Virtual Memory

’22H2

송 인 식

Outline

- Virtual Memory: Concepts
- Virtual Memory: Implementation

Virtualizing Memory

- Example

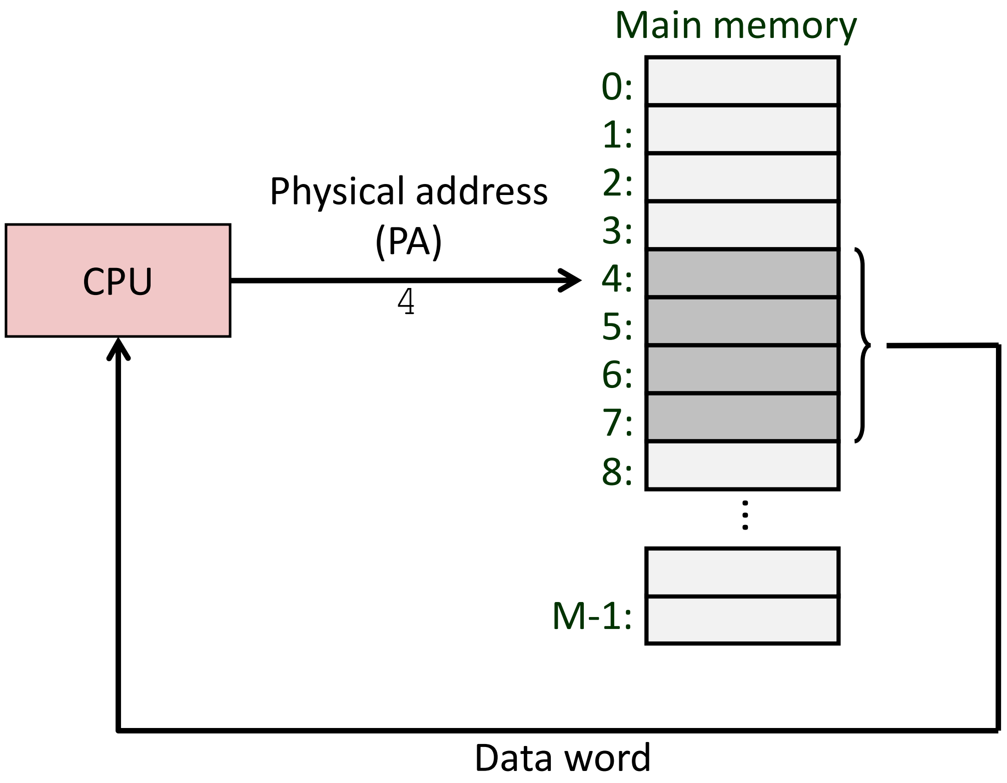
```
#include <stdio.h>
int n = 0;
int main ()
{
        n++;
        printf (“&n = %p, n = %d\n”, &n, n);
}

% ./a.out
&n = 0x0804a024, n = 1
% ./a.out
&n = 0x0804a024, n = 1
```

  – What happens if two users simultaneously run this program?

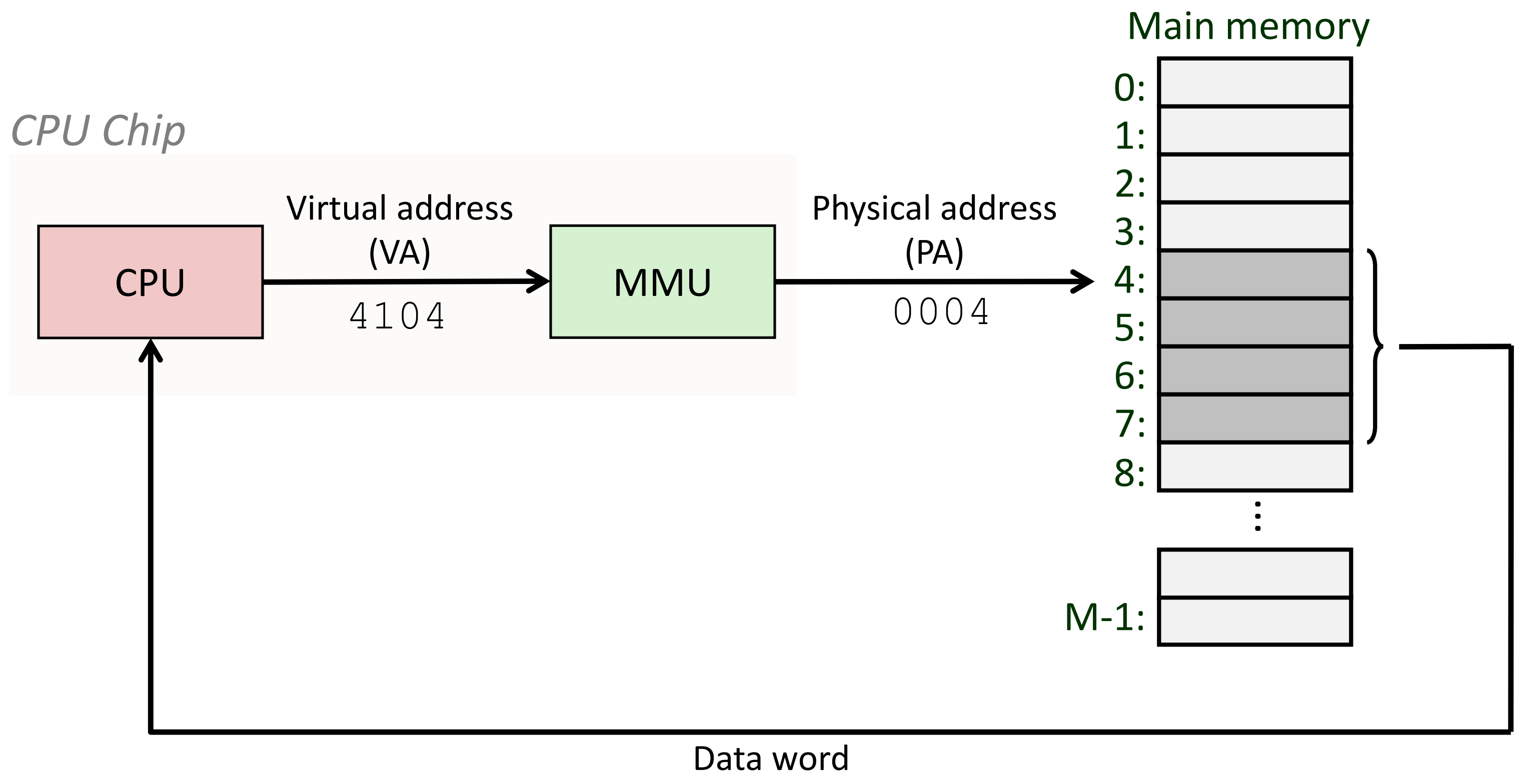
A System Using Physical Addressing

- Used in “simple” systems like embedded microcontrollers in cars, elevators, digital cameras, etc.



A System Using Virtual Addressing

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



Virtual Memory

- Each process has its own virtual address space
  - Large and contiguous
  - Use virtual addresses for memory references
  - Virtual addresses are private to each process
- Address translation is performed at run time
  - From a virtual address to the corresponding physical address
- Supports lazy allocation
  - Physical memory is dynamically allocated or released on demand
  - Programs execute without requiring their entire address space to be resident in physical memory

Address Spaces

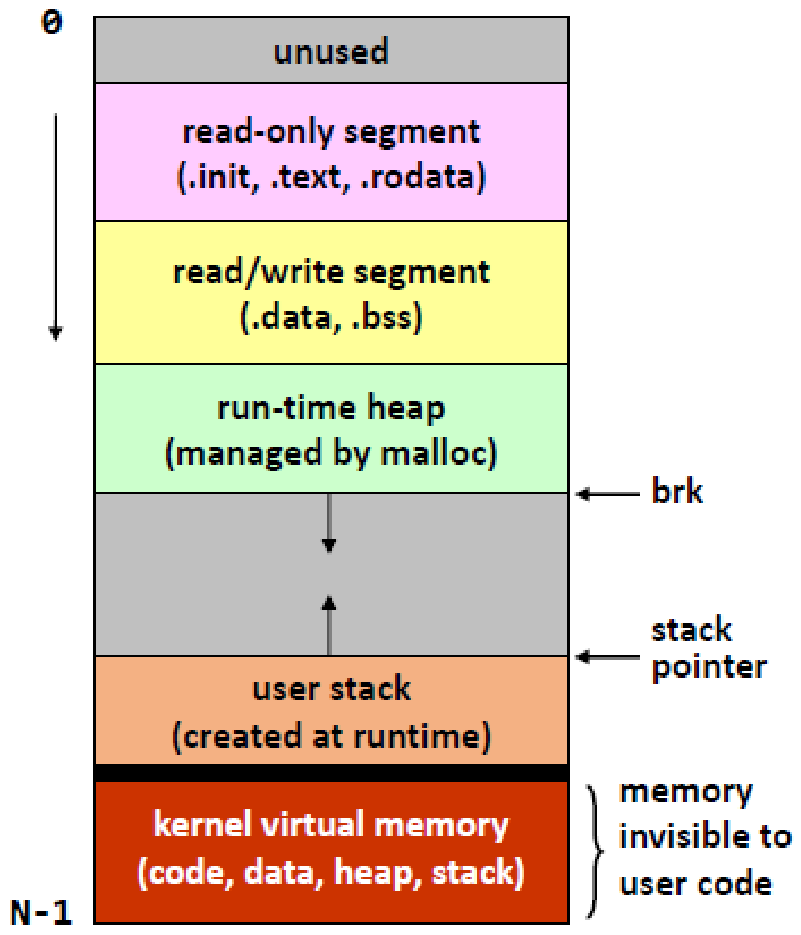
- Linear address space: Ordered set of contiguous non-negative integer addresses:
  {0, 1, 2, 3 … }

- 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}

(Virtual) Address Space

- Process' abstract view of memory
  - OS provides illusion of private address space to each process
  - Contains all of the memory state of the process
  - Static area
    - Allocated on exec()
    - Code & Data
  - Dynamic area
    - Allocated at runtime
    - Can grow or shrink
    - Heap & Stack

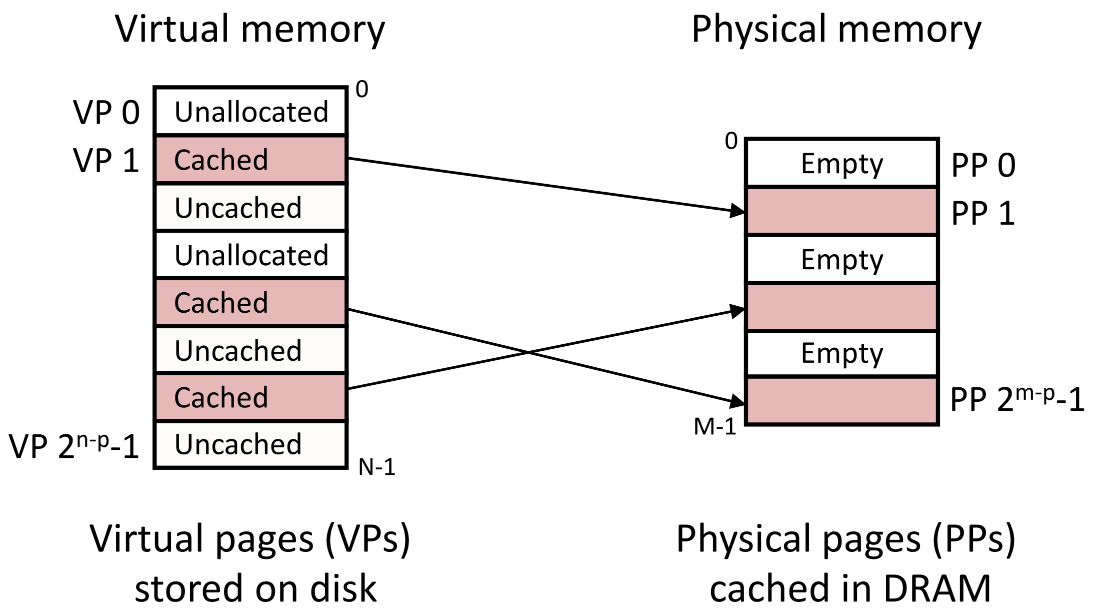

0
unused
read-only segment (.init, .text, .rodata)
read/write segment (.data, .bss)
run-time heap (managed by malloc)
brk
stack pointer
user stack (created at runtime)
kernel virtual memory (code, data, heap, stack)
memory invisible to user code
N-1

Why Virtual Memory (VM)?

