

CSC 252: Computer Organization

Spring 2026: Lecture 20

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

VM Concepts Summary

- Conceptually, *virtual memory* is an array of N *pages*
 - Each virtual page is either in physical memory, or on disk, or unallocated
- The physical memory (PM) is an array of M *pages* stored in DRAM.
- Page size is the same for VM and PM
- $M \ll N$
- On a 64-bit machine, virtual memory size = 2^{64}
- Physical memory size is much much smaller:
 - iPhone 8: 2 GB (2^{31})
 - 15-inch Macbook Pro 2017: 16 GB (2^{34})
- Store only the most frequently used pages in the physical memory
- If a page is not on the physical memory, have to first swap it from the disk to the DRAM.

Calculate Bits in VA and PA

- In a 64-bit machine, VA is 64-bit long. Assuming PM is 4 GB. Assuming 4 KB page size.
- How many bits for page offset?
 - 12. Same for VM and PM
- How many bits for Virtual Page Number?
 - 52, i.e., 2^{52} virtual pages
- How many bits for Physical Page Number?
 - 20, i.e., 2^{20} physical pages



Calculate the Page Table Size

- Assume 4KB page, 4GB virtual memory, each PTE is 8 Bytes
 - $4\text{GB}/4\text{KB} = 1\text{M}$ virtual pages
 - 1M PTEs in a page table
 - 8MB total size per page table
- Do you need a page table for each process?
 - Yes

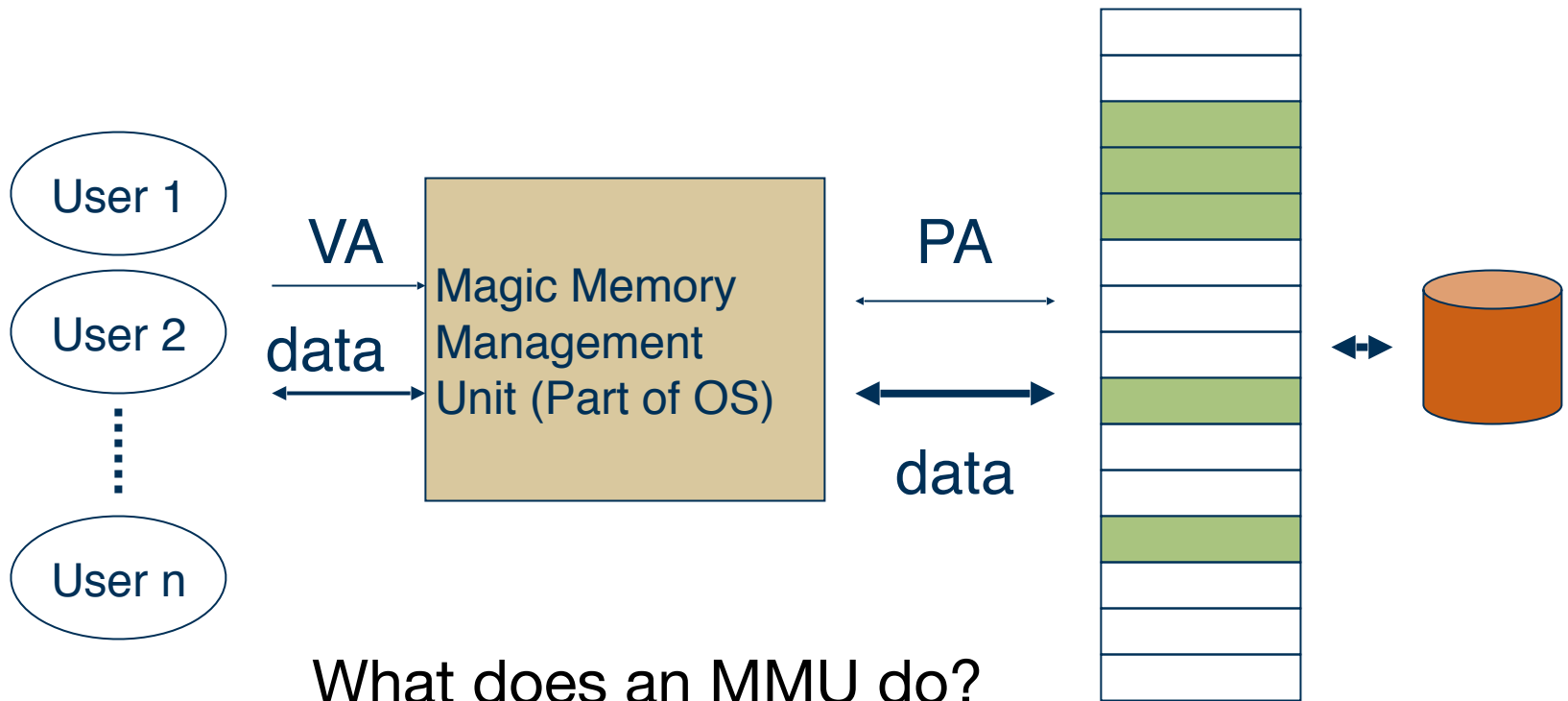
Where Does Page Table Live?

- It needs to be at a specific location where we can find it
 - Some special SRAM?
 - In main memory?
 - On disk?
- ~MBs of a page table *per process*
 - Too big for on-chip SRAM (c.f., a L1 cache is ~32 KB)
 - Too slow to access in disk
 - Put the page table in DRAM, with its start address stored in a special register (Page Table Base Register). More on this later.

Today

- VM basic concepts and operation
- Other critical benefits of VM
- Address translation

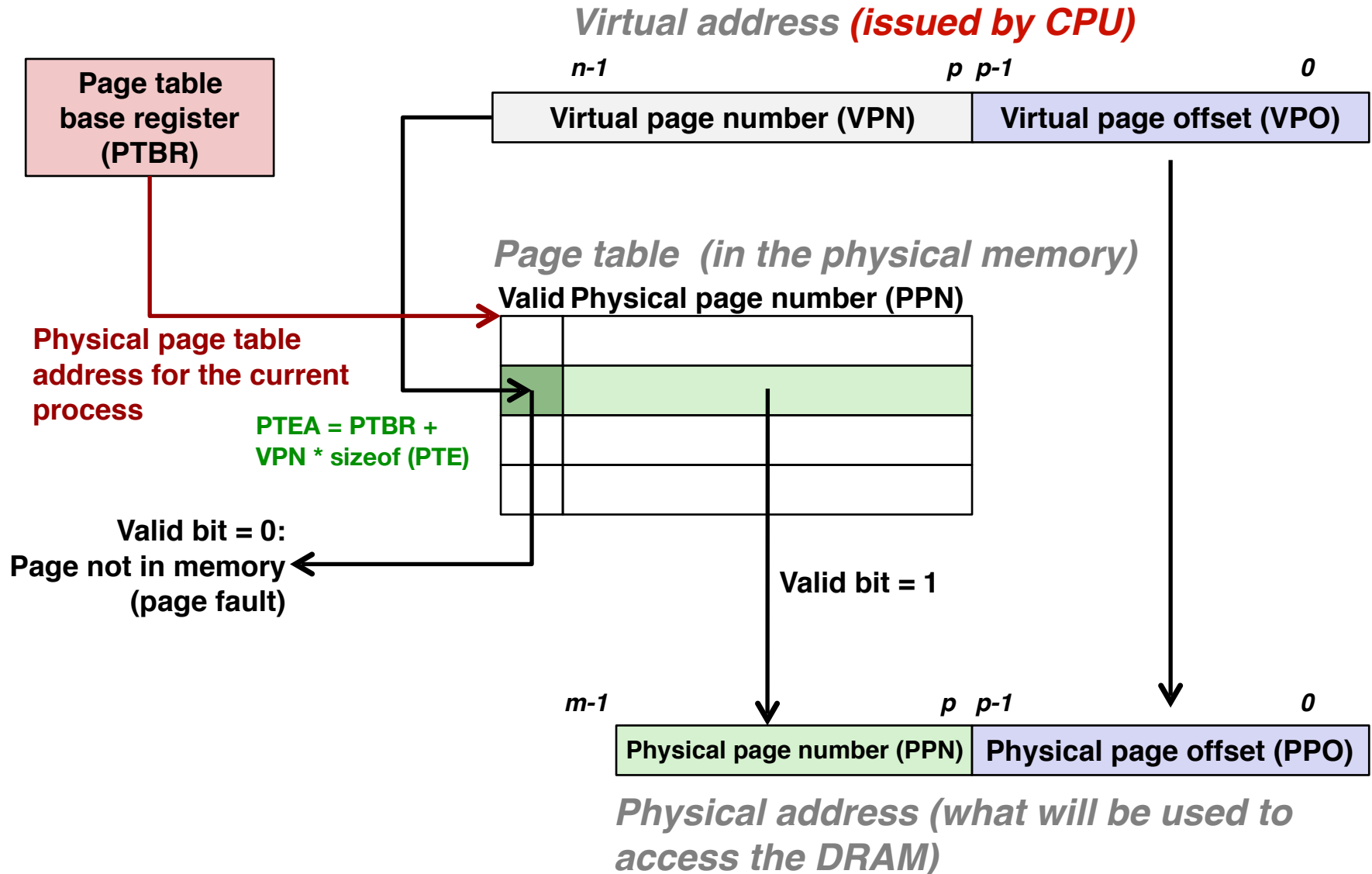
So Far...



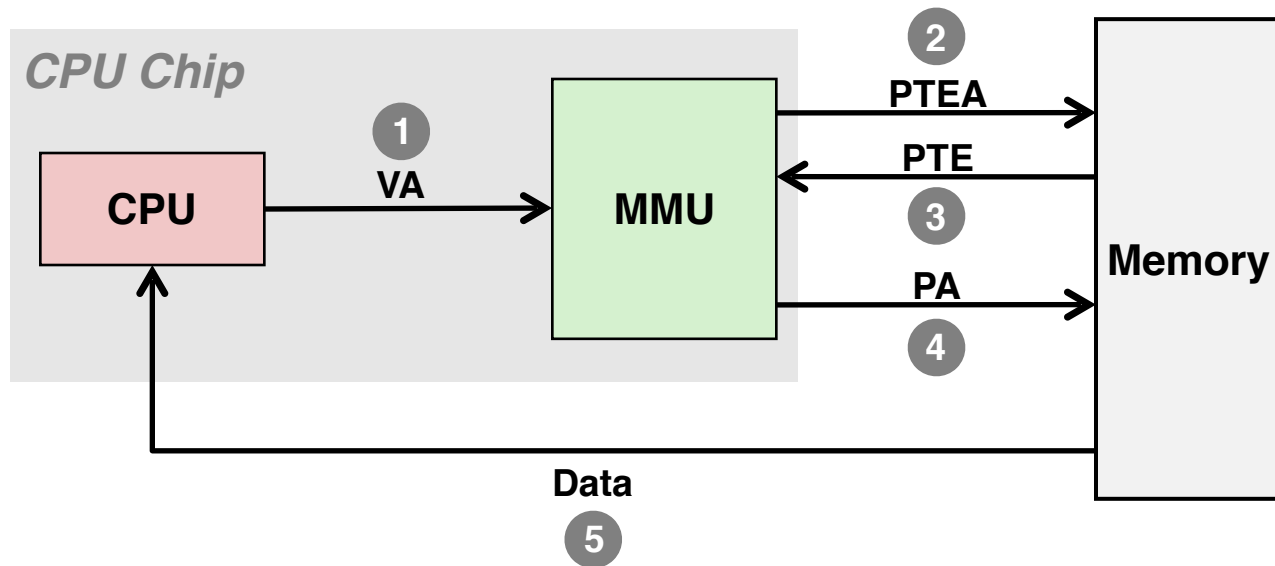
What does an MMU do?

