# CS33 Discussion

Week 9

# A Simple Case

- Run two programs on different cores of the same computer simultaneously
  - Any order is possible!!!

```
MOV $0x1234, %eax
MOV $0x0ff, (%eax)
MOV %ebx, (%eax)
%ebx = ?
```

```
MOV $0x1234, %eax
MOV $0x0cc, (%eax)
MOV %ebx, (%eax)
%ebx = ?
```

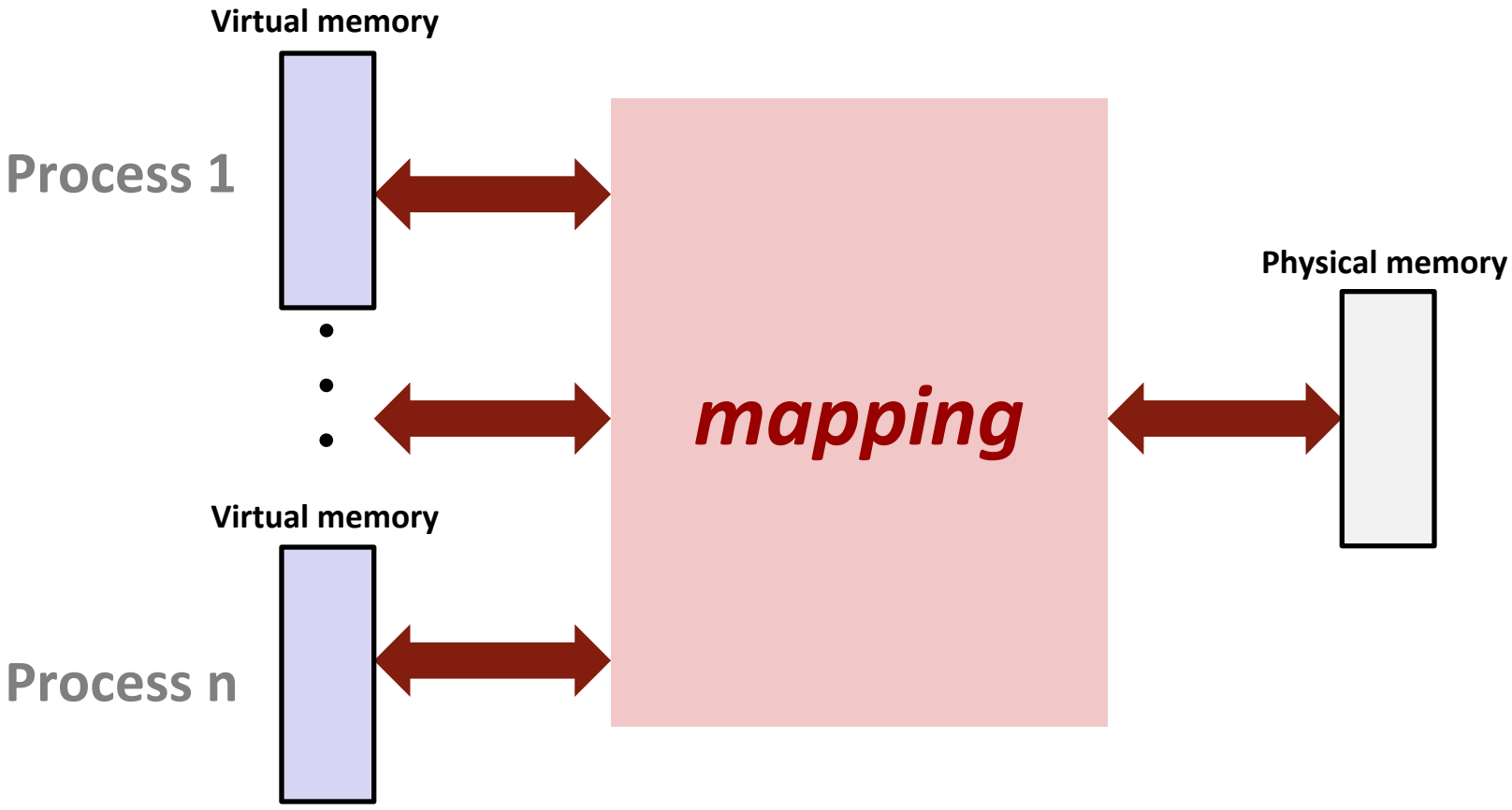
**No matter what the actual execution flow is, the left ebx = 0xff, the right ebx = 0xcc**

# Why no interferences at all?

- Multicore Processor
  - Each core has its own REGISTER FILE
    - Different physical %ebx-s
- But there is only one main memory!
  - Main memory and Disk are shared by all cores
  - The access addresses are exactly the same
    - %eax <= 0x1234  %eax <= 0x1234

**What you have seen is NOT what has actually been done**

# Solution: Add a level of indirection




- Each process gets its own private memory space
- Addresses all of the previous problems

# Memory Access Ideal/Reality

- It appears that…
  - Mov $0x0ff, (%eax) => Store 0xff to Memory Address 0x1234
  - Mov $0x0cc, (%eax) => Store 0xcc to Memory Address 0x1234
  - Address Conflict! Write twice.
- Actually…
  - 0x1234 => Address Translation Table P1 => 0xff1234
  - 0x1234 => Address Translation Table P2 => 0x801234
  - Totally unrelated to each other
- The address a program sent is not the address the memory received

# Address Spaces

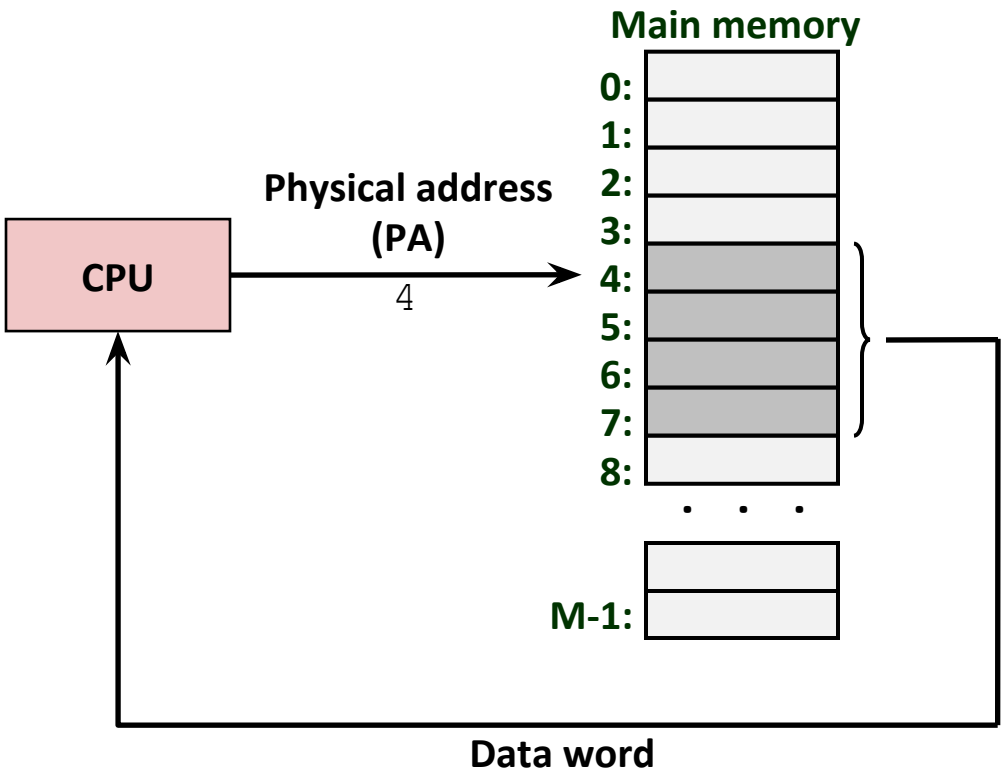
- **Linear address space:** Ordered set of contiguous non-negative integer addresses:
  {0, 1, 2, 3 … }

- **Virtual address space:** Set of $N = 2^n$ virtual addresses
  {0, 1, 2, 3, …, N-1}

- **Physical address space:** Set of $M = 2^m$ physical addresses
  {0, 1, 2, 3, …, M-1}

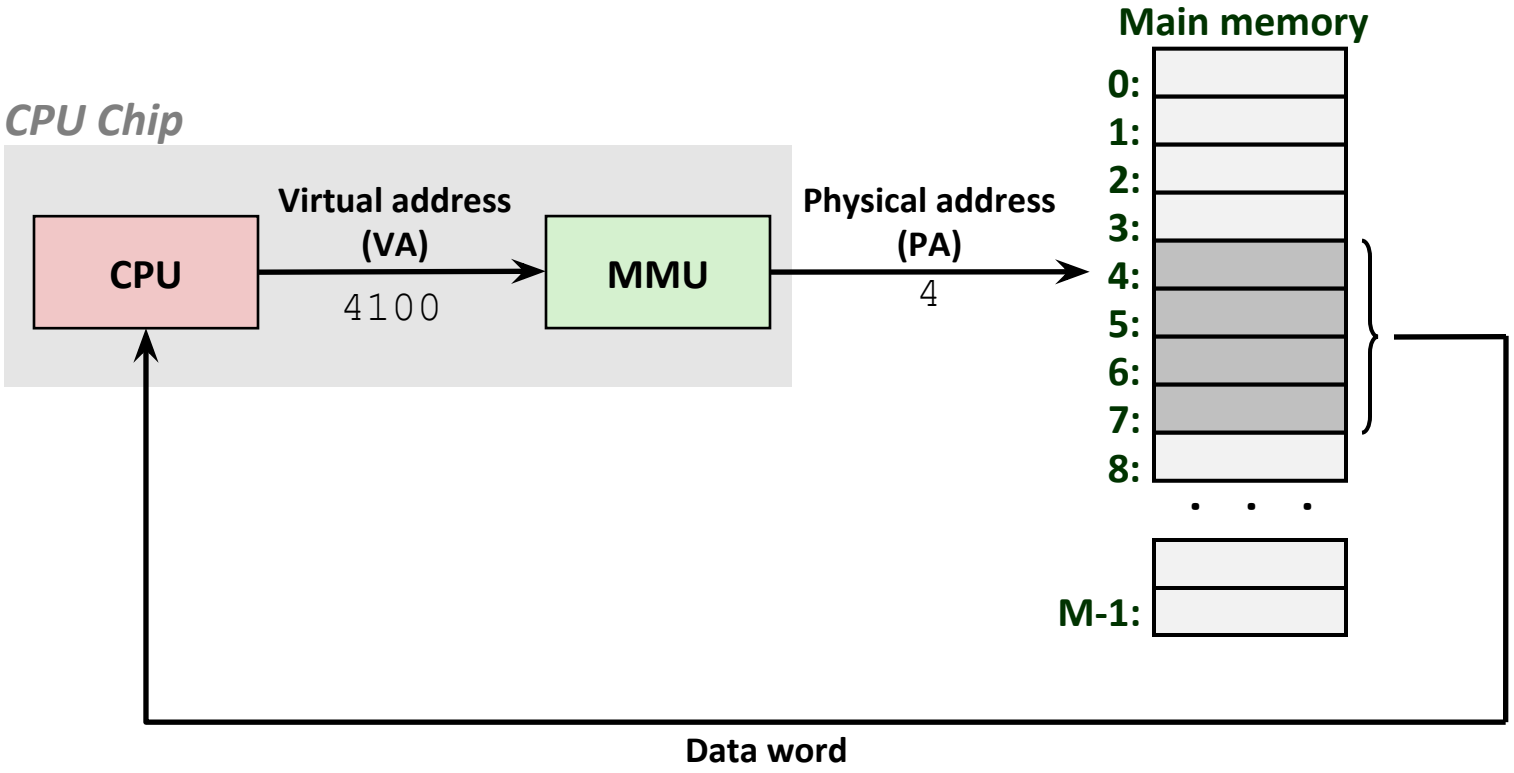
- **Clean distinction between data (bytes) and their attributes (addresses)**
- **Each datum can now have multiple addresses**
- **Every byte in main memory:**
  **one physical address, one (or more) virtual addresses**

