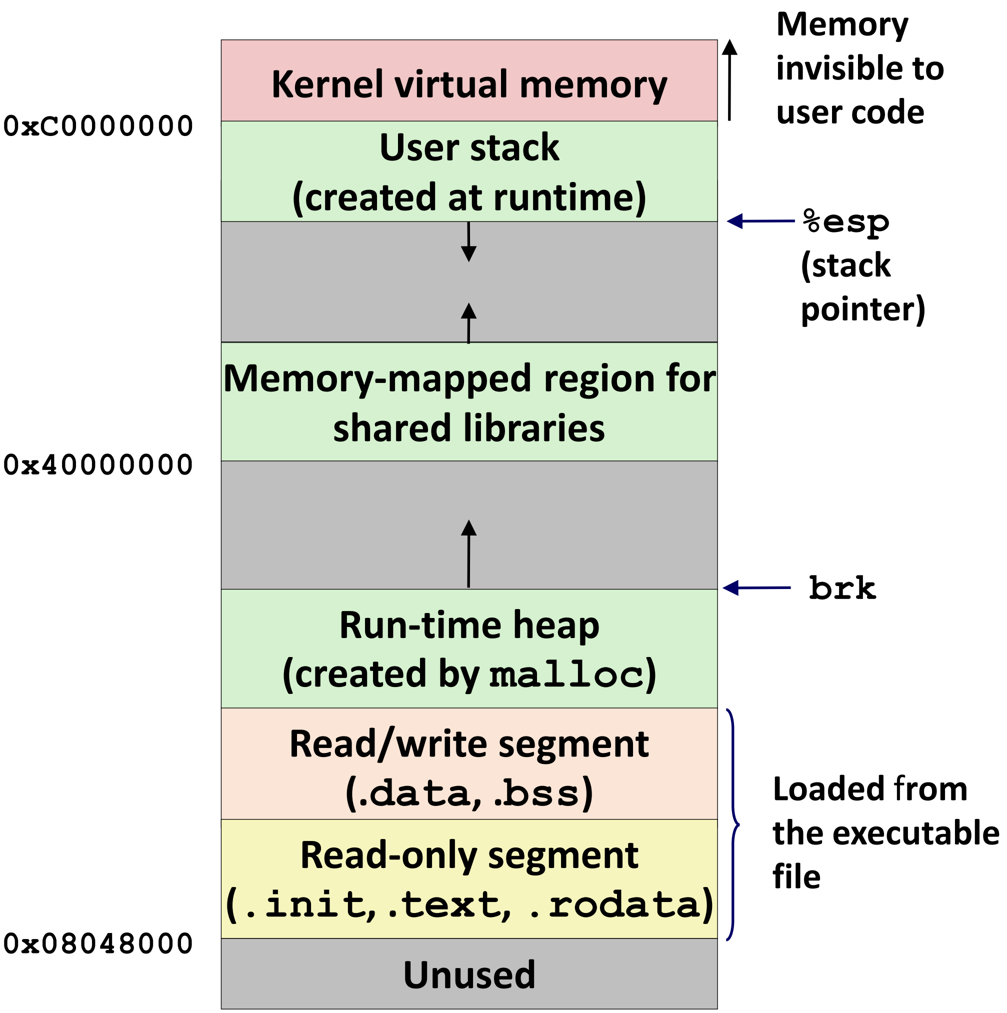
Memory Systems
Lec10 – Virtual Memory

Chin-Fu Nien (粘儆夫)

Module 2: System & Software (con’t)

Virtual Memory Concepts

- We've been viewing memory as a linear array.
- But wait! If you're running 5 processes with stacks at `0xC0000000`, don't their addresses conflict?
- Nope! Each process has its own address space.
- How???



Kernel virtual memory
Memory invisible to user code
0xC0000000
User stack (created at runtime)
%esp (stack pointer)
Memory-mapped region for shared libraries
0x40000000
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Loaded from the executable file
Read-only segment (.init, .text, .rodata)
0x08048000
Unused

[Slides Credit] Karthic Palaniappan, "Recitation 10: Virtual Memory" from "Intro to Computer Systems" course, Fall 2015

Virtual memory concepts

- We define a mapping from the virtual address used by the process to the actual physical address of the data in memory.

