Virtual Memory

15-740 FALL’19

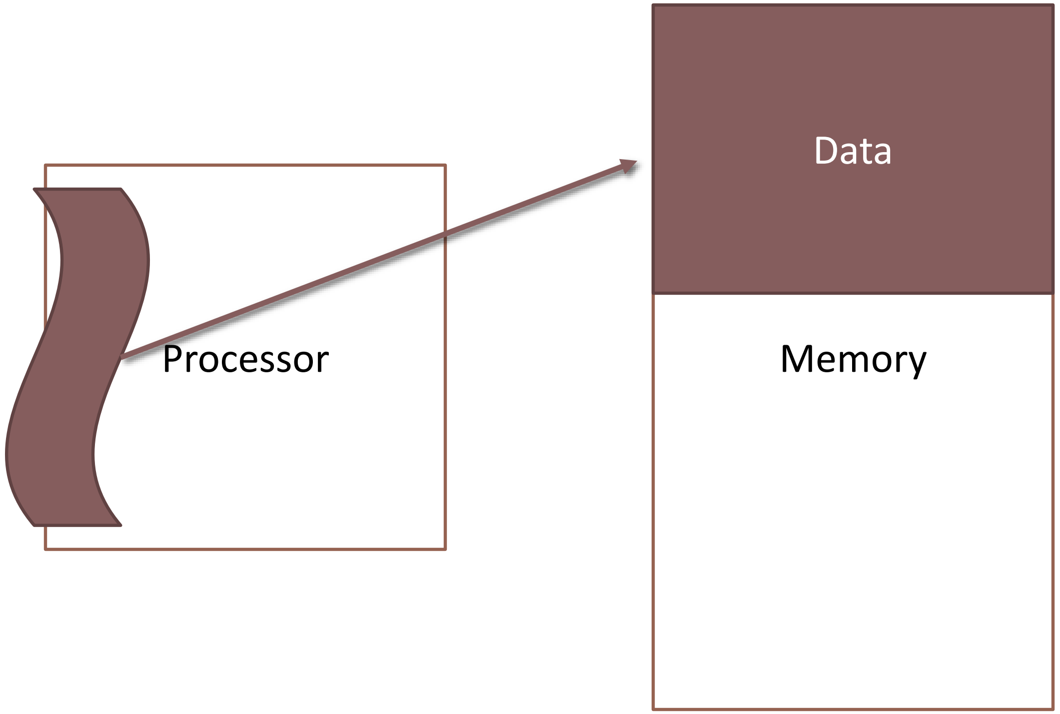
NATHAN BECKMANN

What is virtual memory?

Why is it important?

Back in the beginning…

Programs accessed memory directly

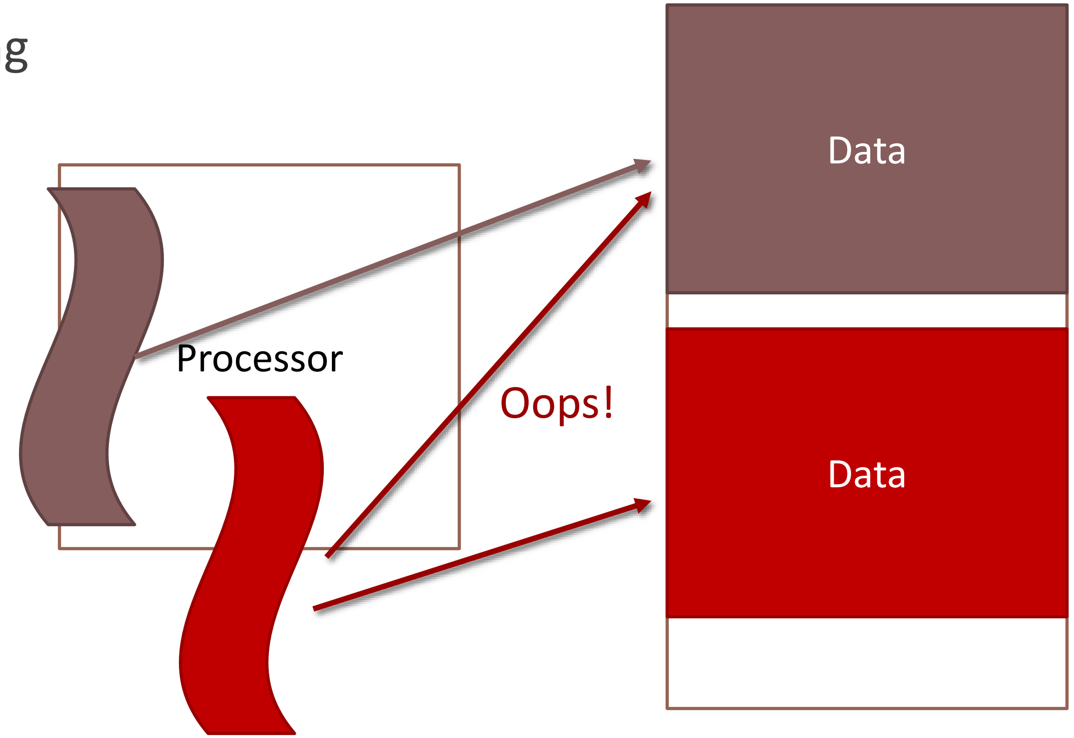


Back in the beginning…

Programs accessed memory directly

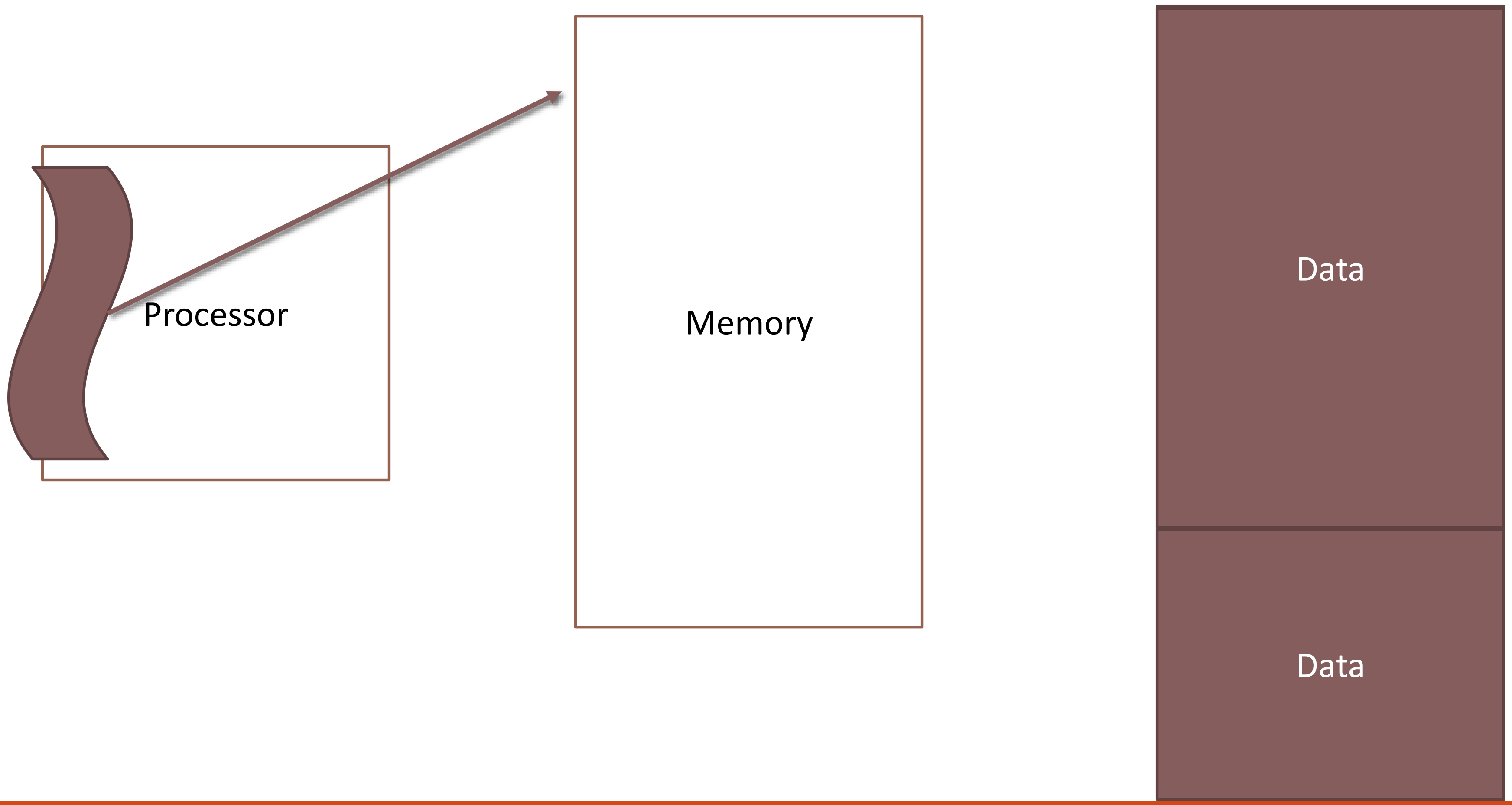
No protection between programs

Complex loading



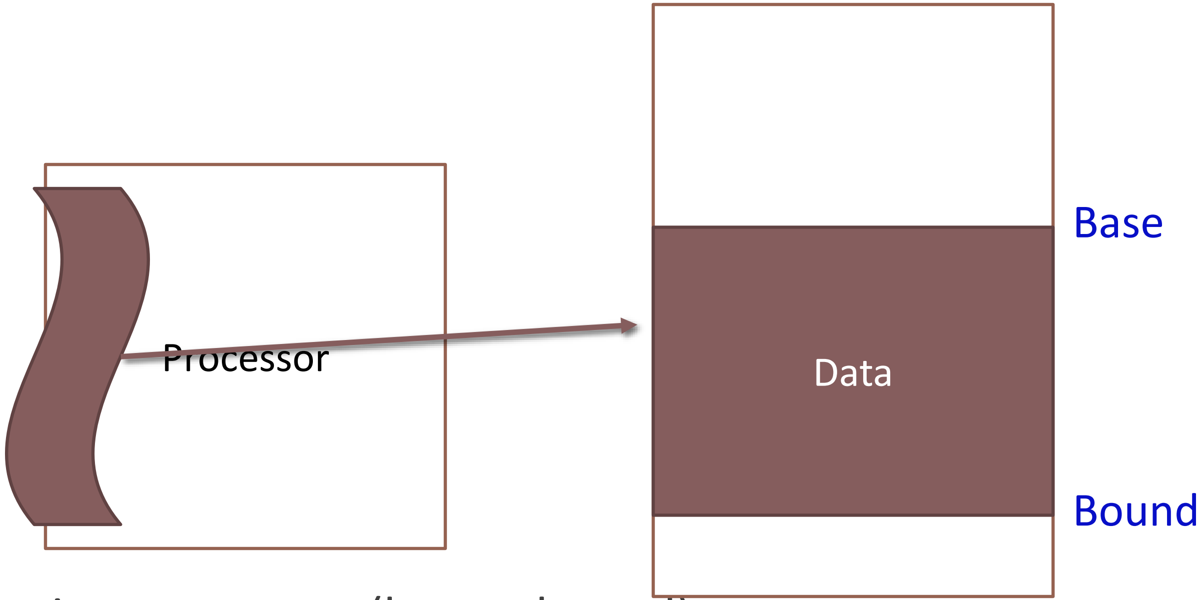
Back in the beginning…

If data was larger than memory, applications dealt with it themselves (eg, overlays)



Back in the beginning…

Virtual memory: give each process the illusion of its own memory



Early implementation: segments (base + bound)

- Memory allocated in contiguous chunks
- Hardware translation
- Inefficient use of space!

Why Virtual Memory?

There are three motivations for Virtual Memory (VM):

1. Allow main memory (DRAM) to act as a “cache” for disk
2. Simplifying memory management
3. Protecting address spaces

But VM works very differently from SRAM caches. Why?

- To understand why, let’s begin with the first motivation
- (Once we understand that, the other aspects of VM will make more sense.)

Motivation 1: DRAM as cache of disk

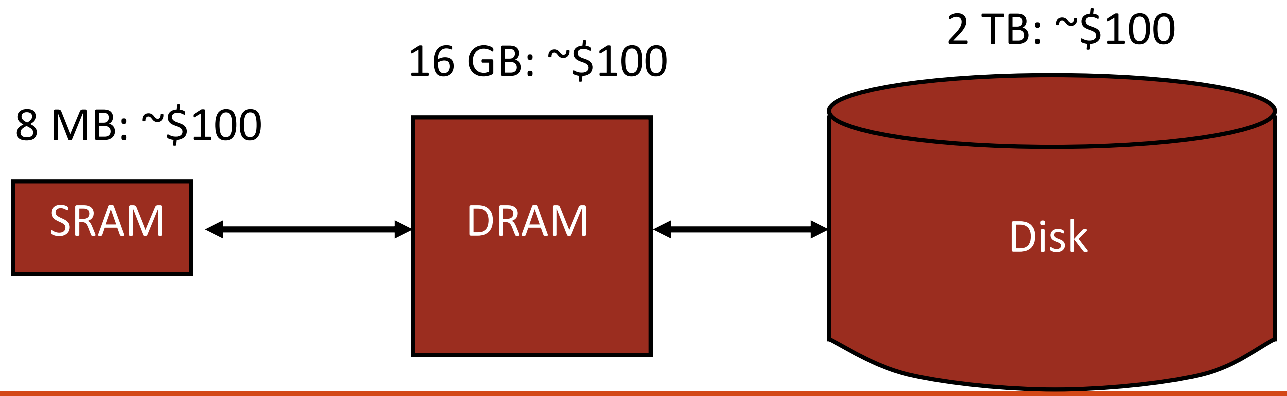
The full address space is quite large:

- 32-bit addresses: ~4,000,000,000 (4 billion) bytes
- 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes

Disk storage is ~100X cheaper than DRAM storage

- 2 TB of DRAM: ~ \$10,000
- 2 TB of disk: ~ \$100

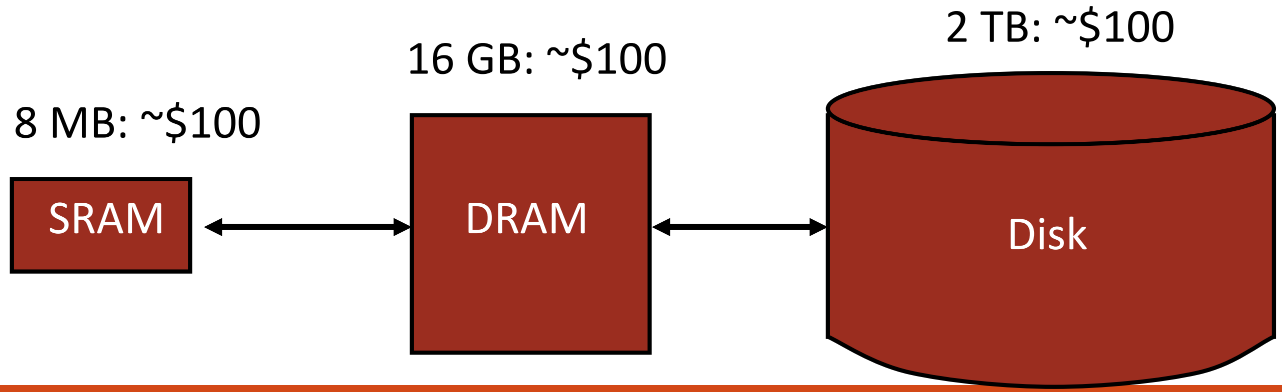
To access very large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk



DRAM vs. SRAM as a “Cache”

DRAM vs. disk is more extreme than SRAM vs. DRAM

- Access **latencies**:
  - DRAM is ~100X slower than SRAM
  - disk is ~100,000X slower than DRAM
- Importance of exploiting **spatial locality**:
  - first byte is ~100,000X slower than successive bytes on disk
  - vs. ~4X improvement for page-mode vs. regular accesses to DRAM
- “Cache” **size**:
  - main memory is ~1000X larger than an SRAM cache
- Different addressing (memory address vs sector address)



Impact of These Properties on Design

If DRAM was to be organized similar to an SRAM cache, how would we set the following design parameters?