# A System Using Physical Addressing




- Used in some “simple” systems, like embedded microcontrollers in cars, elevators, and digital picture frames

# A System Using Virtual Addressing



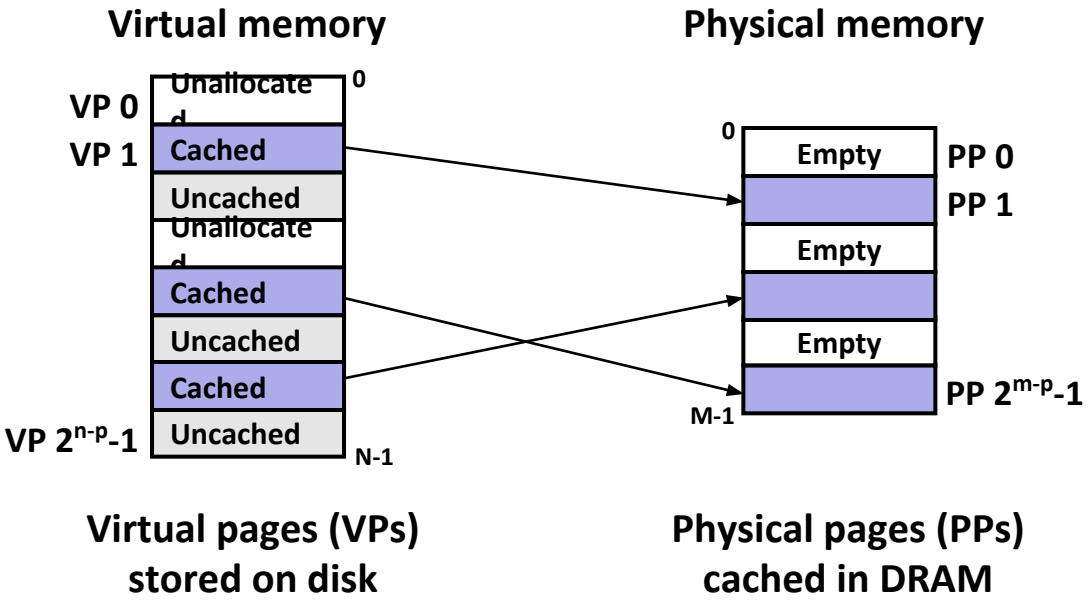

- Used in all modern servers, desktops, and laptops
- One of the great ideas in computer science

# Virtual Memory

- Cache
- Memory Management
- Memory Protection

# (1) VM as a Tool for Caching

- ***Virtual memory*** **is an array of N contiguous bytes that may be stored on disk**
- **The contents of the array on disk are cached in *physical memory* (*DRAM cache*)**
  - These cache blocks are called *pages* (size is P = $2^p$ bytes)



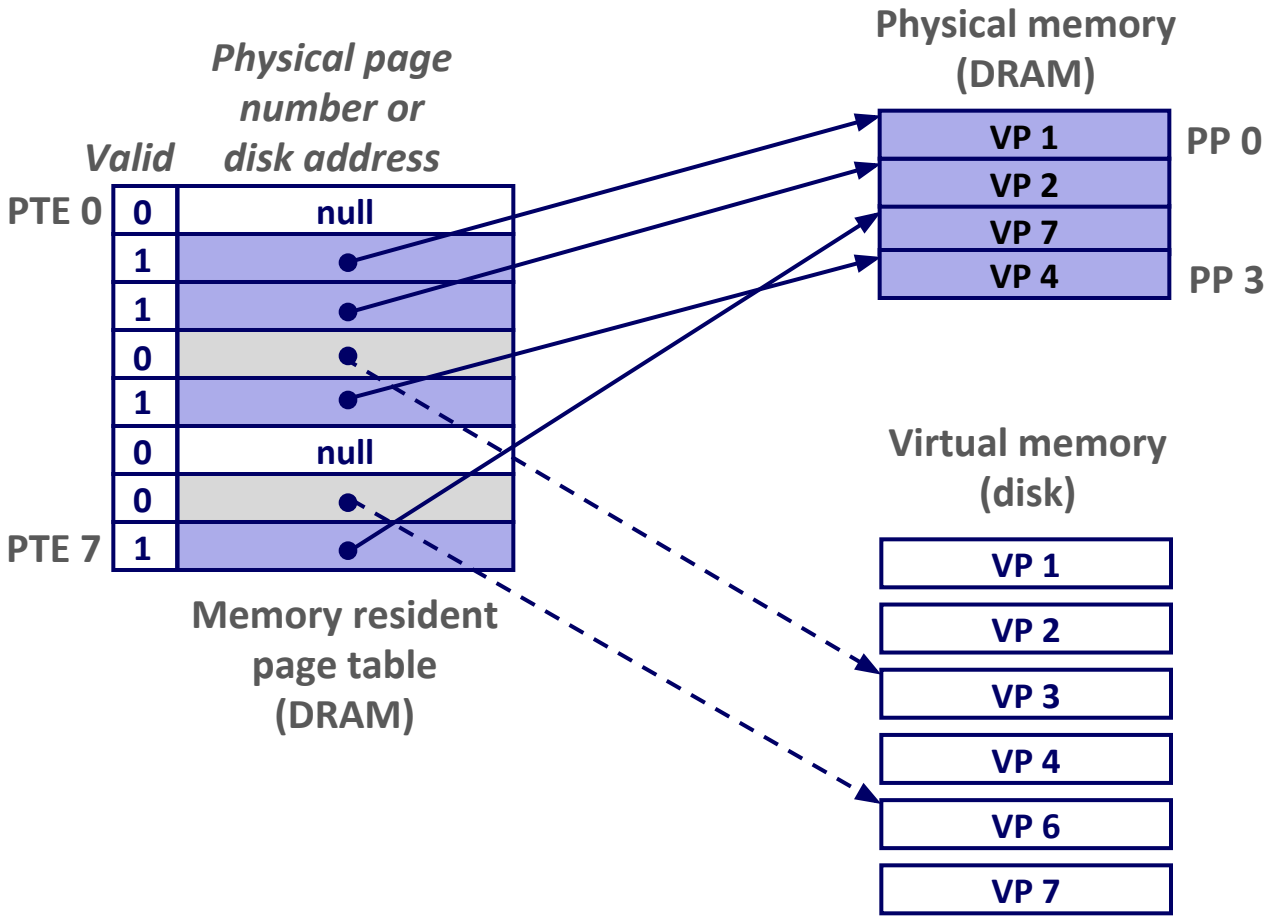
# DRAM Cache Organization

- **DRAM cache organization driven by the enormous miss penalty**
  - DRAM is about ***10x*** slower than SRAM
  - Disk is about ***10,000x*** slower than DRAM

- **Consequences**
  - Large page (block) size: typically 4-8 KB, sometimes 4 MB
  - Fully associative
    - Any virtual page (VP) can be placed in any physical page (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

# Enabling data structure: Page Table

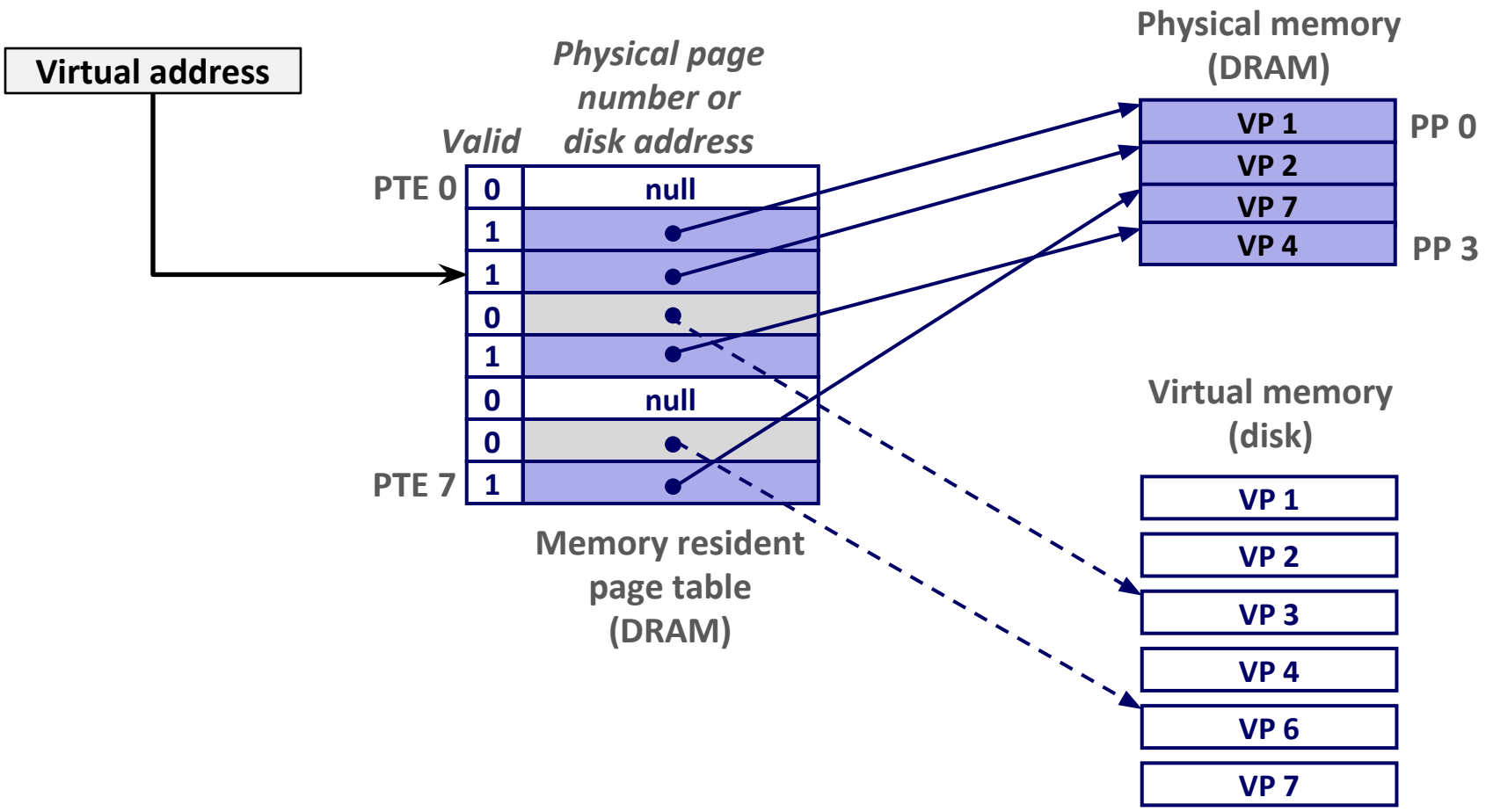
- **A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages (recall cache tags)**
  - Per-process kernel data structure in DRAM