- Translate address from a VA to PA
 - Enforce permissions
 - Fetch from disk

Address Translation With a Page Table



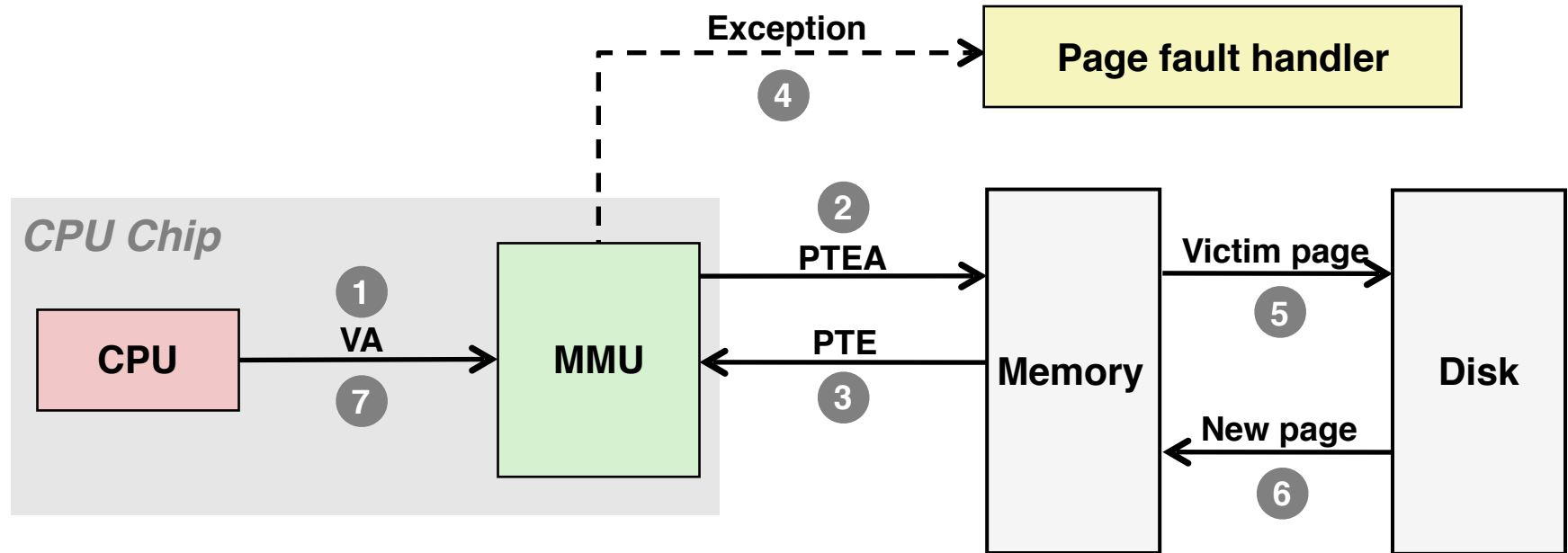
Address Translation: Page Hit



- 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

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

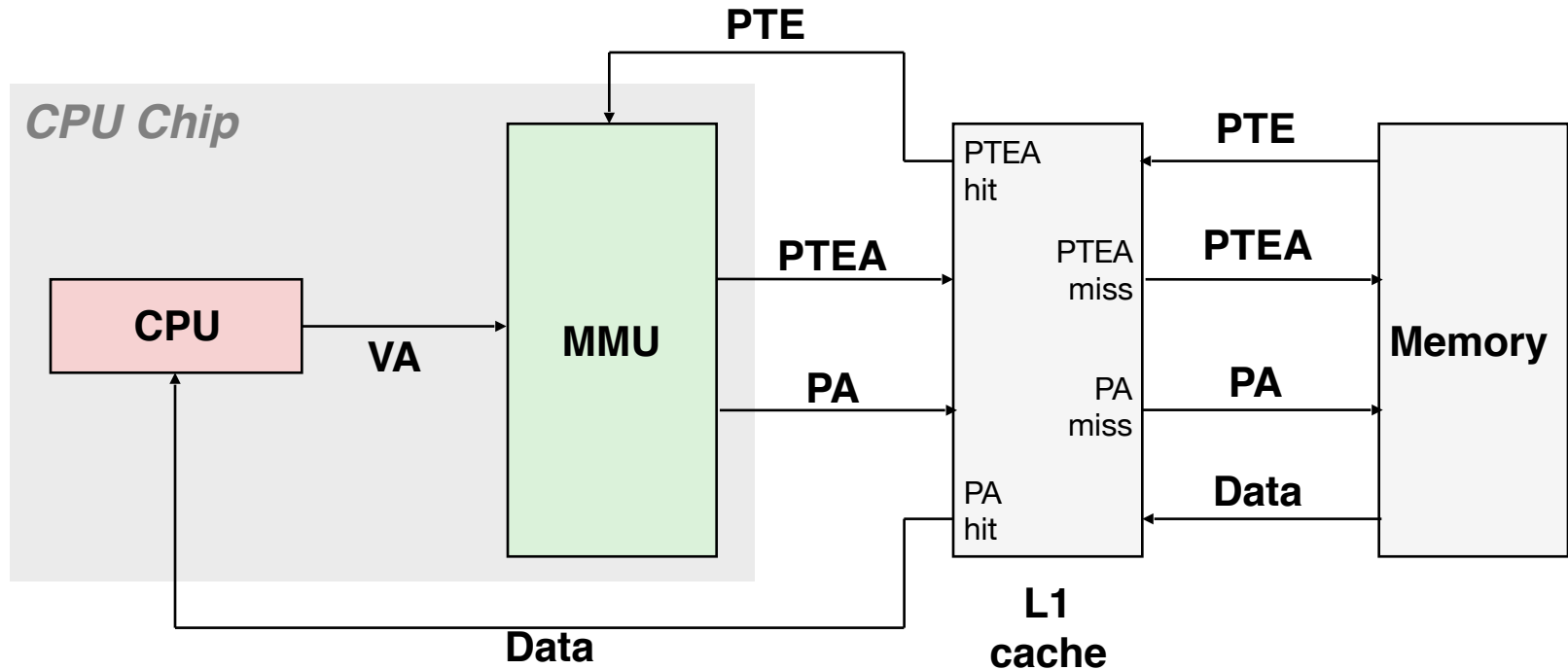
Address Translation: Page Fault



- 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

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

Integrating VM and Cache



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

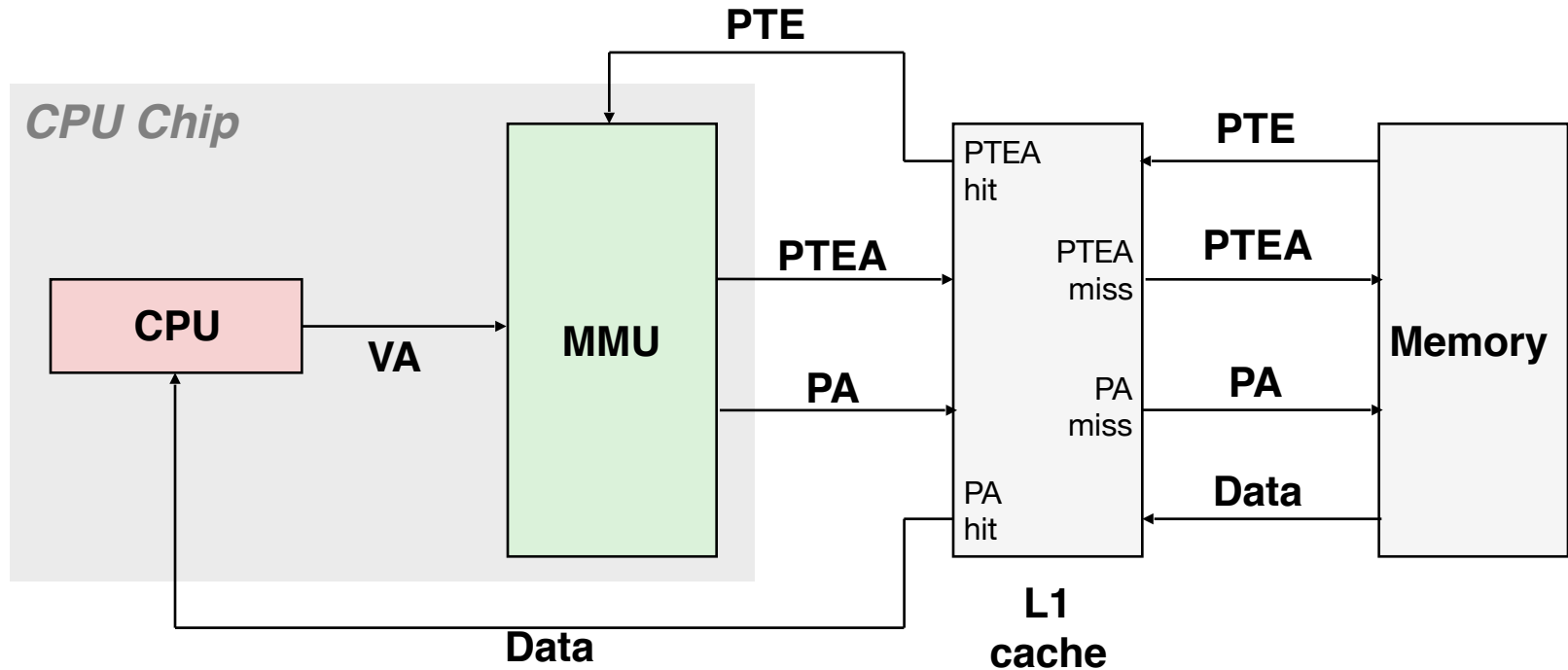
Today

- Three Virtual Memory Optimizations
 - TLB
 - Virtually-indexed, physically-tagged cache
 - Page the page table (a.k.a., multi-level page table)
- Case-study: Intel Core i7/Linux example