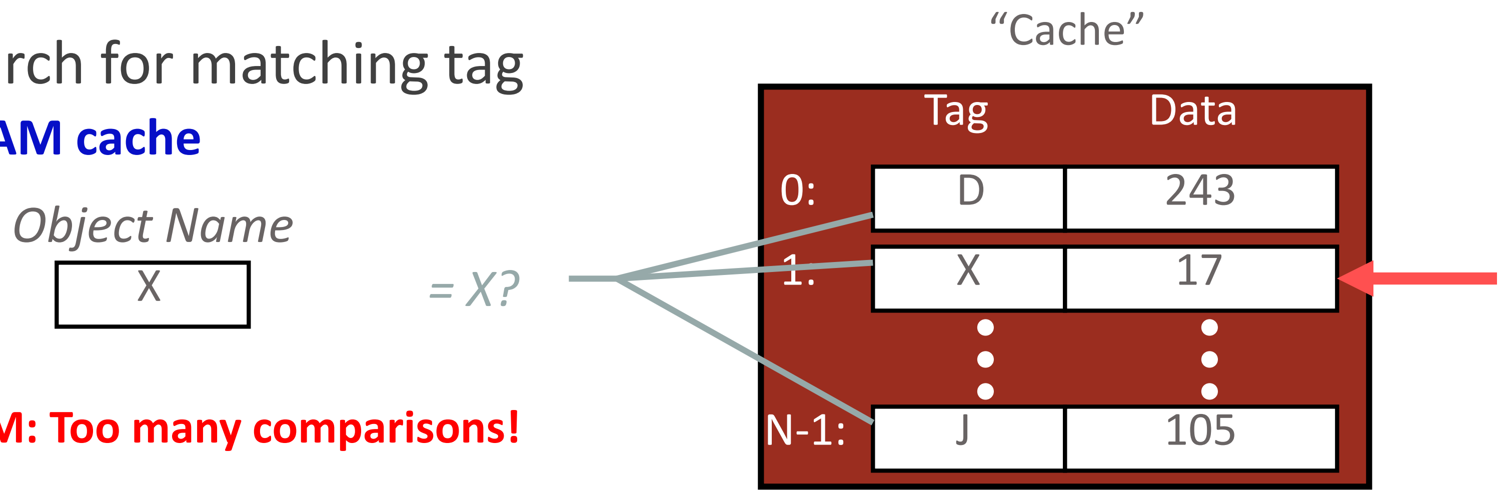
- Line size?
- Associativity?
- Replacement policy (if associative)?
- Write through or write back?

(What would the impact of these choices be on: miss rate, hit time, miss penalty, tag overhead, …)

But how to implement a multi-GB, fully associative cache?

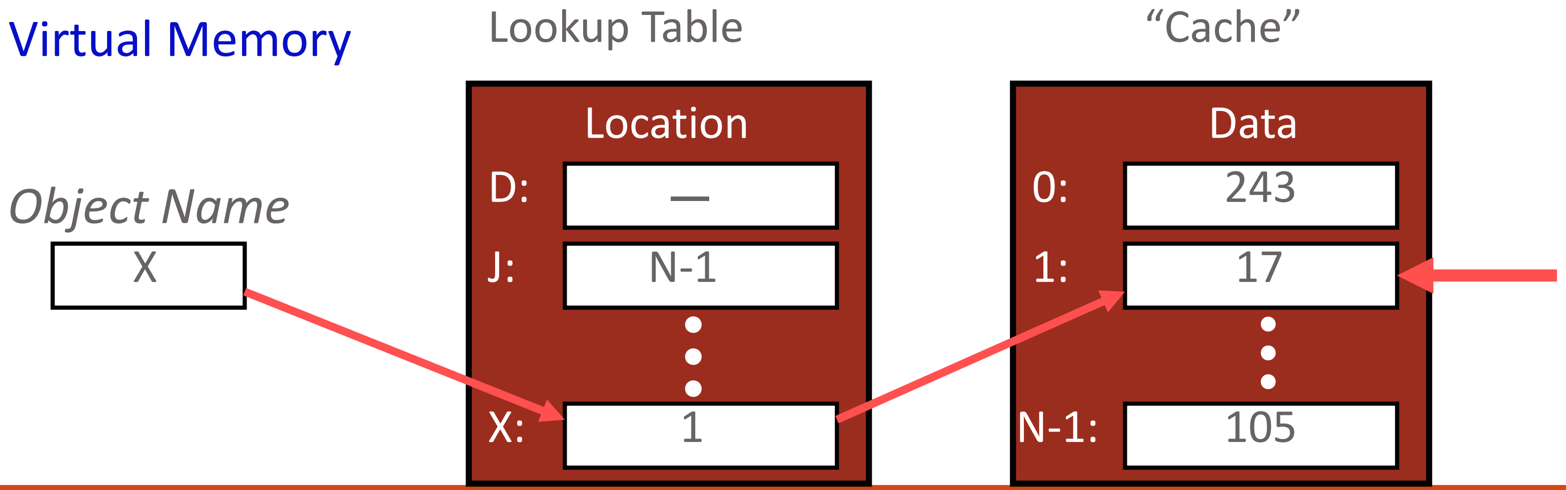
Looking up an object

1. Search for matching tag
   - **SRAM cache**



VM: Too many comparisons!

2. Use indirection to look up actual object location
   - Virtual Memory



Tag Overheads

How many tags? How big are they?

Conventional SRAM cache
- Tags for each <u>cached</u> item
- Tag stores address, other bits

Indirect virtual memory cache
- Tags for <u>every</u> item (cached or not)
- Tag stores location, other bits

Main difference is # of tags
- How to deal with tags for virtual memory?
- Strategy: store them in <u>memory</u> and <u>cache the tags</u>

Address Spaces

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

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

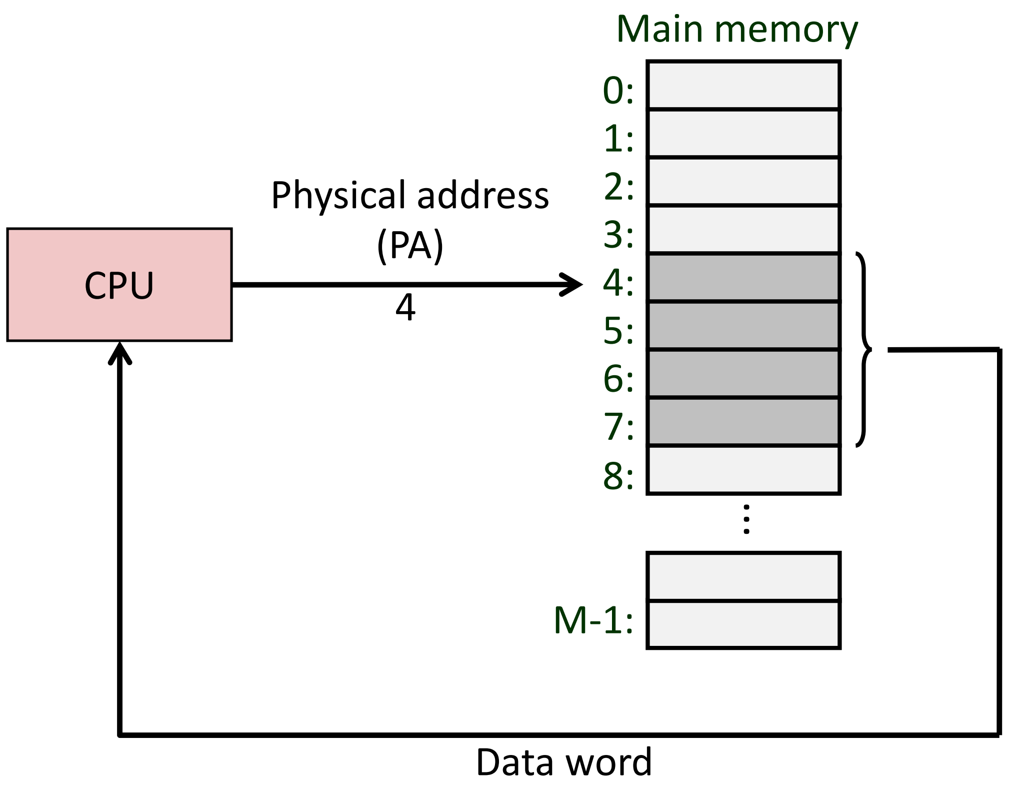
Clean distinction between data (bytes) and their attributes (addresses)

Each datum can now have multiple addresses

Every byte in main memory:
one physical address, one (or more) virtual addresses

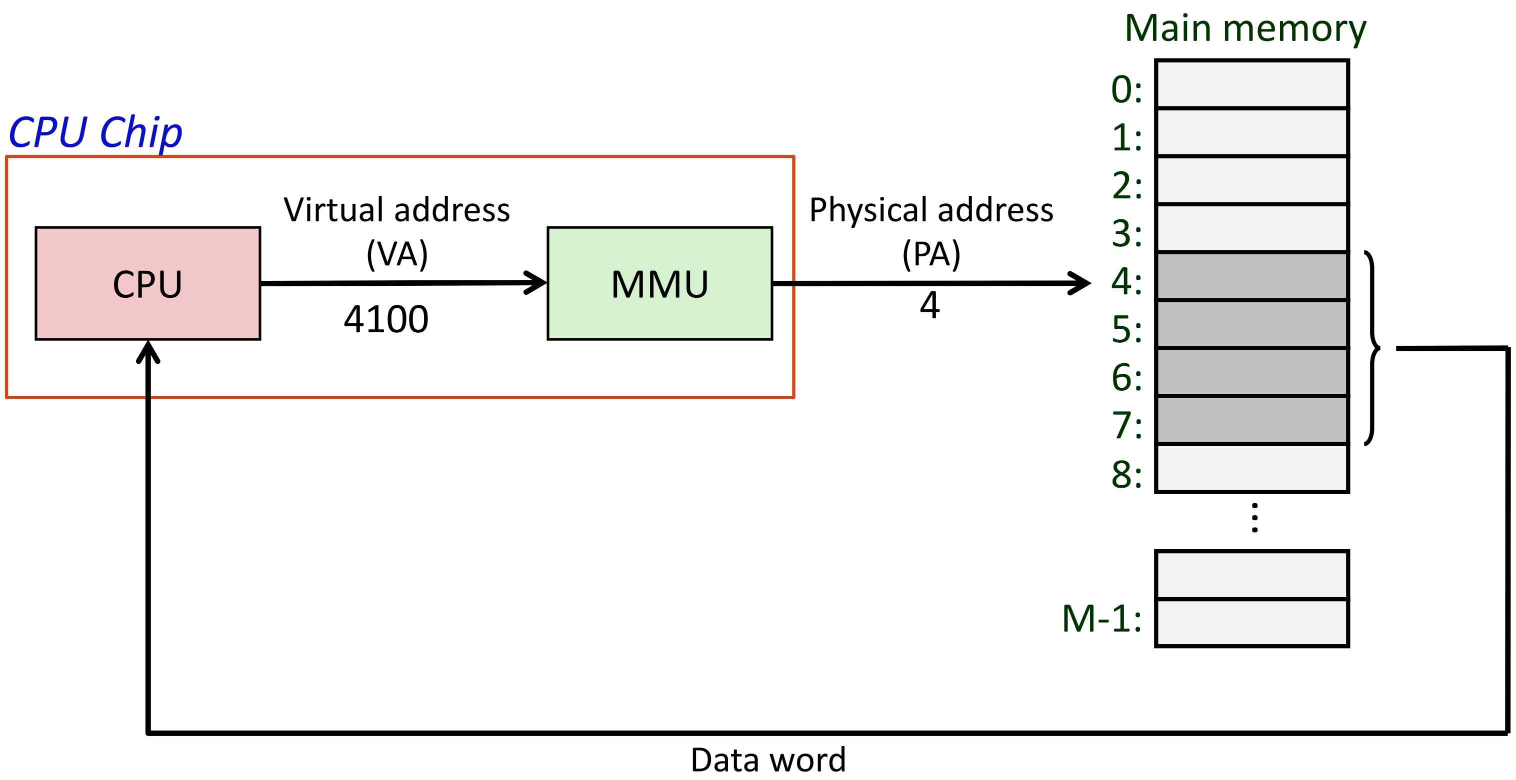
A System Using Physical Addressing

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, laptops, phones, etc.



Why Virtual Memory? (Further Details)

(1) VM allows efficient use of limited main memory (RAM)

- Use RAM as a cache for the parts of a virtual address space
    - some non-cached parts stored on disk
    - some (unallocated) non-cached parts stored nowhere
- Keep only active areas of virtual address space in memory
    - transfer data back and forth as needed

(2) VM simplifies memory management for programmers

- Each process gets a full, private linear address space

(3) VM isolates address spaces

- One process can't interfere with another's memory
    - because they operate in different address spaces
- User process cannot access privileged information
    - different sections of address spaces have different permissions

Motivations for VM Revisited

Recall the 3 motivations for Virtual Memory (VM):

1. Allow main memory (DRAM) to act as a “cache” for disk
2. Simplifying memory management
3. Protecting address spaces

To solve #1, we introduced a new form of *indirection*

This indirection also makes it easy to solve #2 and #3:

- Simplifying memory management:
  - flexible mapping of virtual to physical addresses
- Protecting address spaces:
  - protection information can be stored in the lookup table
  - and enforced before allowing access to physical memory

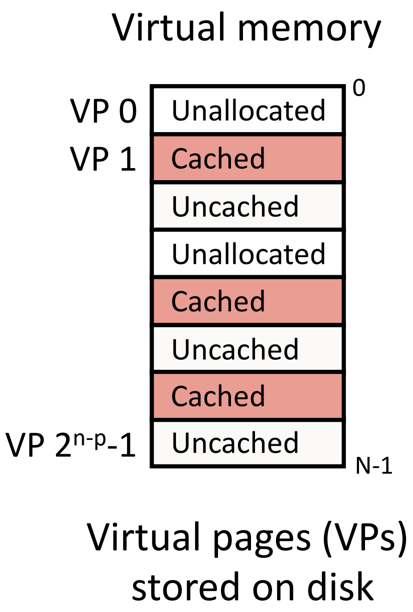
(1) VM as a Tool for Caching

Virtual memory is an array of N contiguous bytes

- You can think of VM backed by storage on disk

The contents of the array on disk are cached in *physical memory* (ie, *DRAM as a cache*)

- These cache blocks are called *pages* (size is $P = 2^p$ bytes)