# /proc/<procID>/maps

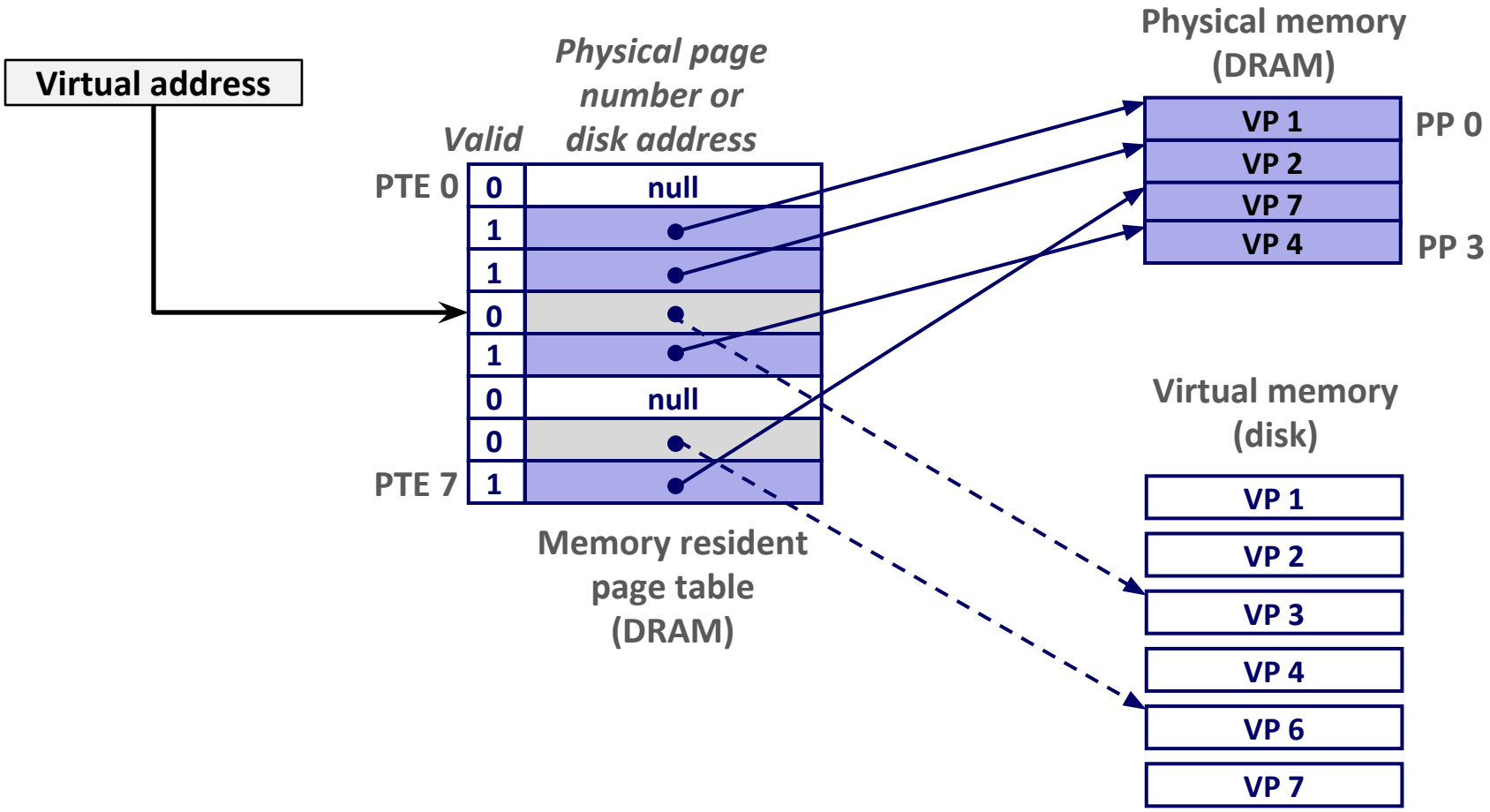
# Page Hit

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



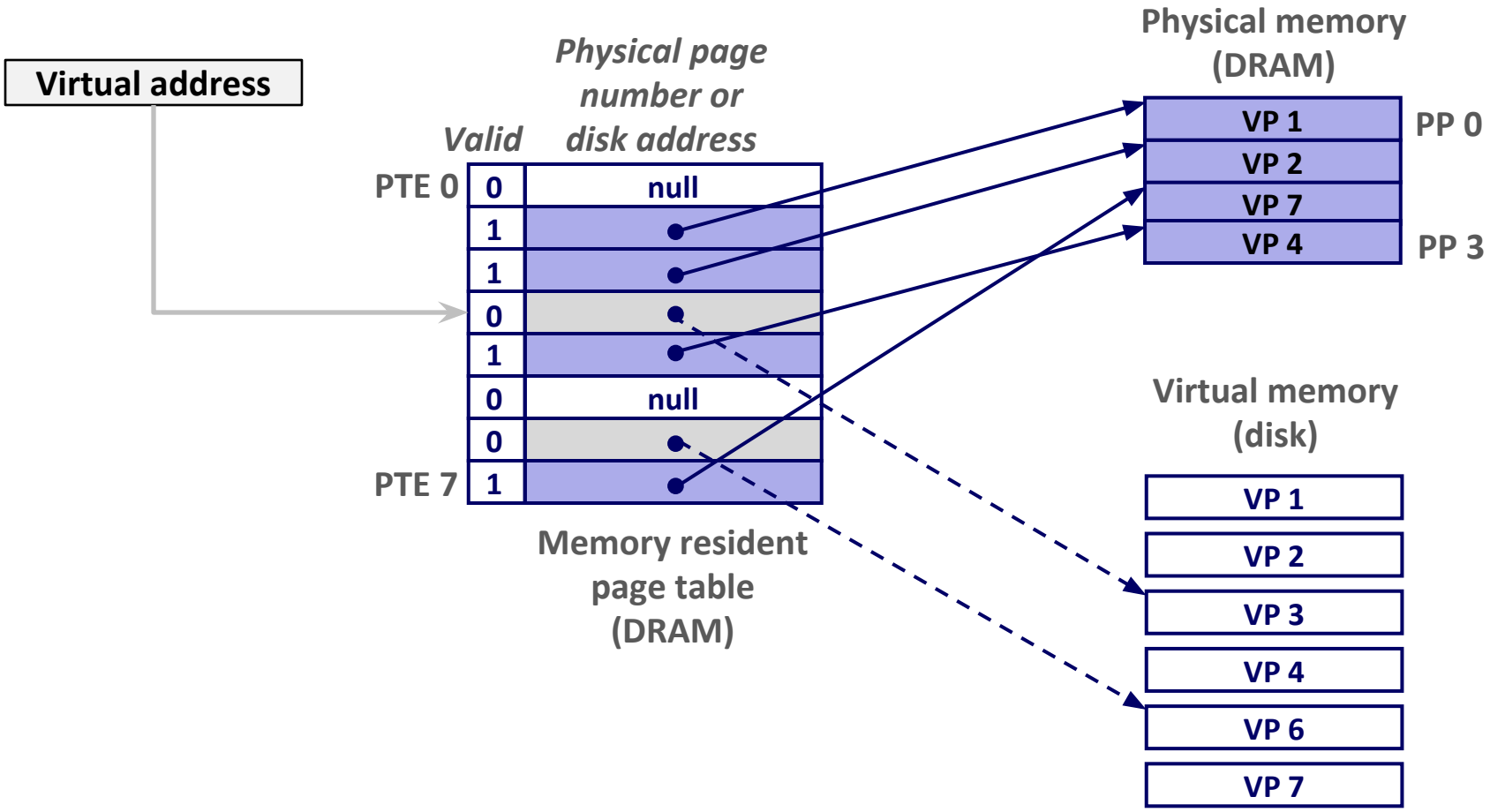
# Page Fault

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



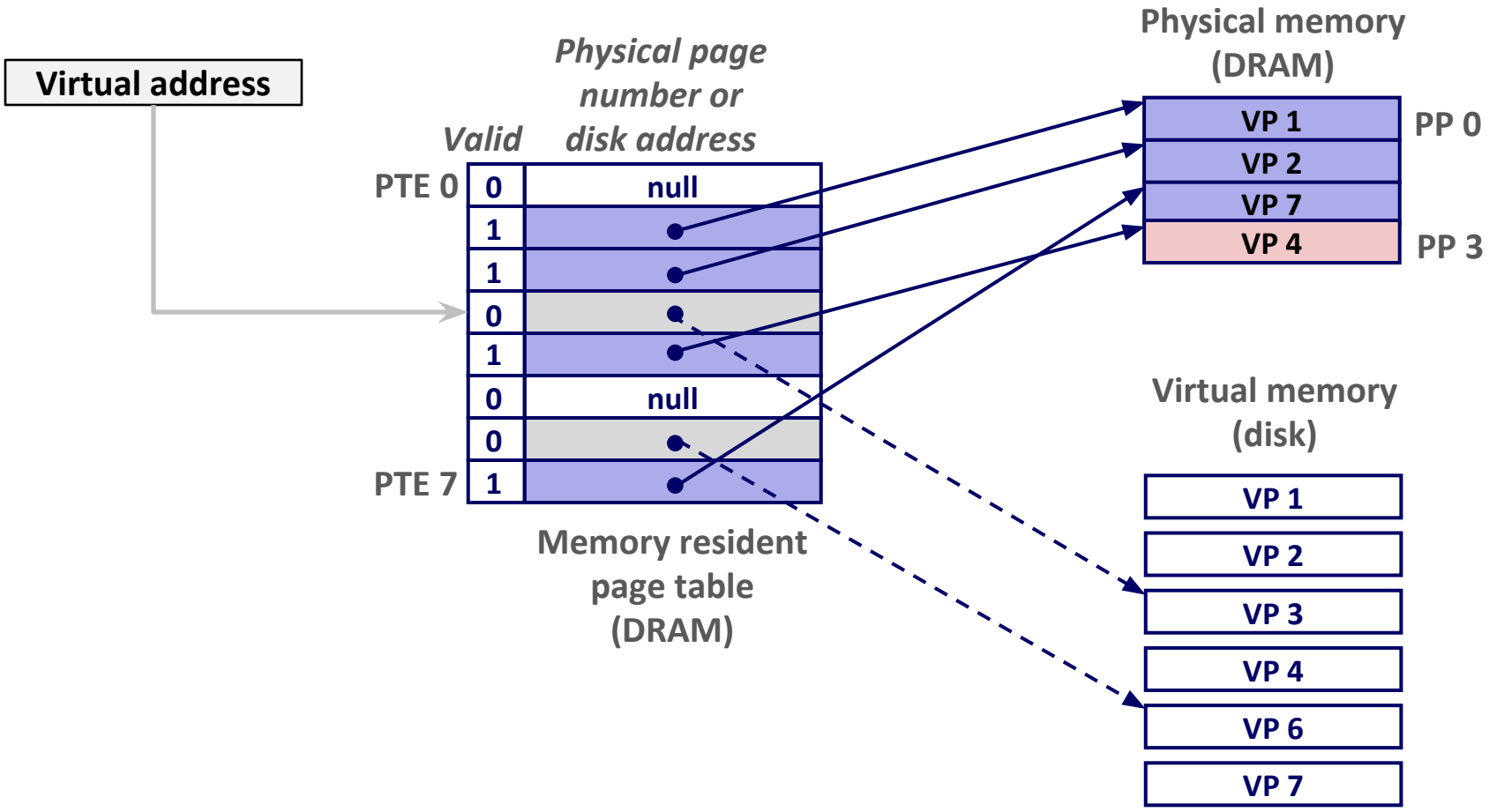
# Handling Page Fault

- Page miss causes page fault (an exception)



