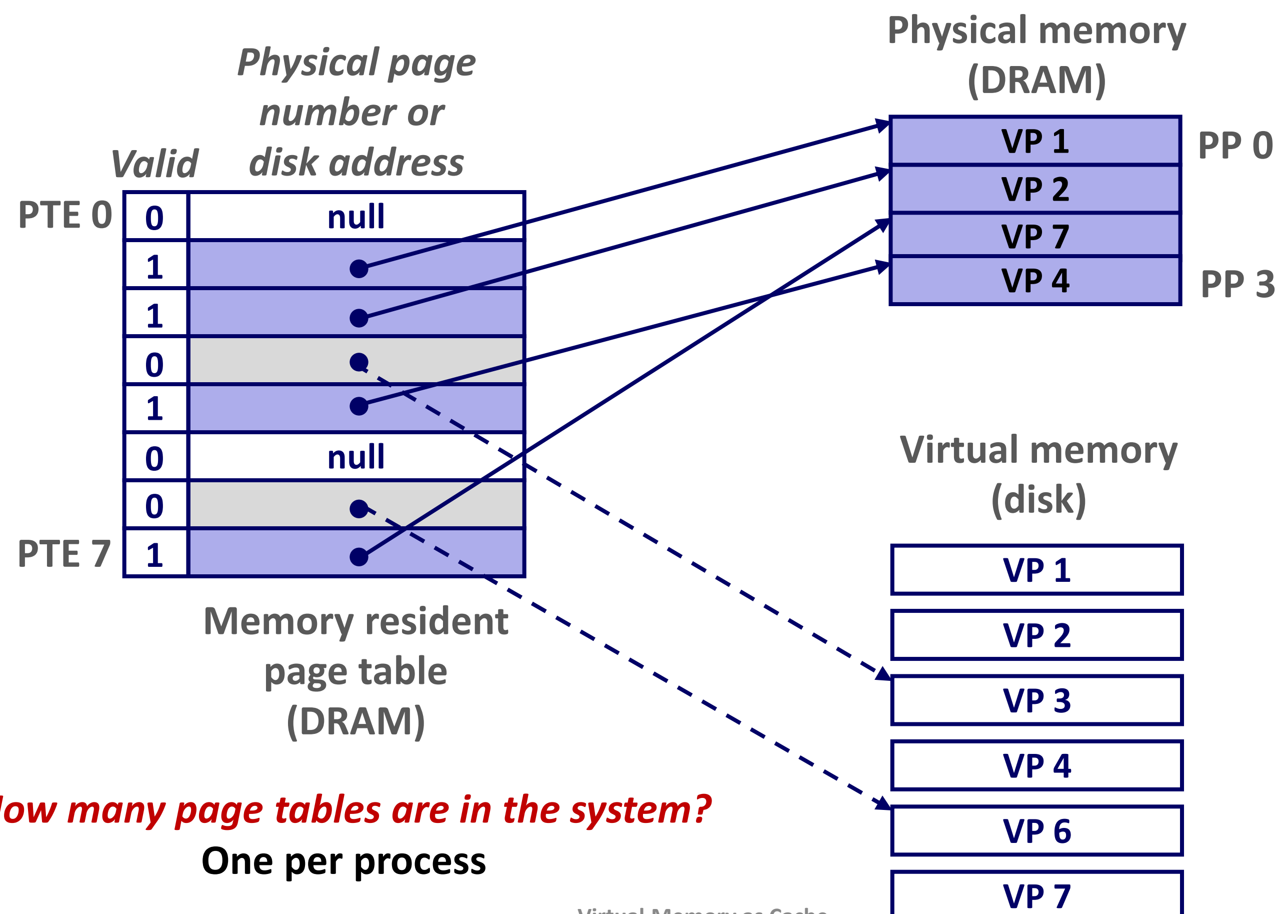
Indexing into the "DRAM Cache"



How do we perform the VA -> PA translation?