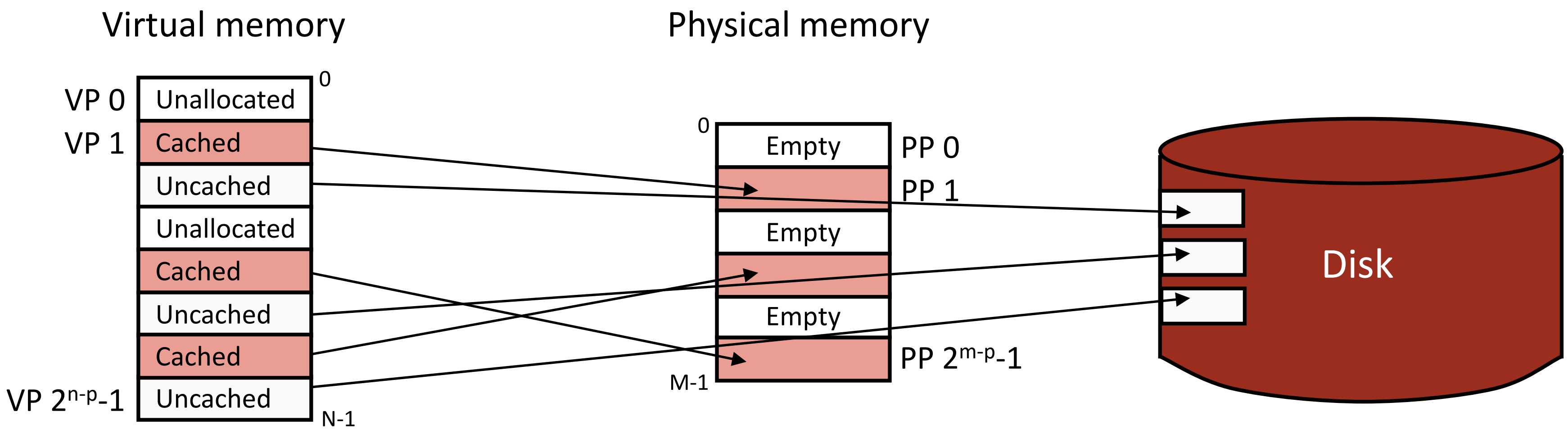
(1) VM as a Tool for Caching

Virtual memory is an array of N contiguous bytes

- You can think of VM backed by storage on disk

The contents of the array on disk are cached in *physical memory* (ie, *DRAM as a cache*)

- These cache blocks are called *pages* (size is $P = 2^p$ bytes)



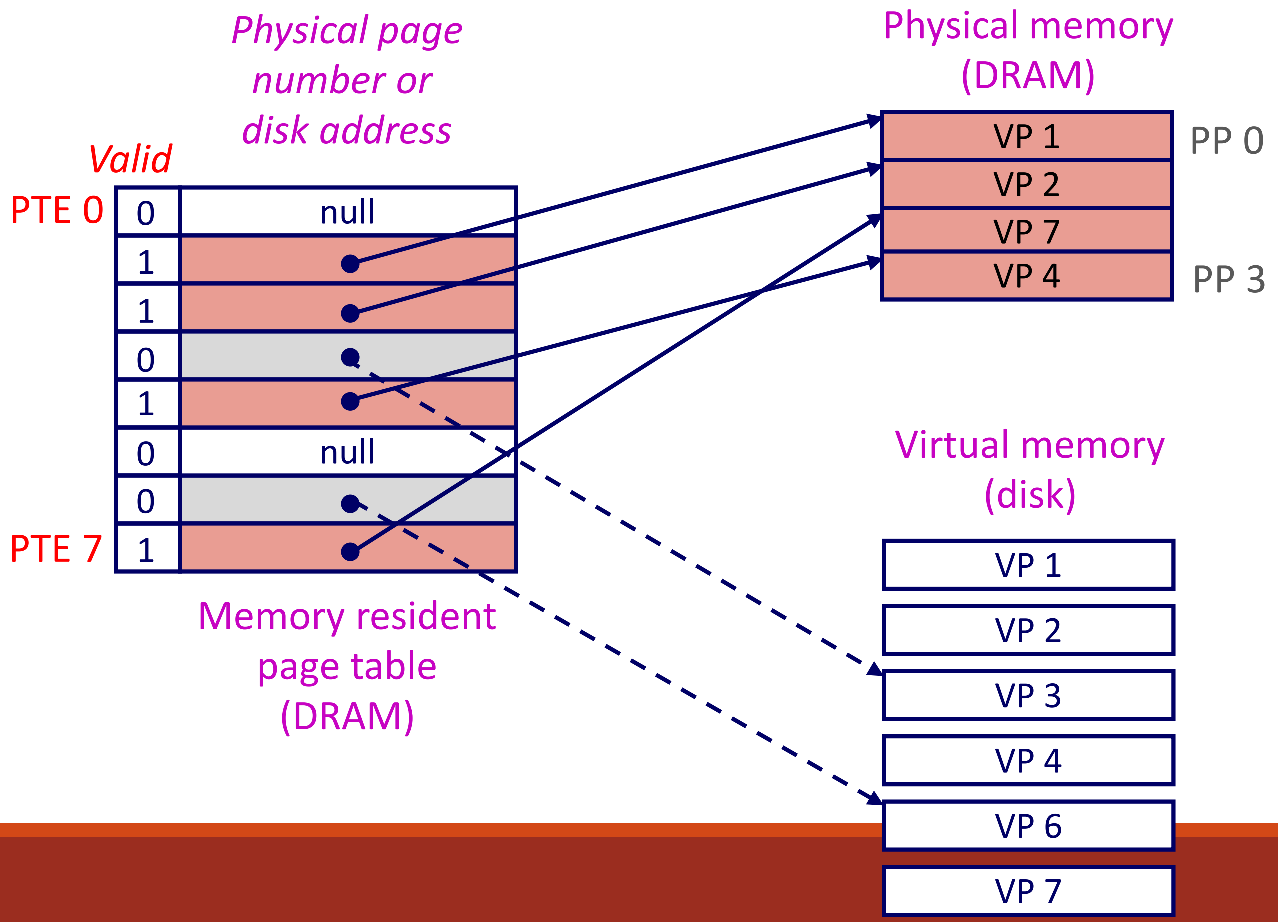
Virtual pages (VPs) stored on disk

Physical pages (PPs) / page frames cached in DRAM

Enabling Data Structure: Page Table

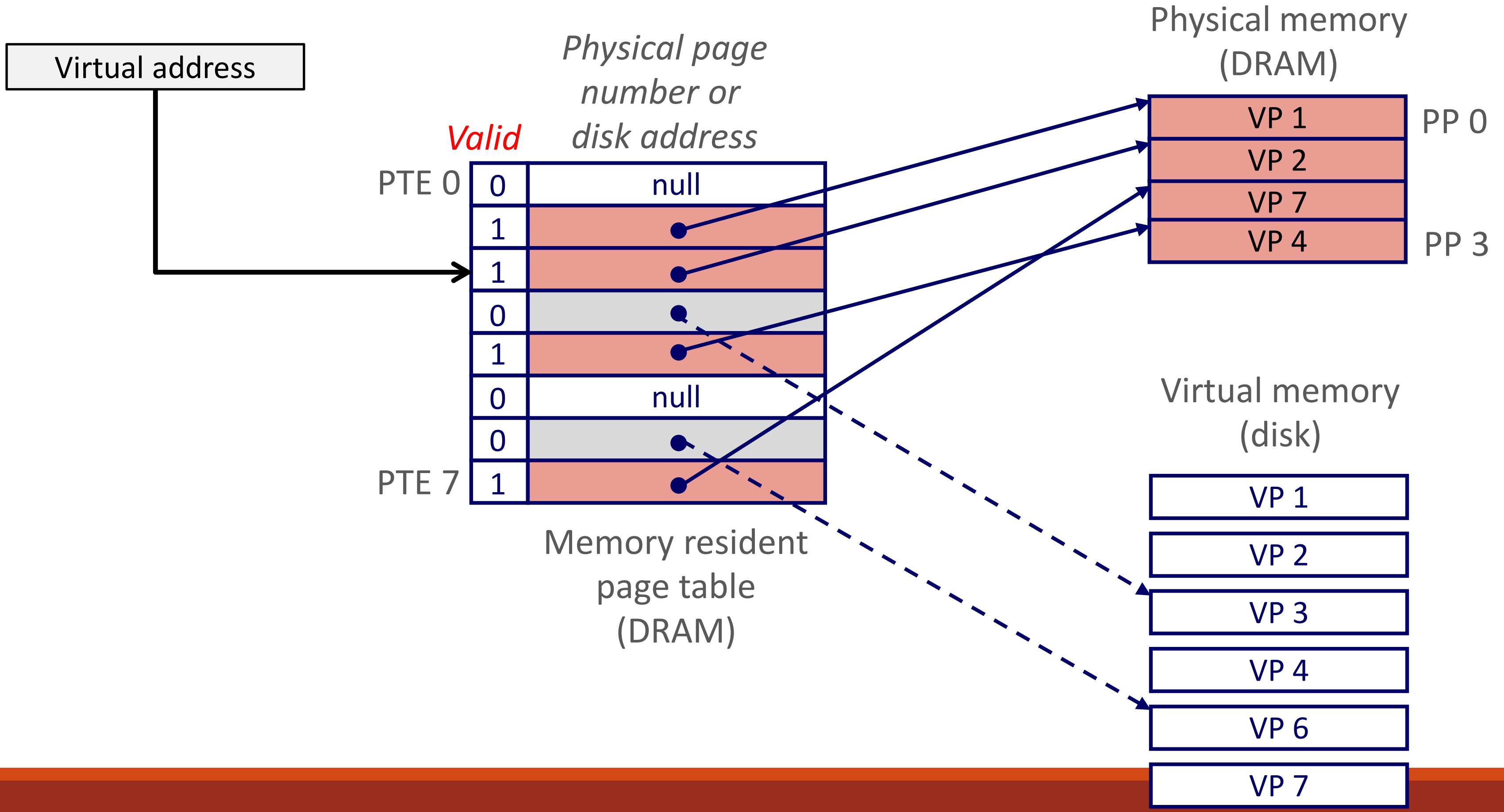
Page table: an array of page table entries (PTEs) that maps virtual pages to physical pages

- (Page table == tags)
- Per-process kernel data structure in DRAM



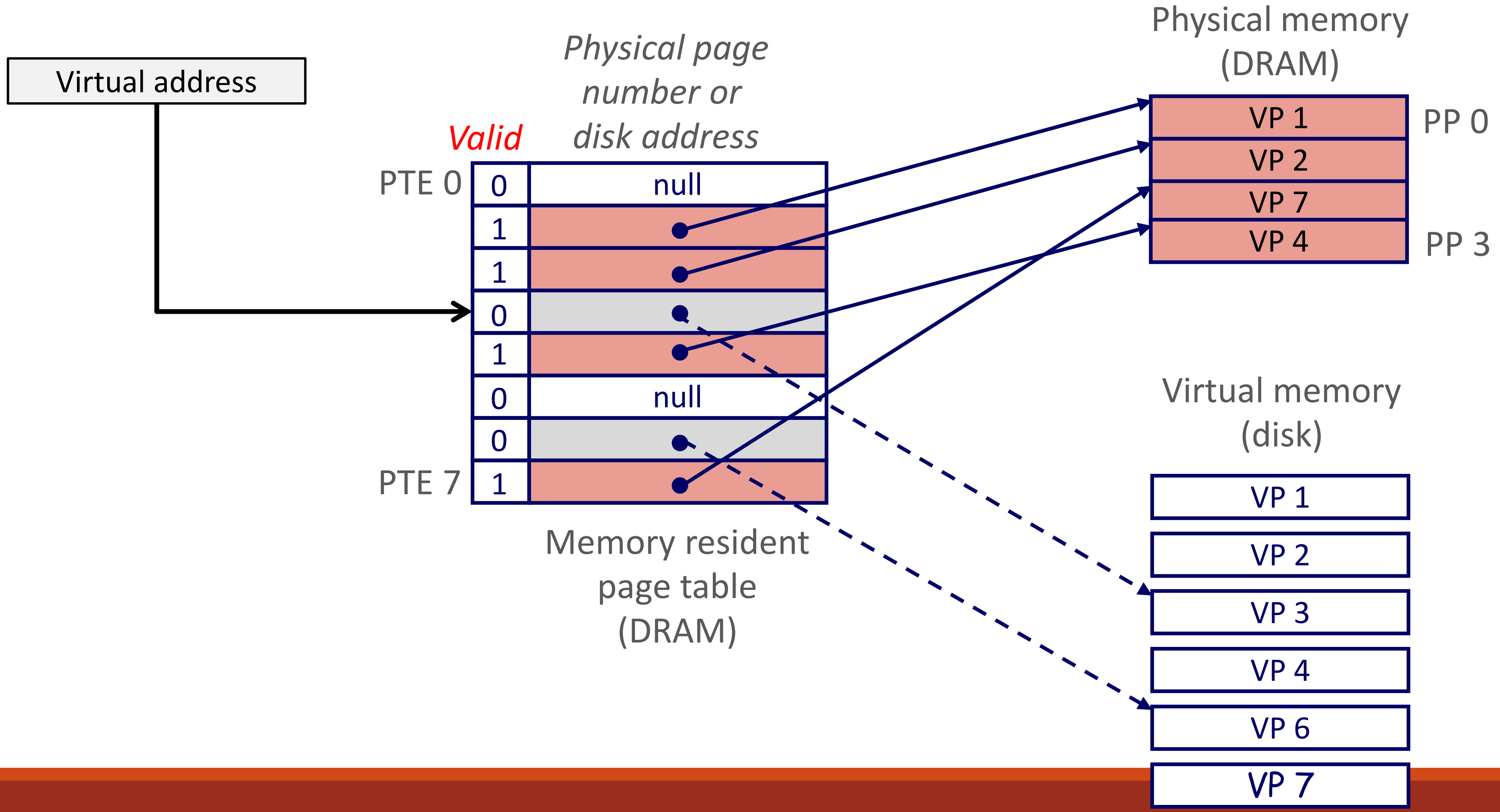
Page Hit

Page hit: reference to VM word that is in physical memory (== cache hit)