Speeding up Address Translation

- Problem: Every memory load/store requires two memory accesses: one for PTE, another for real
 - The PTE access is kind of an overhead
 - Can we speed it up?
- Page table entries (PTEs) are already cached in L1 data cache like any other memory data. But:
 - PTEs may be evicted by other data references
 - PTE hit still requires a small L1 delay

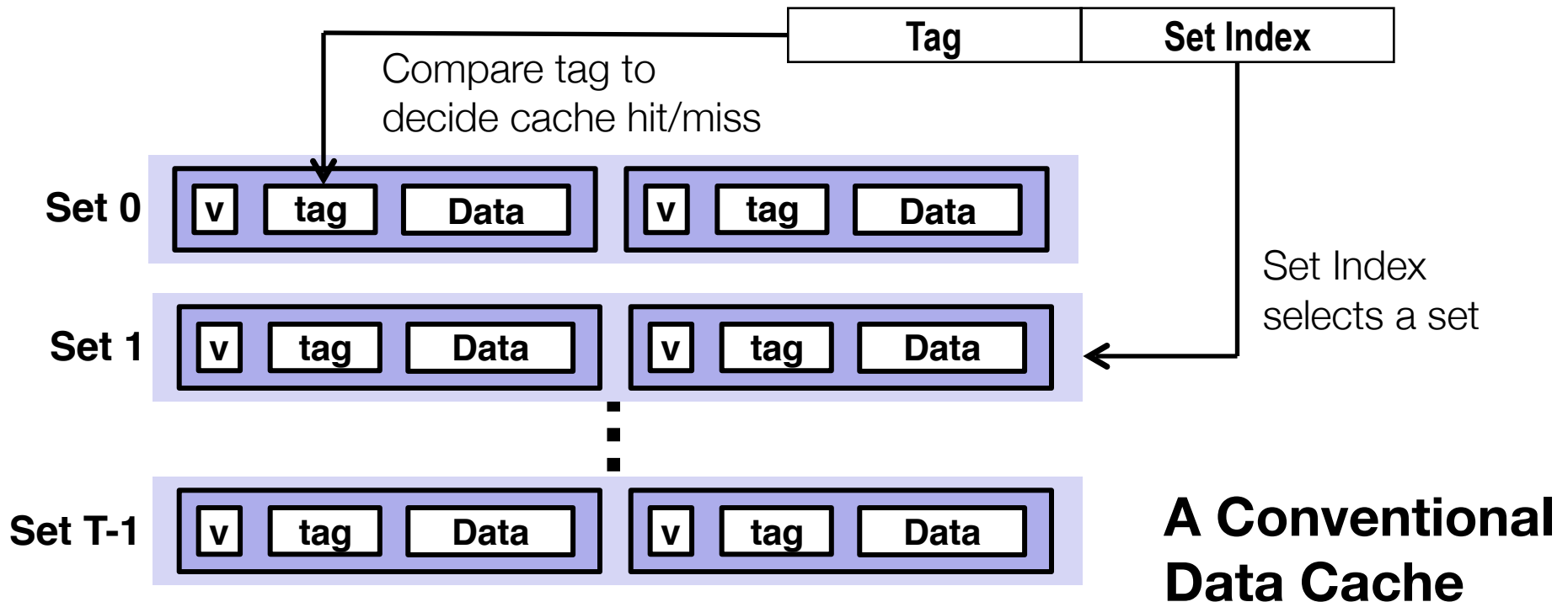
Recall: Page Table is Cached



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

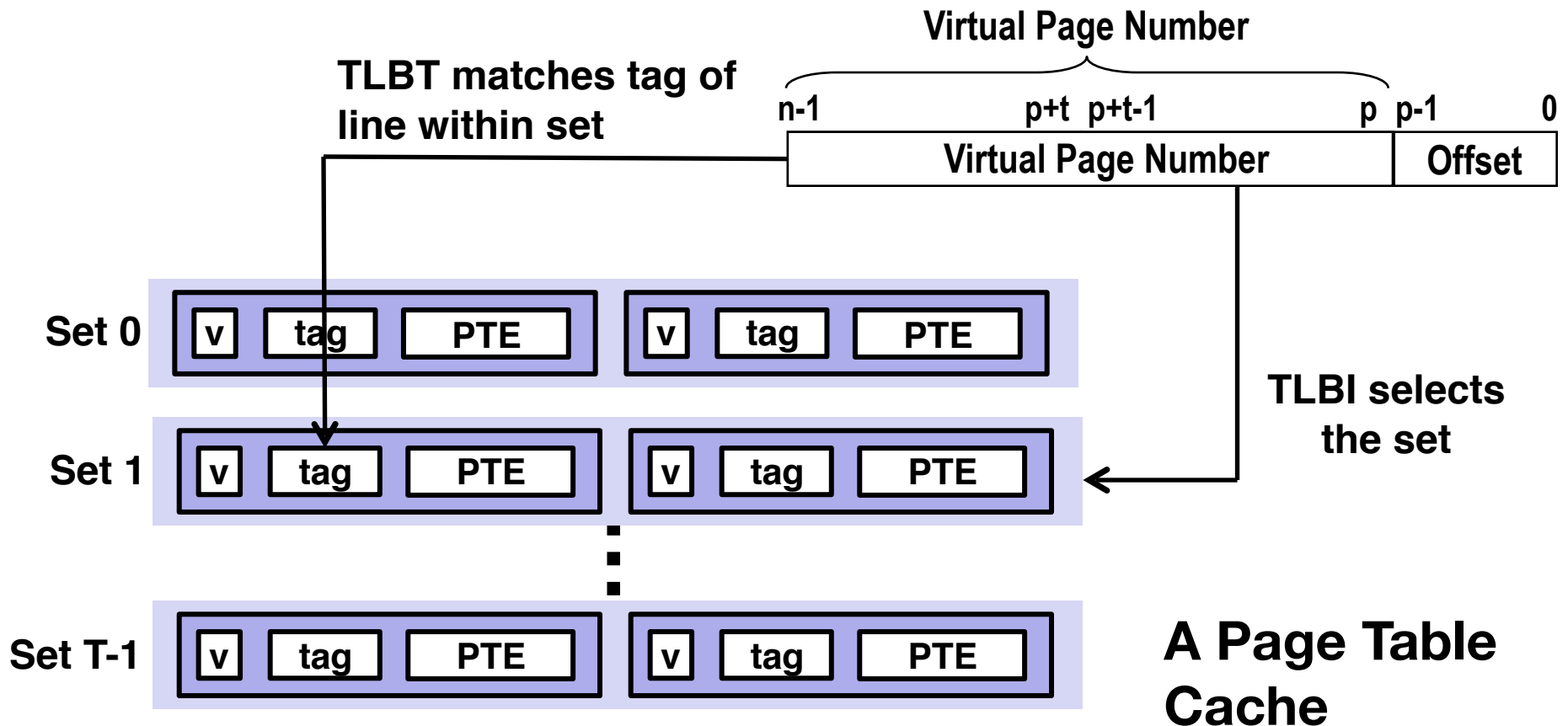
Speeding up Translation with a TLB

- Solution: *Translation Lookaside Buffer* (TLB)
 - Think of it as a dedicated cache for page table
 - Small set-associative hardware cache in MMU
 - Contains complete page table entries for a small number of pages

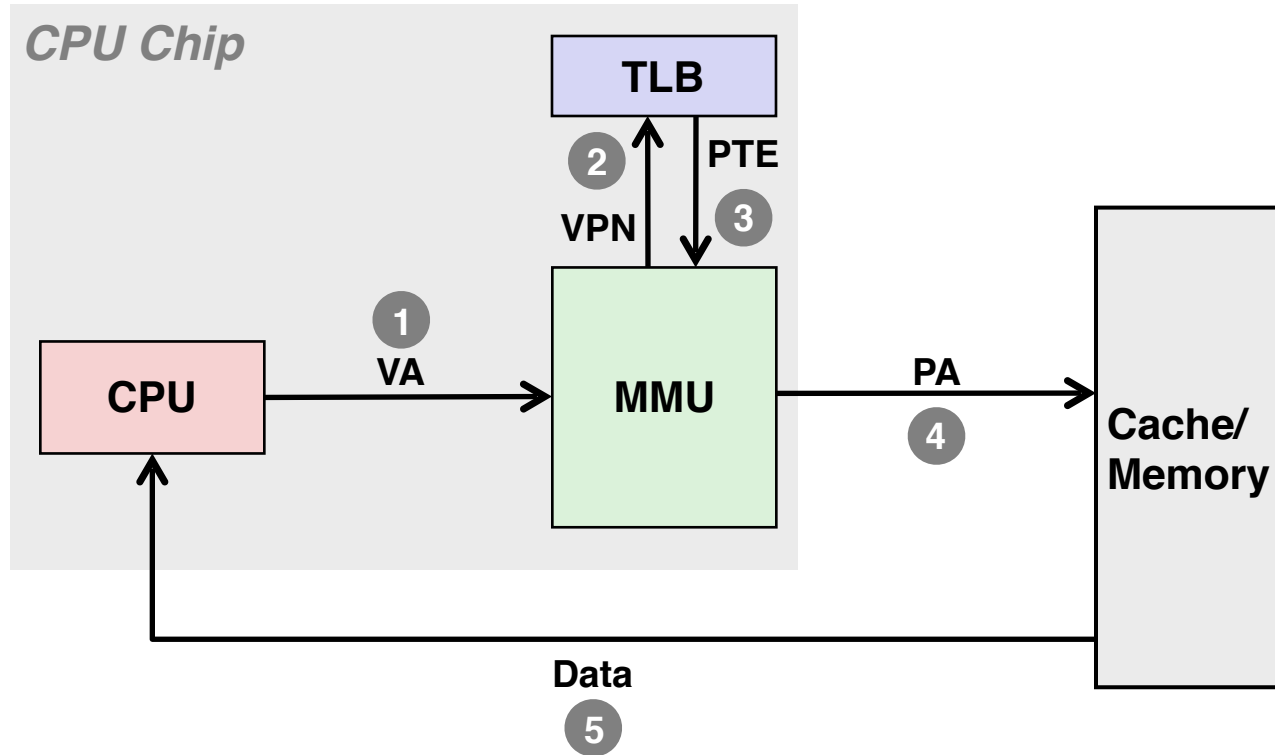


Accessing the TLB

- MMU uses the Virtual Page Number portion of the virtual address to access the TLB:

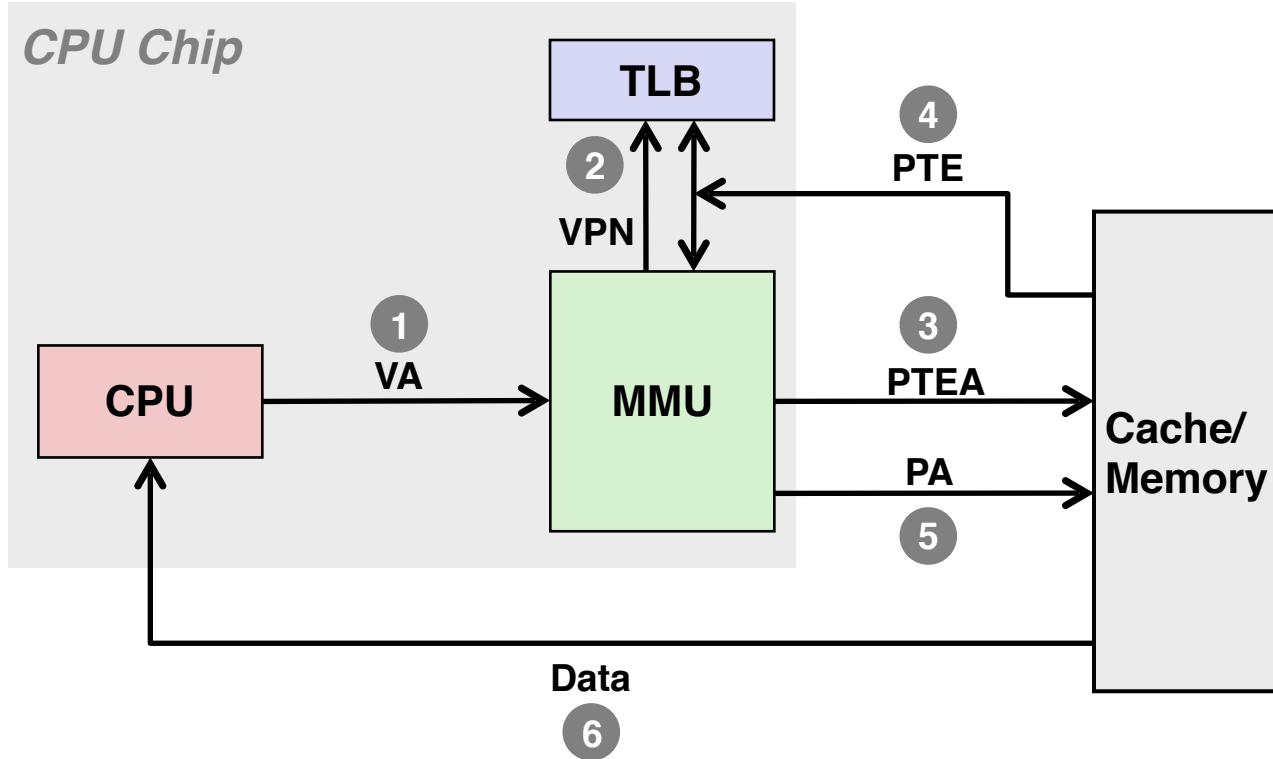


TLB Hit



A TLB hit eliminates a memory access

TLB Miss

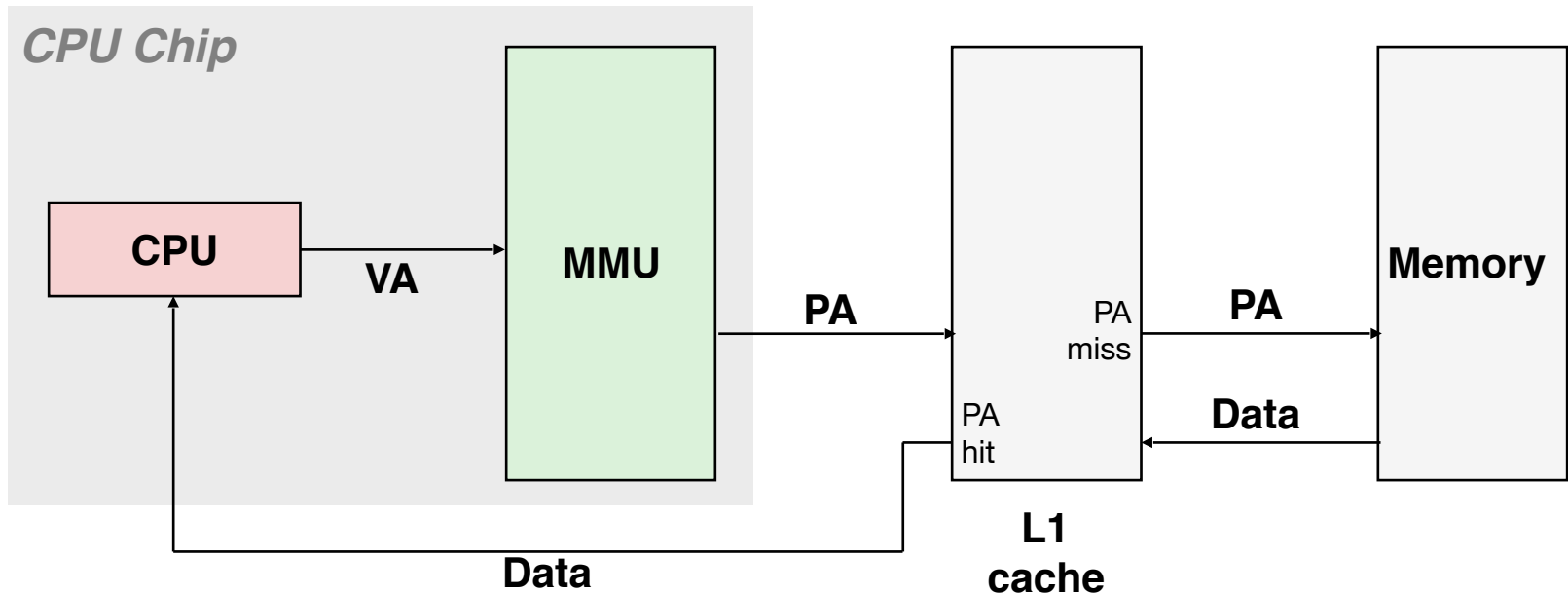


Today

- Three Virtual Memory Optimizations
 - TLB
 - Virtually-indexed, physically-tagged cache
 - Page the page table (a.k.a., multi-level page table)
- Case-study: Intel Core i7/Linux example

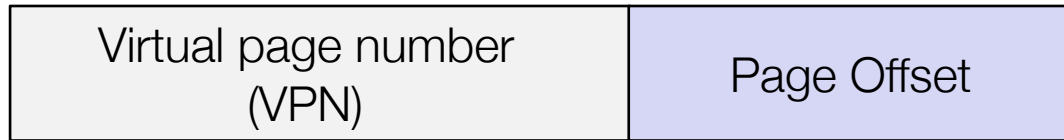
Performance Issue in VM

- Address translation and cache accesses are serialized
 - First translate from VA to PA
 - Then use PA to access cache
 - Slow! Can we speed it up?

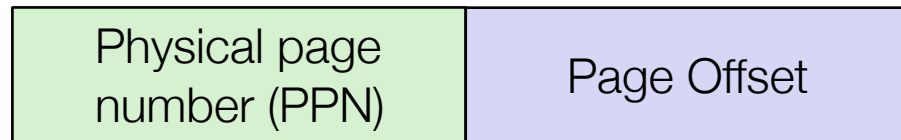


Observe Address Translation

Virtual
Address



Physical
Address



Unchanged!!

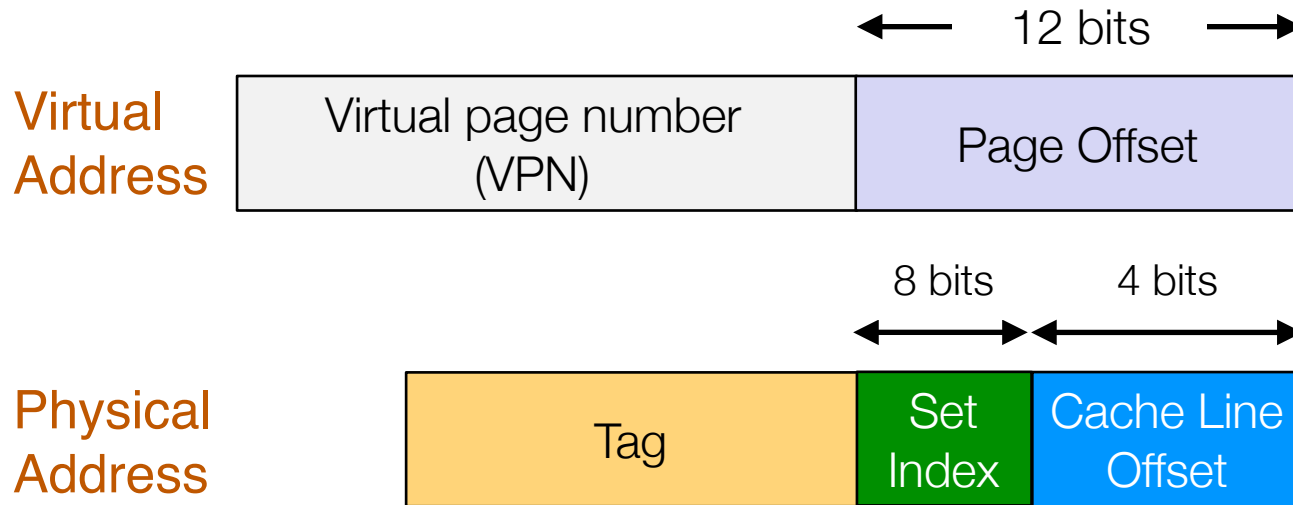


Virtually-Indexed,
Physically-Tagged
Cache

**L1
cache**

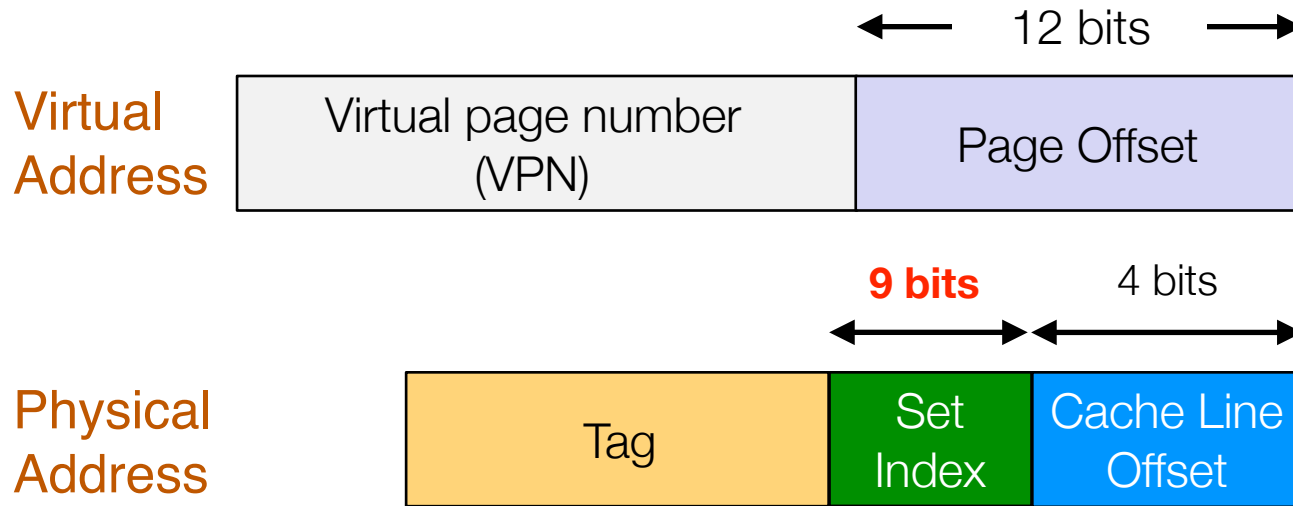
- Set Index + Cache Line Offset = Page Offset
- Indexing into cache in parallel with translation (TLB access)
- If TLB hits, can get the data back in one cycle

Any Implications?