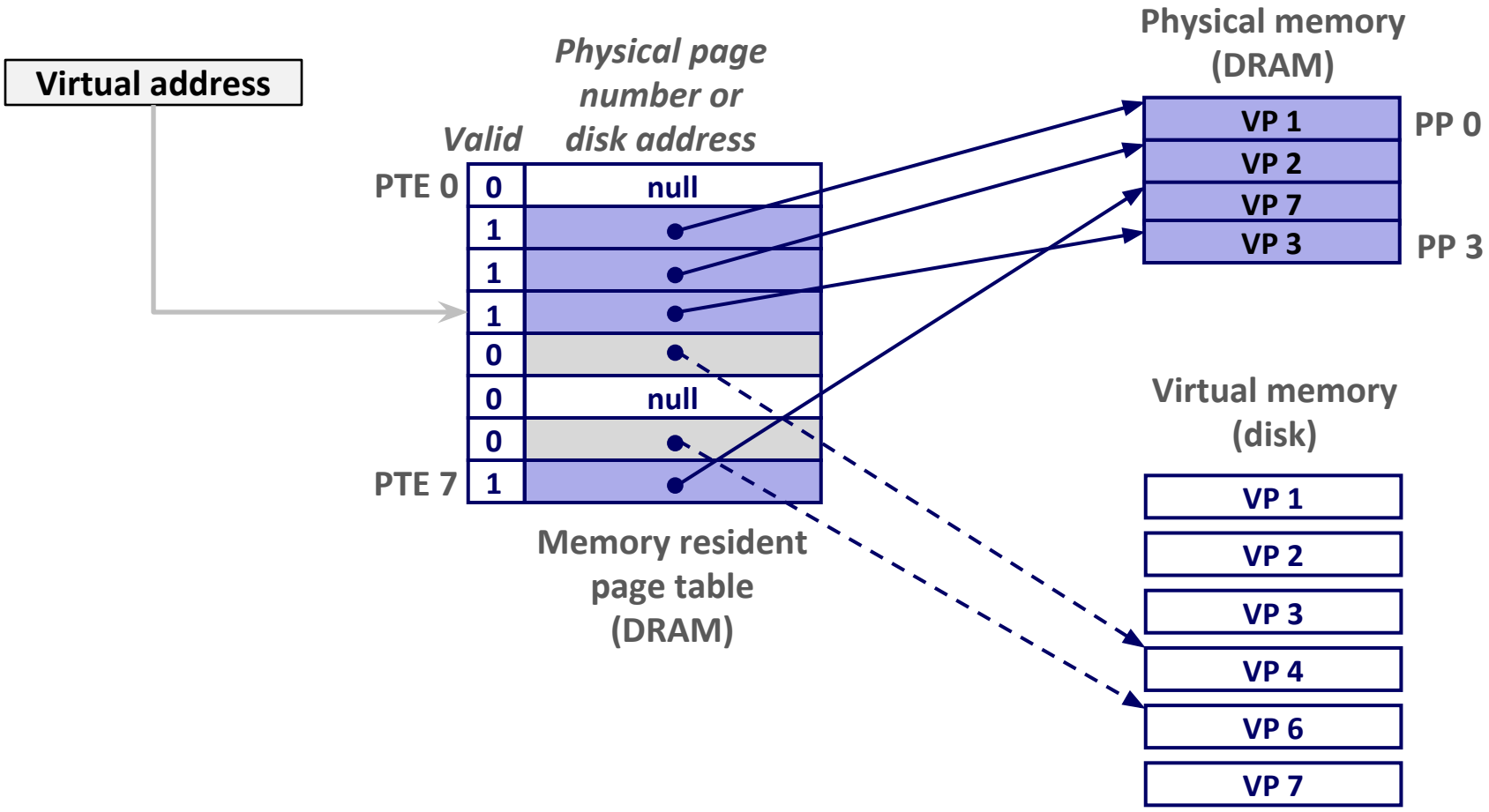
# Handling Page Fault

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



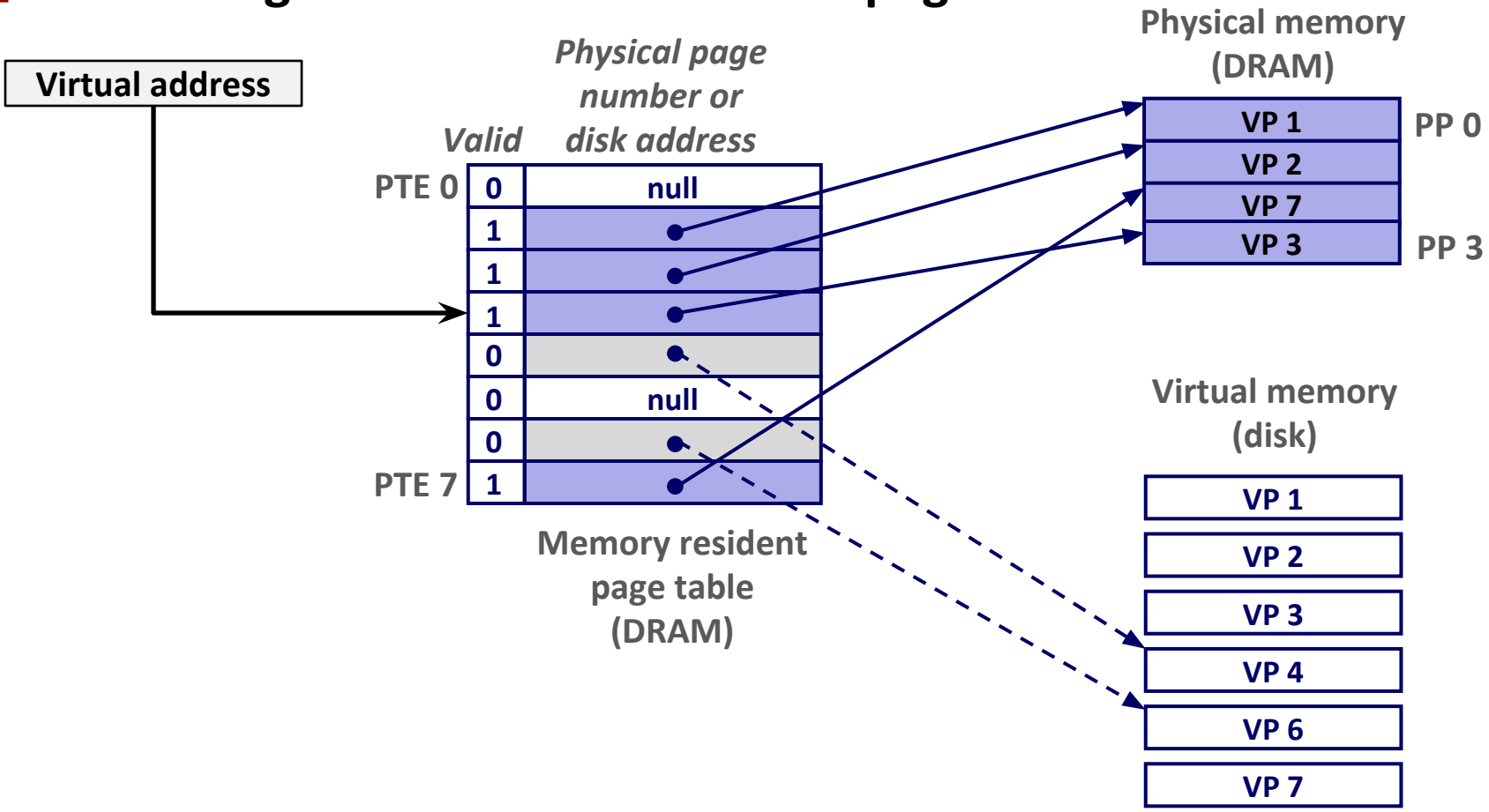
# Handling Page Fault

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



# 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!

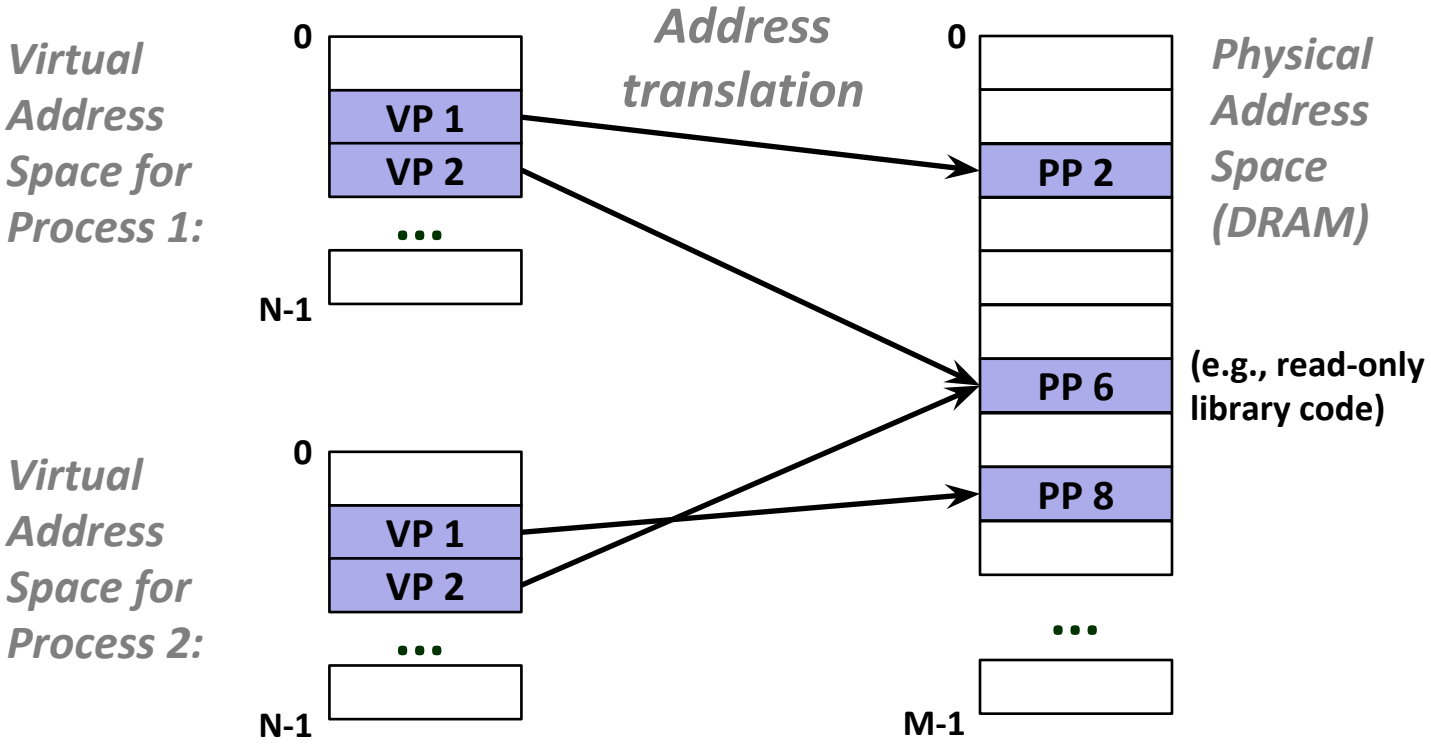


# Locality to the Rescue Again!

- **Virtual memory works because of locality**
- **At any point in time, programs tend to access a set of active virtual pages called the *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 moved (copied) in and out continuously