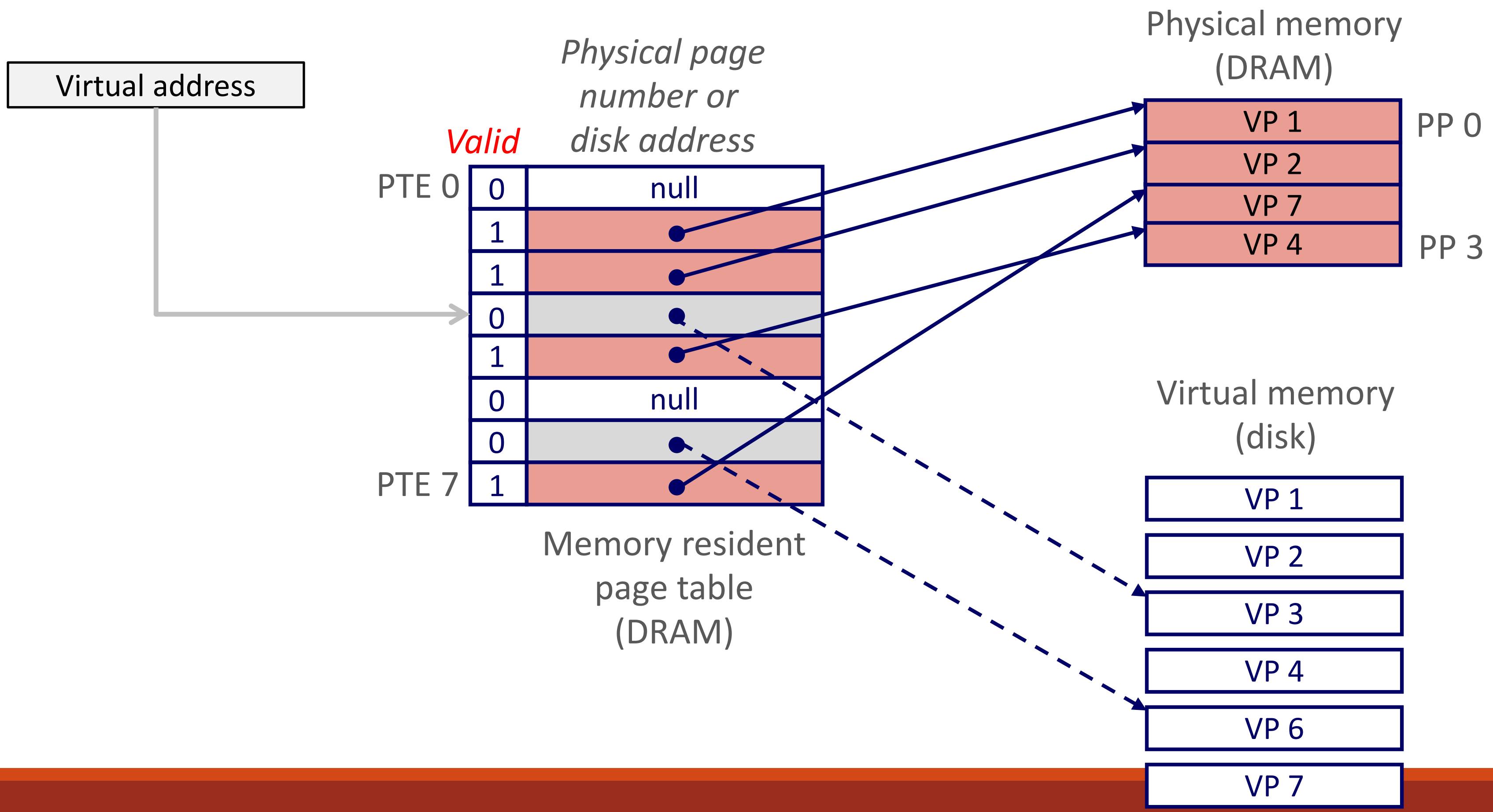
Page Fault

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



Handling Page Fault

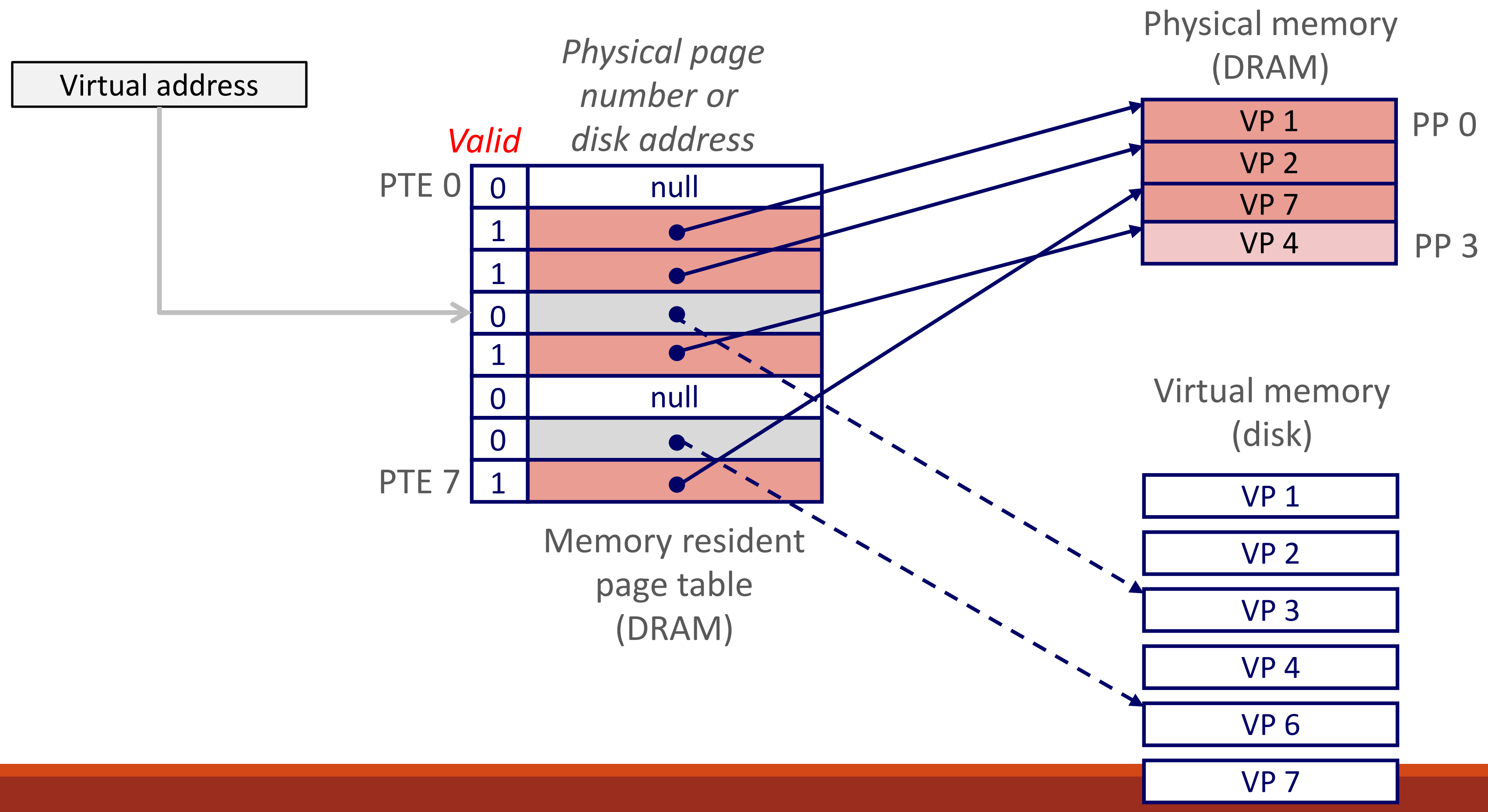
Page miss causes page fault (an exception – invoking software!)



Handling Page Fault

Page miss causes page fault

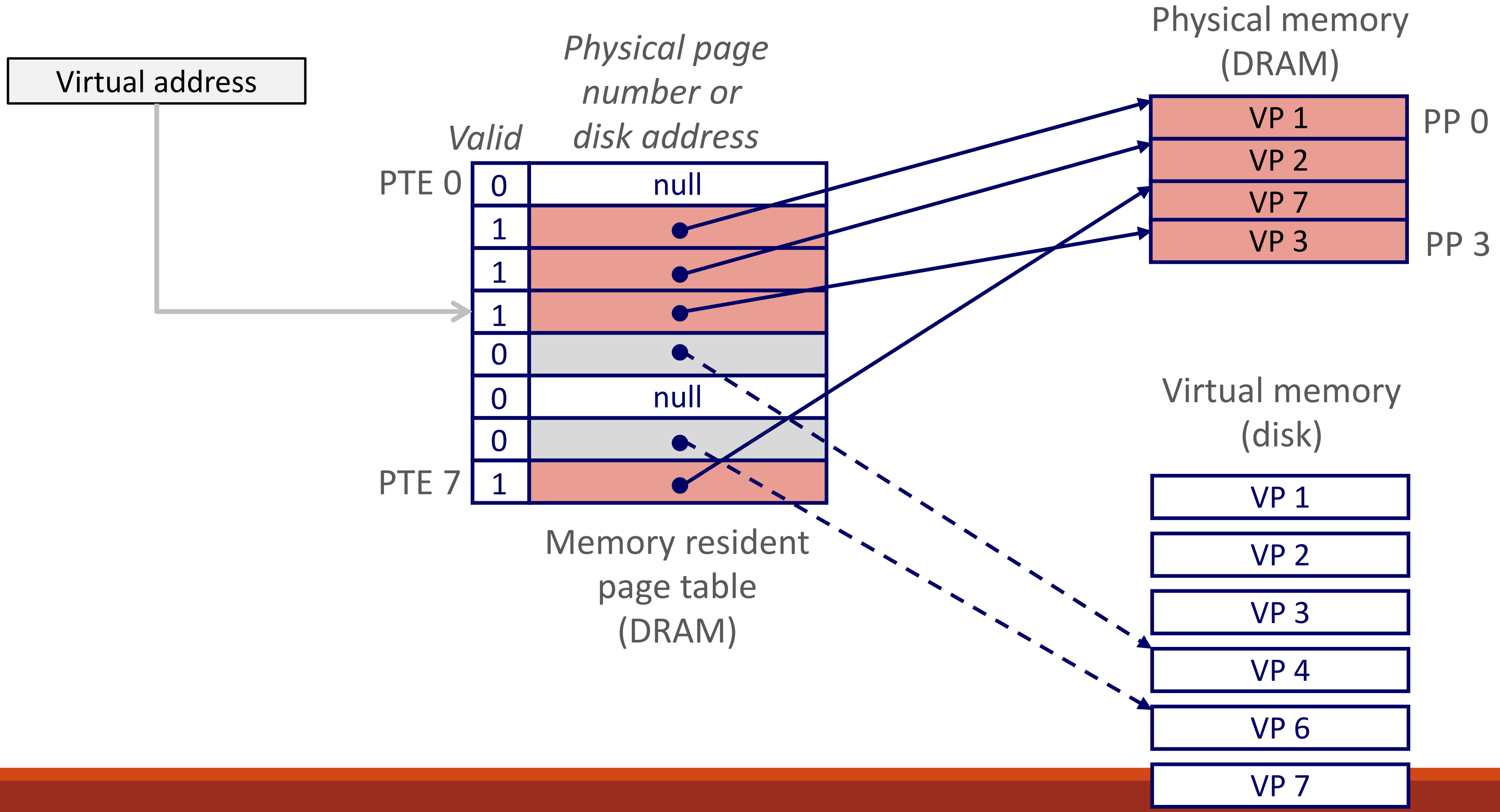
Page fault handler selects a victim to be evicted (here VP 4)



Handling Page Fault

Page miss causes page fault

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

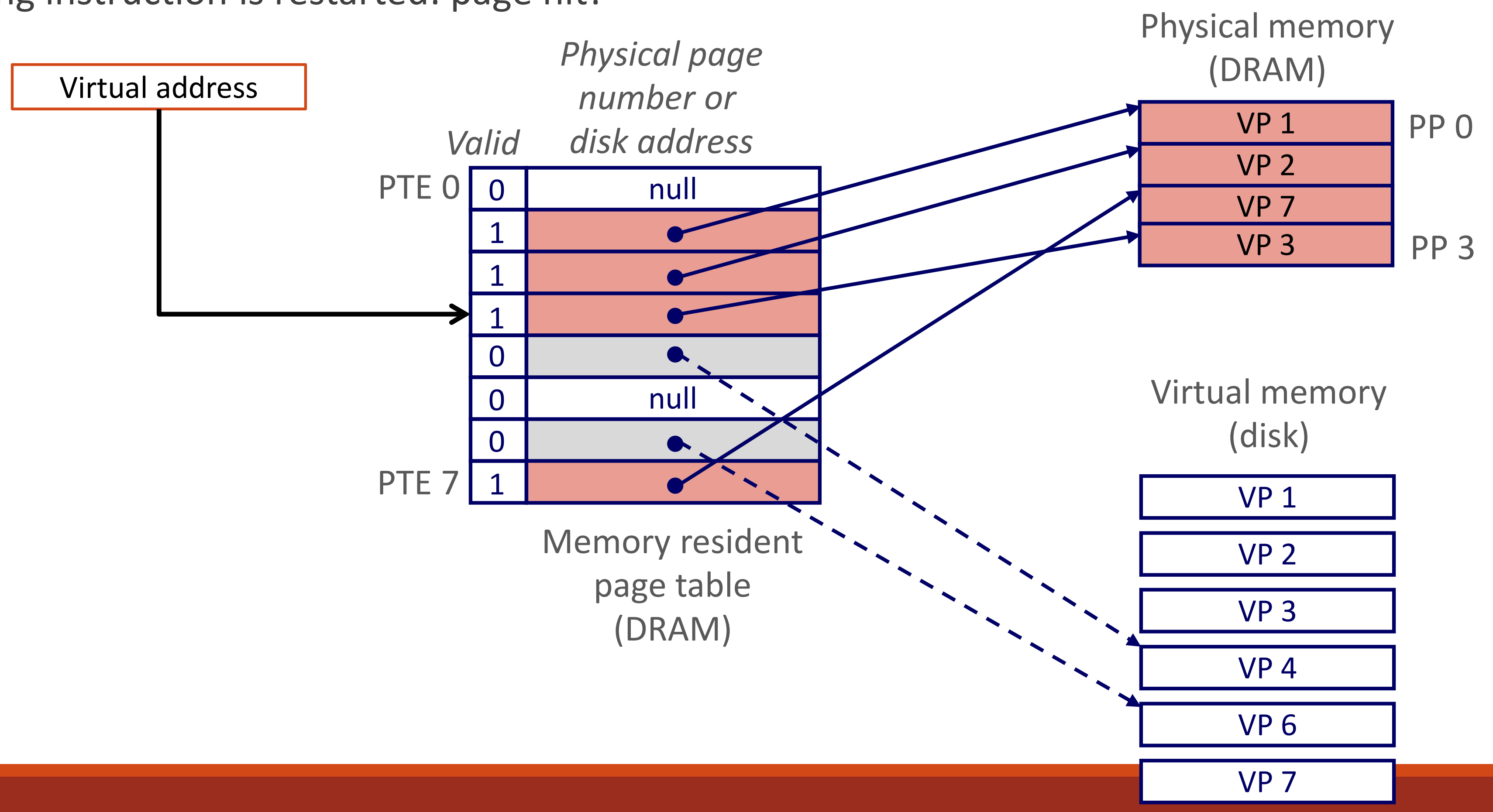


Handling Page Fault

Page miss causes page fault

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

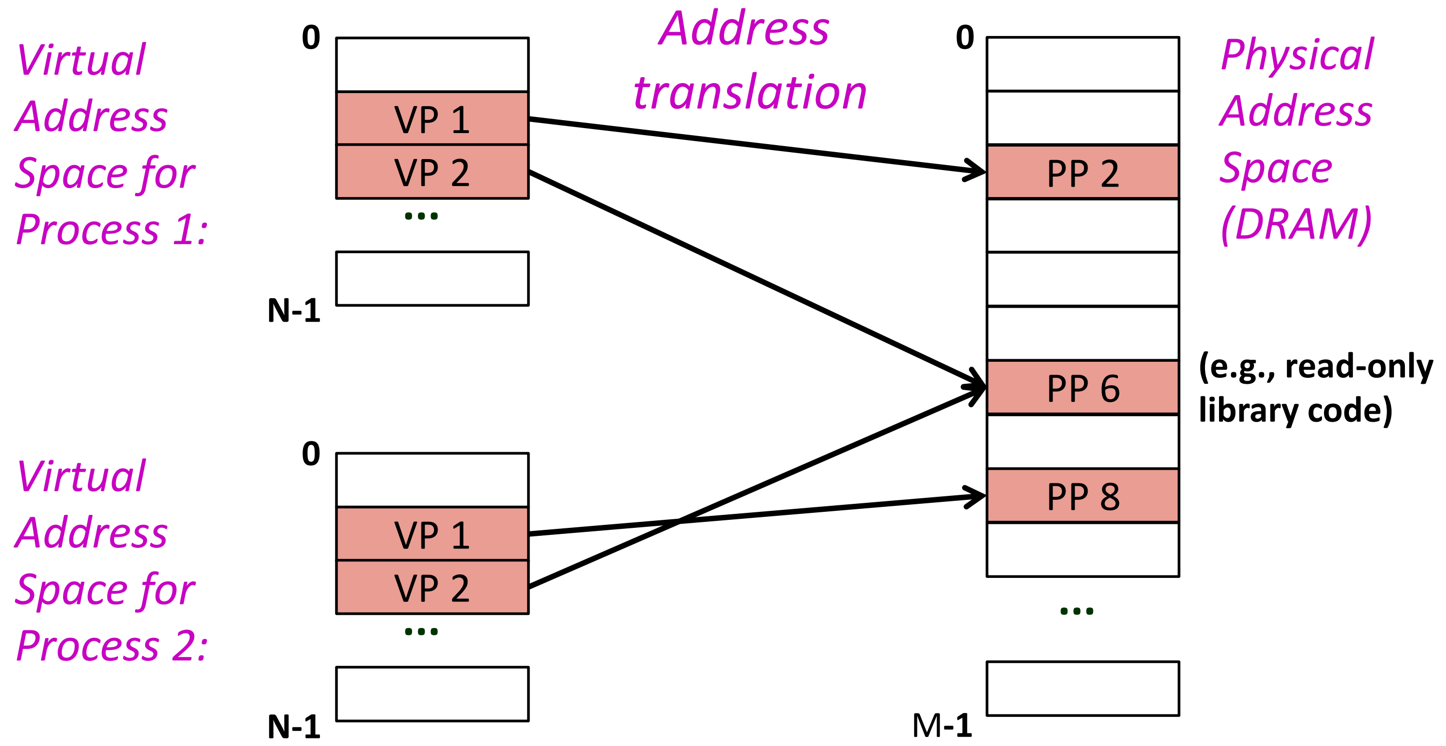
If (SUM(working set sizes) > main memory size)