- Assuming 4K page size, cache line size is 16 bytes.
- Set Index = 8 bits. Can only have 256 Sets => Limit cache size
- Increasing cache size then requires increasing associativity
 - Not ideal because that requires comparing more tags
- Solutions?

Any Implications?



- What if we use 9 bits for Set Index? More Sets now.
- How can this still work?
- The least significant bit in VPN and PPN must be the same
- That is: an even VA must be mapped to an even PA, and an odd VA must be mapped to an odd PA

Today

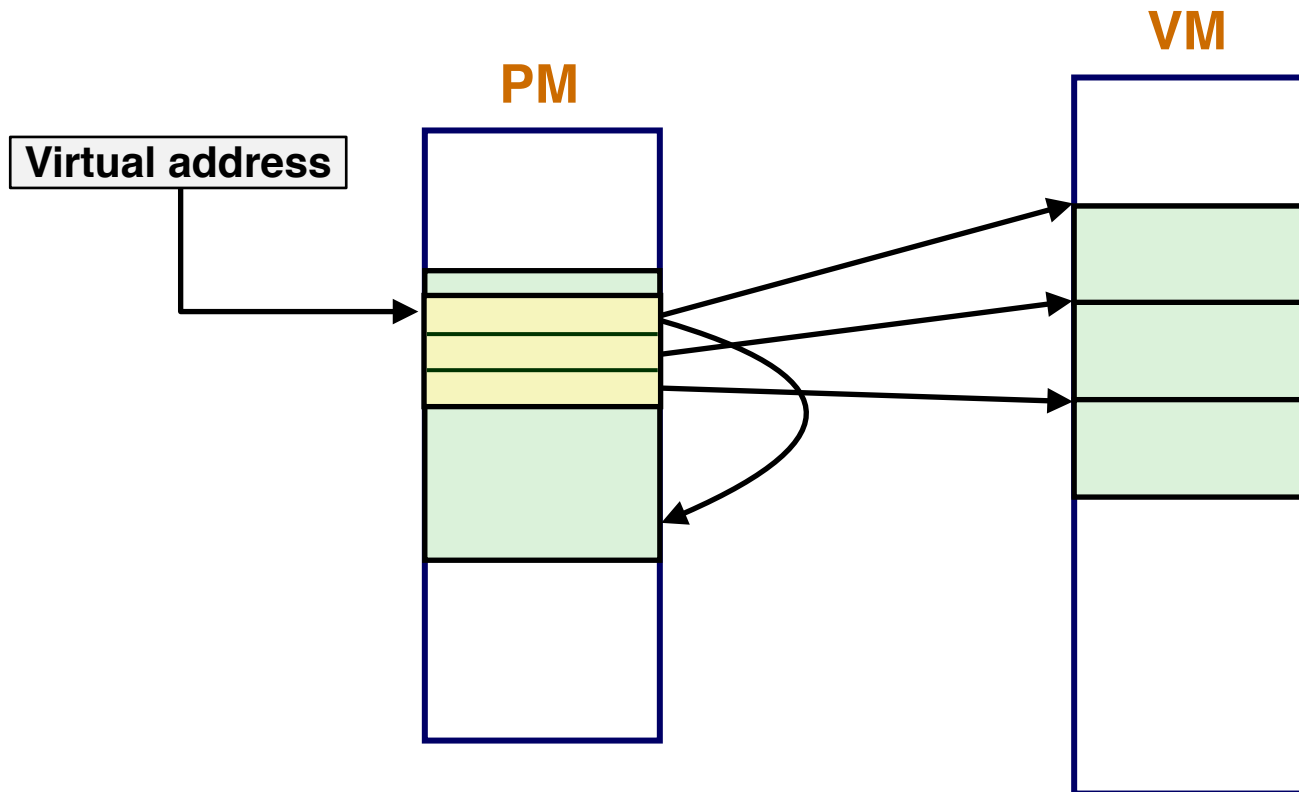
- Three Virtual Memory Optimizations
 - TLB
 - Virtually-indexed, physically-tagged cache
 - Page the page table (a.k.a., multi-level page table)
- Case-study: Intel Core i7/Linux example

Where Does Page Table Live?

- It needs to be at a specific location where we can find it
 - In main memory, with its start address stored in a special register (PTBR)
- Assume 4KB page, 48-bit virtual memory, each PTE is 8 Bytes
 - 2^{36} PTEs in a page table
 - 512 GB total size per page table??!!
- Problem: Page tables are huge
 - One table per process!
 - Storing them all in main memory wastes space

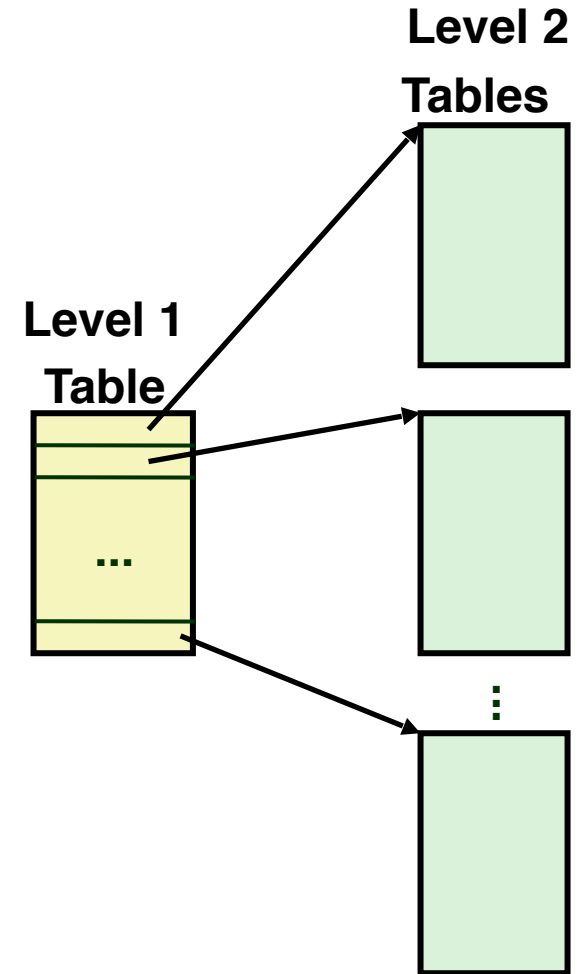
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entries in main memory
- Idea: Put page table in Virtual Memory and swap it just like data

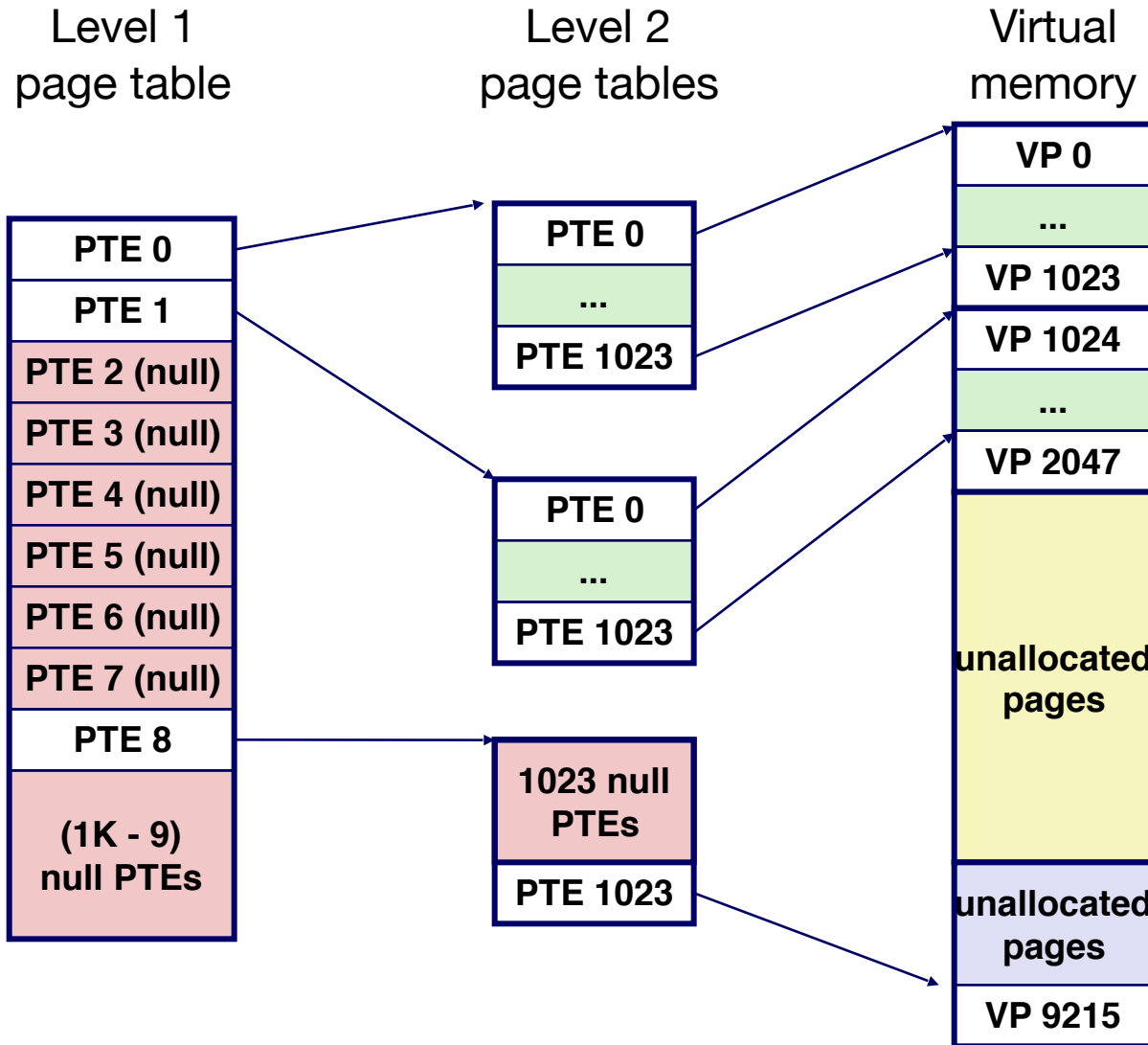


Effectively: A 2-Level Page Table

- Level 1 table:
 - Always in physical memory at a known location.
 - Each L1 PTE points to the start address of a L2 page table.
 - Bring that table to memory on-demand.
- Level 2 table:
 - Each PTE points to an actual data page