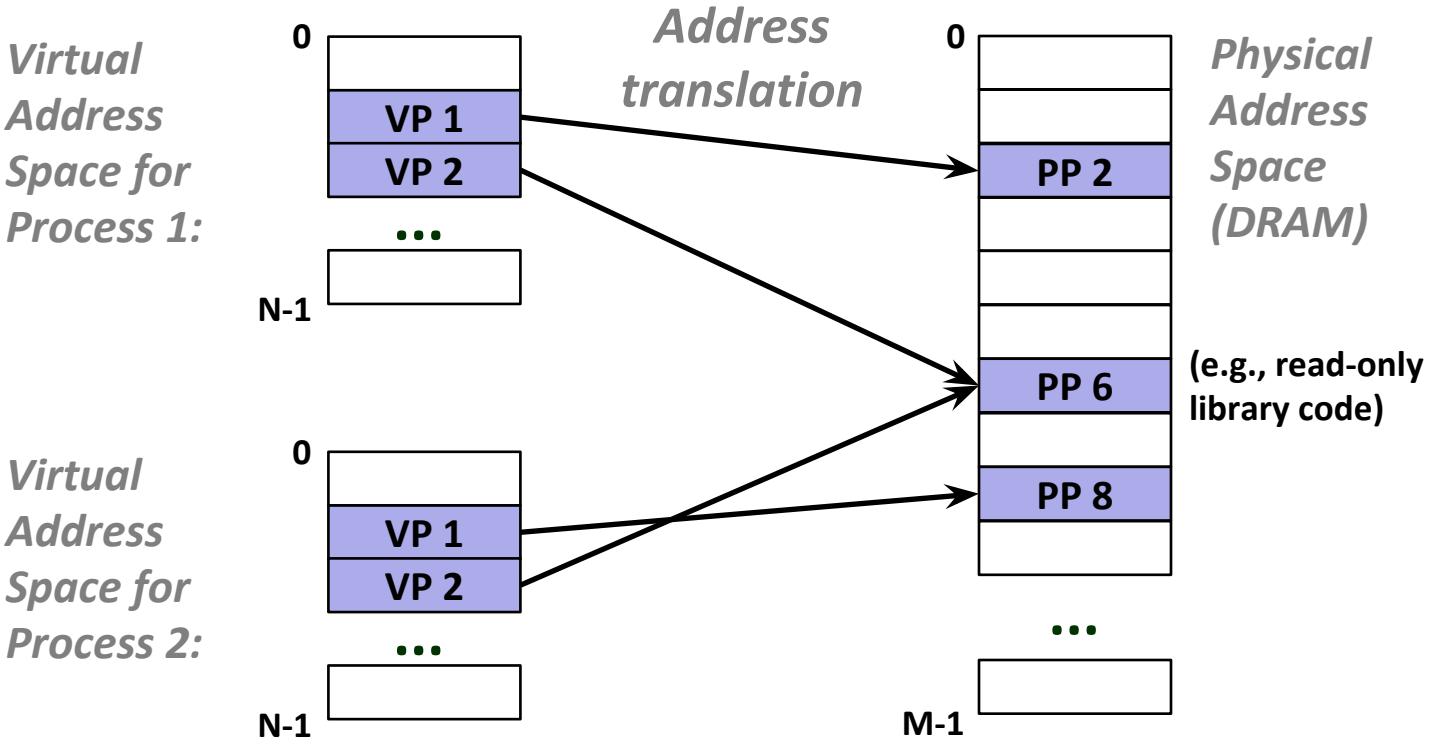
# (2) VM as a Tool for Memory Management

- **Key idea: 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



# Simplifying allocation and sharing

- **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 multiple virtual pages to the same physical page (here: PP 6)

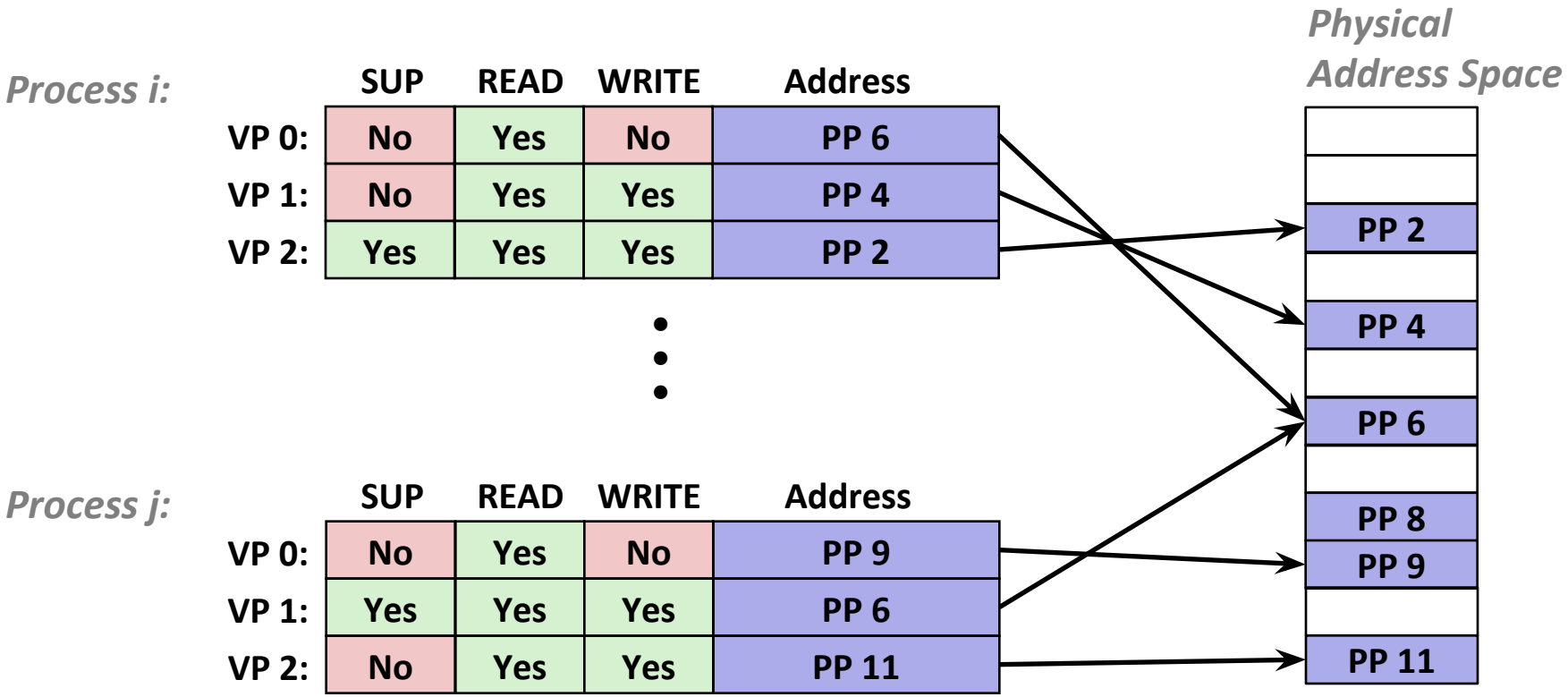


# Caching vs Mapping

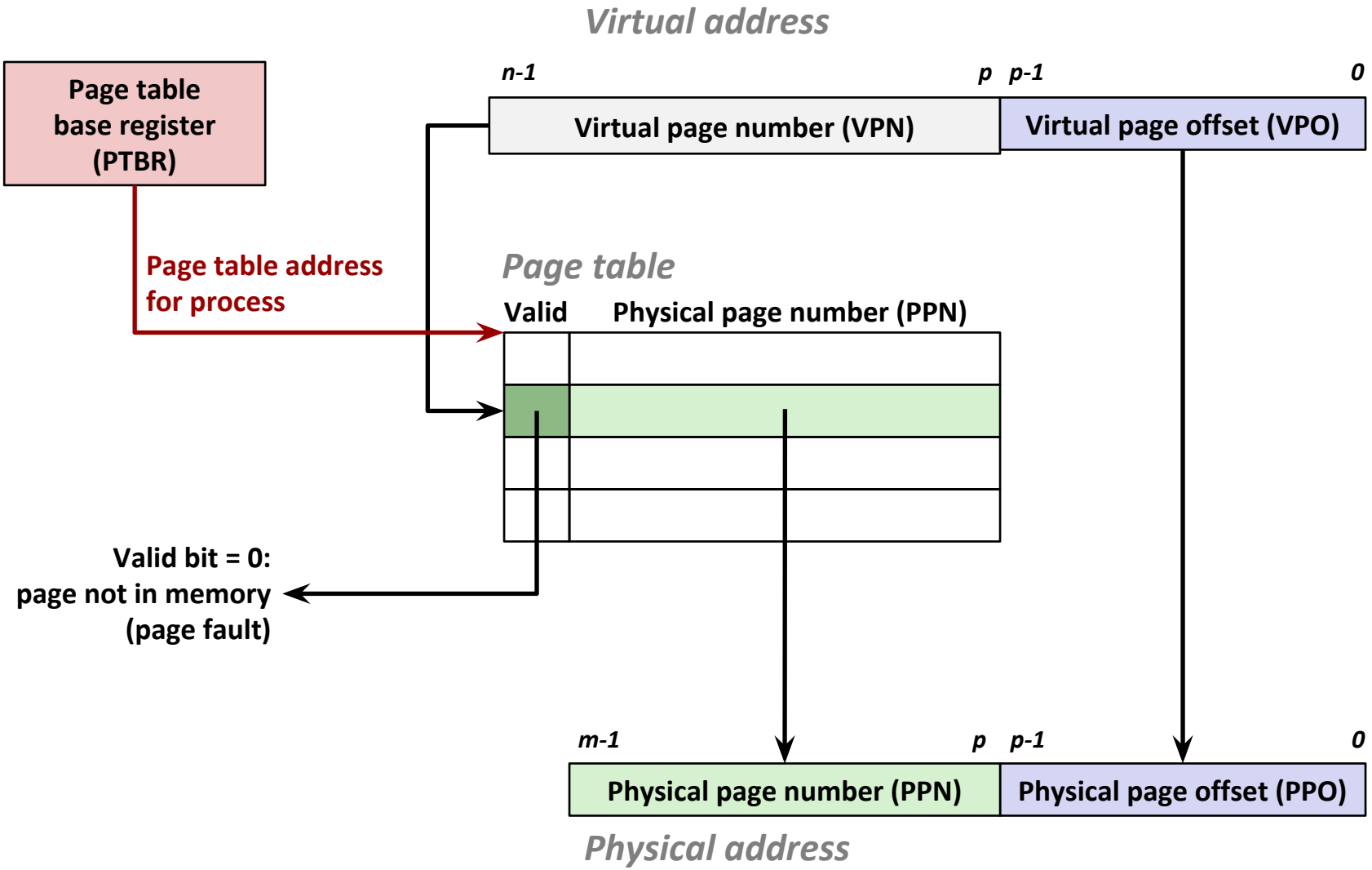
- Functionality
  - Is Caching necessary?
    - Computers can definitely work with CPU+Disk only, if you can bear on the fact that each memory access takes 1 million cycles
  - Is Mapping necessary?
    - May be… If you are willing to make the whole software/hardware by yourself
- Performance
  - Does Caching help?
    - That's why we need this complicated thing, though it makes CS33 Final really hard
  - Does Mapping help?
    - You can say that the most significant speedup is to make things work, but generally there is some overhead for maintaining the complicated mapping mechanism