- VM as a tool for caching
  - Uses main memory efficiently
  - Use DRAM as a cache for parts of a virtual address space
- VM as a tool for memory management
  - Simplifies memory management
  - Each process gets the same uniform linear address space
- VM as a tool for memory protection
  - Isolates address spaces
  - One process can’t interfere with another’s memory
  - User program cannot access privileged kernel information and code

VM as a Tool for Caching

- *Virtual memory* is an array of N contiguous bytes 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)



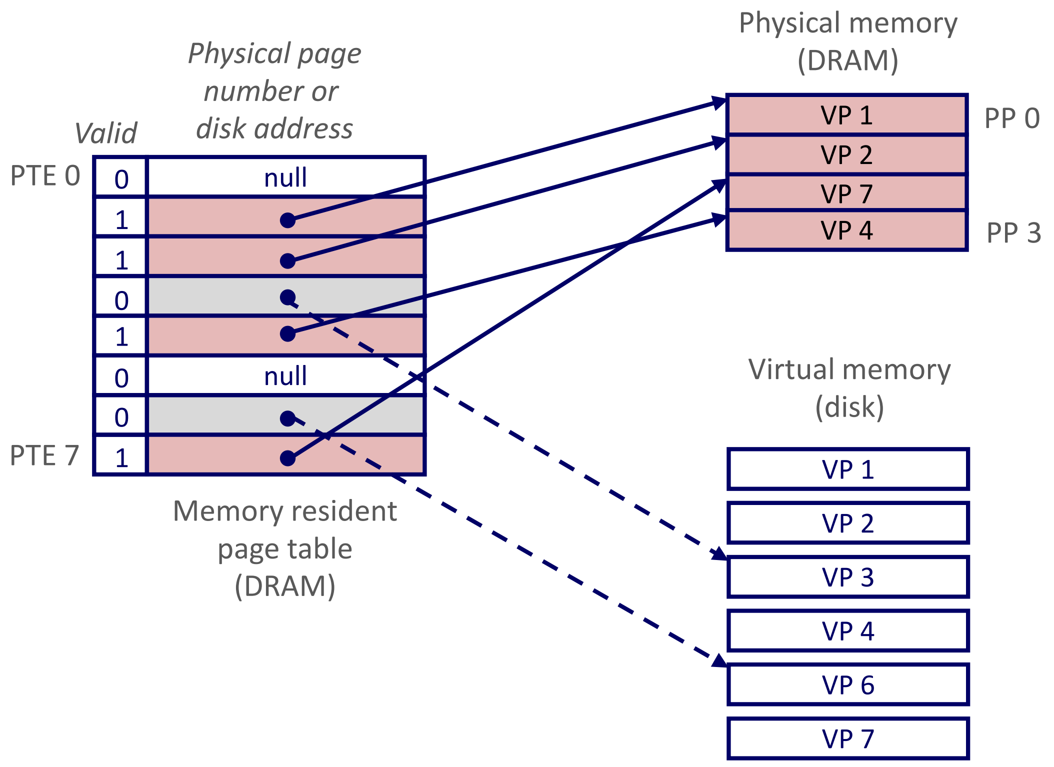
DRAM Cache Organization

- 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 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