- *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



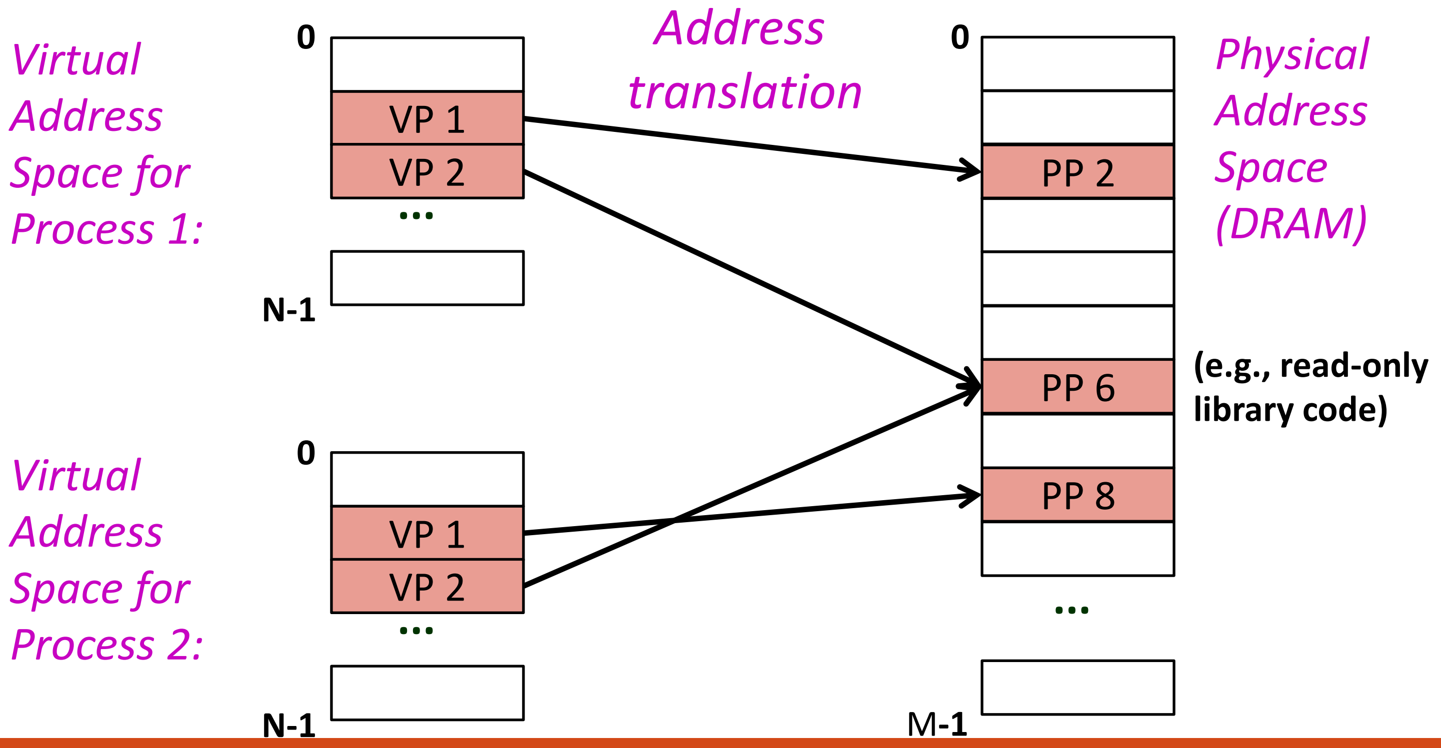
Simplifying allocation and sharing

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

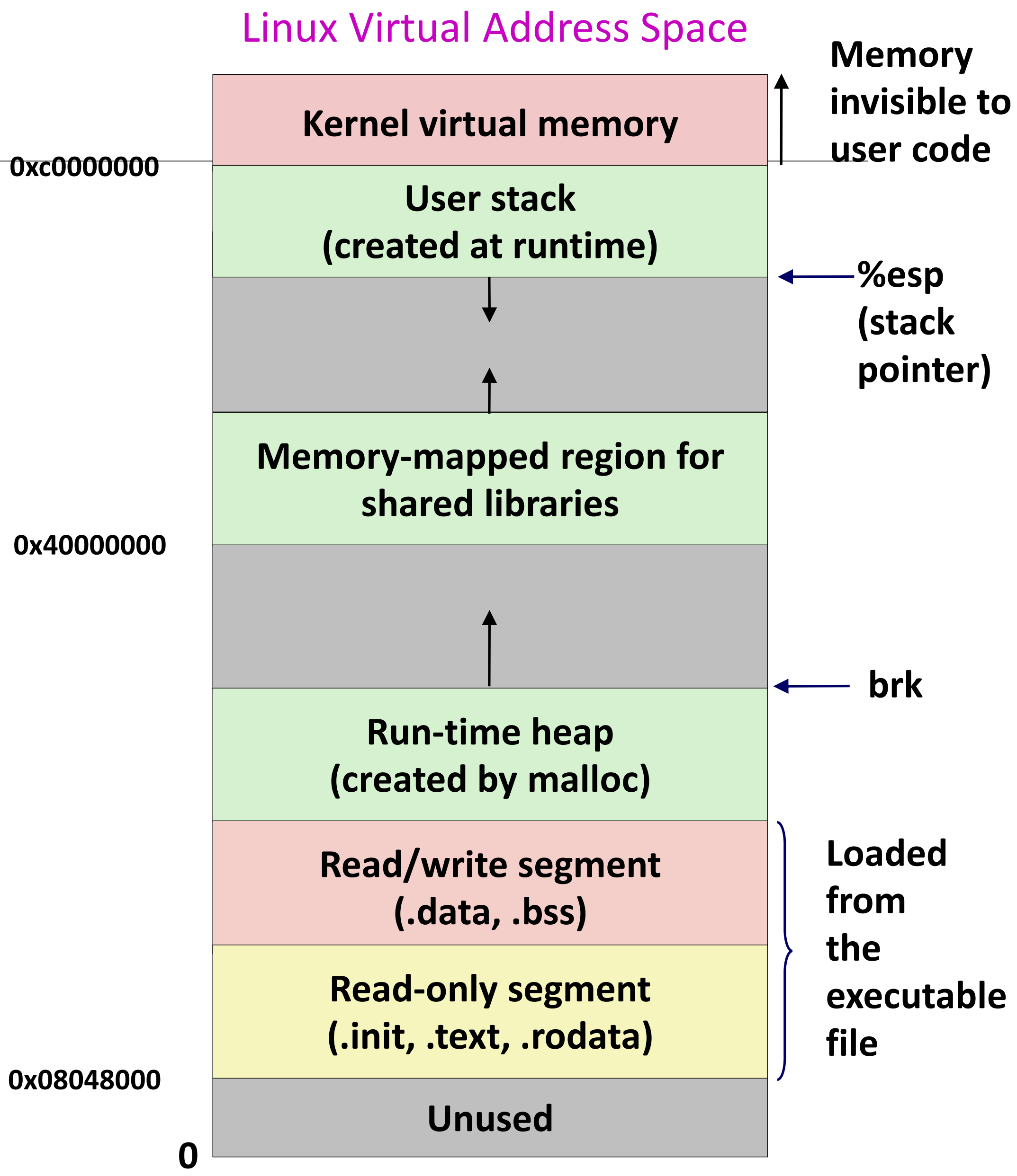
Sharing code and data among processes

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



Simplifying Linking

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



(3) 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)

Process i:

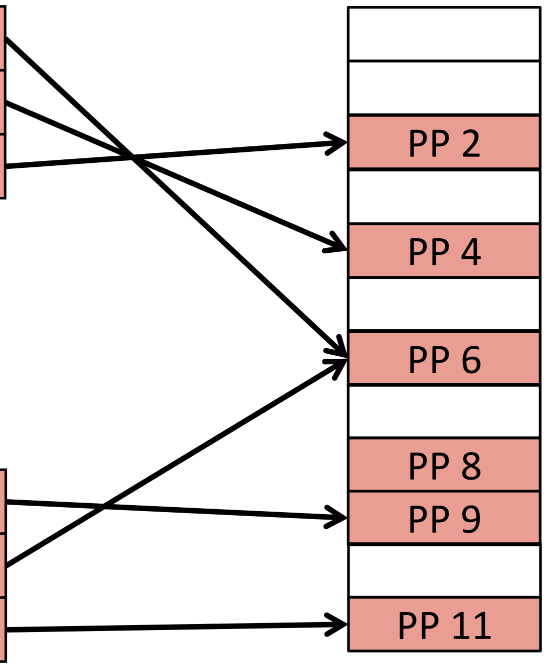
	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 6
VP 1:	No	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	PP 2

⋮

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

Physical Address Space



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 point to check permissions

VM Address Translation

Virtual Address Space

- $V = \{0, 1, \ldots, N-1\}$

Physical Address Space

- $P = \{0, 1, \ldots, M-1\}$

Address Translation

- ***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

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)

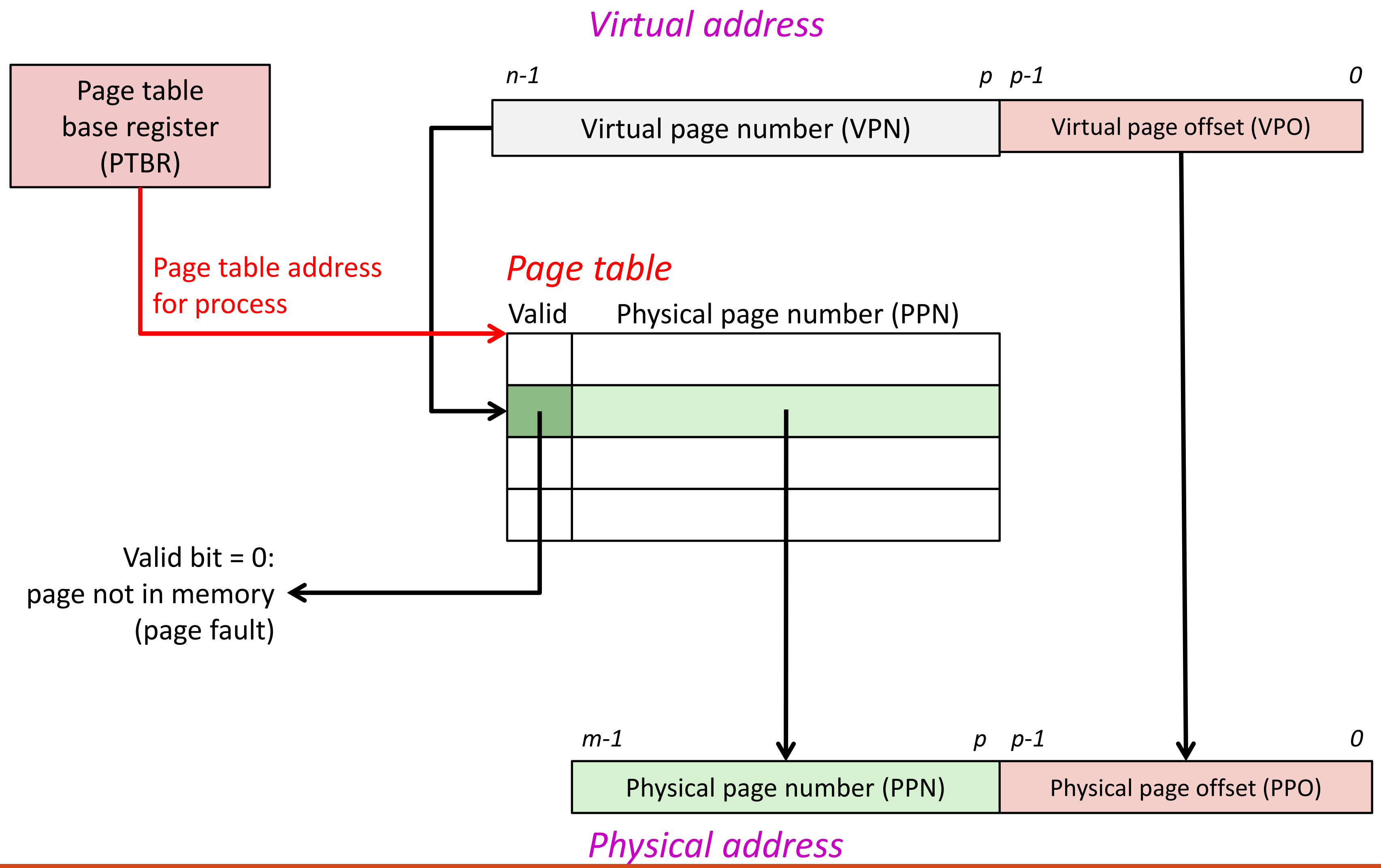
Components of the virtual address (VA)