# VM as a Tool for Memory Protection

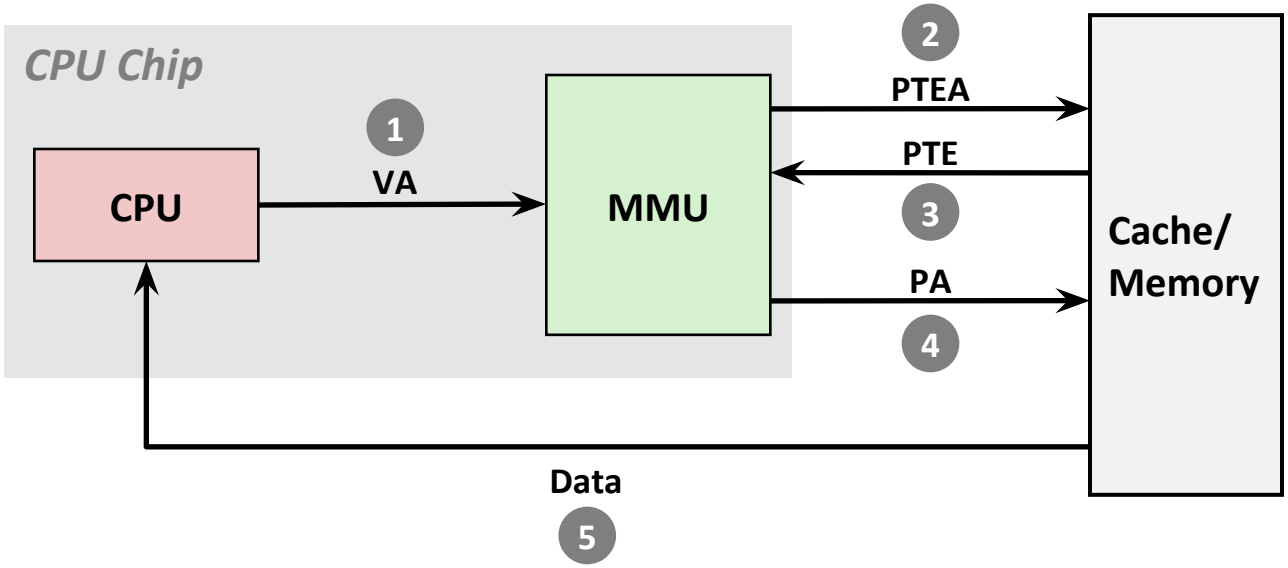
- **Extend PTEs with permission bits**
- **Page fault handler checks these before remapping**
  - If violated, send process SIGSEGV (segmentation fault)



# Address Translation With a Page Table



# Address Translation: Page Hit



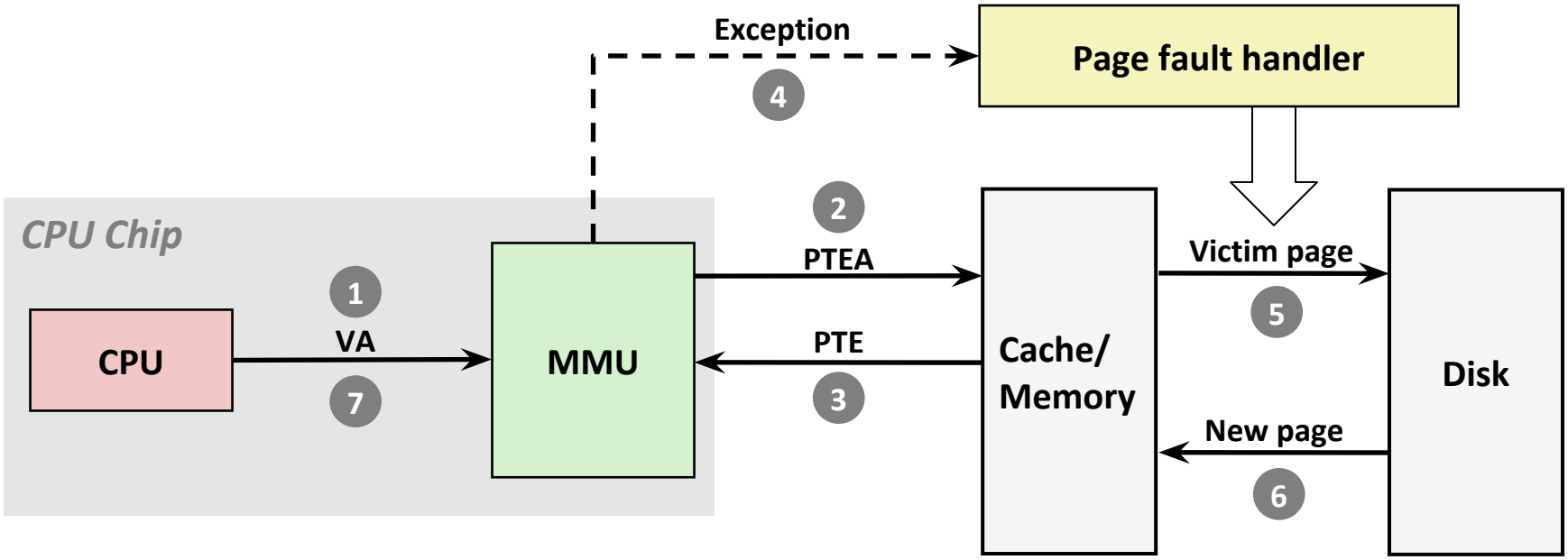

1) Processor sends virtual address to MMU

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)

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

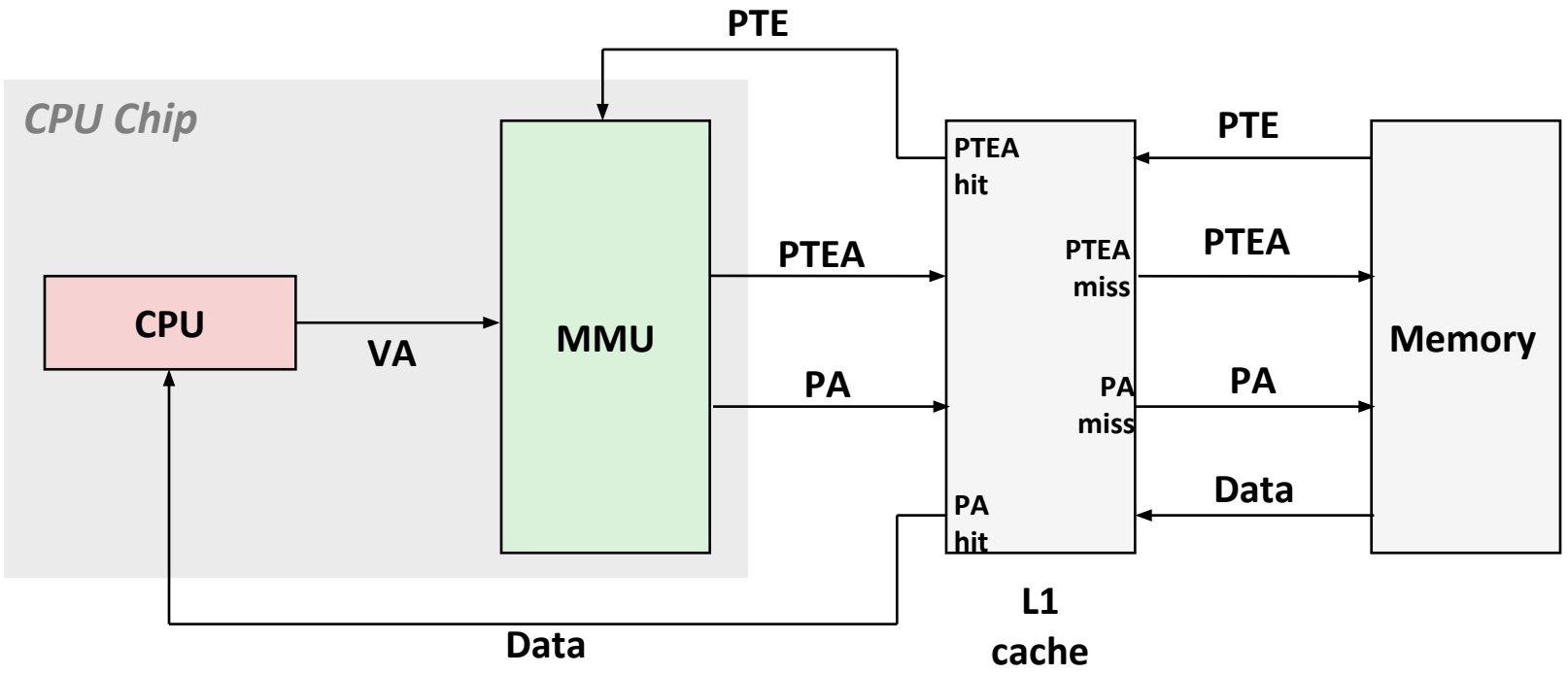
7) Handler returns to original process, restarting faulting instruction

# Views of virtual memory

- **Programmer's view of virtual memory**
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes

- **System view of virtual memory**
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions

# Integrating VM and Cache

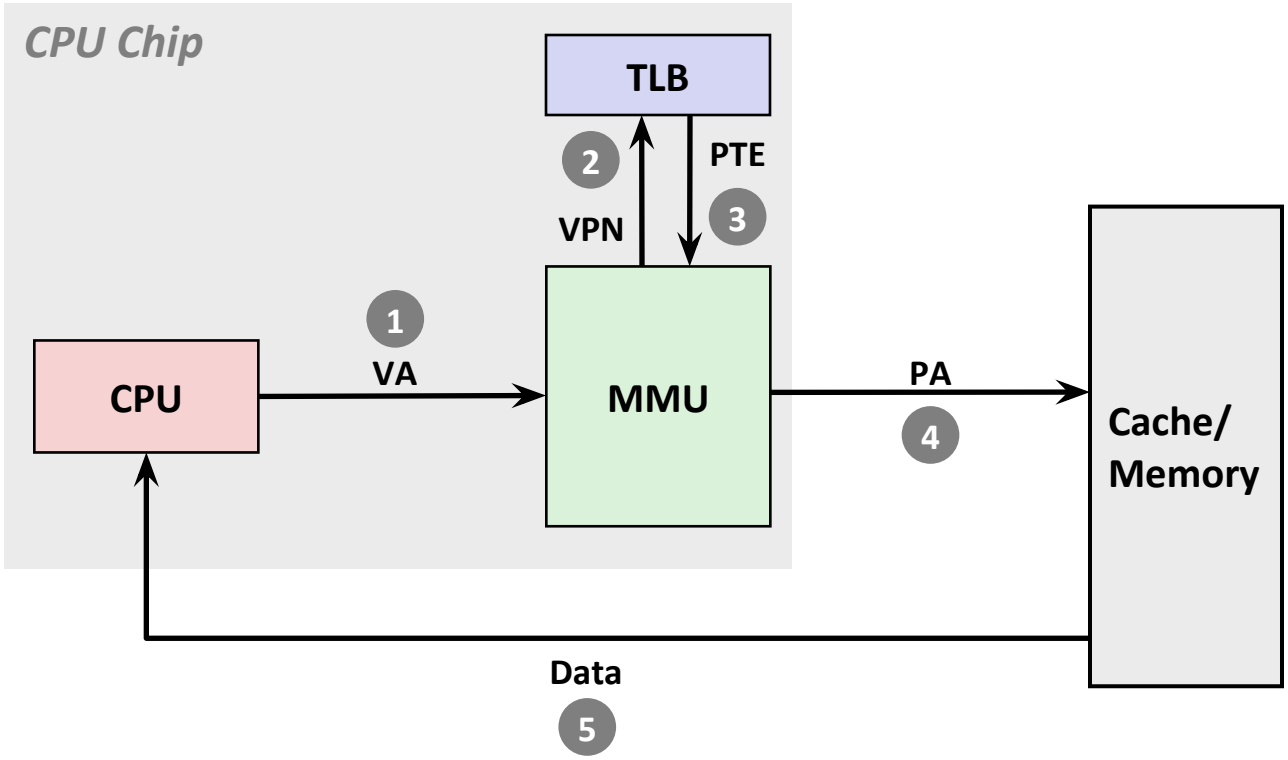



*VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address*

# Speeding up Translation with a TLB

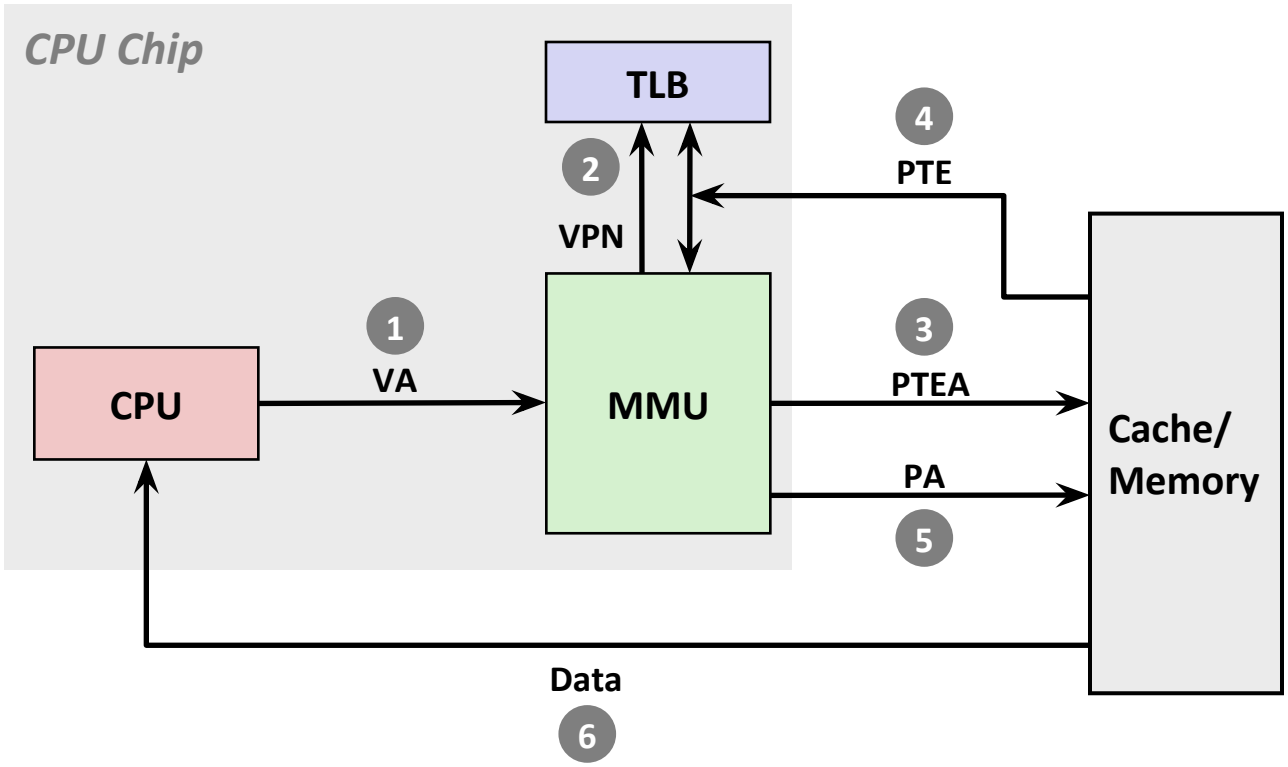
- **Page table entries (PTEs) are cached in L1 like any other memory word**
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay
- **Solution: *Translation Lookaside Buffer* (TLB)**
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page 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)**

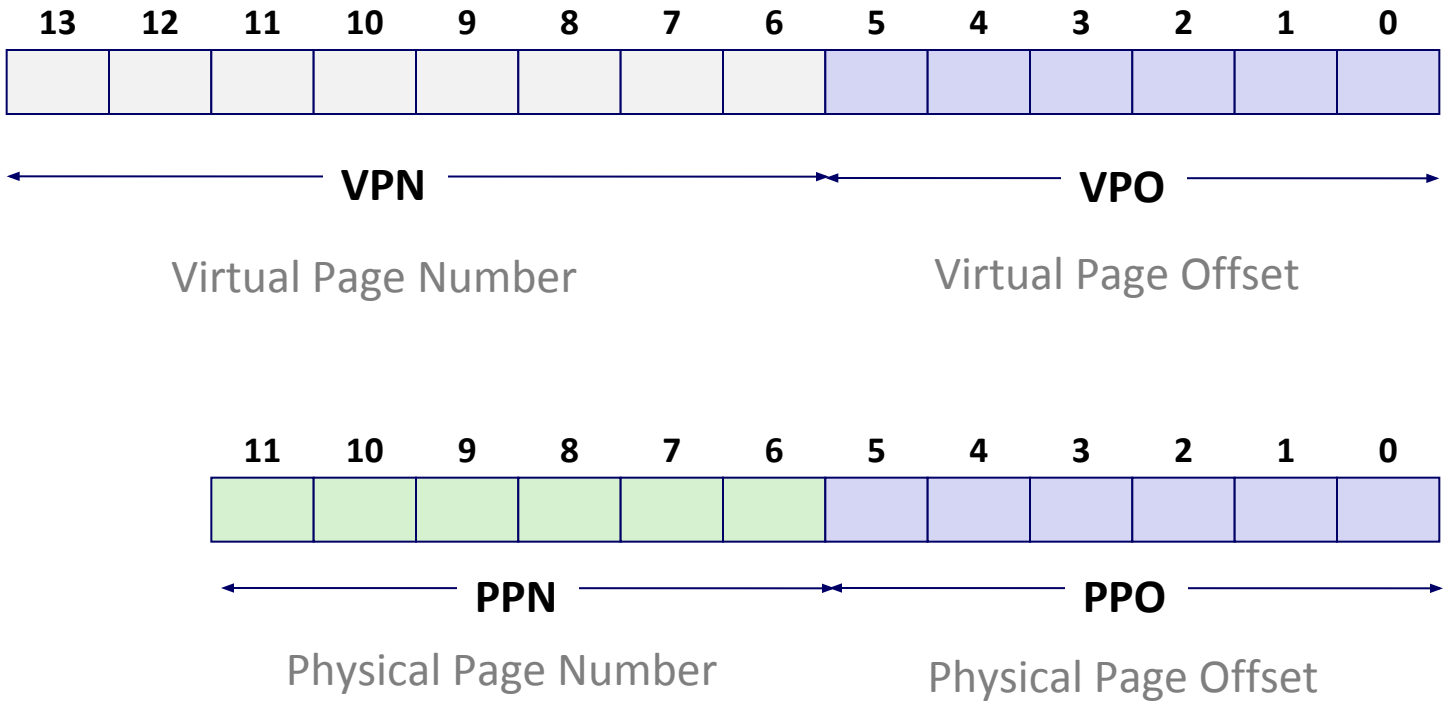
Fortunately, TLB misses are rare. Why?

# Symbols

- **Basic Parameters**
  - **N = $2^n$** : Number of addresses in virtual address space
  - **M = $2^m$** : Number of addresses in physical address space
  - **P = $2^p$** : Page size (bytes)
- **Components of the virtual address (VA)**
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
- **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number
  - **CO**: Byte offset within cache line
  - **CI:** Cache index
  - **CT**: Cache tag

# Simple Memory System Example

- **Addressing**
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes



# Simple Memory System Page Table

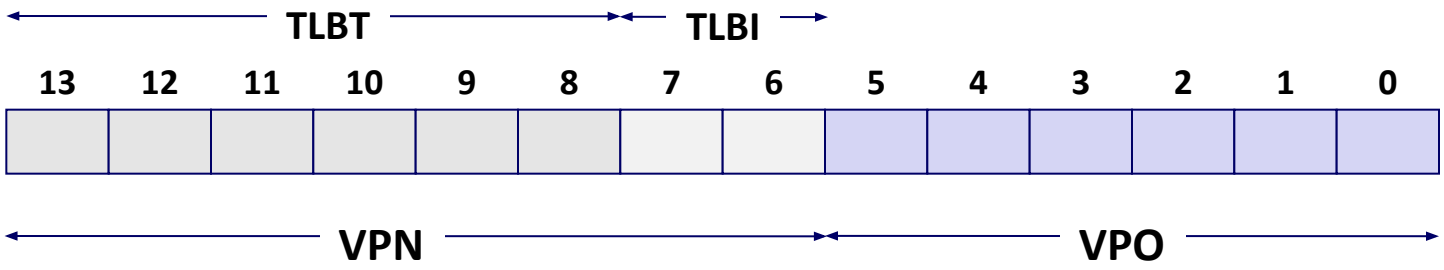
Only show first 16 entries (out of 256)

| *VPN* | *PPN* | *Valid* |
|---|---|---|
| 00 | 28 | 1 |
| 01 | – | 0 |
| 02 | 33 | 1 |
| 03 | 02 | 1 |
| 04 | – | 0 |
| 05 | 16 | 1 |
| 06 | – | 0 |
| 07 | – | 0 |

| *VPN* | *PPN* | *Valid* |
|---|---|---|
| 08 | 13 | 1 |
| 09 | 17 | 1 |
| 0A | 09 | 1 |
| 0B | – | 0 |
| 0C | – | 0 |
| 0D | 2D | 1 |
| 0E | 11 | 1 |
| 0F | 0D | 1 |