Address Translation: Page Tables

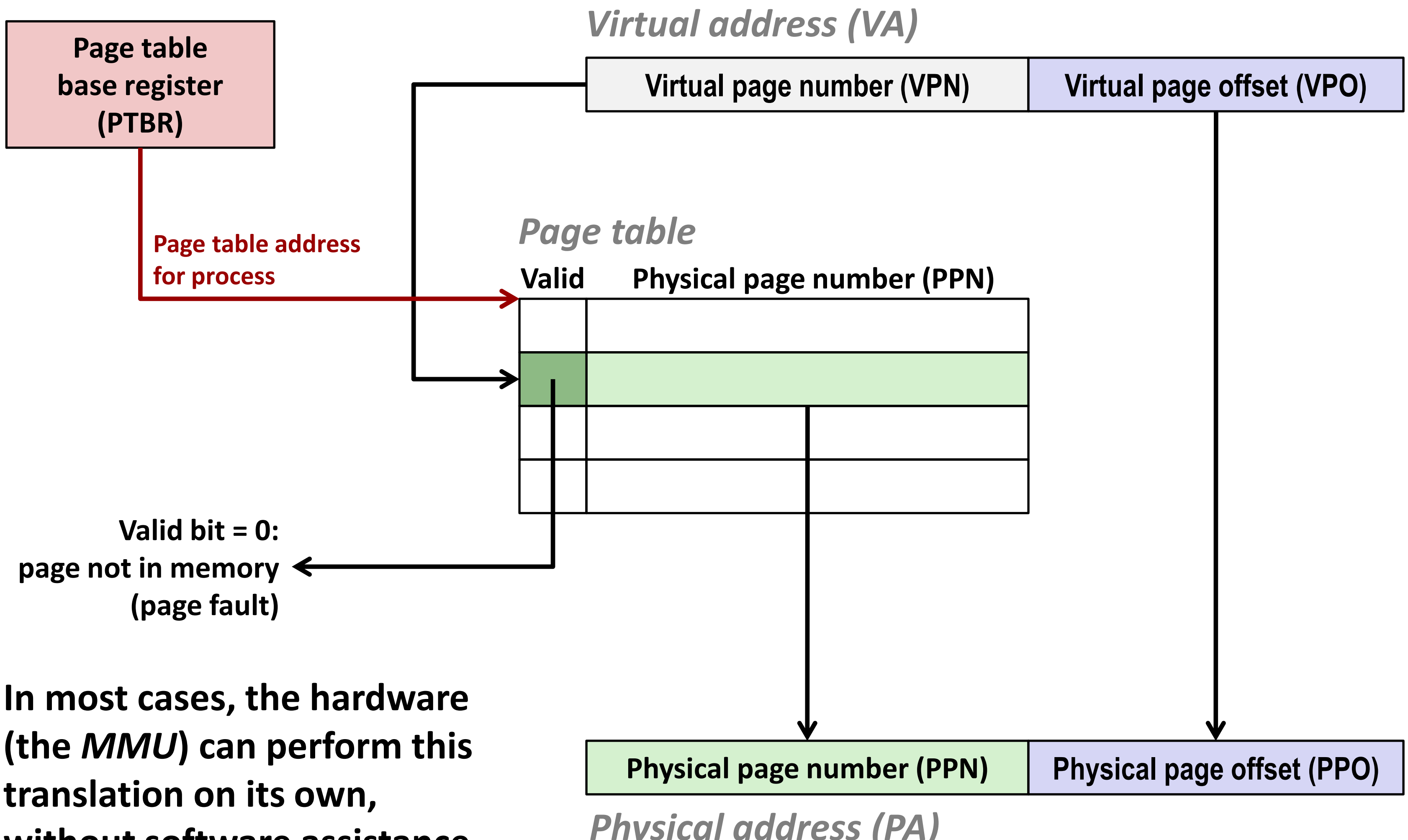
- A ***page table*** (PT) is an array of ***page table entries*** (PTEs) that maps virtual pages to physical pages.



How many page tables are in the system?