- **VPO**: Virtual page offset
- **VPN**: Virtual page number

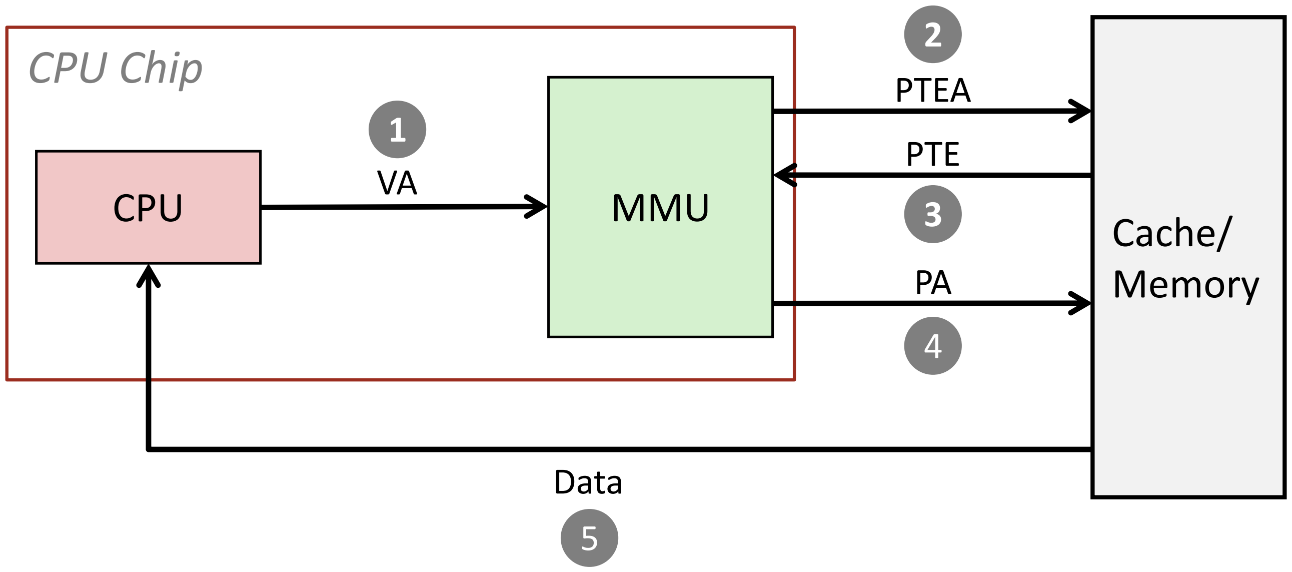
Components of the physical address (PA)

- **PPO**: Physical page offset (same as VPO, usually V/P dropped)
- **PPN:** Physical page number

Address Translation With a Page Table



Address Translation: Page Hit



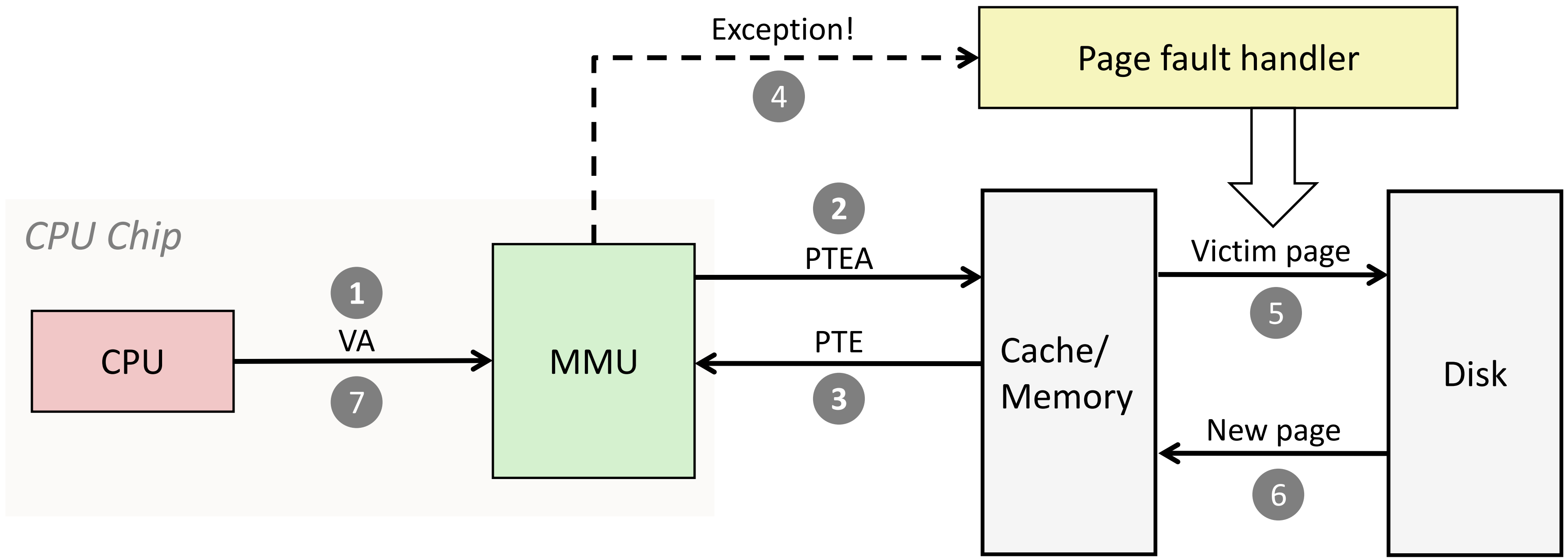
1) Processor sends virtual address to MMU

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

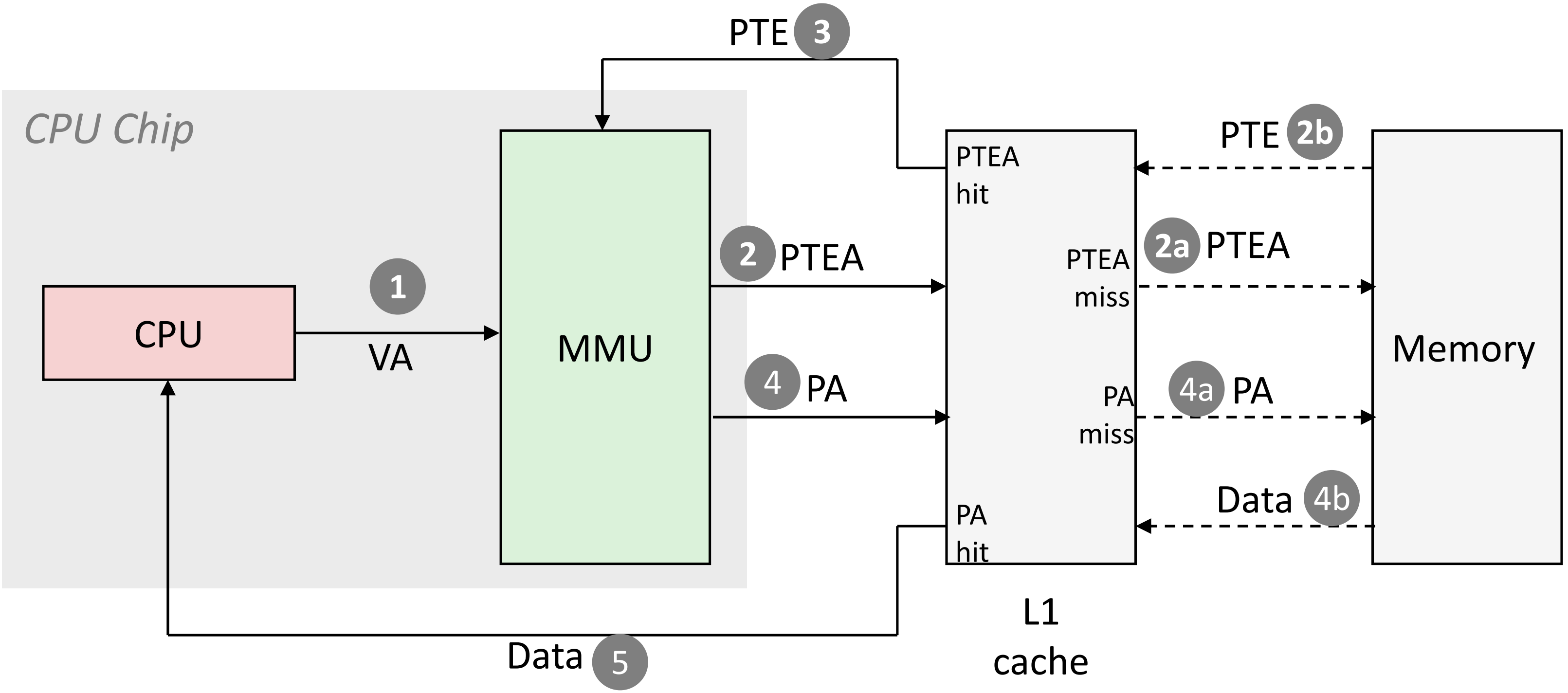
7) Handler returns to original process, restarting faulting instruction

Question:

Are the PTEs cached like other memory accesses?

Yes (and no: see next question)

Integrating VM and Cache



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

Question:

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 a TLB

Page table entries (PTEs) are cached in L1 like any other memory word

- PTEs may be evicted by other data references
- But even PTE hit still requires a small L1 delay!

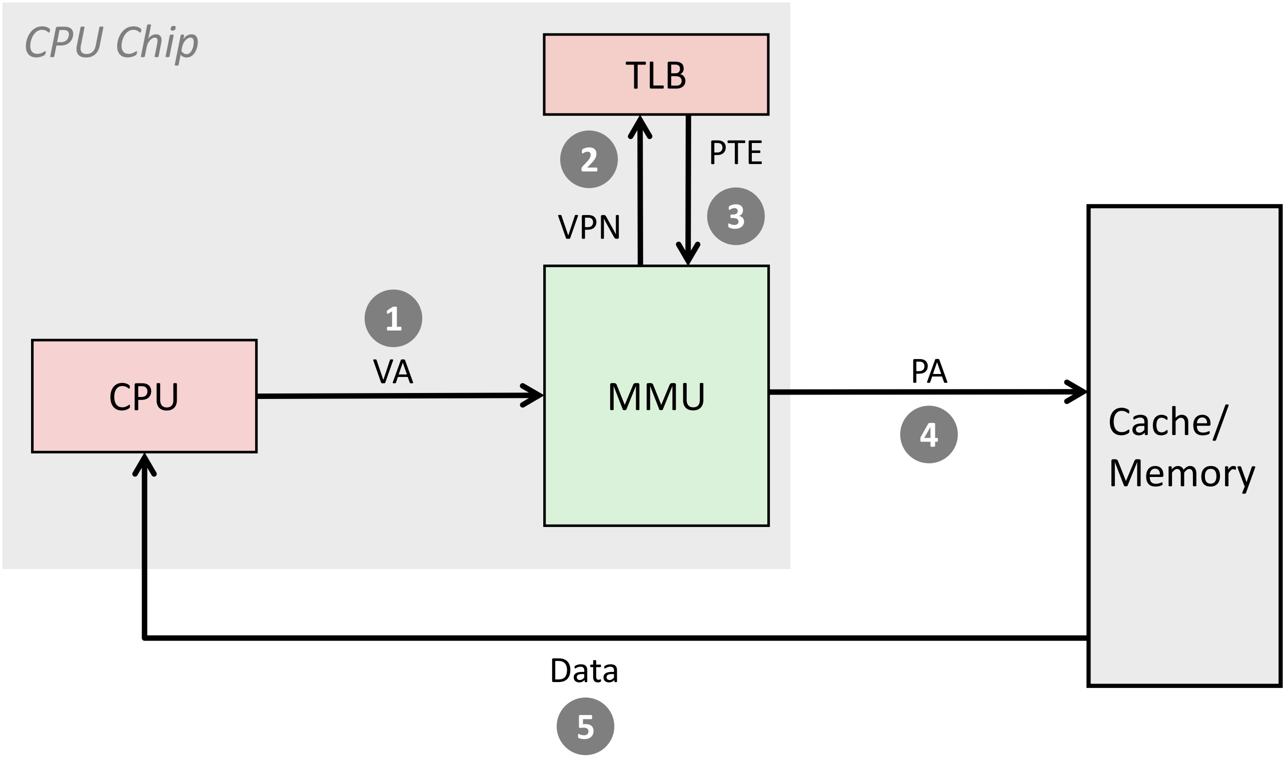
Solution: *Translation Lookaside Buffer* (TLB)

- Small hardware cache in MMU
- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages

TLBs only need a few entries. Why?

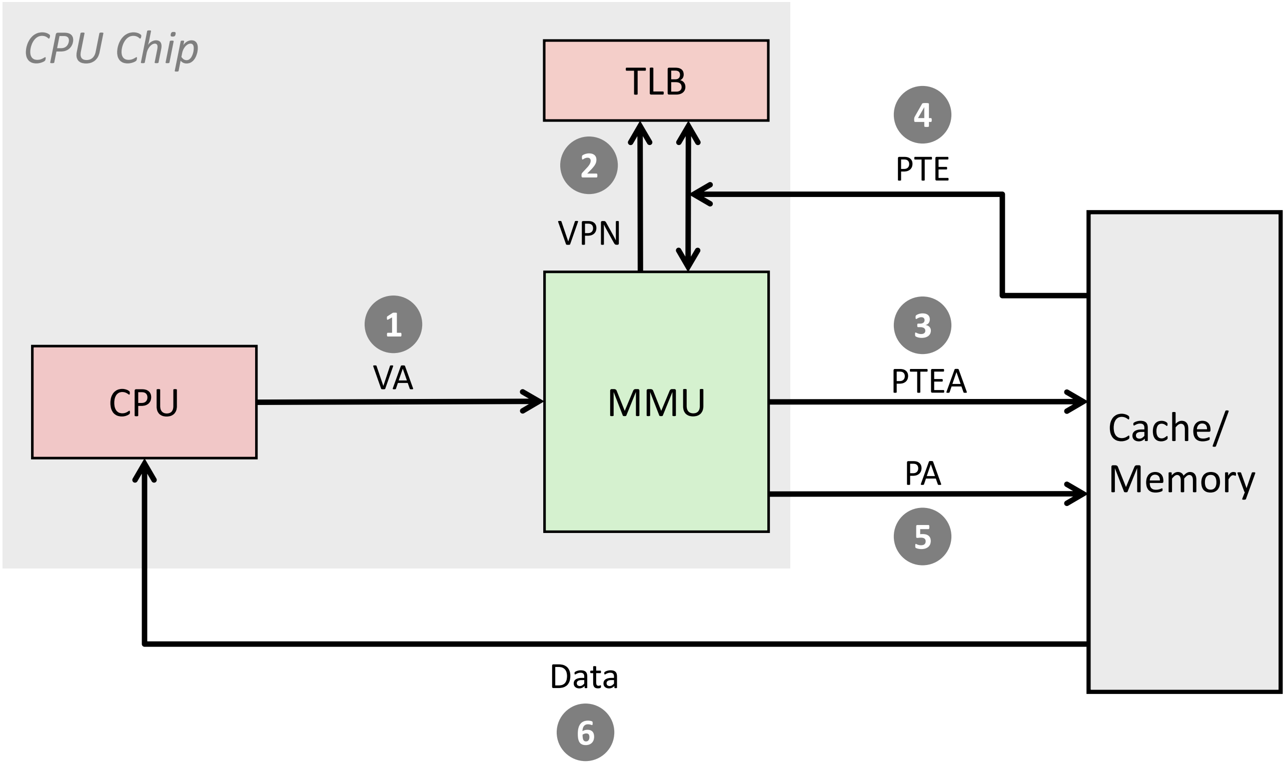
- 512 entries ➔ reach of 512 * 4KB = 2MB
- *But less and less true as main memory gets large and page sizes don't scale!*

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.

TLB Coherence

Observation: Page tables rarely change
- “Read mostly” data

…and only change at clearly defined points
- Viz., page faults

TLBs are not kept coherent by hardware

Software (OS) invalidates TLB entries when PTEs change
- Called a “TLB shootdown”
- Mechanism: inter-process interrupt (IPI)

Question(s):

Isn’t accessing memory still slower than without VM?