# Simple Memory System TLB

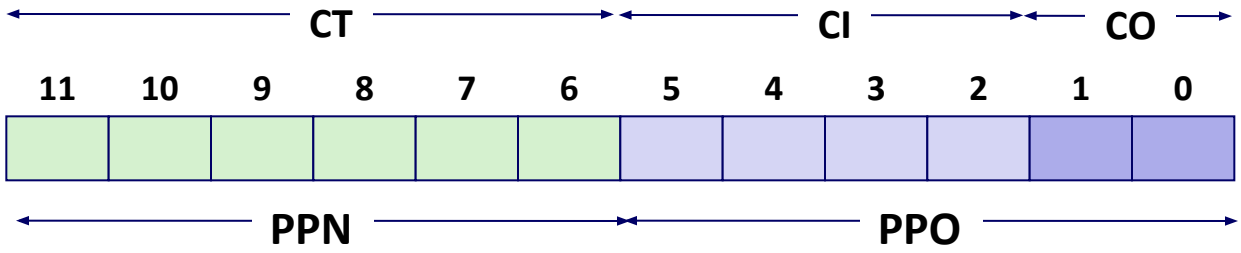
- 16 entries
- 4-way associative




| *Set* | *Tag* | *PPN* | *Valid* | *Tag* | *PPN* | *Valid* | *Tag* | *PPN* | *Valid* | *Tag* | *PPN* | *Valid* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 03 | – | 0 | 09 | 0D | 1 | 00 | – | 0 | 07 | 02 | 1 |
| 1 | 03 | 2D | 1 | 02 | – | 0 | 04 | – | 0 | 0A | – | 0 |
| 2 | 02 | – | 0 | 08 | – | 0 | 06 | – | 0 | 03 | – | 0 |
| 3 | 07 | – | 0 | 03 | 0D | 1 | 0A | 34 | 1 | 02 | – | 0 |

# Simple Memory System Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped

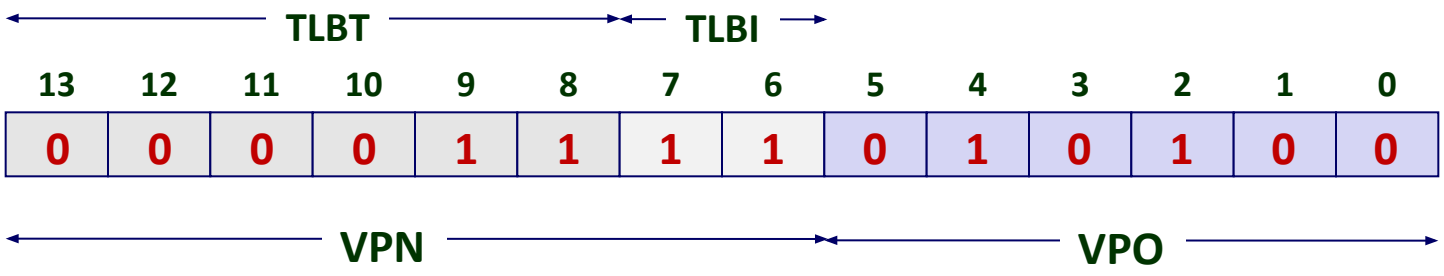



| Idx | Tag | Valid | B0 | B1 | B2 | B3 |
|---|---|---|---|---|---|---|
| 0 | 19 | 1 | 99 | 11 | 23 | 11 |
| 1 | 15 | 0 | – | – | – | – |
| 2 | 1B | 1 | 00 | 02 | 04 | 08 |
| 3 | 36 | 0 | – | – | – | – |
| 4 | 32 | 1 | 43 | 6D | 8F | 09 |
| 5 | 0D | 1 | 36 | 72 | F0 | 1D |
| 6 | 31 | 0 | – | – | – | – |
| 7 | 16 | 1 | 11 | C2 | DF | 03 |

| Idx | Tag | Valid | B0 | B1 | B2 | B3 |
|---|---|---|---|---|---|---|
| 8 | 24 | 1 | 3A | 00 | 51 | 89 |
| 9 | 2D | 0 | – | – | – | – |
| A | 2D | 1 | 93 | 15 | DA | 3B |
| B | 0B | 0 | – | – | – | – |
| C | 12 | 0 | – | – | – | – |
| D | 16 | 1 | 04 | 96 | 34 | 15 |
| E | 13 | 1 | 83 | 77 | 1B | D3 |
| F | 14 | 0 | – | – | – | – |

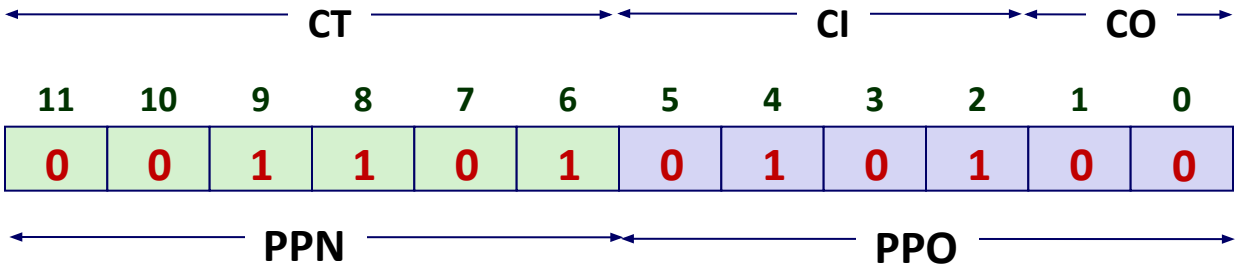
# Address Translation Example #1

## Virtual Address: 0x03D4




VPN 0x0F TLBI 0x3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

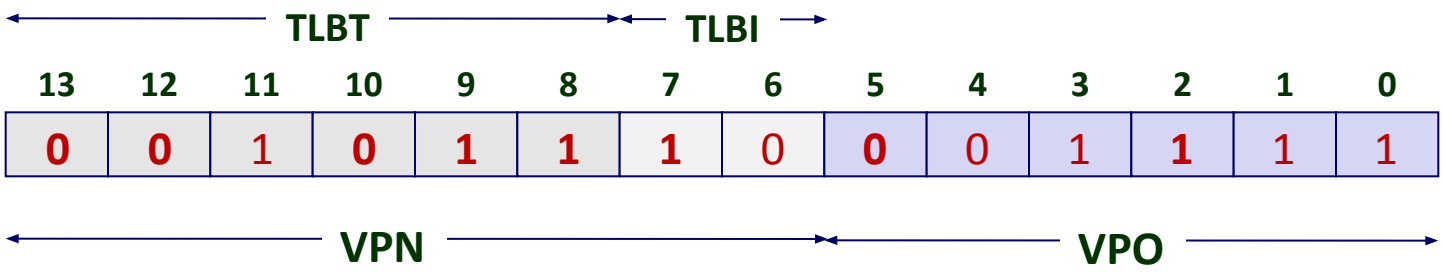
## Physical Address




CO 0 CI 0x5 CT 0x0D Hit? Y Byte: 0x36

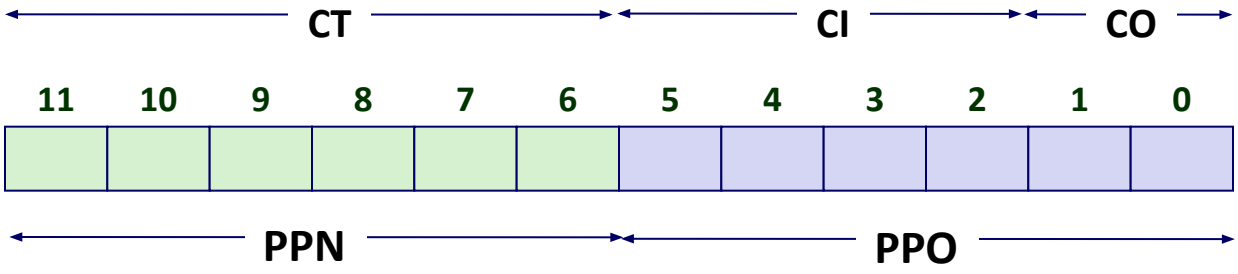
# Address Translation Example #2

## Virtual Address: 0x0B8F




VPN 0x2E TLBI ___ 2 TLBT ___0x0B TLB Hit? __ N Page Fault? __ Y PPN: ____TBD

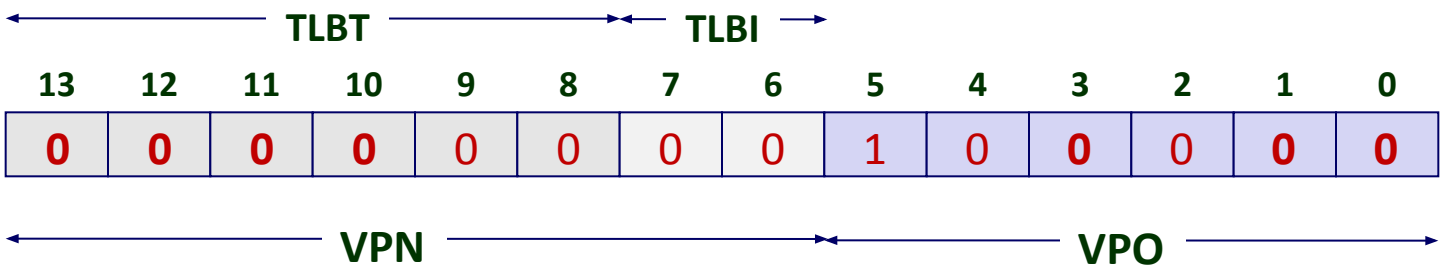
## Physical Address




CO ___ CI___ CT ____ Hit? __ Byte: ____

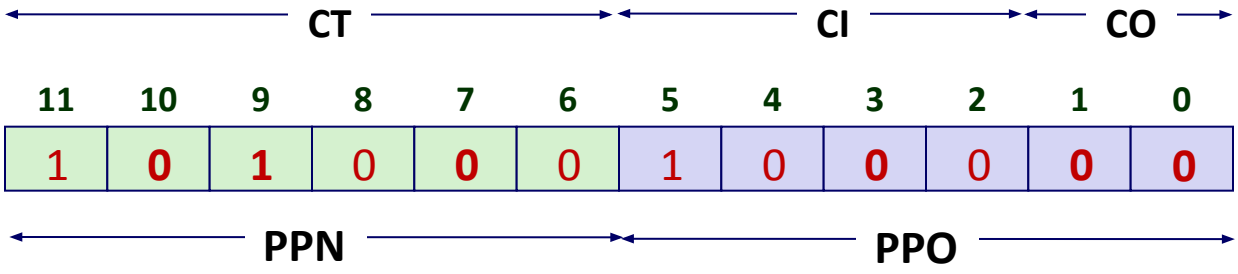
# Address Translation Example #3

## Virtual Address: 0x0020




VPN 0x00 TLBI ___ 0 TLBT ___0x00 TLB Hit? __ N Page Fault? __ N PPN: ___0x28

## Physical Address




CO___0 CI___0x8 CT ___0x28 Hit? __ N Byte: ___Mem