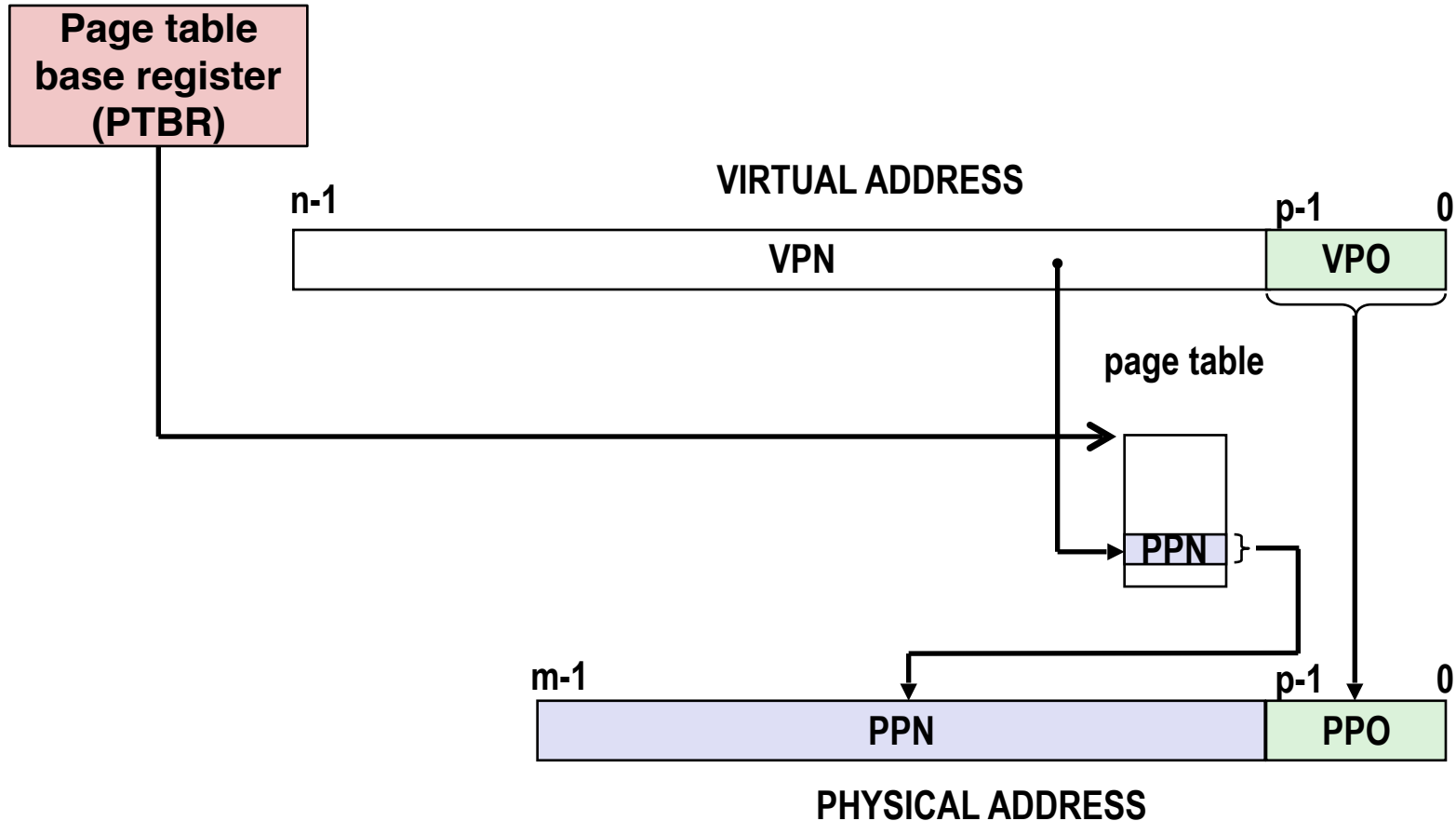
A Two-Level Page Table Hierarchy



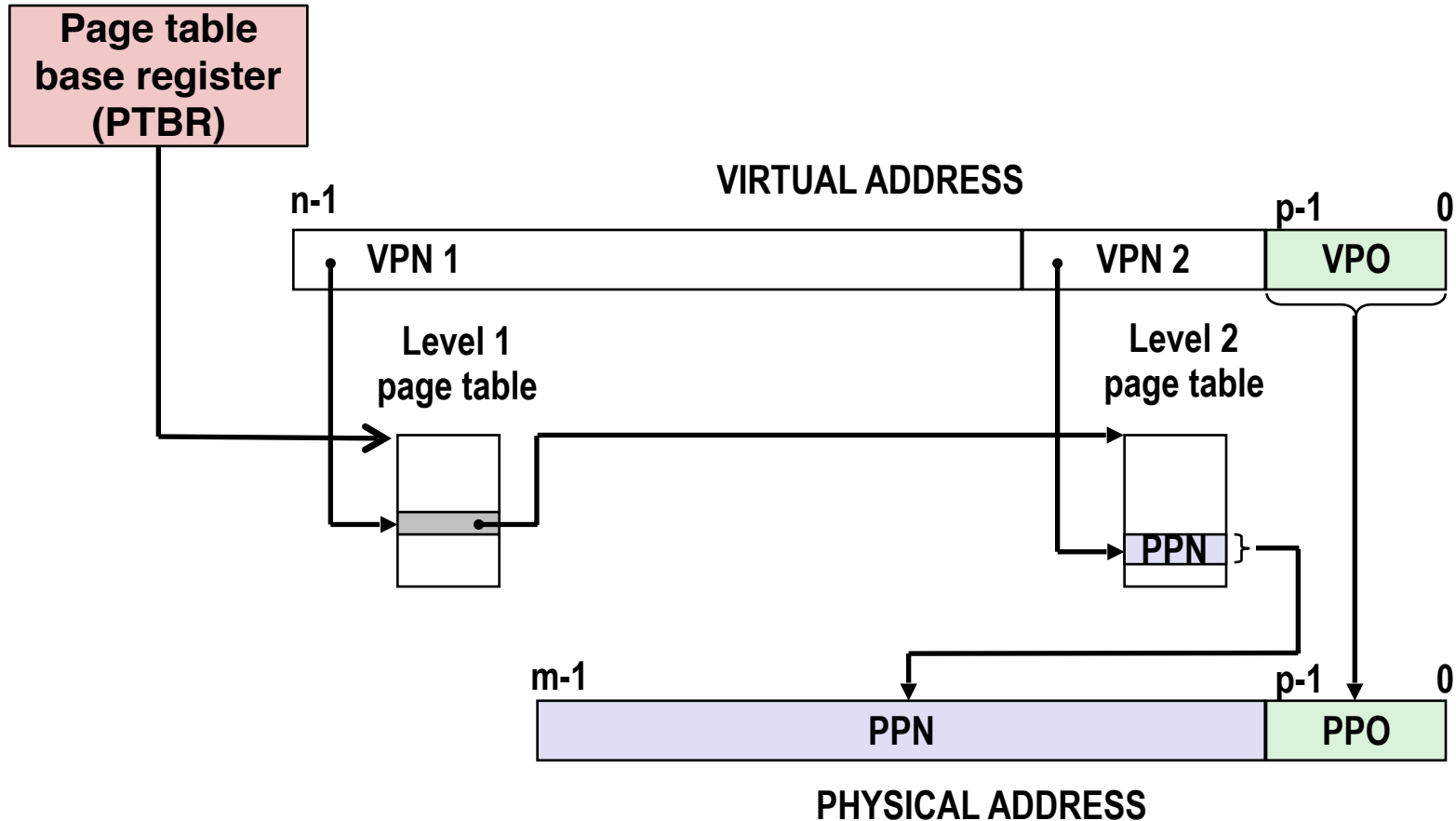
- Level 2 page table size:
 - $2^{32} / 2^{12} * 4 = 4 \text{ MB}$
- Level 1 page table size:
 - $(2^{32} / 2^{12} * 4) / 2^{12} * 4 = 4 \text{ KB}$

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

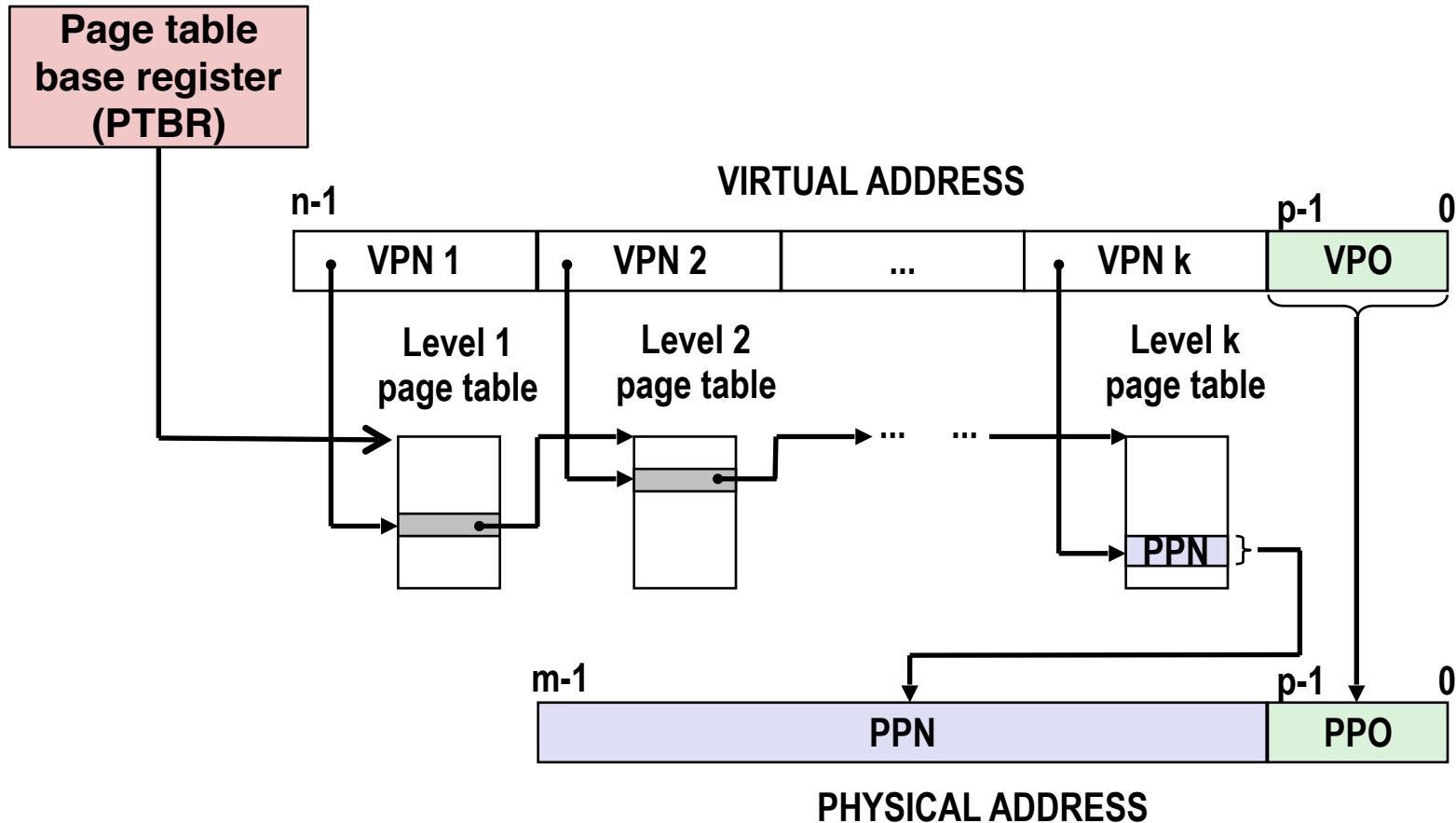
How to Access a 2-Level Page Table?