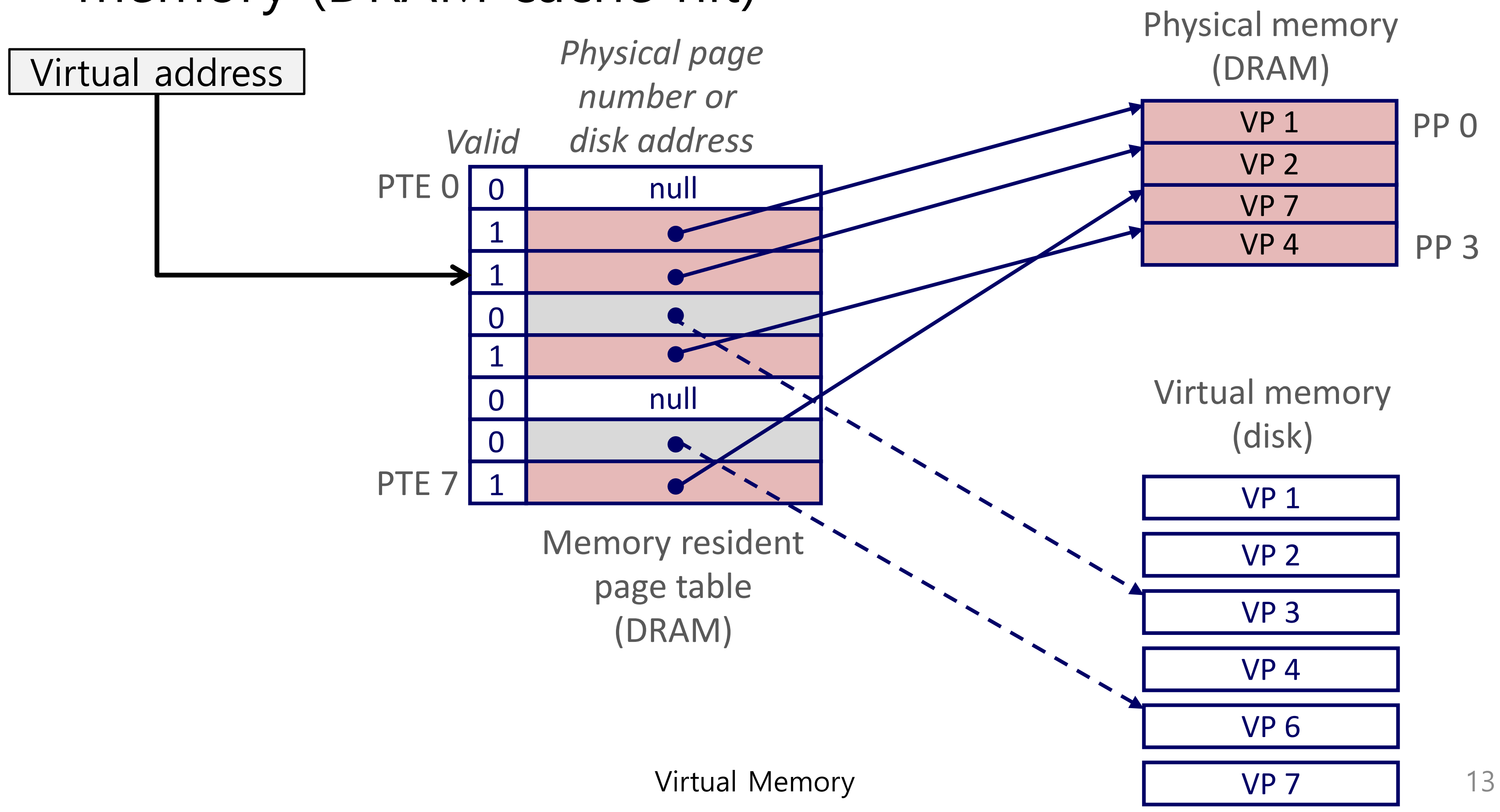
Enabling data structure: Page Table

- 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



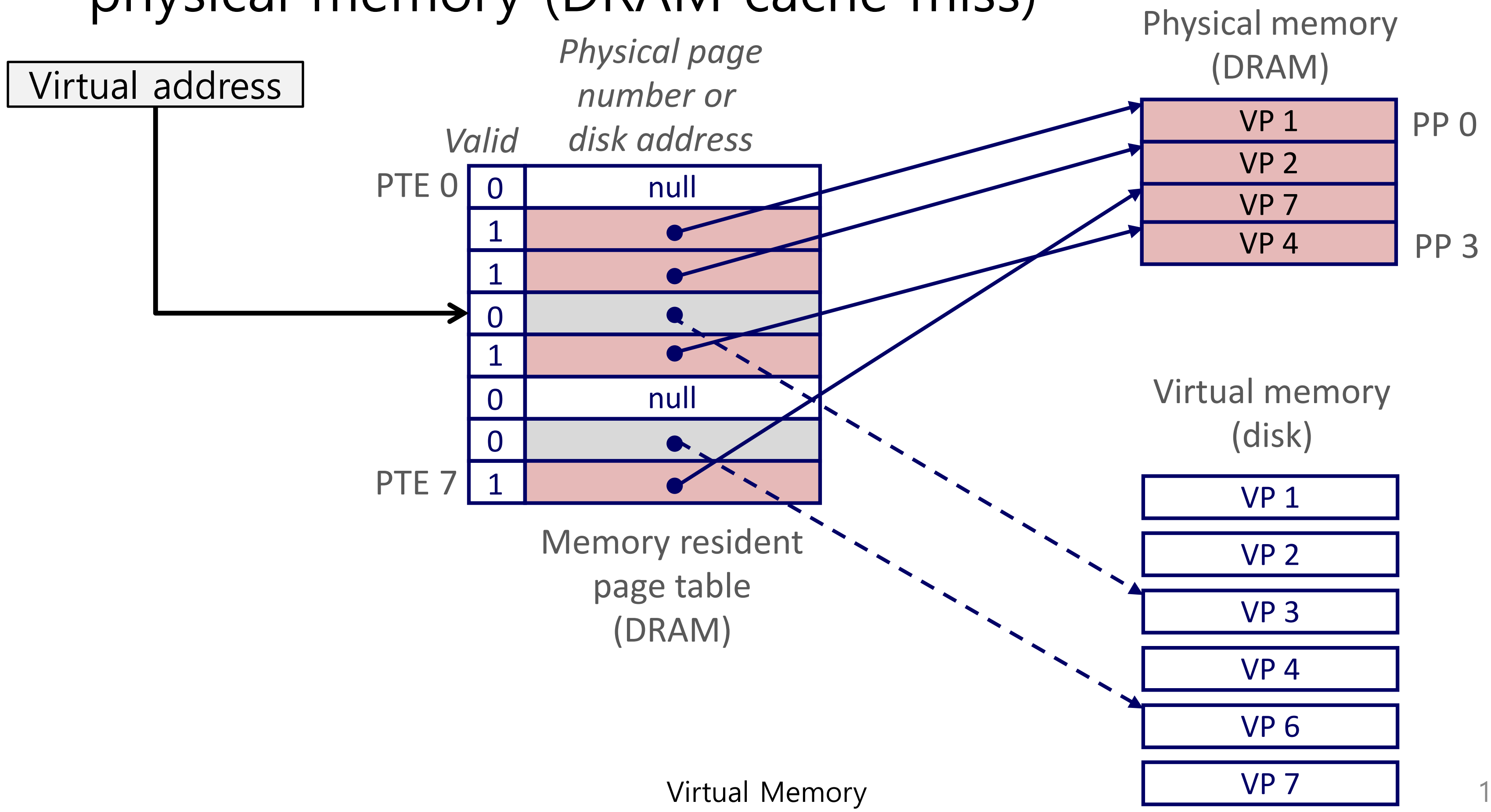
Page Hit

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



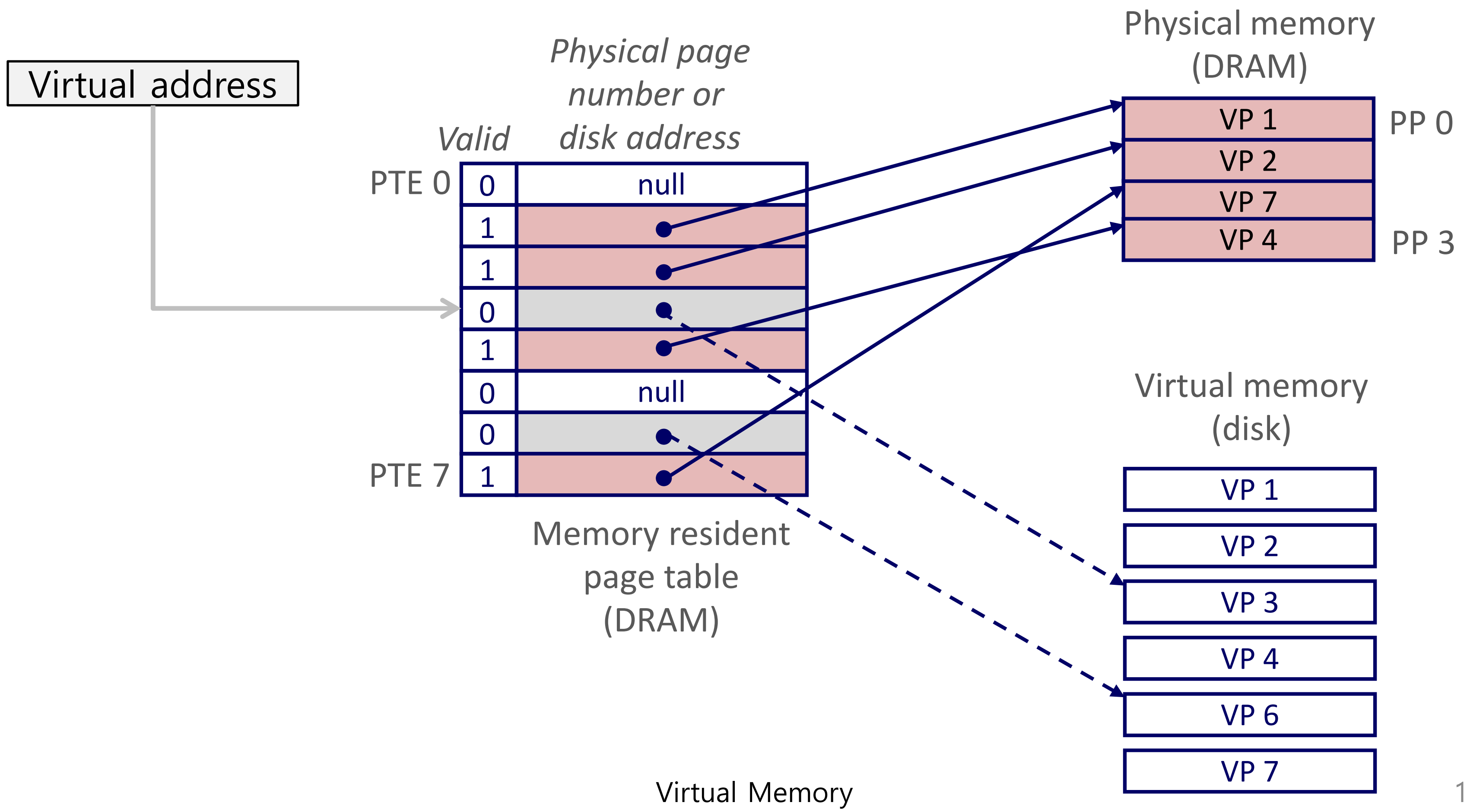
Page Fault

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



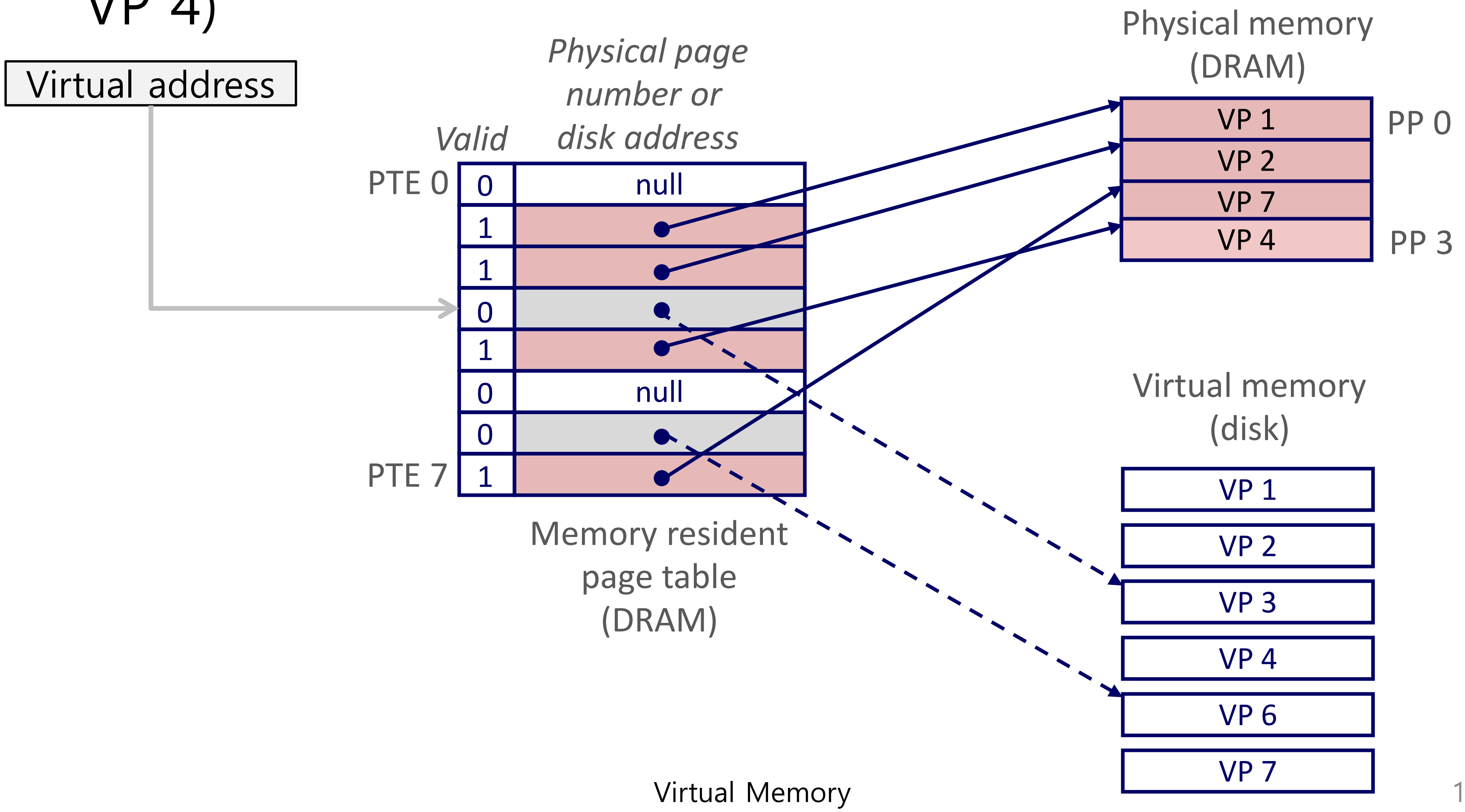
Handling Page Fault #1

- Page miss causes page fault (an exception)



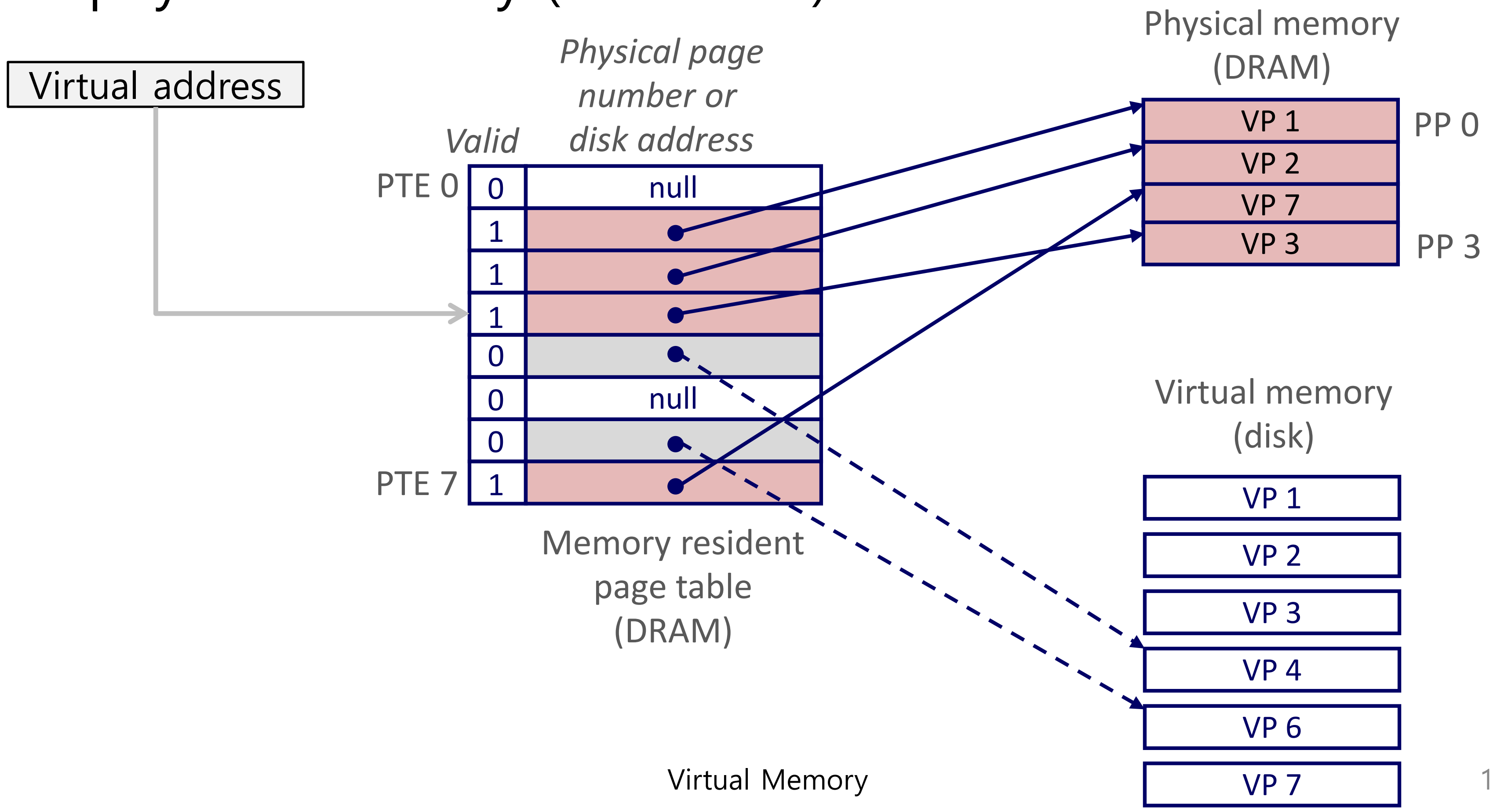
Handling Page Fault #2

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



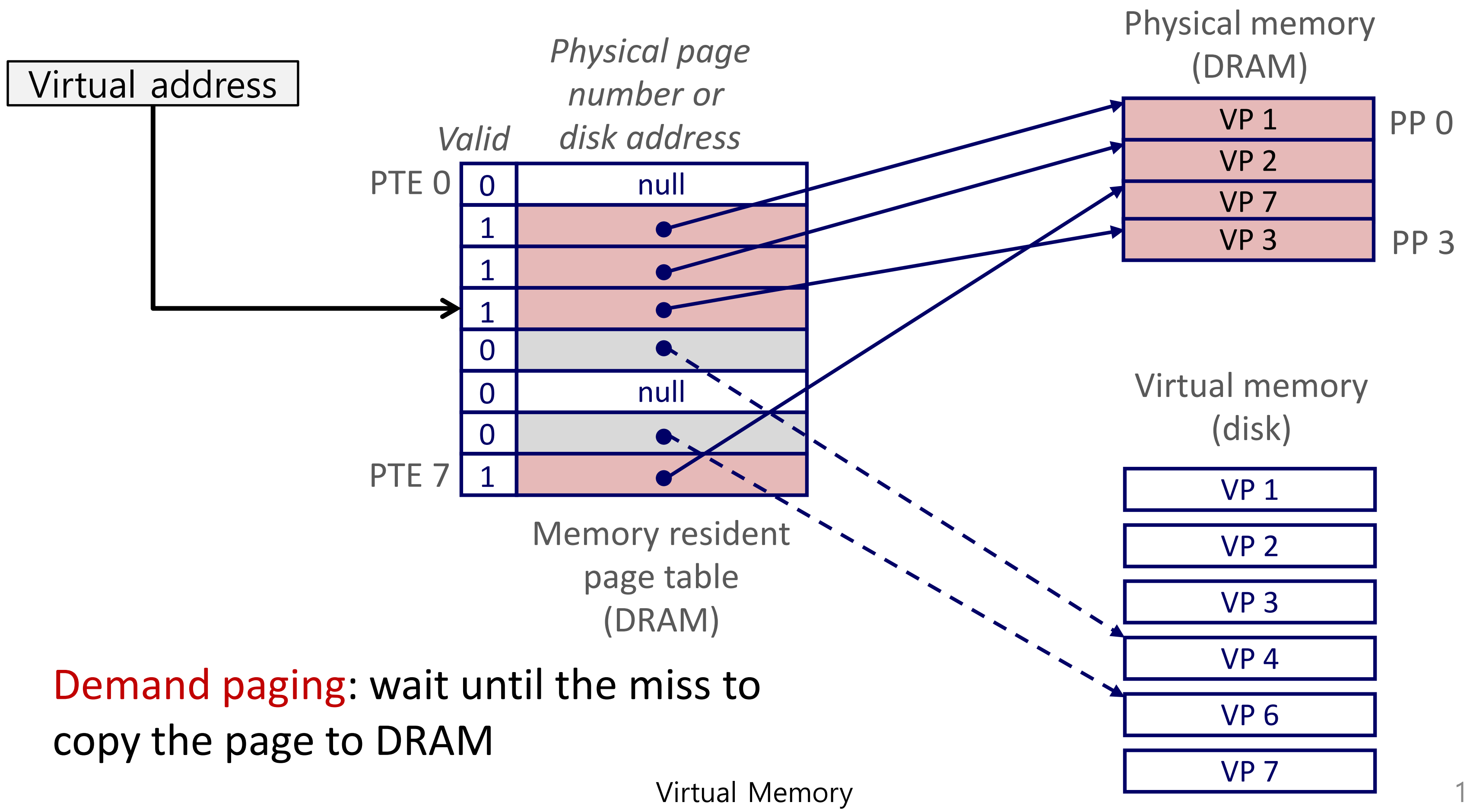
Handling Page Fault #3

- Page fault handler loads the faulting page into physical memory (here VP 3)