Yes, if TLB is on critical path

- Instead, access cache using virtual addresses

Problem solved?

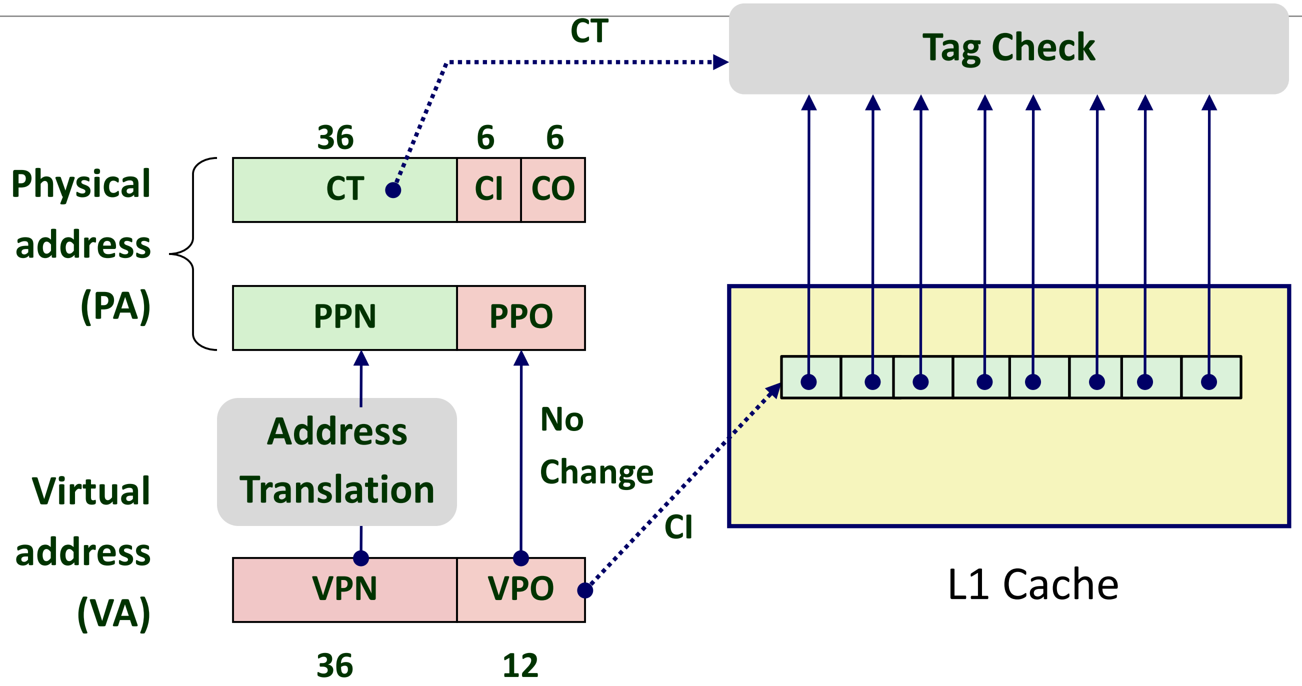
No, <u>synonyms</u> and <u>homonyms</u> lose coherence (!!)

- Synonym: different VAs, same PAs
- Homonym: same VAs, different PAs

Common solution: **index** the cache using <u>virtual</u> address but <u>tag</u> using physical addresses (VIPT)

- What must we guarantee for this to work?

Speeding Up L1 Access



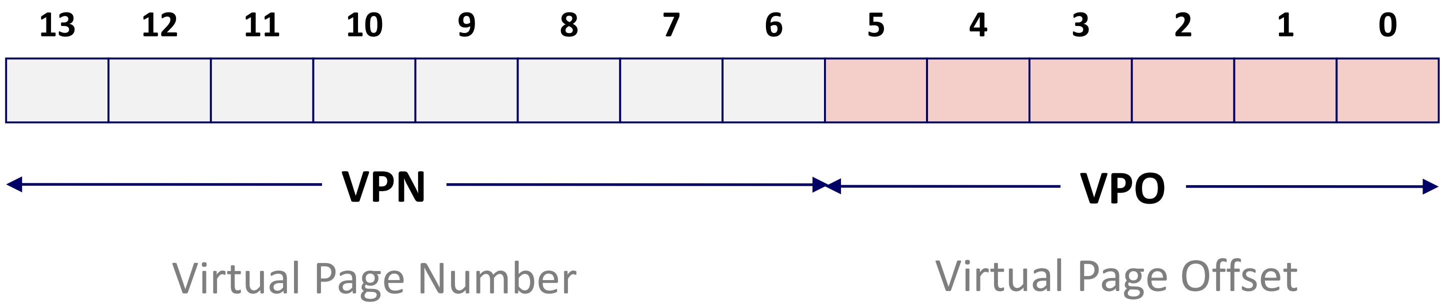
Bits that determine cache index are **identical in virtual and physical address**
- Can index into cache while address translation taking place
- TLB hit rate >> cache hit rate, so tag generally available

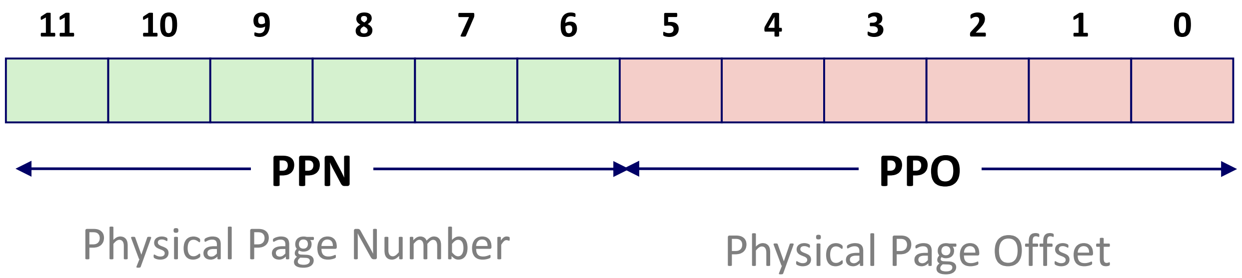
Cache carefully sized to make this possible
- Each cache way ≤ page size ➔ forced associativity in L1s
- But see: *SIPT: Speculatively Indexed, Physically Tagged* by Zheng et al, HPCA'18

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)

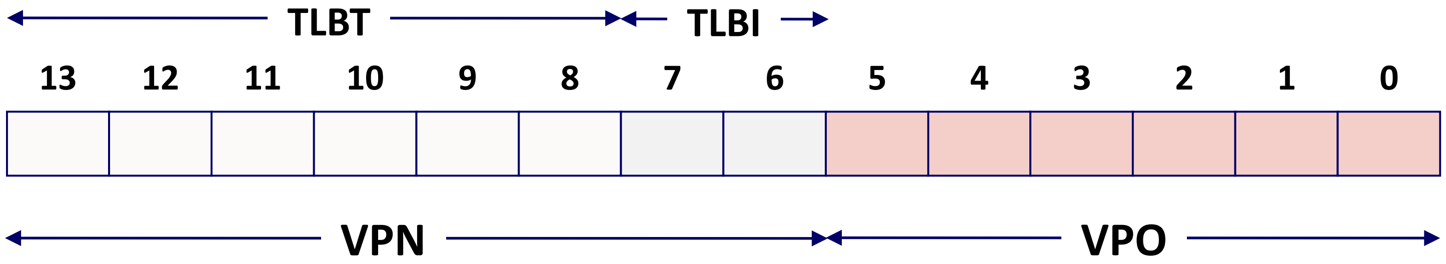
VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

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



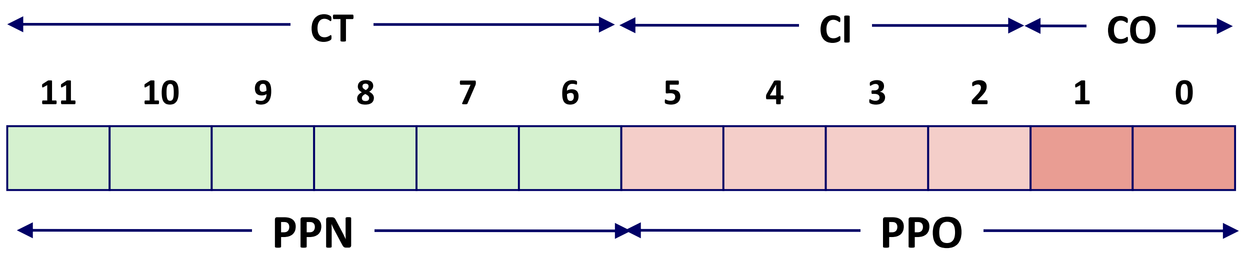
Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

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`

	TLBT						TLBI						
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0
VPN								VPO					

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

Physical Address

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0
PPN						PPO					

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

Address Translation Example #2

Virtual Address: `0x0B8F`

TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0	0	0	1	1	1	1
VPN								VPO					

VPN ___ **0x2E** TLBI ___ 2 TLBT ____ **0x0B** TLB Hit? __ N Page Fault? __ Y PPN: ____ **TBD**

Physical Address

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
PPN						PPO					

CO ___ CI___ CT ____ Hit? __ Byte: ____

Address Translation Example #3

Virtual Address: `0x0020`

	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	TLBT						TLBI							
	0	0	0	0	0	0	0	0	1	0	0	0	0	0
	VPN								VPO					

VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? N Page Fault? N PPN: 0x28

Physical Address

	11	10	9	8	7	6	5	4	3	2	1	0
	CT						CI				CO	
	1	0	1	0	0	0	1	0	0	0	0	0
	PPN						PPO					

CO 0 CI 0x8 CT 0x28 Hit? N Byte: Mem

Question:

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:

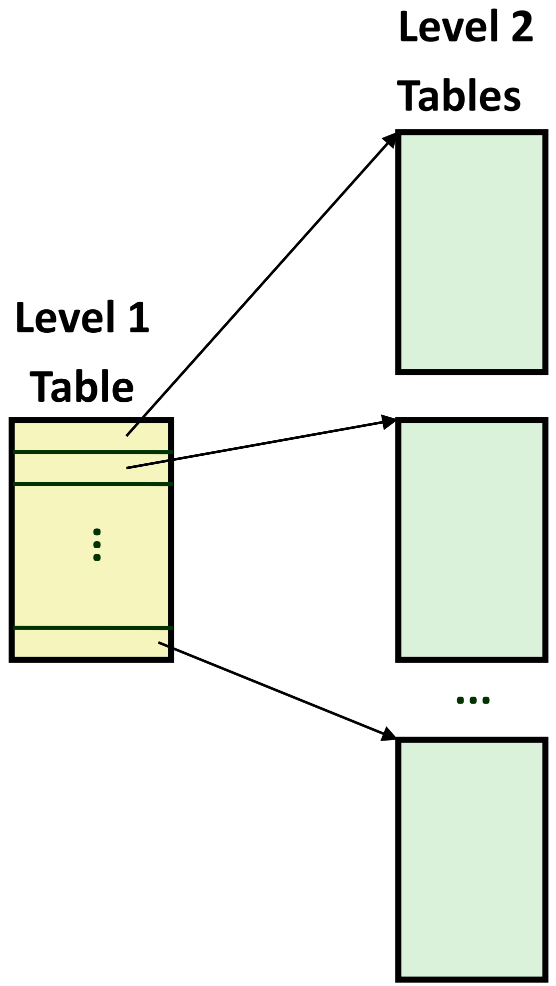
- 4KB (2^{12}) page size, 64-bit address space, 8-byte PTE

Problem:

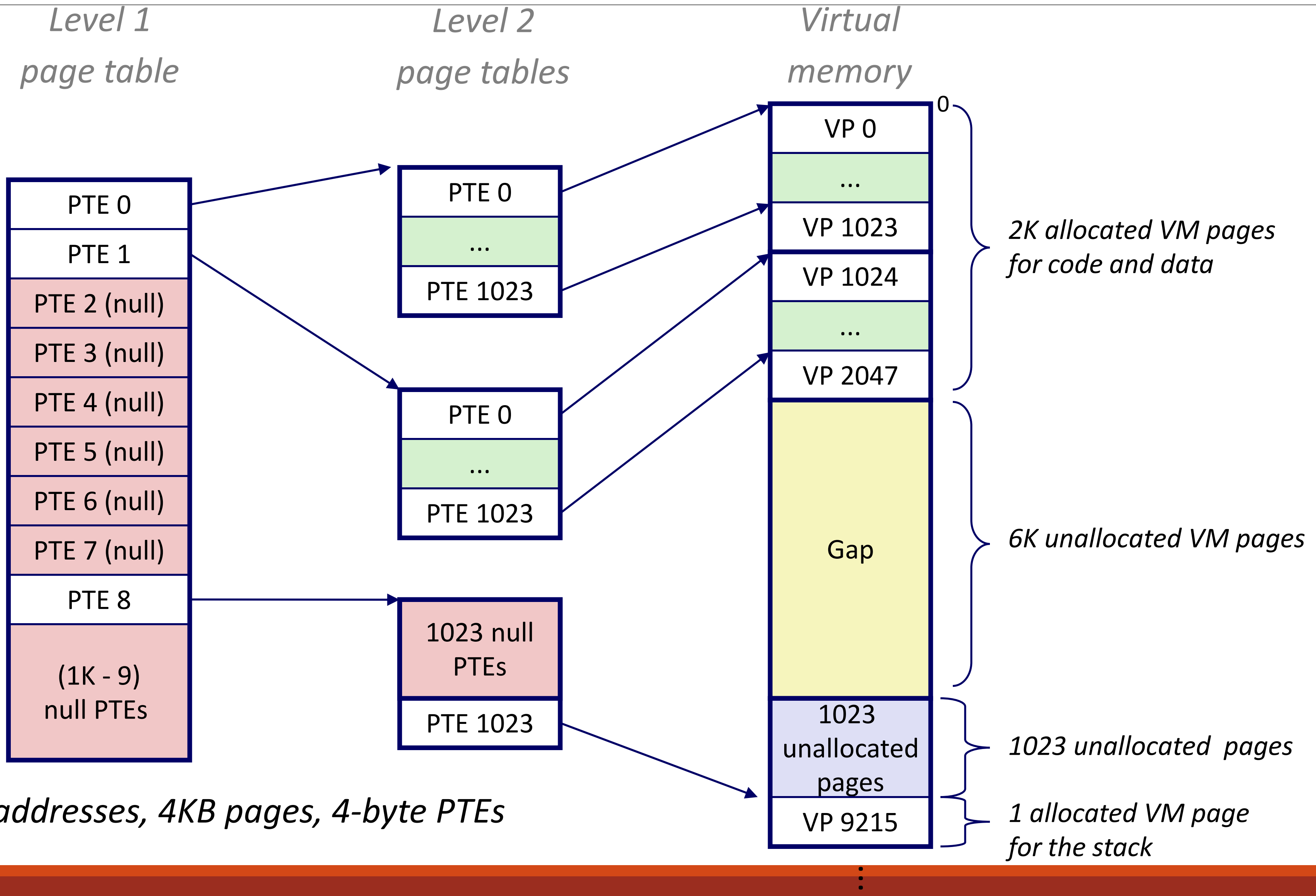
- Would need a 32,000 TB page table!
  - $2^{64} * 2^{-12} * 2^{3} = 2^{55}$ bytes

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)