How to Access a 2-Level Page Table?



Translating with a k-level Page Table



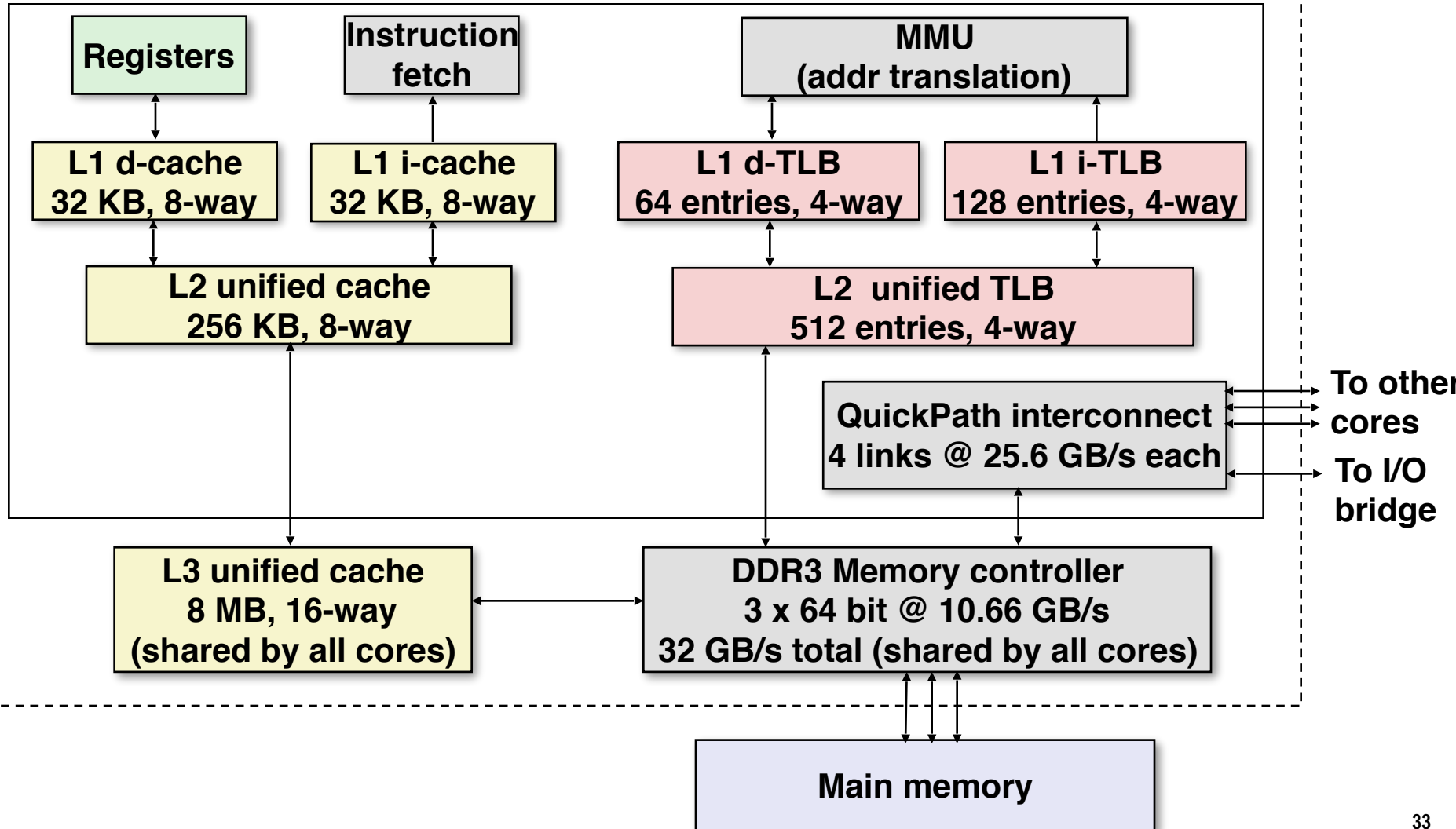
Today

- Three Virtual Memory Optimizations
 - TLB
 - Virtually-indexed, physically-tagged cache
 - Page the page table (a.k.a., multi-level page table)
- Case-study: Intel Core i7/Linux example