Handling Page Fault #4

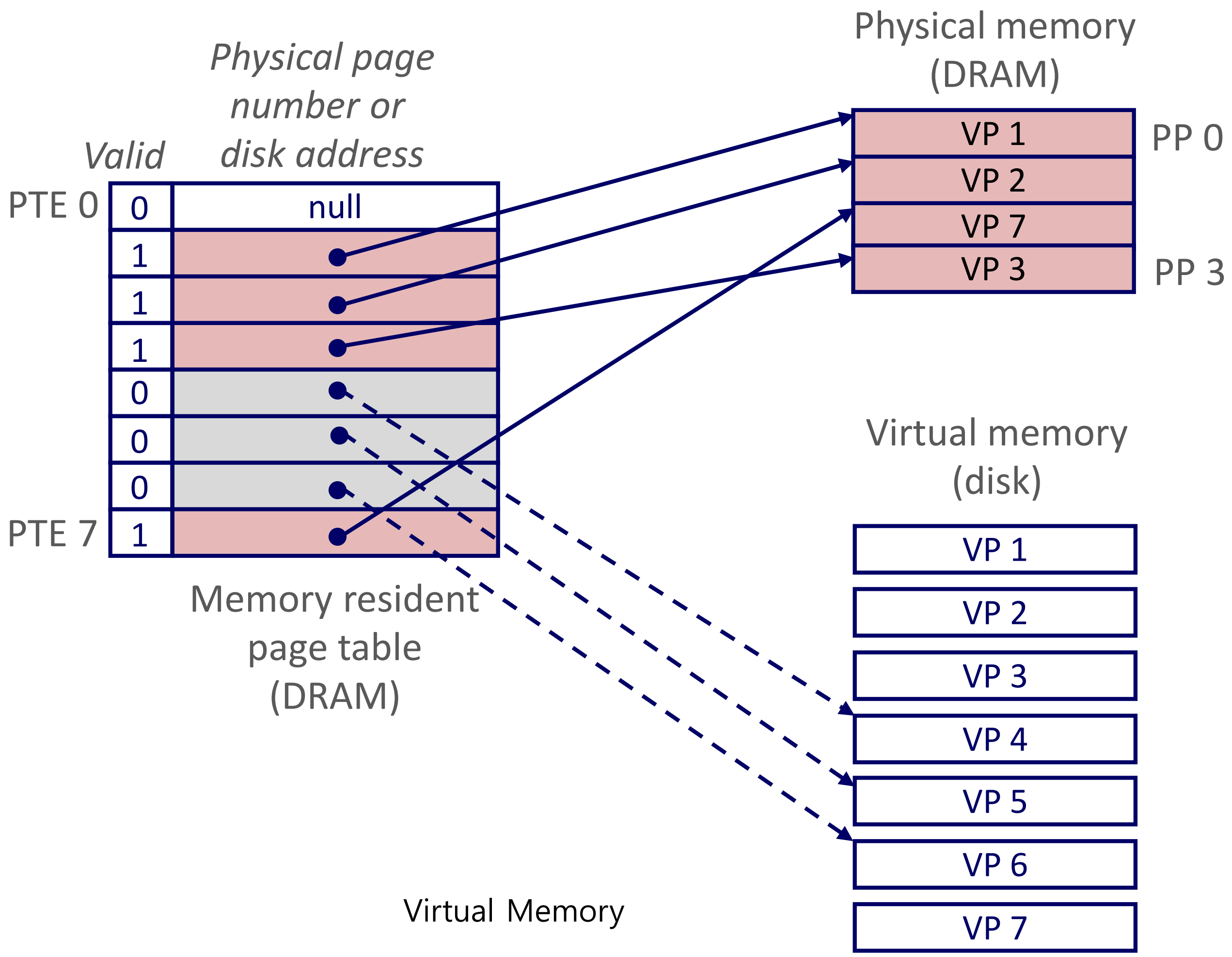
- Offending instruction is restarted: page hit!



Demand paging: wait until the miss to copy the page to DRAM

Allocating Pages

- Allocating a new page (VP 5) of virtual memory.

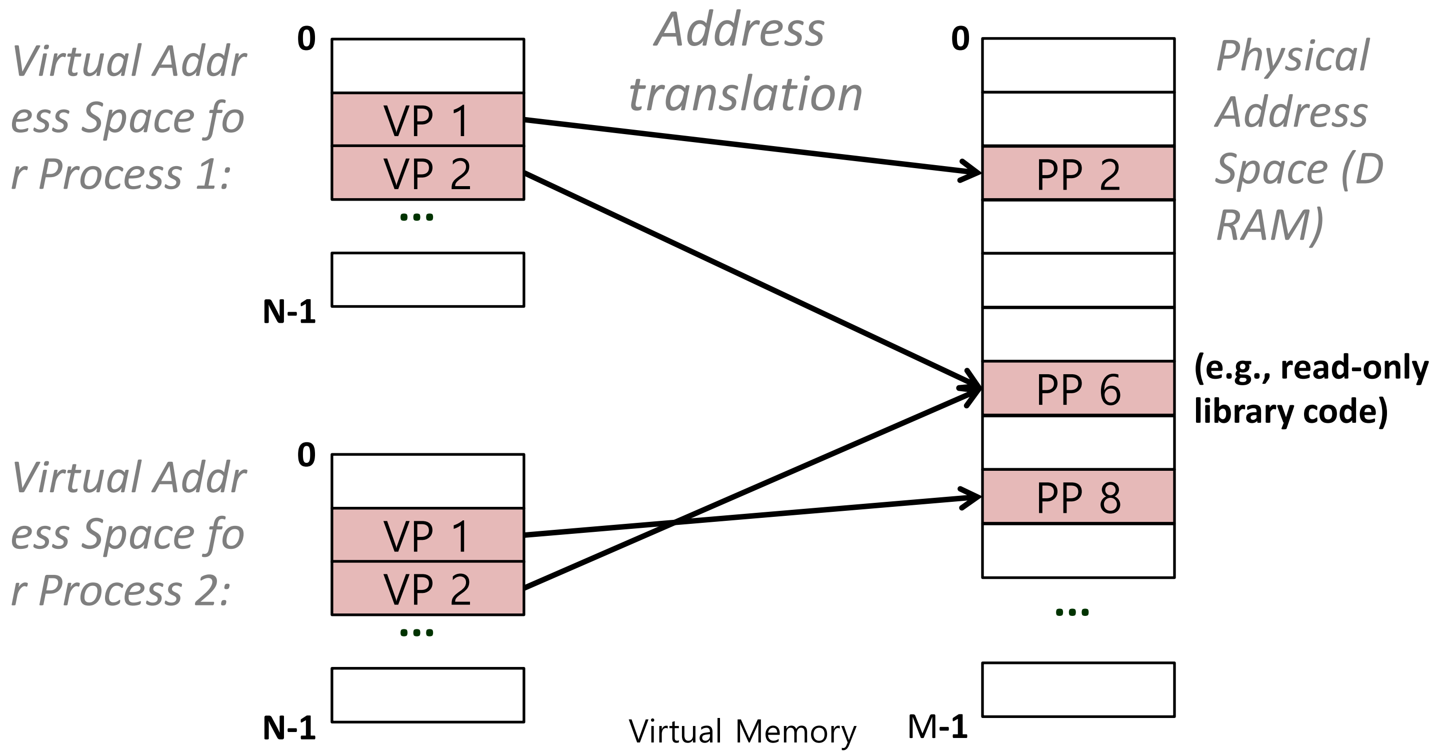


Locality to the Rescue Again!

- 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

VM as a Tool for Memory Management

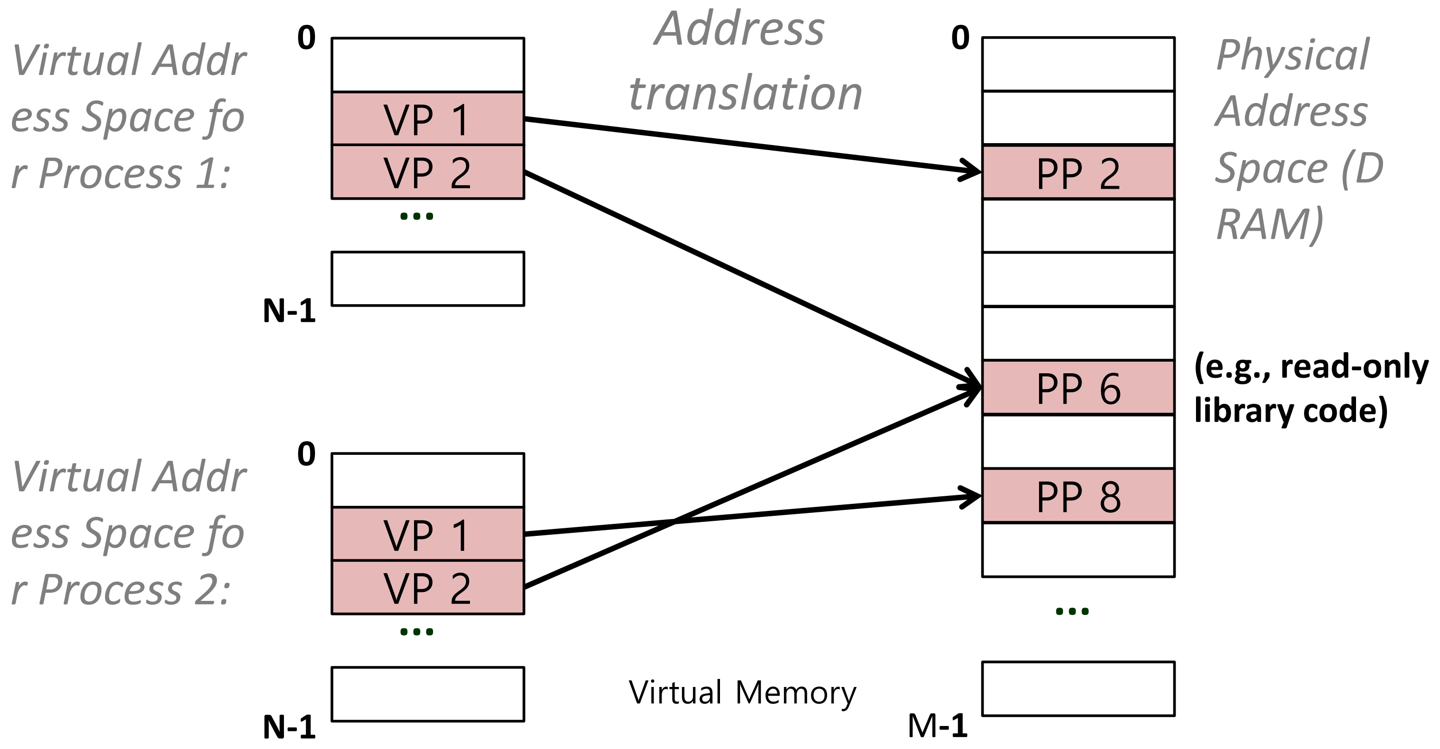
- Key idea: each process has its own virtual address space
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory
    - Well chosen mappings simplify memory allocation and management



Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address translation
0
PP 2
PP 6
PP 8
...
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

VM as a Tool for Memory 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 multiple virtual pages to the same physical page (here: PP 6)

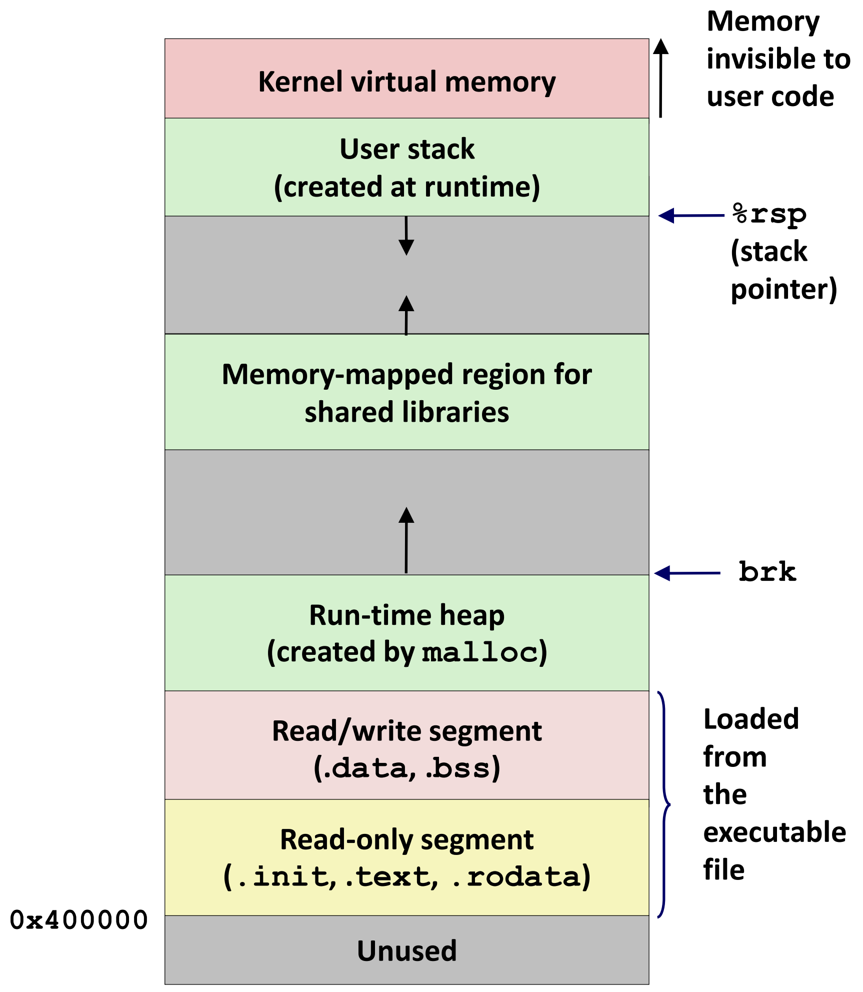

Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address translation
0
PP 2
PP 6
PP 8
...
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