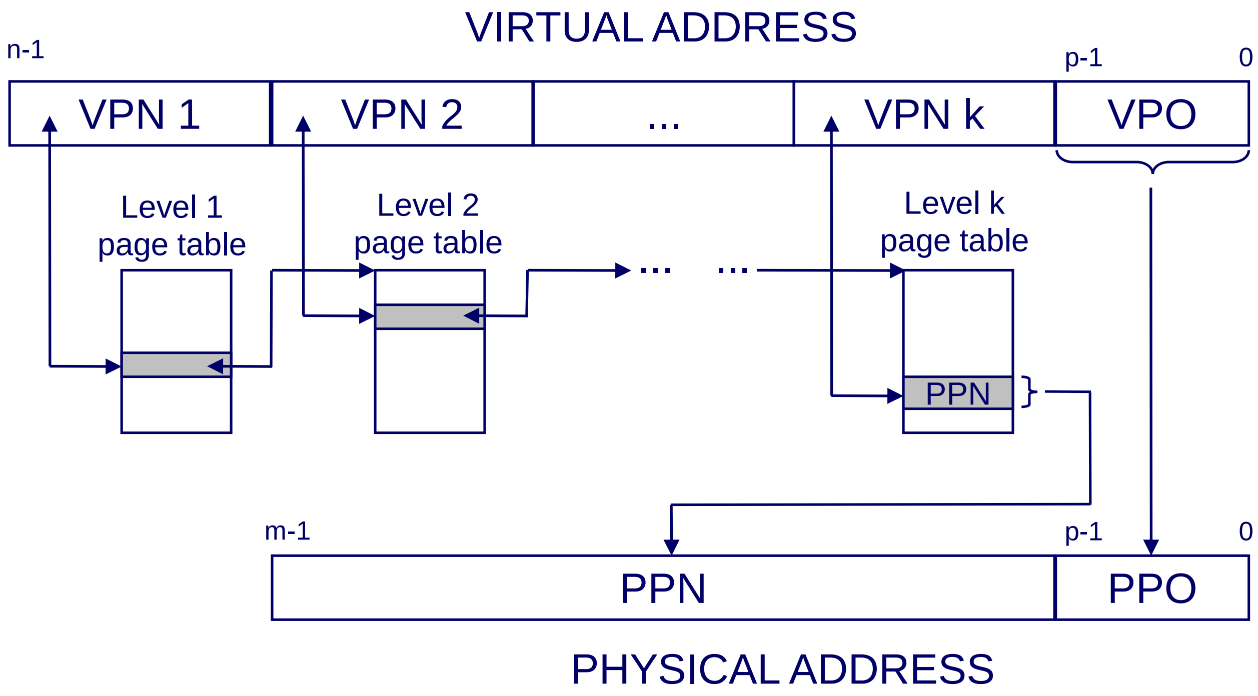


A Two-Level Page Table Hierarchy

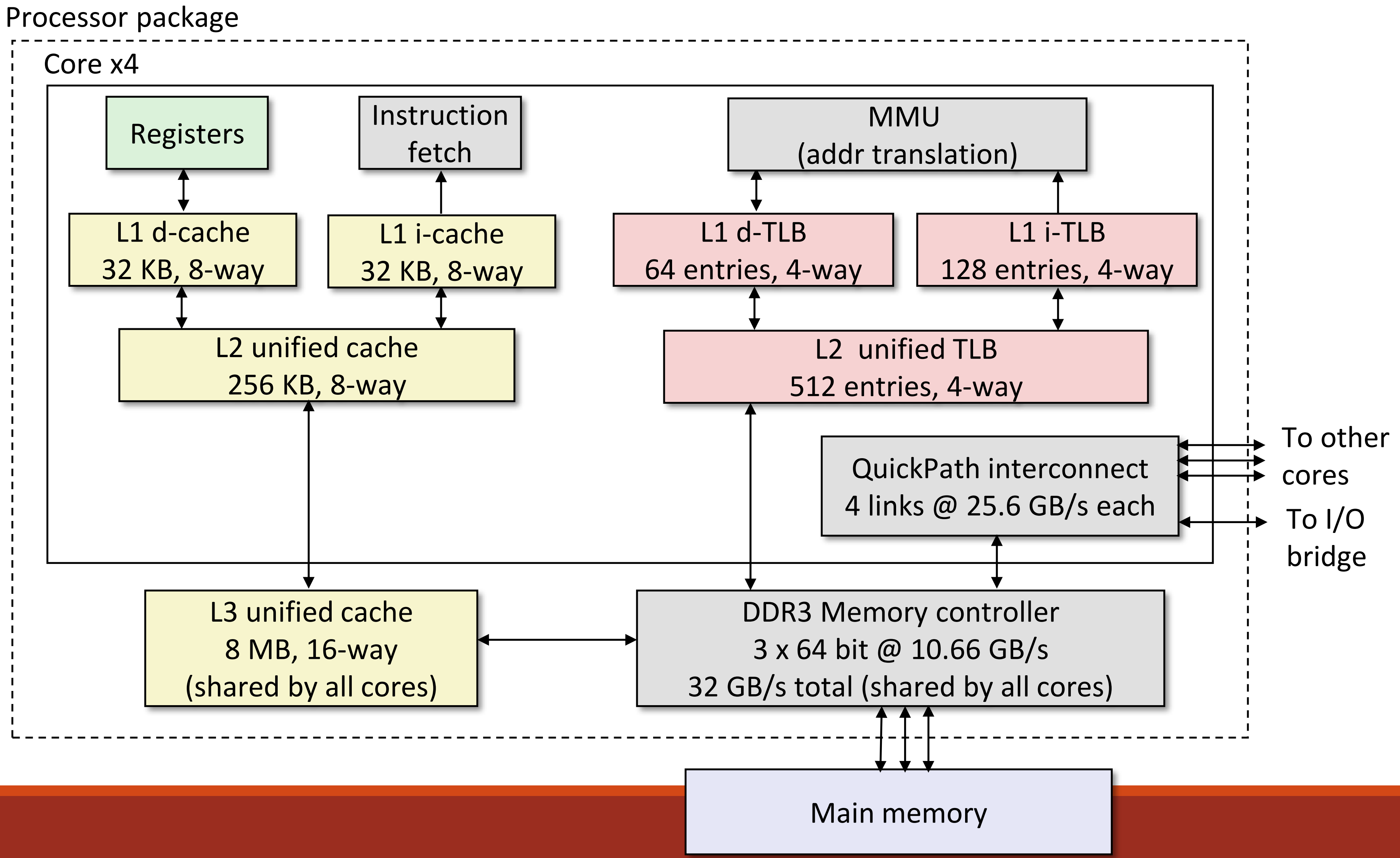


32 bit addresses, 4KB pages, 4-byte PTEs

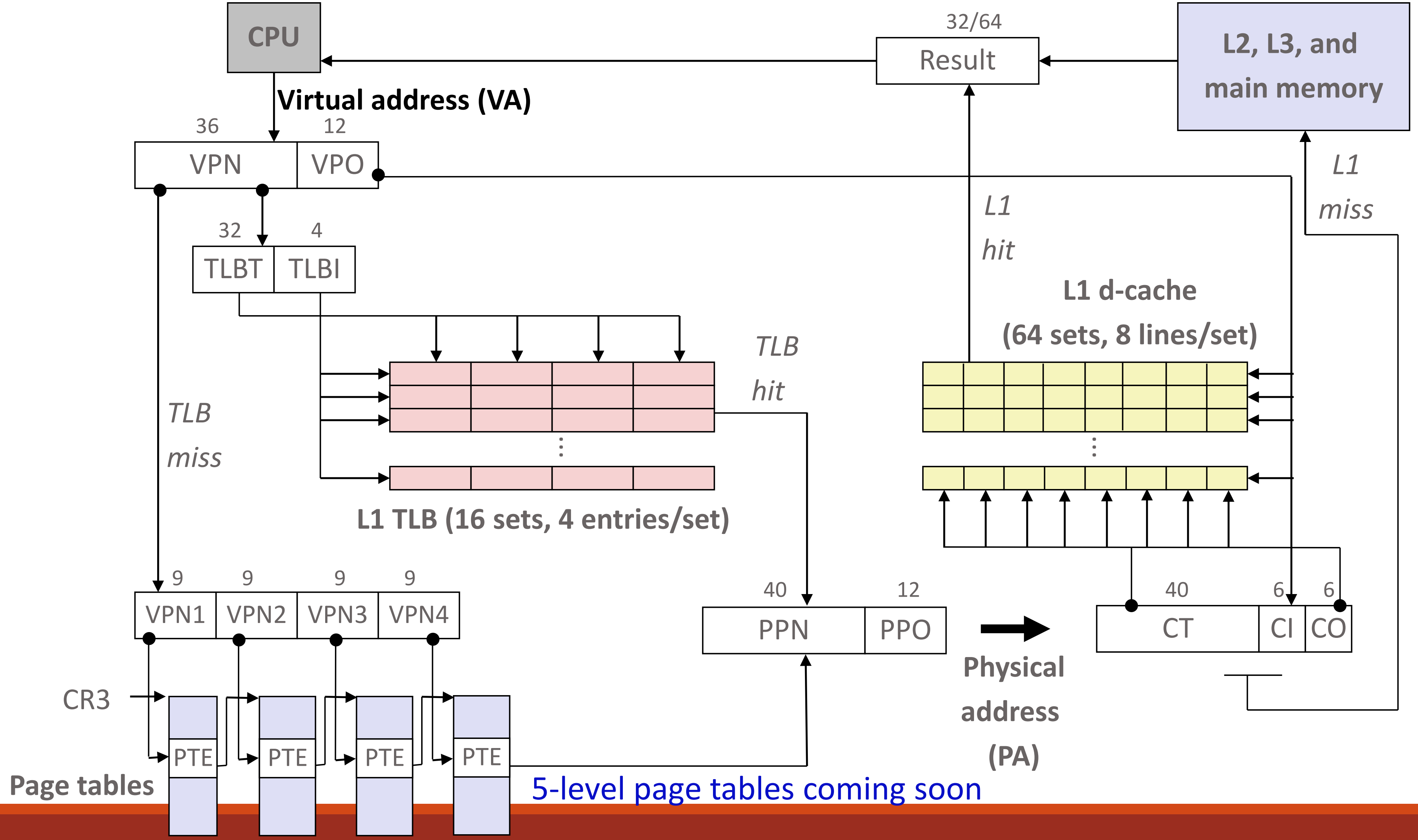
Translating with a k-level Page Table



Intel Core i7 Memory System

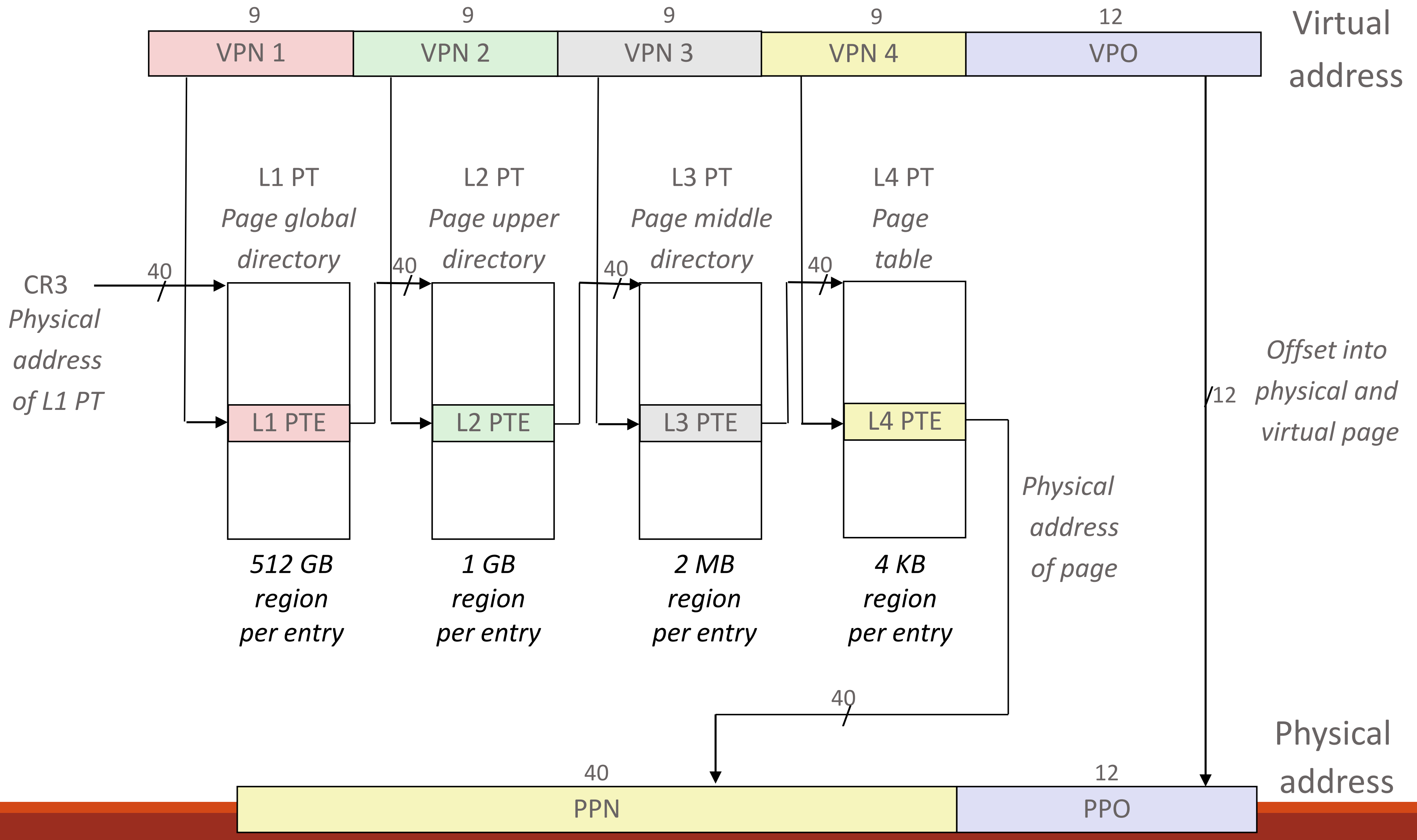


End-to-end Core i7 Address Translation



5-level page tables coming soon

Core i7 Page Table Translation



Core i7 Level 1-3 Page Table Entries

63	62 … 52	51 … 12	11 … 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page table physical base address	Unused	G	PS		A	CD	WT	U/S	R/W	P=1

Available for OS (page table location on disk)	P=0

Each entry references a 4K child page table

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

CD: Caching disabled or enabled for the child page table.

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

PS: Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

G: Global page (don't evict from TLB on task switch)

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

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

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

CD: Cache disabled (1) or enabled (0)

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

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

G: Global page (don’t evict from TLB on task switch)

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

Recent research into virtual memory

Problems:

- Many levels of indirection are slow
- Hard to map large (many GBs) working sets with small TLB

Recall motivations for page-based VM

- Make more efficient use of limited DRAM ?
- Simplify memory management for programmers ✓
- Protection between programs ✓

Mechanisms to map large memory regions

- Huge pages (2MB, 1GB), supported in current hardware & Linux
- Segments *[Karakostas et al, ISCA’15]*
- Fine-grain coalescing of pages into continuous regions *[Park et al, ISCA’17]*

These ideas compromise on DRAM efficiency & require contiguous allocations in physical memory

Virtual Memory Summary

Virtual memory several important problems

- Efficient use of physical memory
- Simplifies memory management
- Protection in shared systems

Implemented by using DRAM as a cache

- Design differs from processor caches

Scaling up this abstraction involves many tricks throughout the processor

- Improving the VM abstraction is an active research area