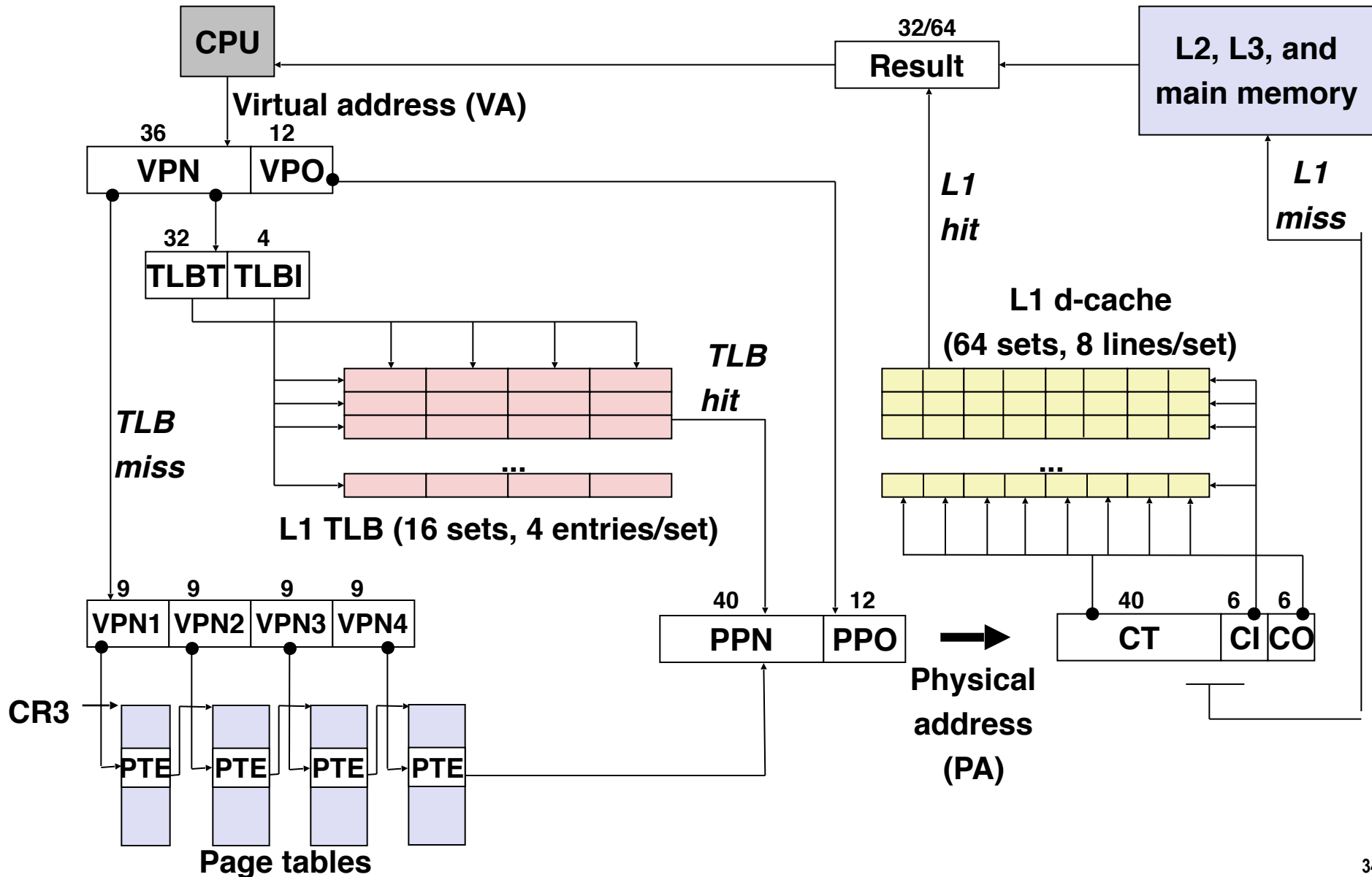
Intel Core i7 Memory System

Processor package

Core x4



End-to-End Core i7 Address Translation

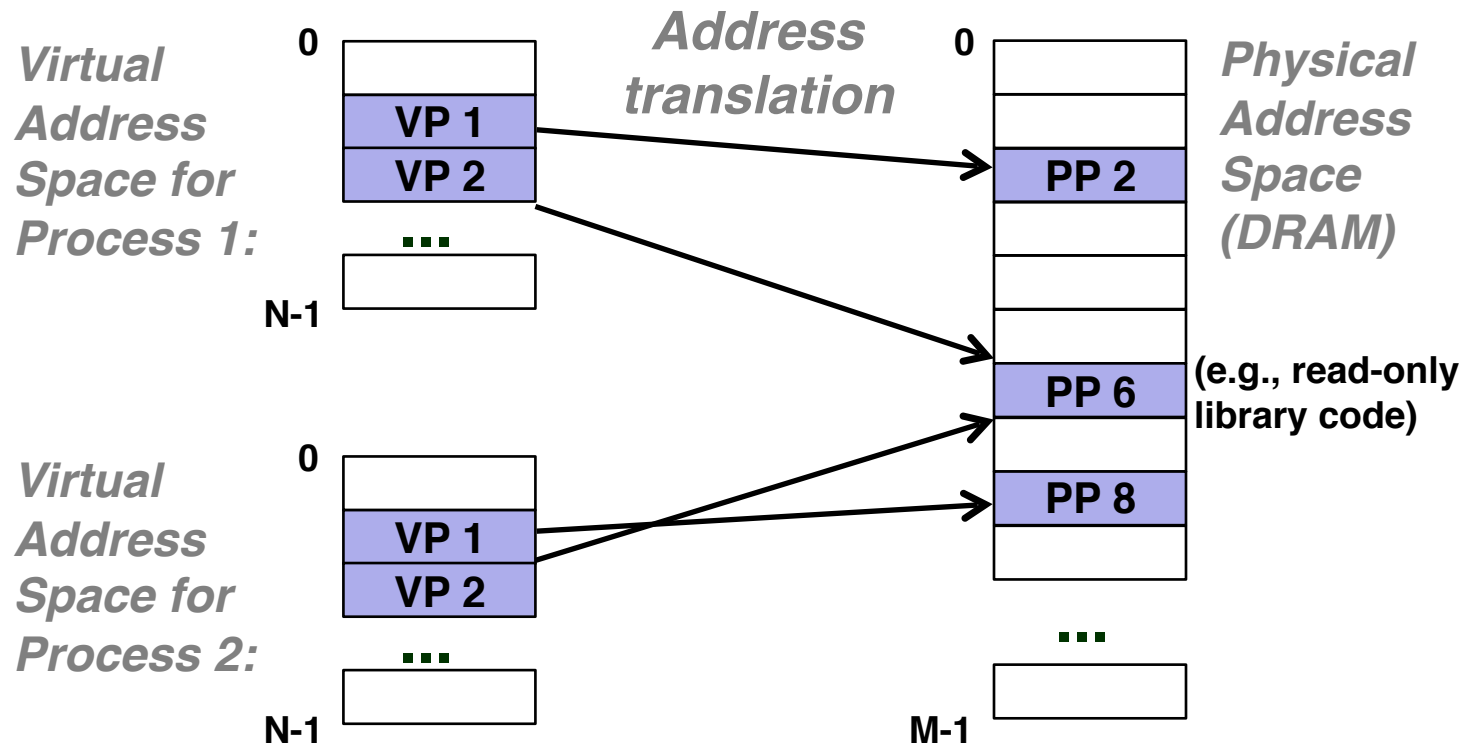


Today

- VM basic concepts and operation
- Other critical benefits of VM
- Address translation

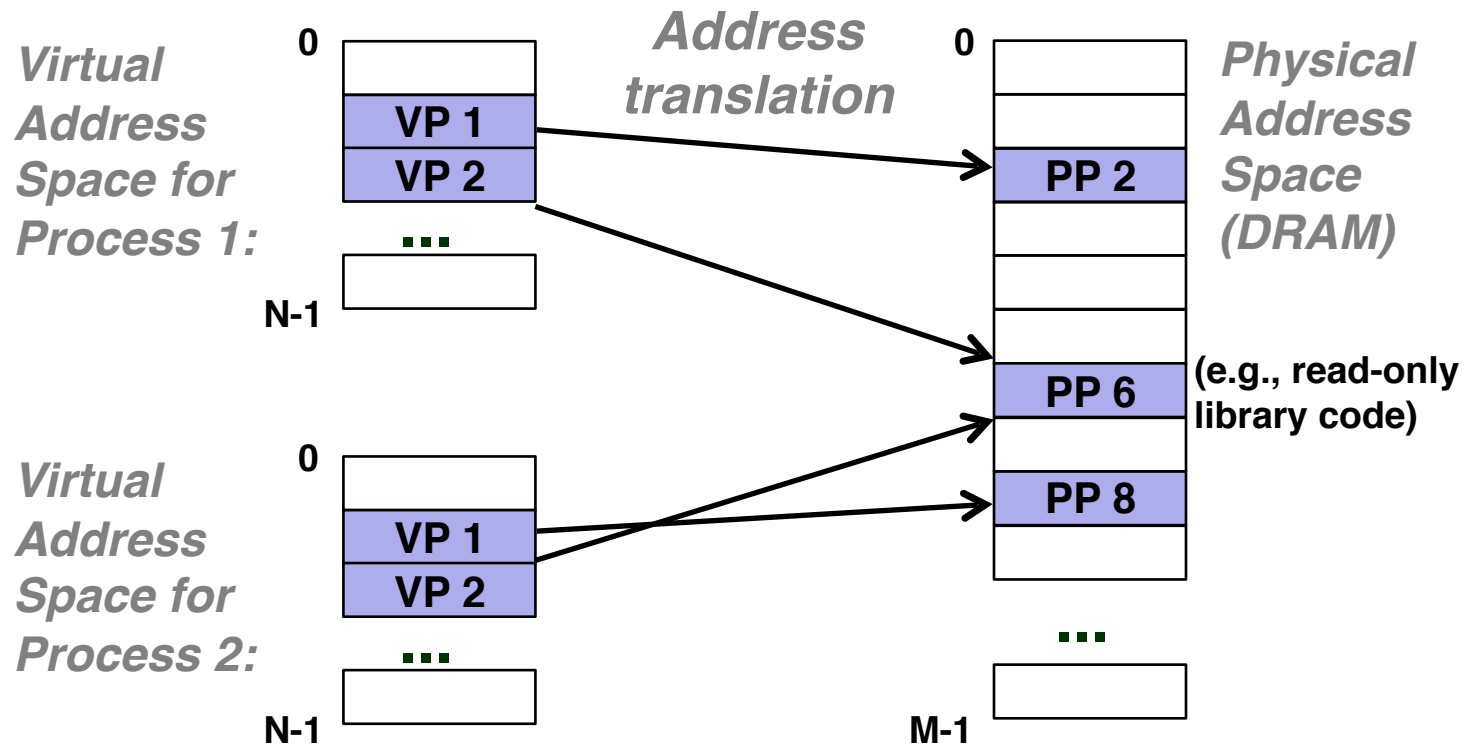
VM as a Tool for Memory Management

- Each process has its own virtual address space
 - It can view memory as a simple linear array
 - Mapping scatters addresses through physical memory
 - Well-chosen mappings can improve locality



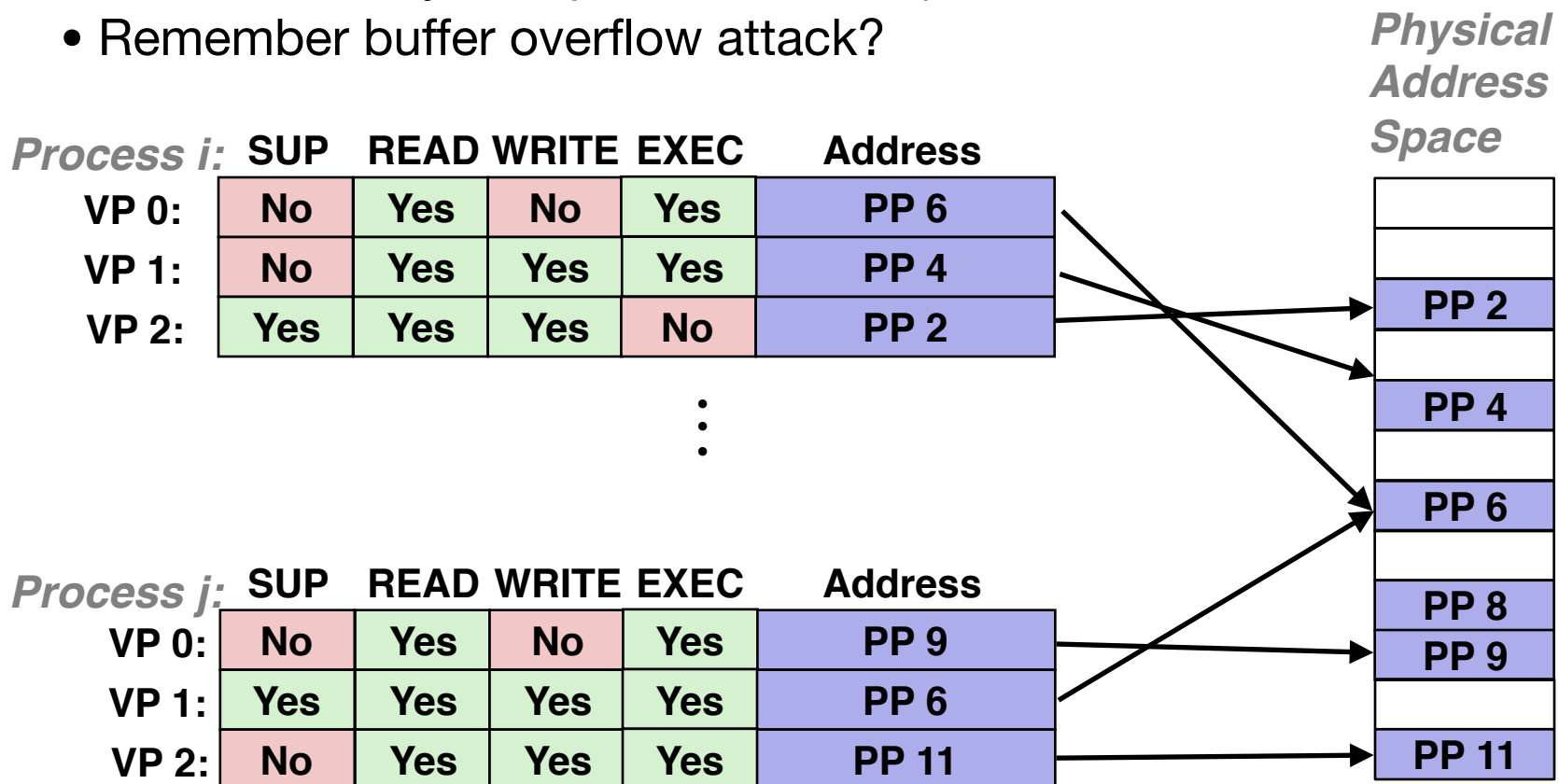
Virtual Memory Enables Sharing

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



VM Provides Further Protection Opportunities

- Extend PTEs with permission bits
- MMU checks these bits on each access (read/write/executable/accessible only in supervisor mode?)
- Remember buffer overflow attack?



Core i7 Level 4 Page Table Entries

63	62	52	51	12	11	9	8	7	6	5	4	3	2	1	0
XD	Unused	Page physical base address				Unused	G		D	A	CD	WT	U/S	R/W	P=1
Available for OS (page location on disk)															P=0

Each entry references a 4K child page. Significant fields:

P: Child page is present in memory (1) or not (0)

R/W: Read-only or read-write access permission for child page

U/S: User or supervisor mode access

WT: Write-through or write-back cache policy for this page

A: Reference bit (set by MMU on reads and writes, cleared by software)

D: Dirty bit (set by MMU on writes, cleared by software)

Page physical base address: 40 most significant bits of physical page address (forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.