Virtual Memory

Simplifying Linking and Loading

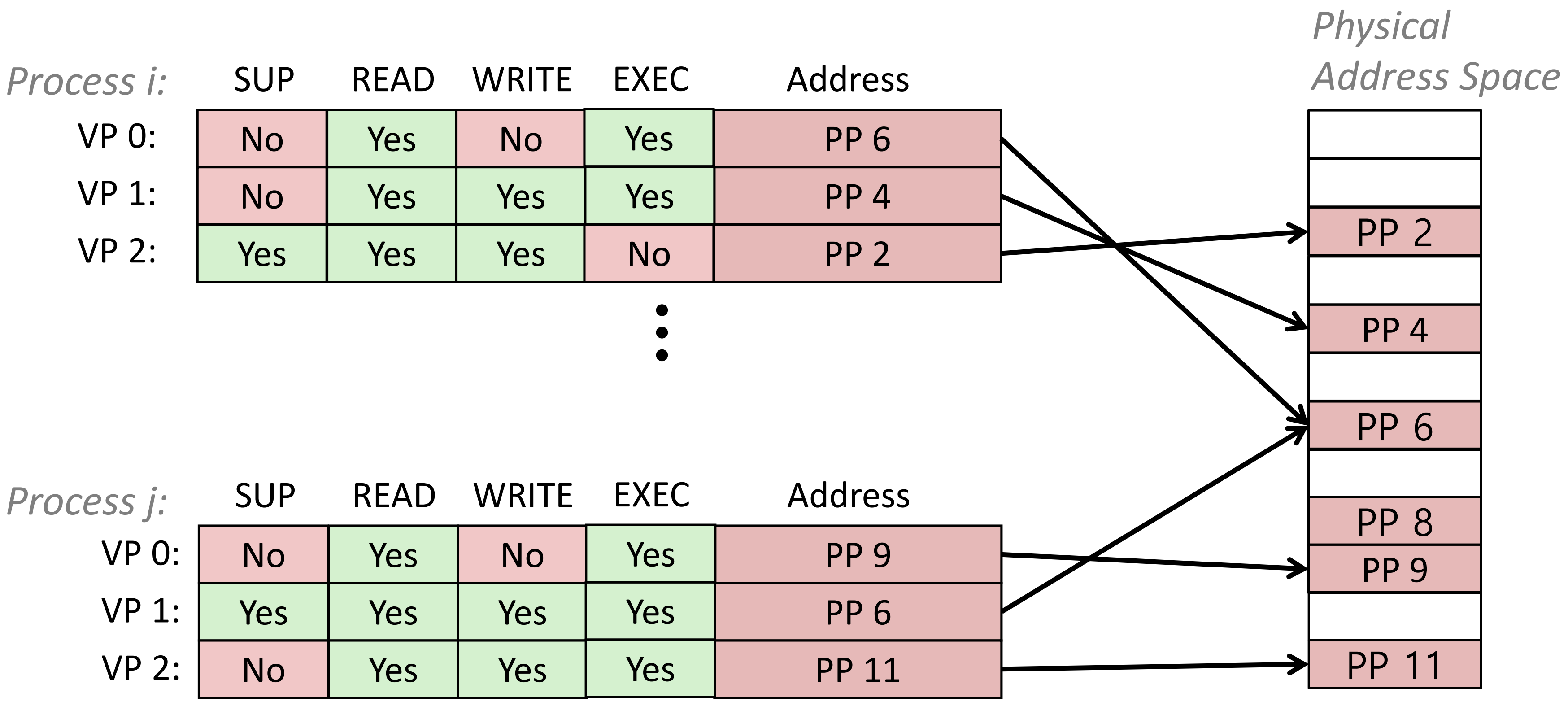
- Linking
  - Each program has similar virtual address space
  - Code, stack, and shared libraries always start at the same address

- 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


Memory invisible to user code
Kernel virtual memory
User stack (created at runtime)
%rsp (stack pointer)
Memory-mapped region for shared libraries
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Read-only segment (.init, .text, .rodata)
Loaded from the executable file
0x400000
Unused

VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- MMU checks these bits on each access



Process i:

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

⋮

Process j:

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

Outline

- Virtual Memory: Concepts
- Virtual Memory: Implementation

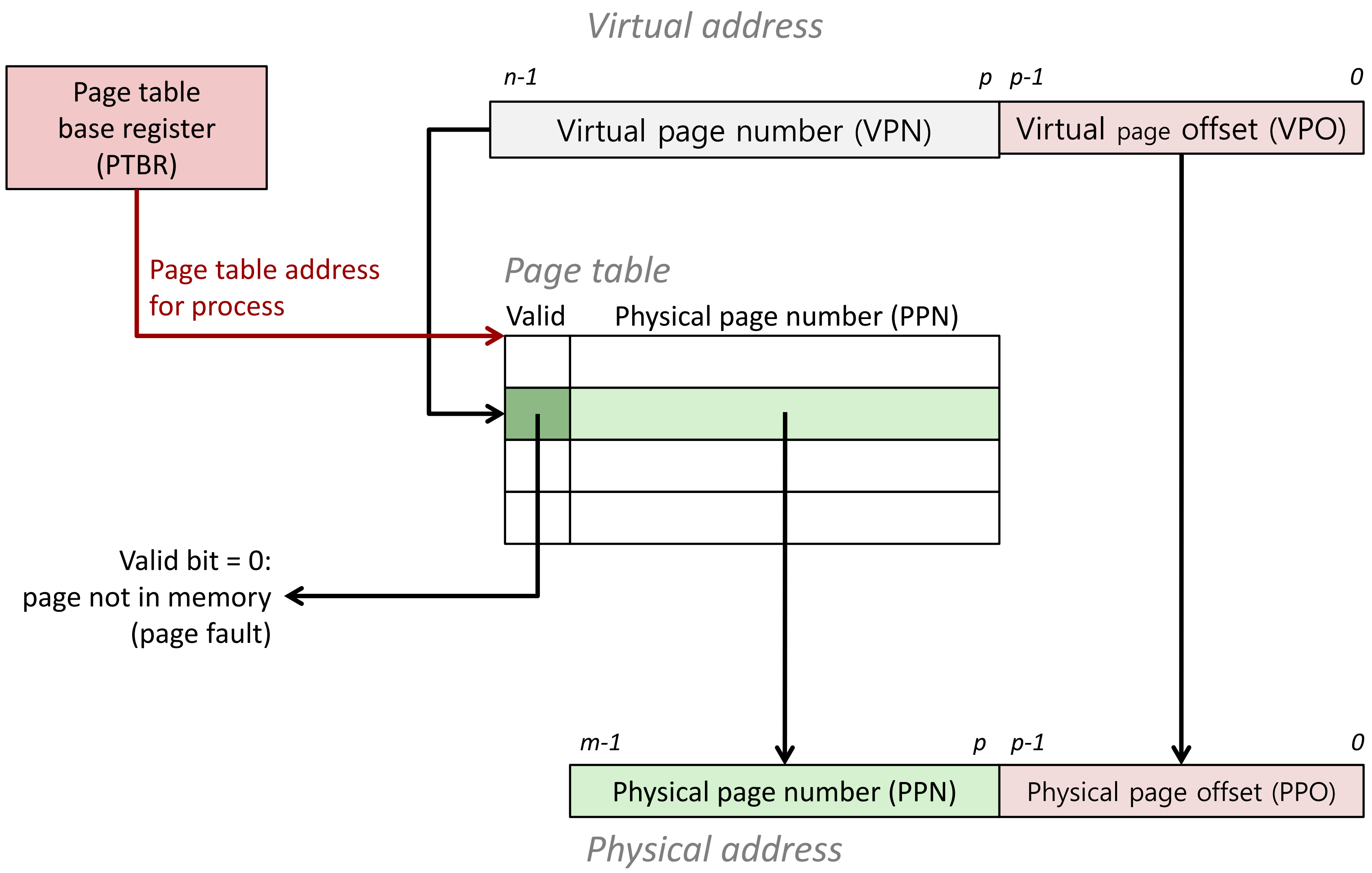
VM Address Translation

- Virtual Address Space
  - $V = \{0, 1, \ldots, N-1\}$
- Physical Address Space
  - $P = \{0, 1, \ldots, M-1\}$
- Address Translation
  - $\mathbf{MAP}: V \rightarrow P \cup \{\varnothing\}$
  - For virtual address ***a***:
    - $MAP(a) = a'$ if data at virtual address ***a*** is at physical address ***a'*** in ***P***
    - $MAP(a) = \varnothing$ if data at virtual address ***a*** is not in physical memory
      - Either invalid or stored on disk

Summary of Address Translation Symbols

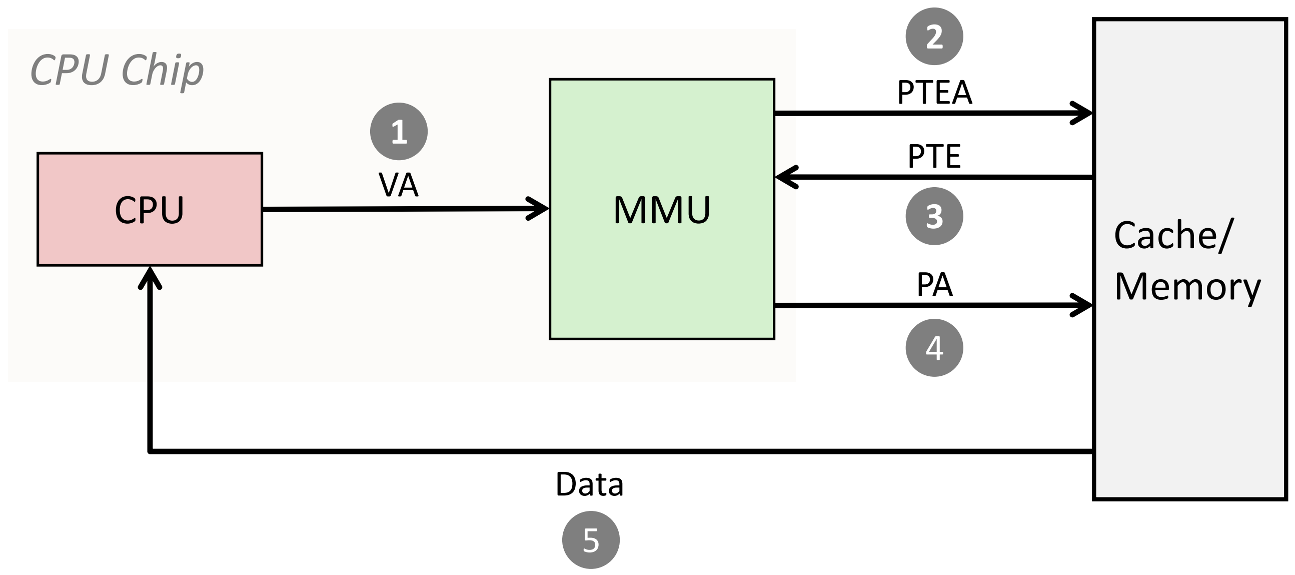
- Basic Parameters
  - **$N = 2^n$** : Number of addresses in virtual address space
  - **$M = 2^m$** : Number of addresses in physical address space
  - **$P = 2^p$** : Page size (bytes)
- Components of the virtual address (VA)
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
  - **TLBI**: 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

Address Translation With a Page Table



Virtual address
n-1
p
p-1
0
Virtual page number (VPN)
Virtual page offset (VPO)
Page table base register (PTBR)
Page table address for process
Page table
Valid
Physical page number (PPN)
Valid bit = 0: page not in memory (page fault)
m-1
p
p-1
0
Physical page number (PPN)
Physical page offset (PPO)
Physical address

Address Translation: Page Hit



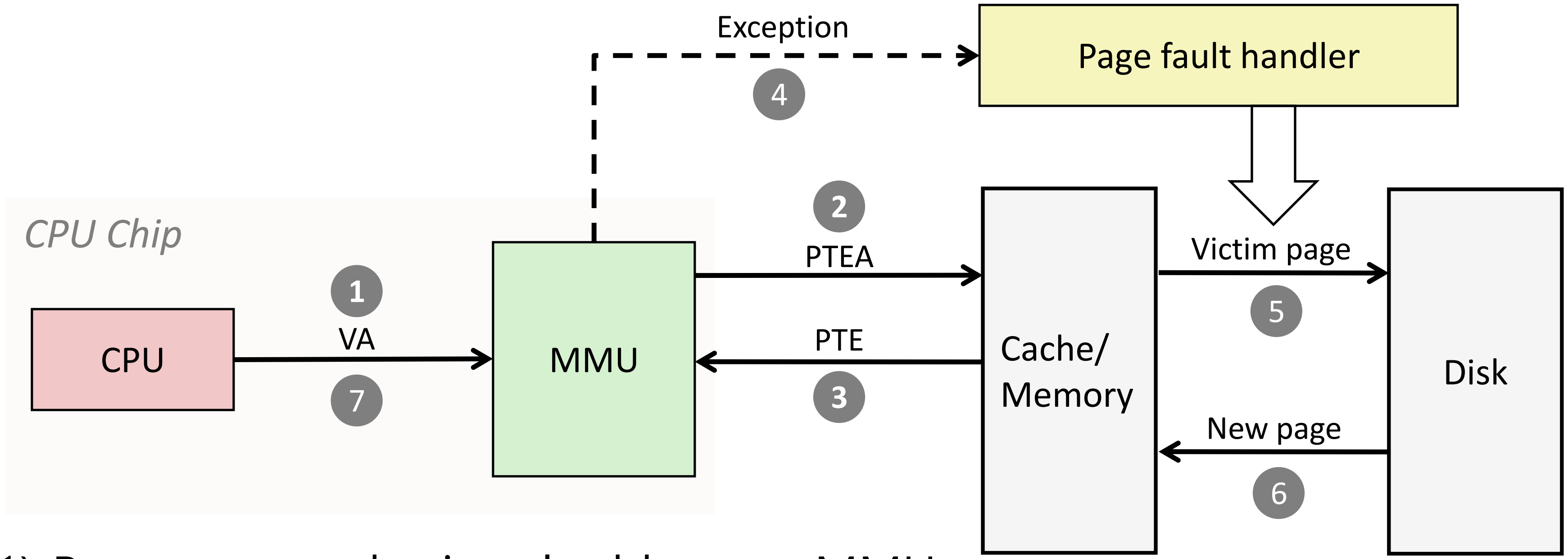
1) Processor sends virtual address to MMU