One per process

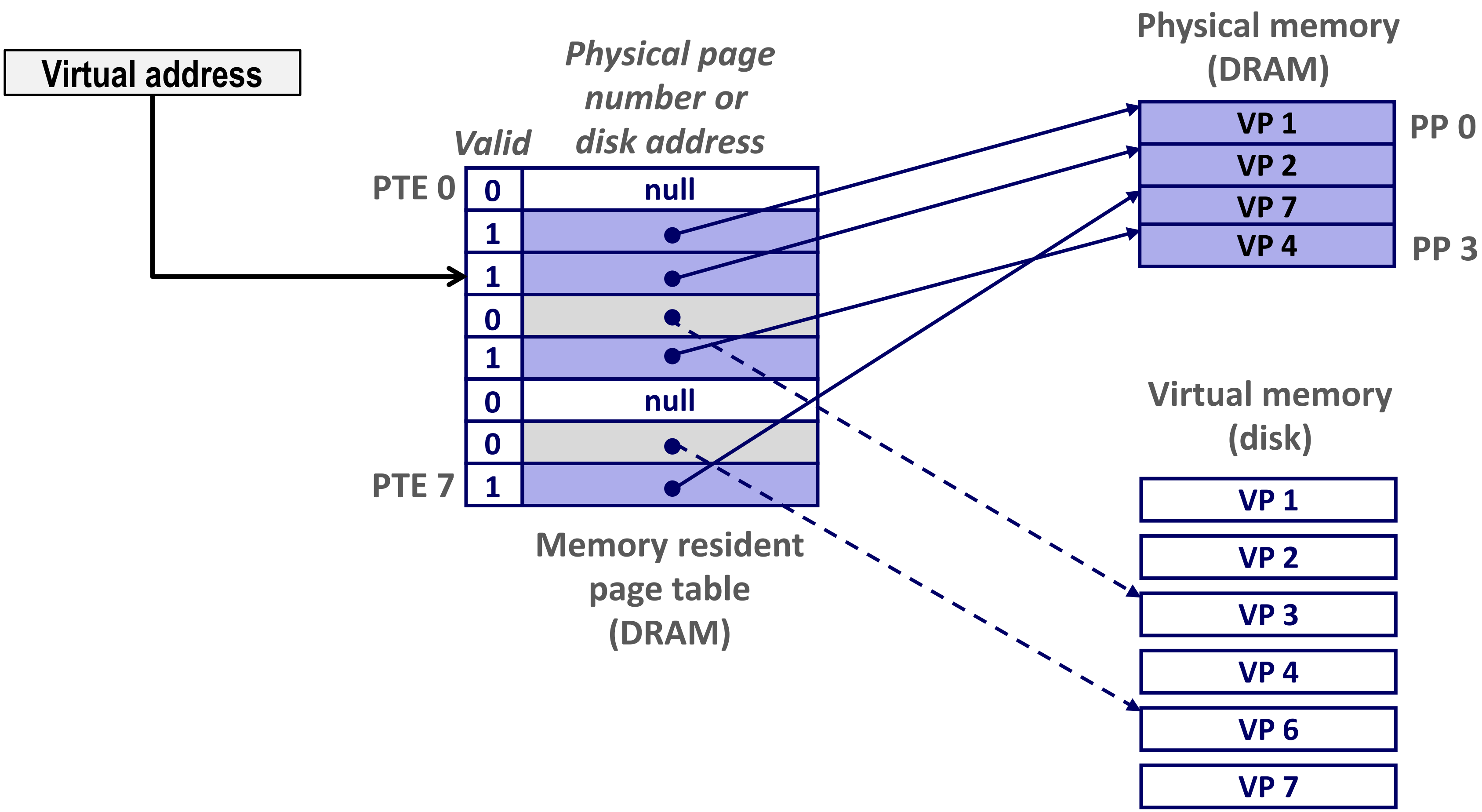
Address Translation With a Page Table



In most cases, the hardware (the *MMU*) can perform this translation on its own, without software assistance