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

4) MMU sends physical address to cache/memory

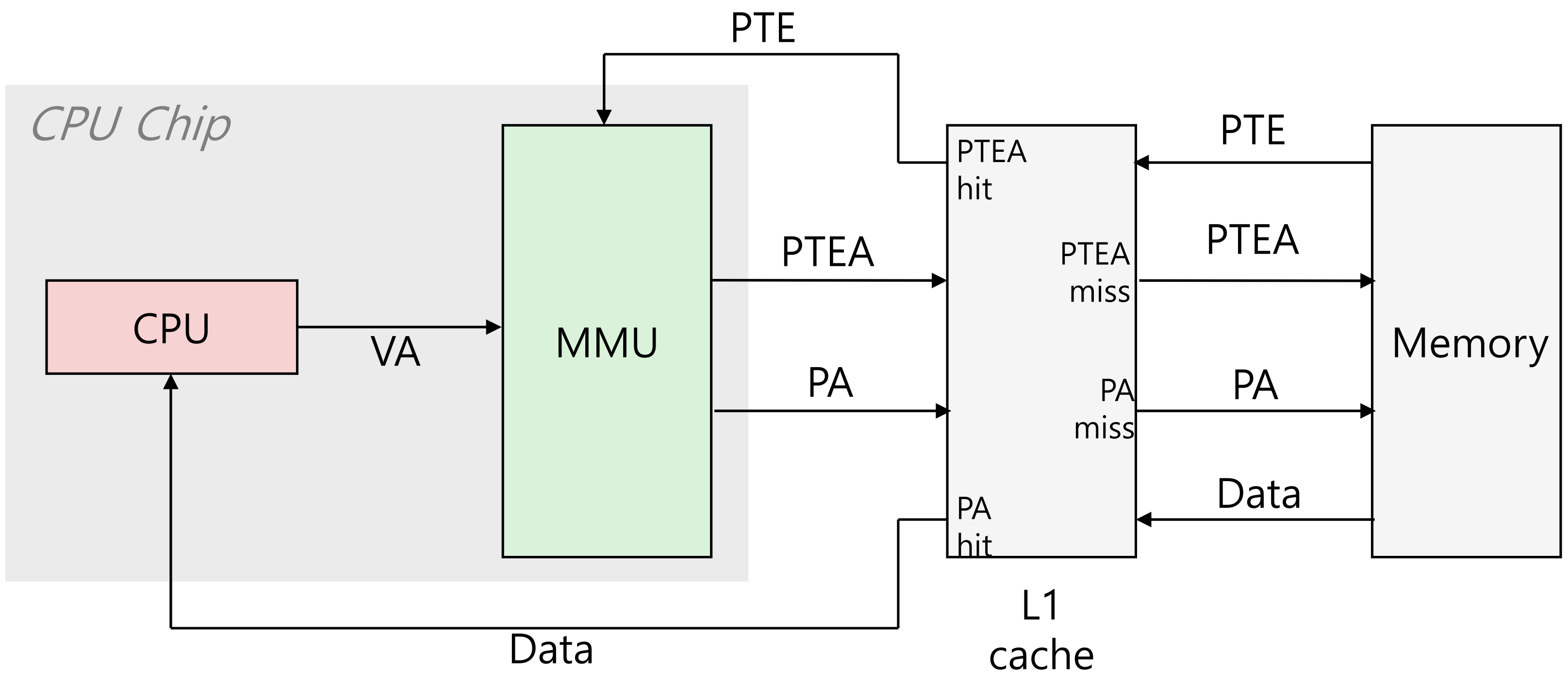
5) Cache/memory sends data word to processor

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

Integrating VM and Cache



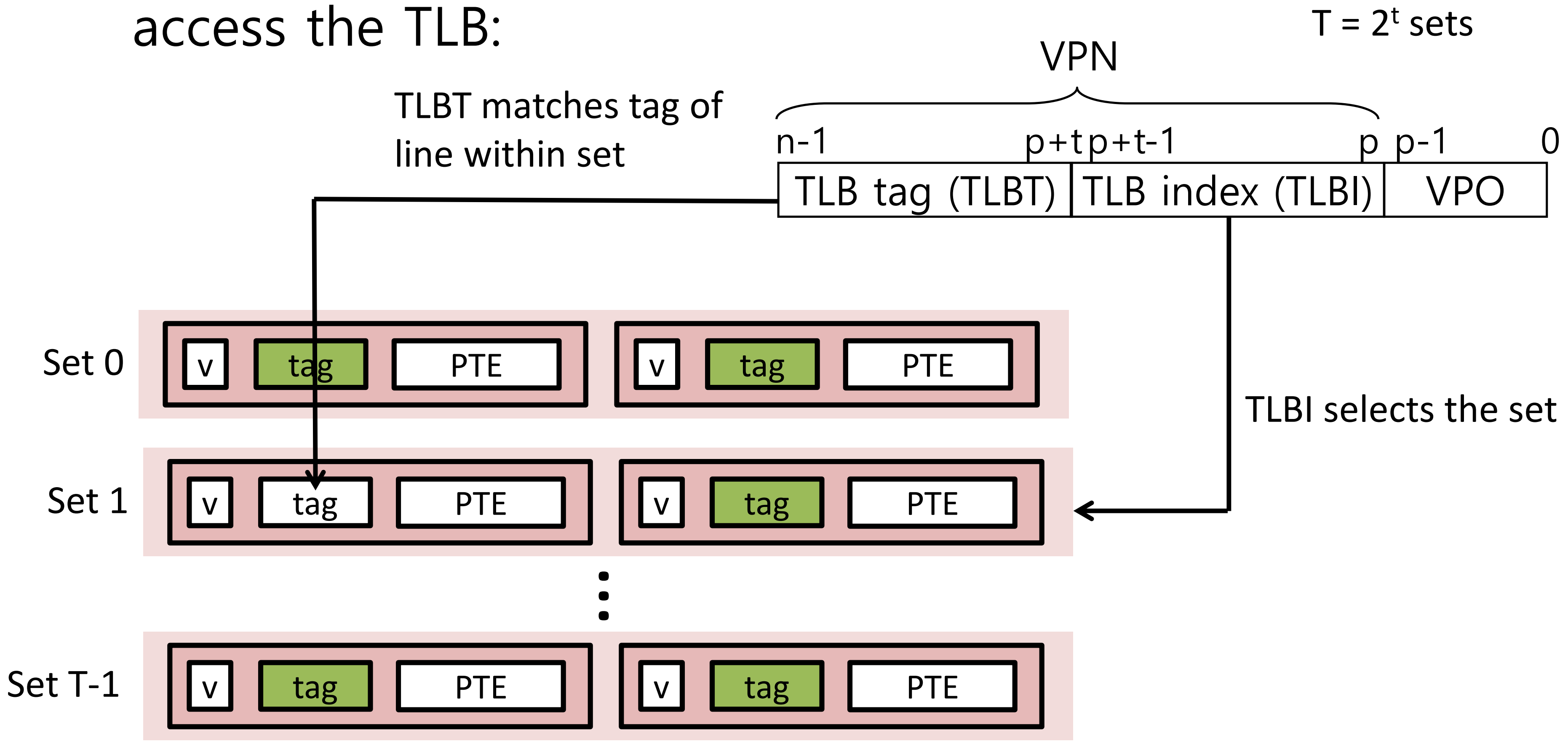
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Speeding up Translation with a TLB

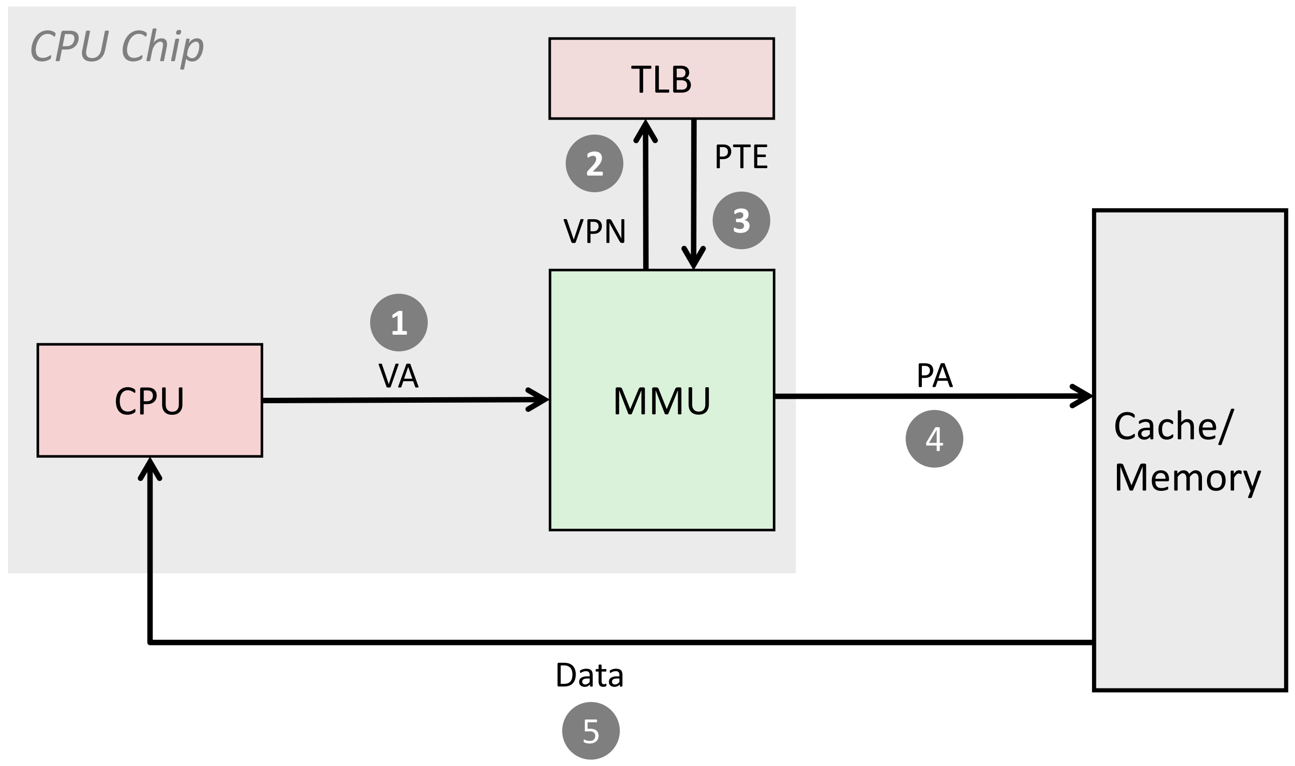
- Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay
- Solution: *Translation Lookaside Buffer* (TLB)
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers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

Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB:

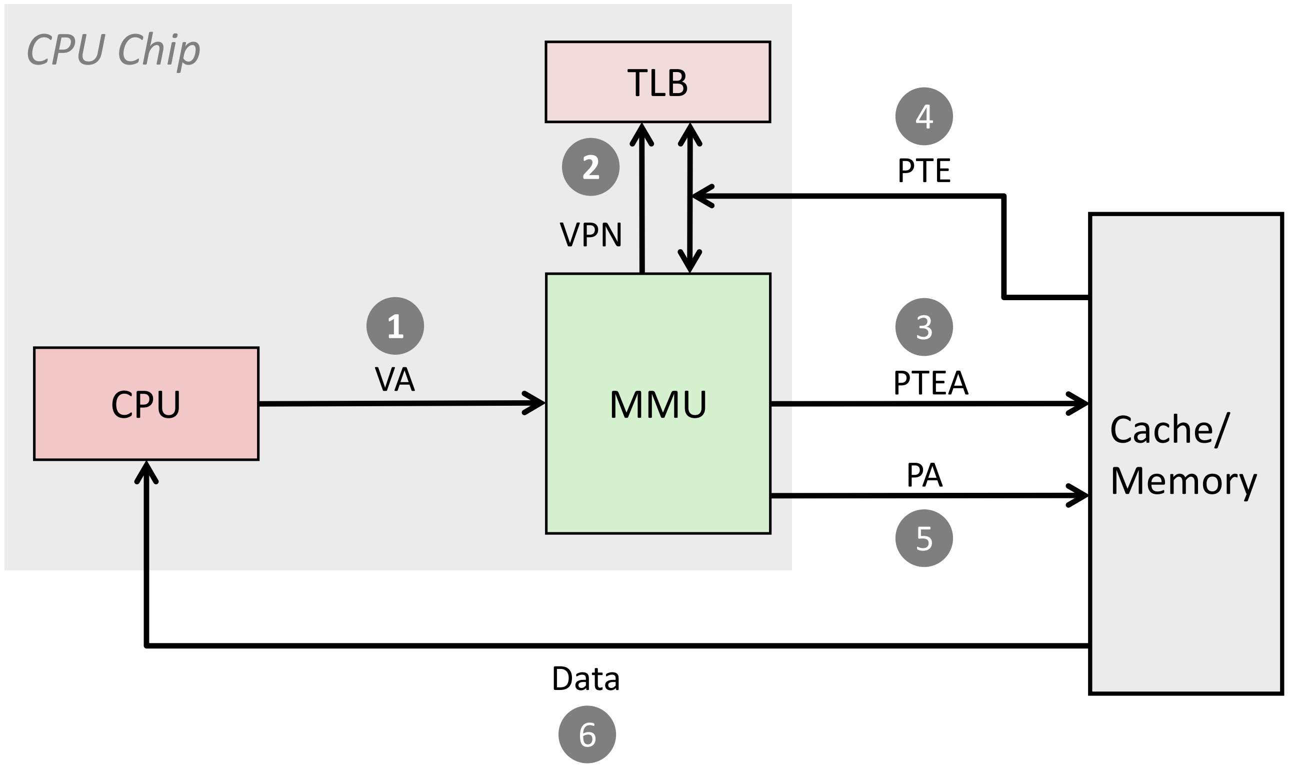


TLB Hit



A TLB hit eliminates a memory access

TLB Miss

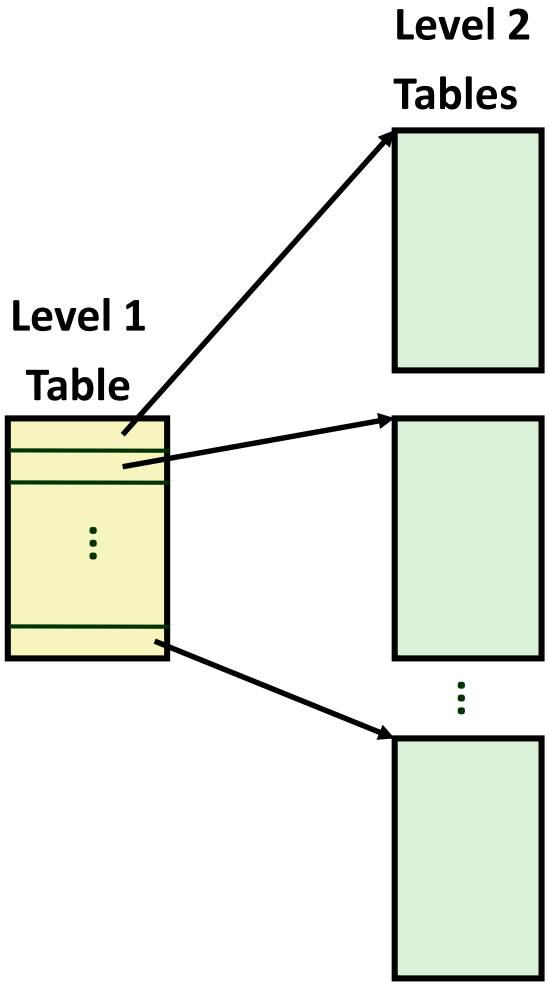


A TLB miss incurs an additional memory access (the PTE)

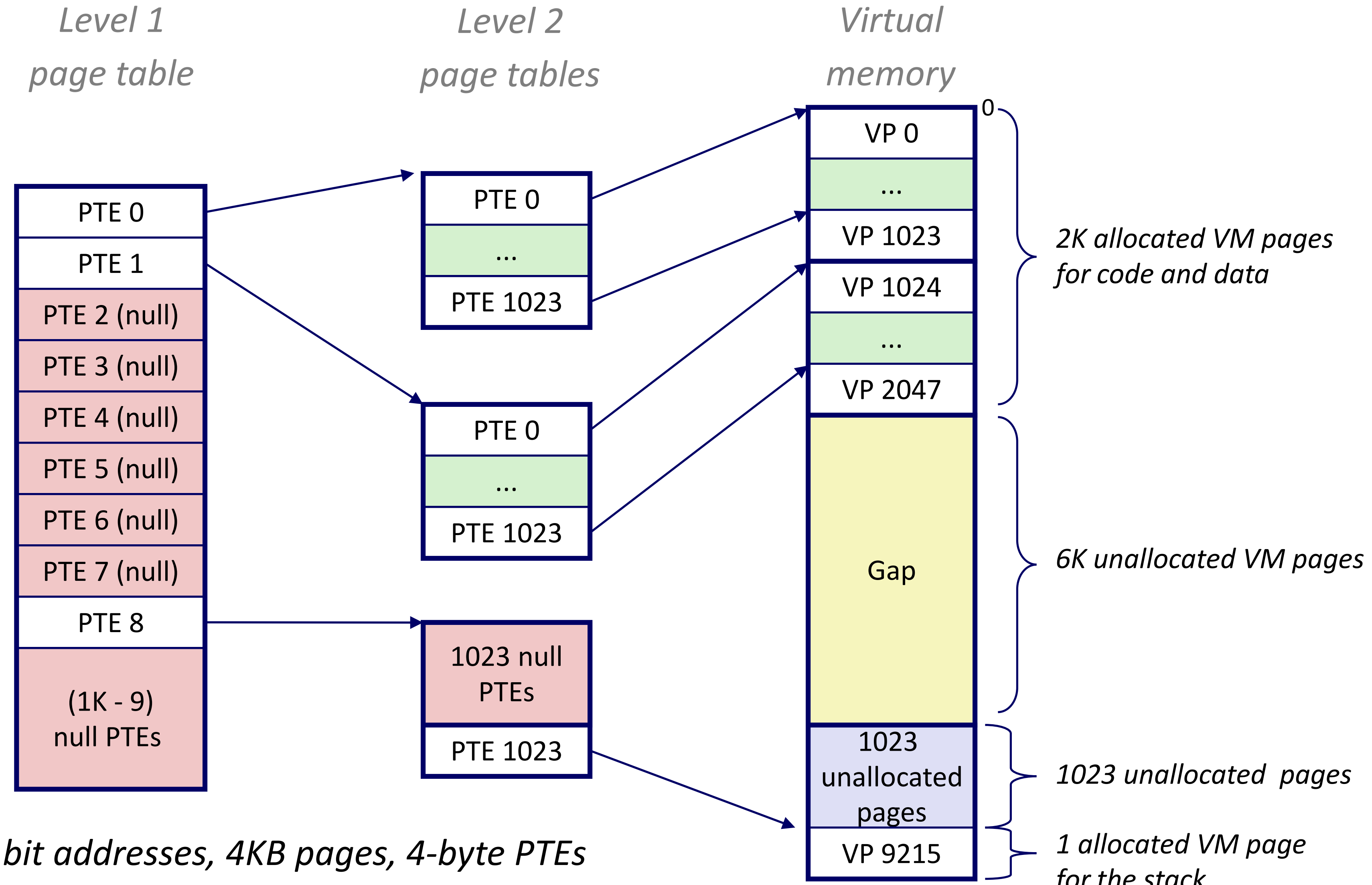
Fortunately, TLB misses are rare. Why?

Multi-Level Page Tables

- Suppose:
  - 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE
- Problem:
  - Would need a 512 GB page table!
    - $2^{48} * 2^{-12} * 2^{3} = 2^{39}$ bytes
- Common solution: Multi-level page table
- 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)