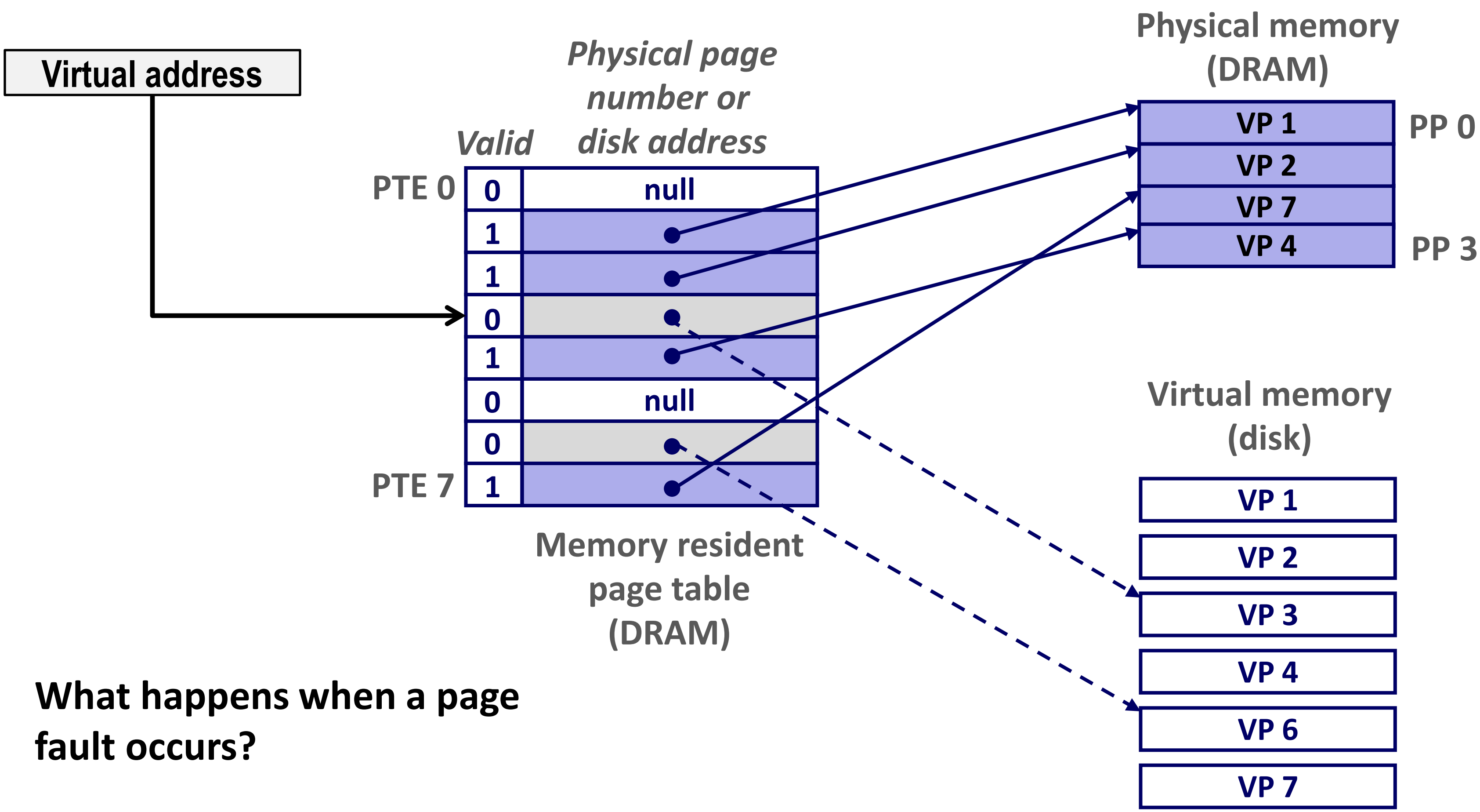
Page Hit

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



Page Fault

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



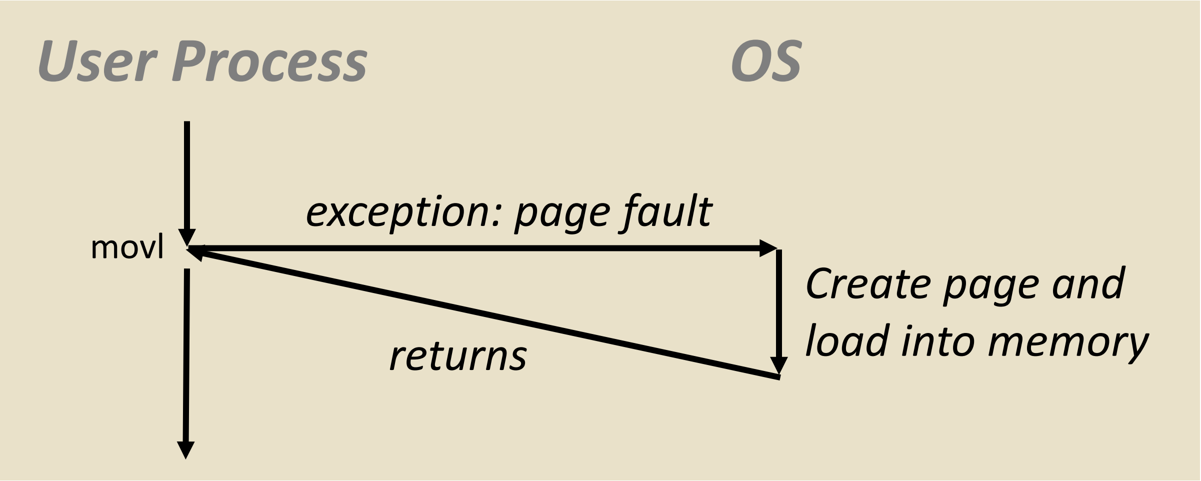
What happens when a page fault occurs?

Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user's memory is currently on disk

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

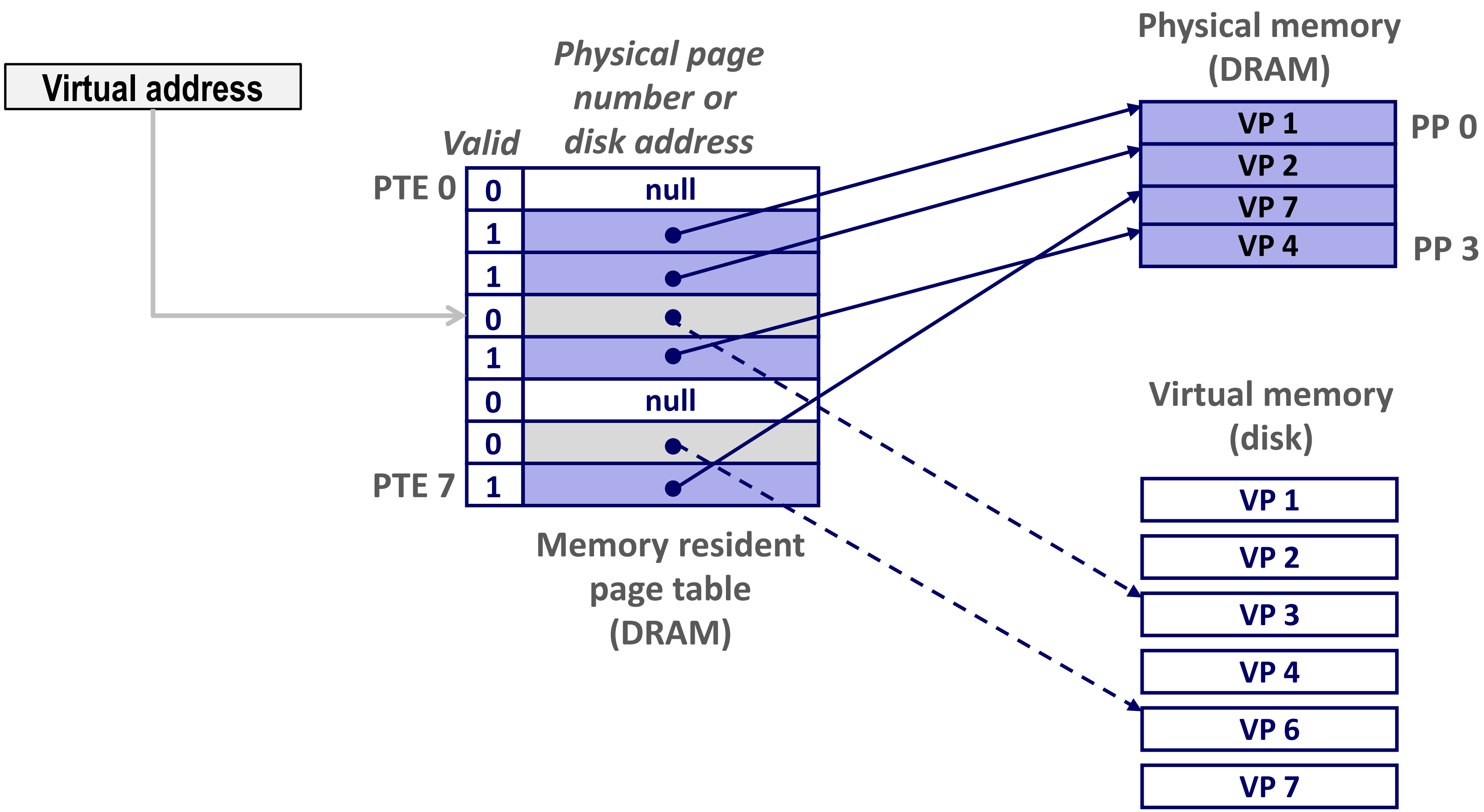
```
80483b7:        c7 05 10 9d 04 08 0d    movl    $0xd,0x8049d10
```