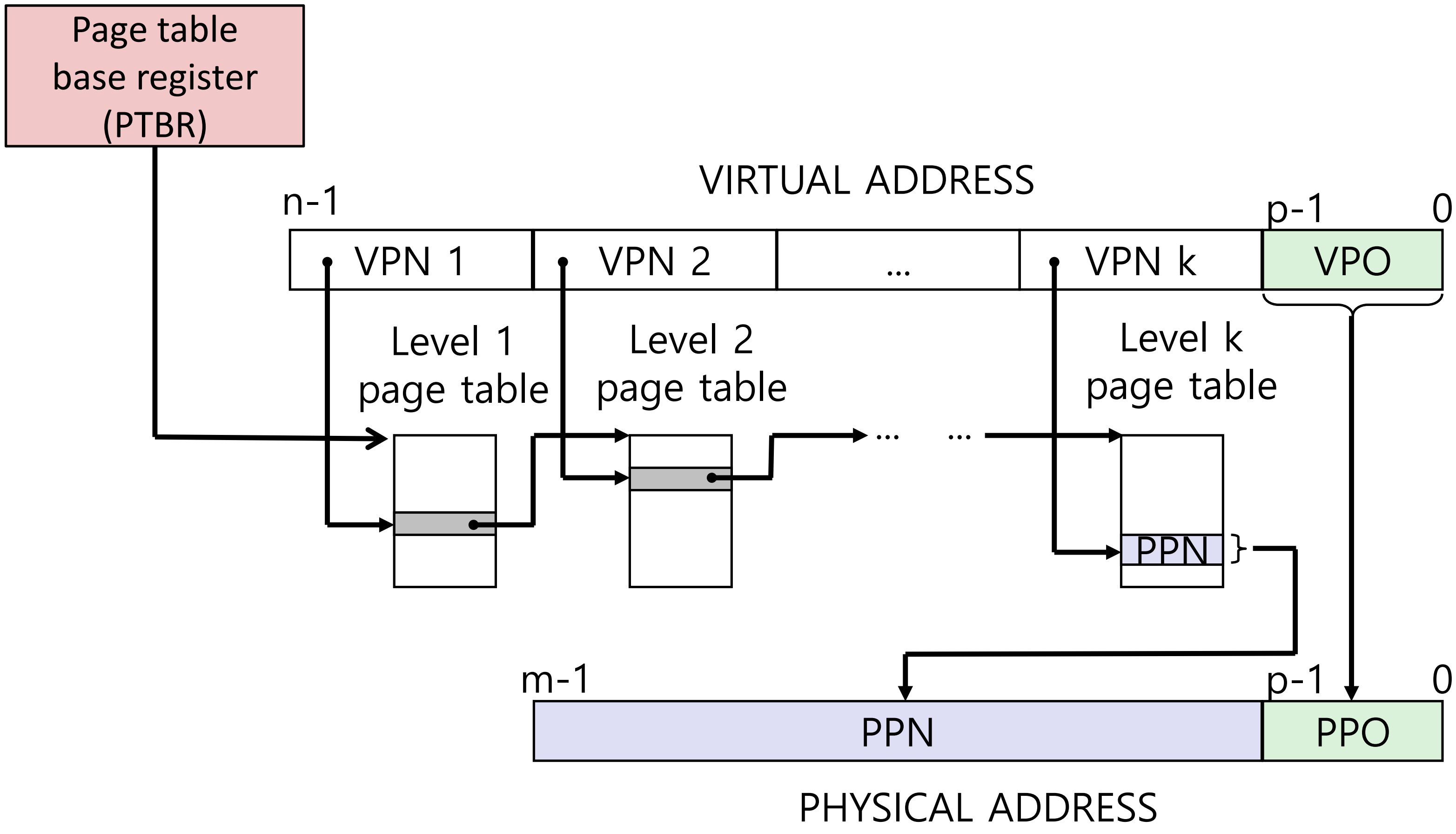


A Two-Level Page Table Hierarchy

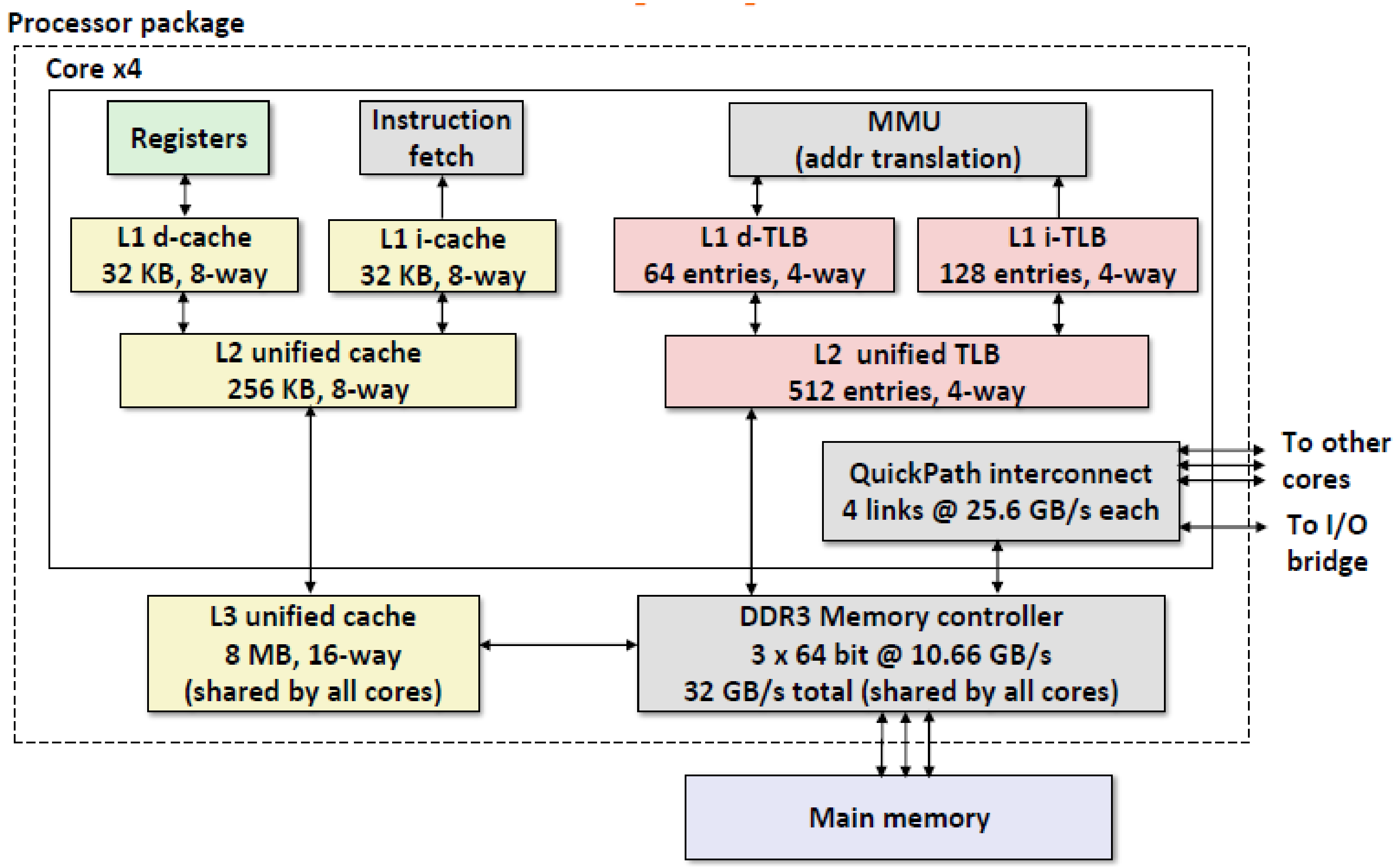


32 bit addresses, 4KB pages, 4-byte PTEs

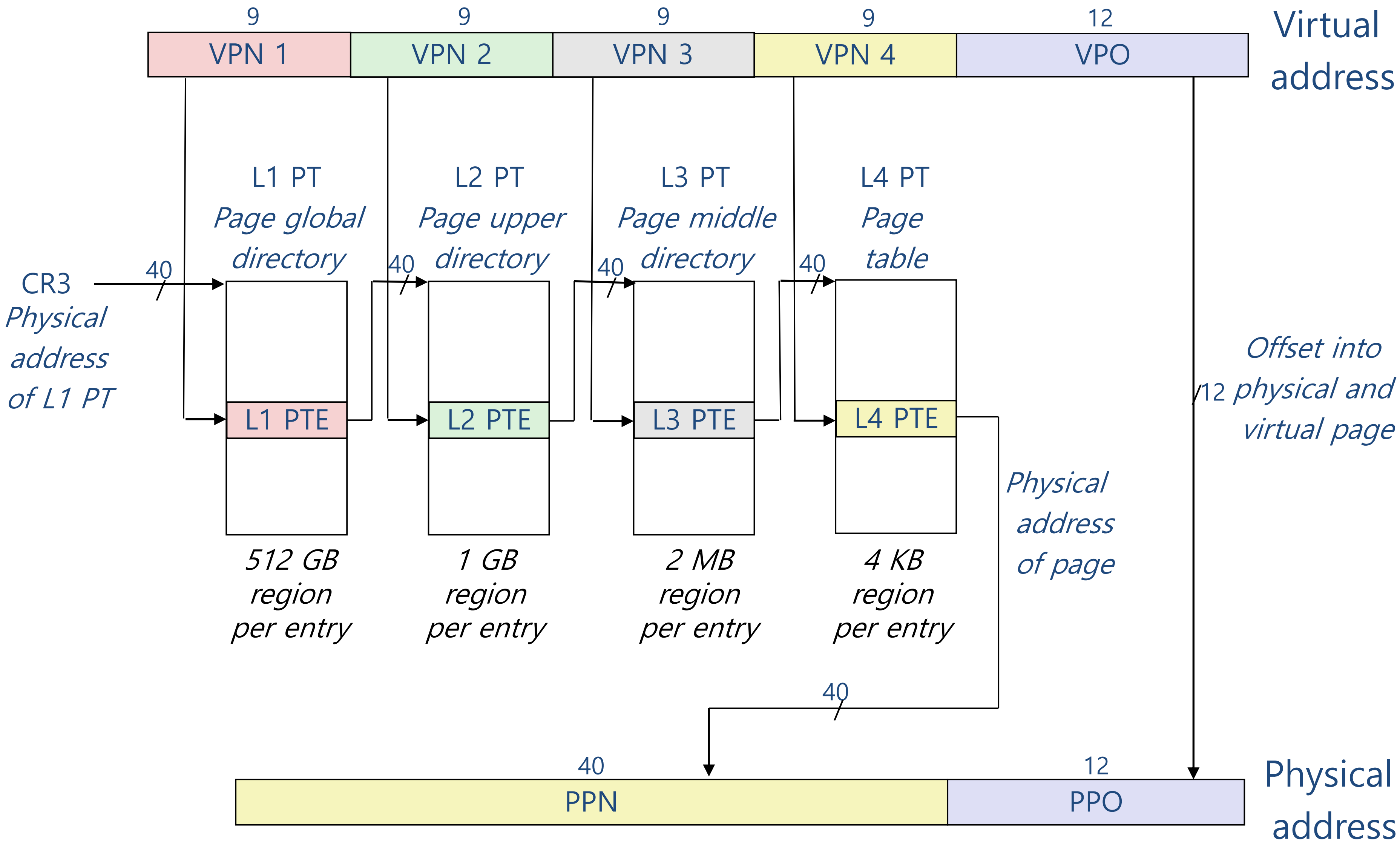
Translating with a k-level Page Table



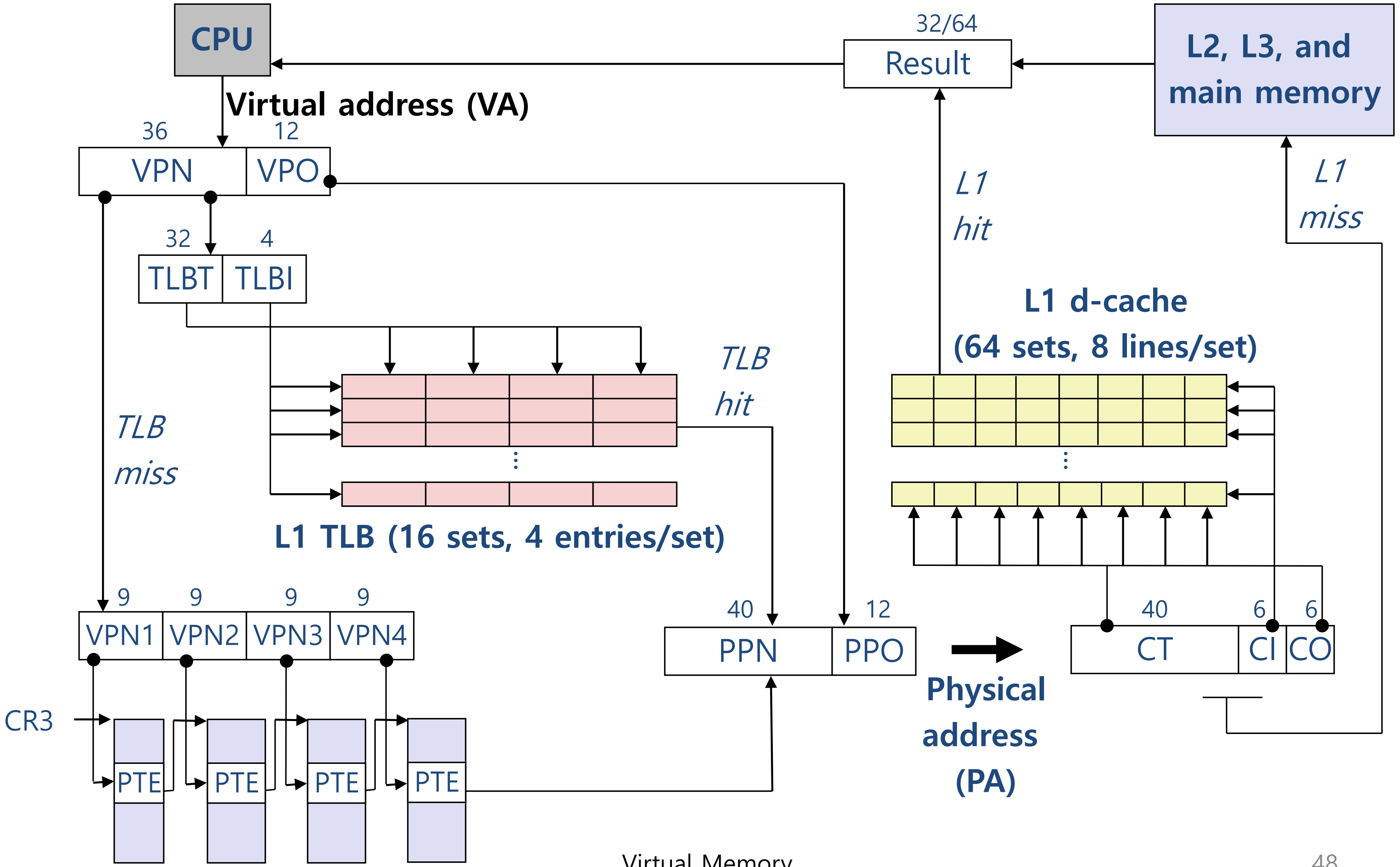
Intel Core i7 memory System


Processor package
Core x4
Registers
Instruction fetch
MMU (addr translation)
L1 d-cache 32 KB, 8-way
L1 i-cache 32 KB, 8-way
L1 d-TLB 64 entries, 4-way
L1 i-TLB 128 entries, 4-way
L2 unified cache 256 KB, 8-way
L2 unified TLB 512 entries, 4-way
QuickPath interconnect 4 links @ 25.6 GB/s each
To other cores
To I/O bridge
L3 unified cache 8 MB, 16-way (shared by all cores)
DDR3 Memory controller 3 x 64 bit @ 10.66 GB/s 32 GB/s total (shared by all cores)
Main memory

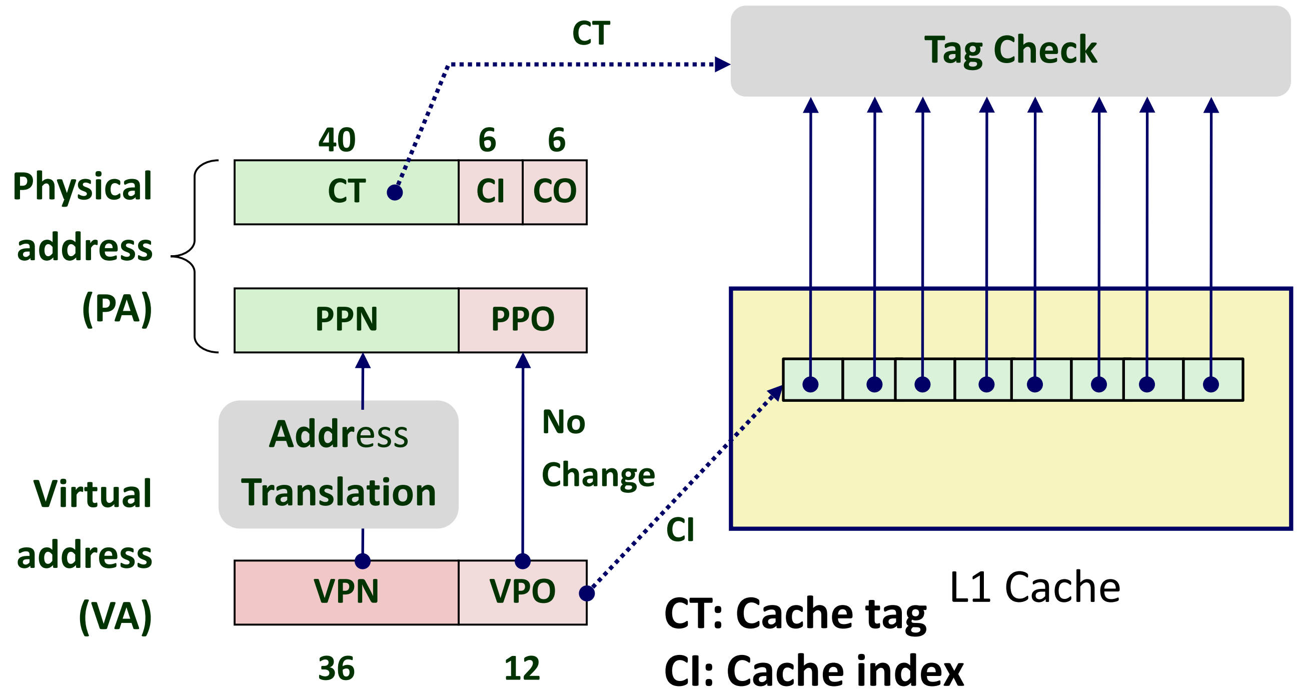
Core i7 Page Table Translation



End-to-end Core i7 Address Translation



Cute Trick for Speeding Up L1 Access



CT: Cache tag
CI: Cache index
CO: Byte offset within cache line

- Observation
  - Bits that determine CI identical in virtual and physical address
  - Can index into cache while address translation taking place
  - Generally we hit in TLB, so PPN bits (CT bits) available next
  - "Virtually indexed, physically tagged"
  - Cache carefully sized to make this possible

Summary

- 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

Questions?