- Page handler must load page into physical memory
- Returns to faulting instruction: **`mov`** is executed again!
- Successful on second try

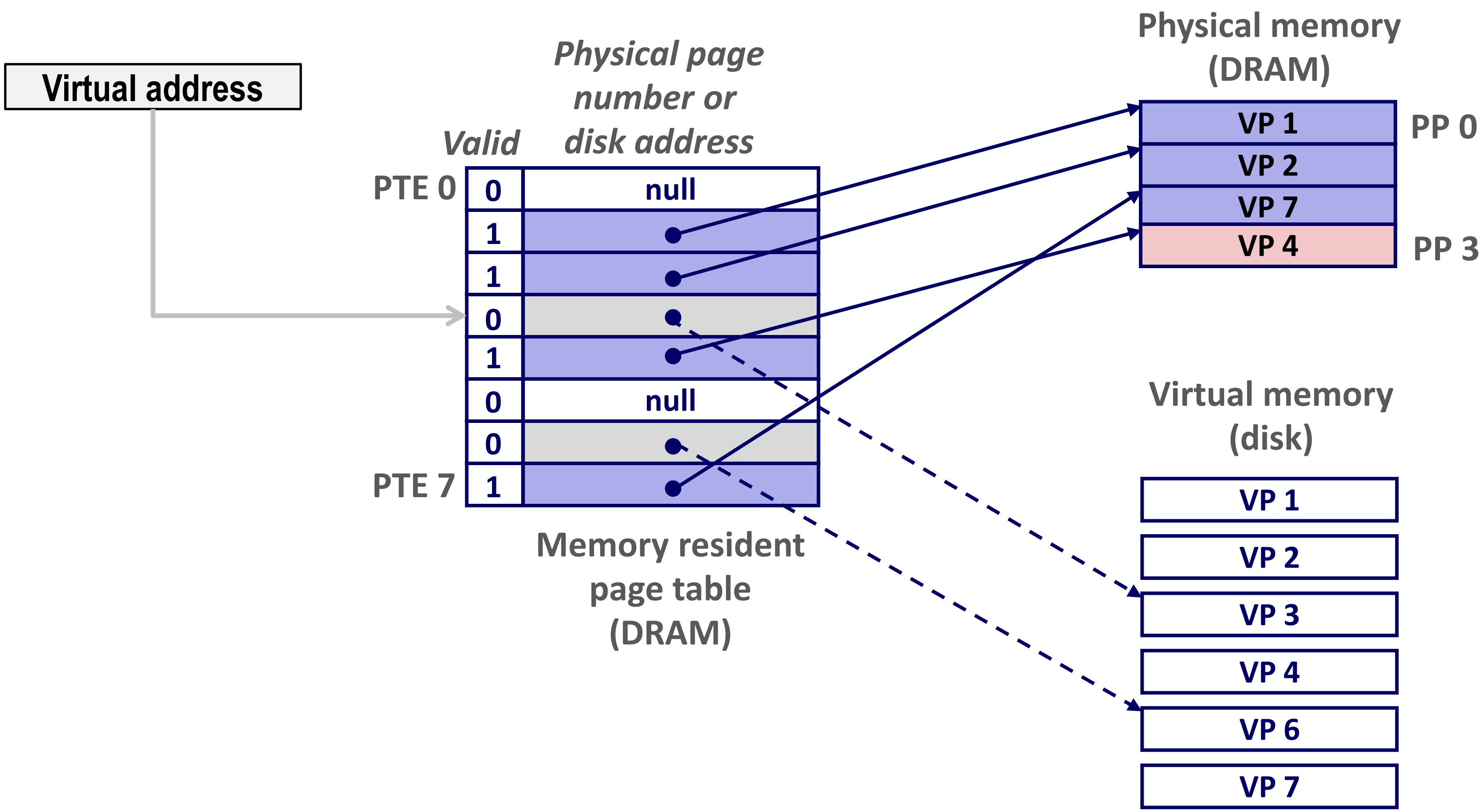
Handling Page Fault

- Page miss causes page fault (an exception)



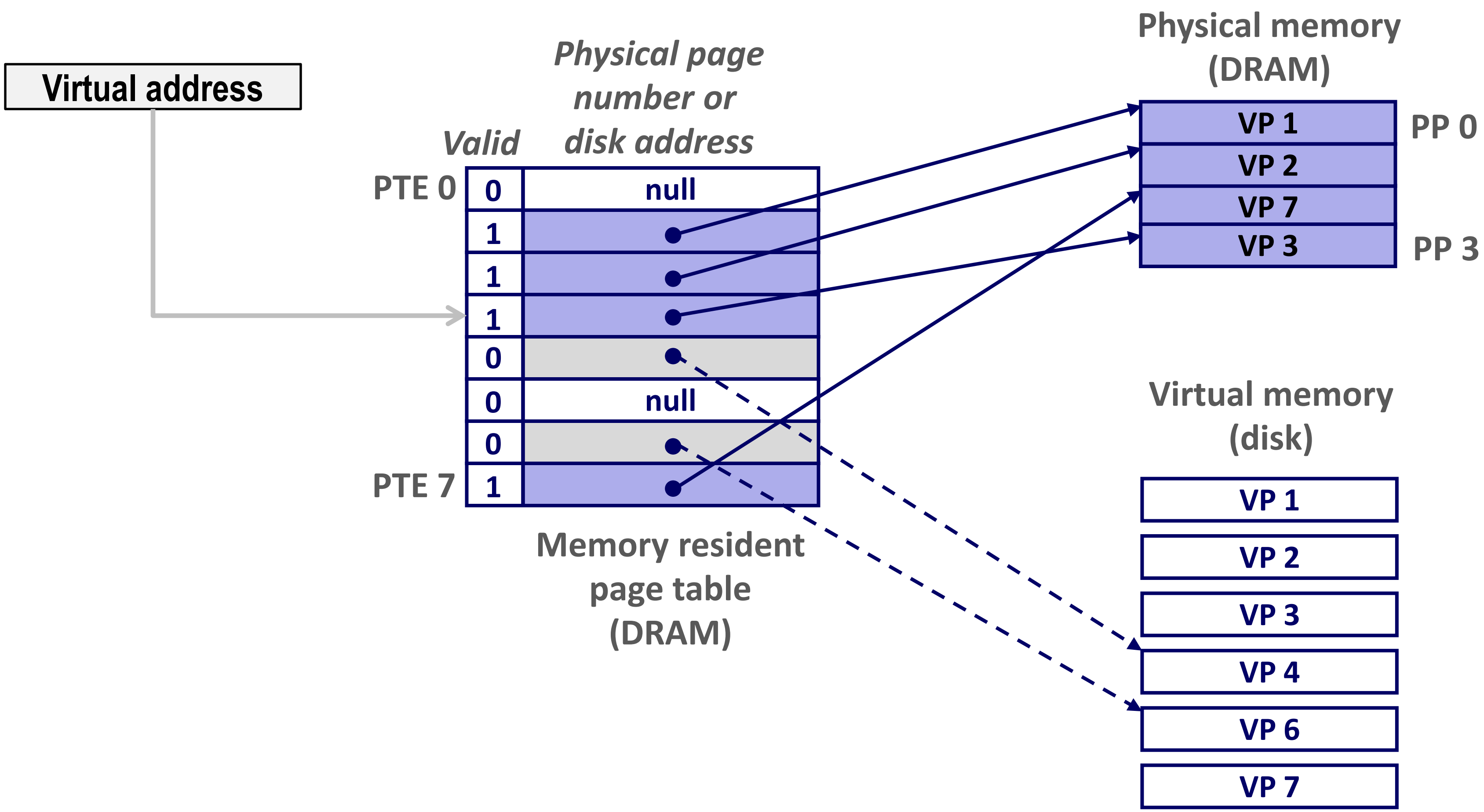
Handling Page Fault

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



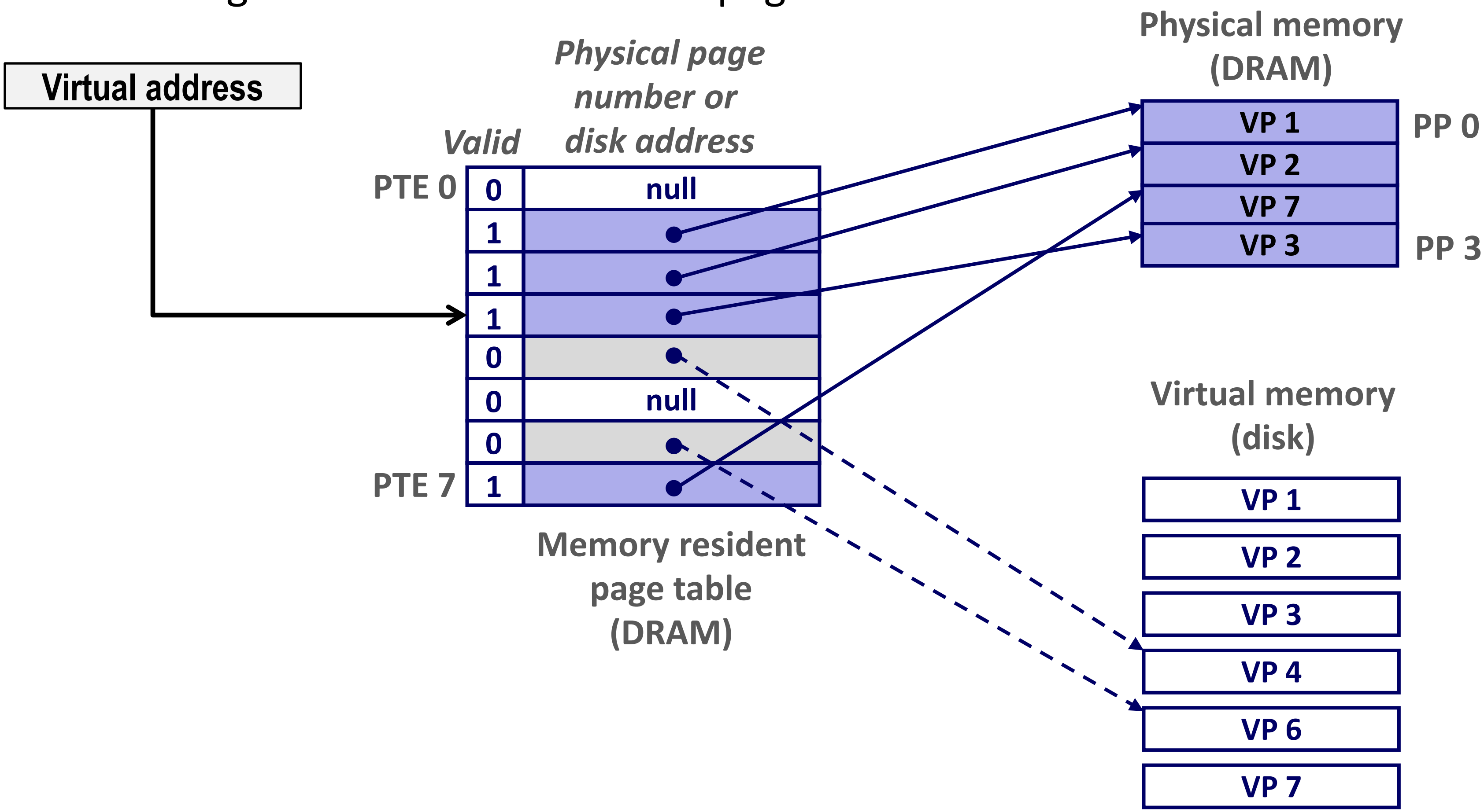
Handling Page Fault

- 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)
- Offending instruction is restarted: page hit!



Why does it work? Locality

- **Virtual memory works well 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***
  - 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