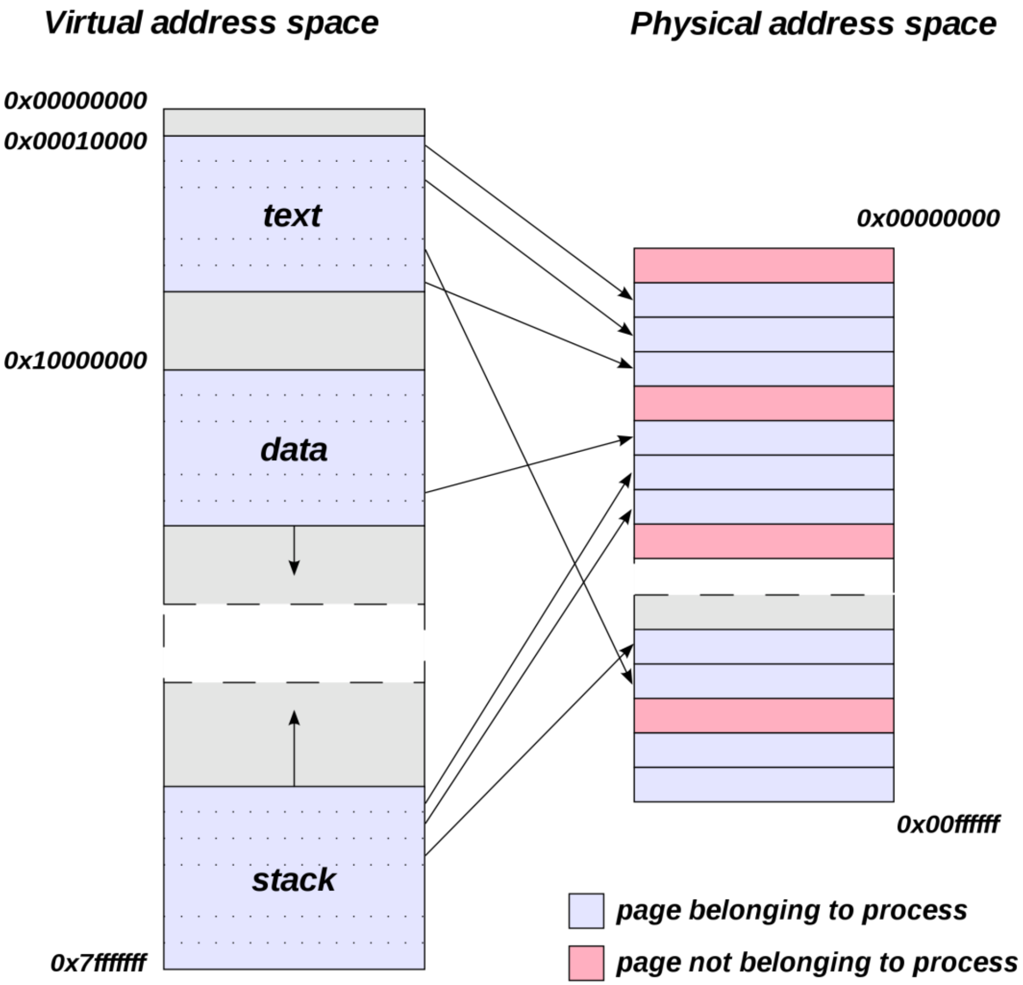


Image: http://en.wikipedia.org/wiki/File:Virtual_address_space_and_physical_address_space_relationship.svg

[Slides Credit] Karthic Palaniappan, “Recitation 10: Virtual Memory” from “Intro to Computer Systems” course, Fall 2015

Virtual Memory: Concepts

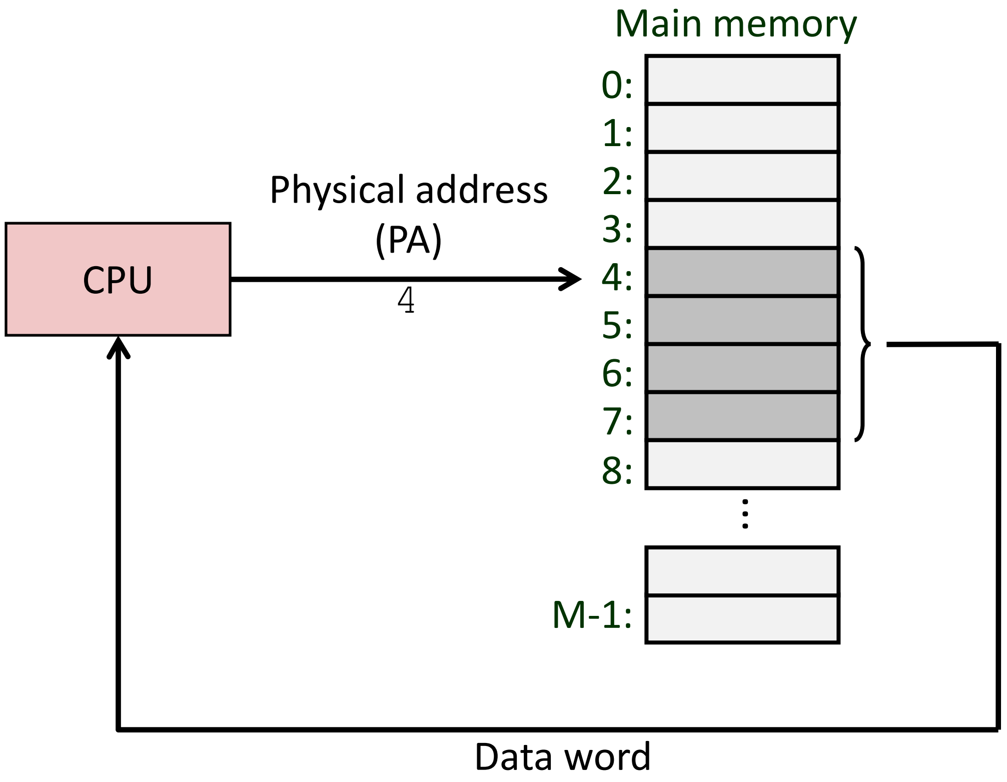
The content of this part is mainly from:
Randal E. Bryant and David R. O'Hallaron, “Computer Systems: A Programmer's Perspective,” 3/e.

(本節內容改自Prof. Randal E. Bryant and David R. O’Hallaron 17th Lectures課程講義)

Today

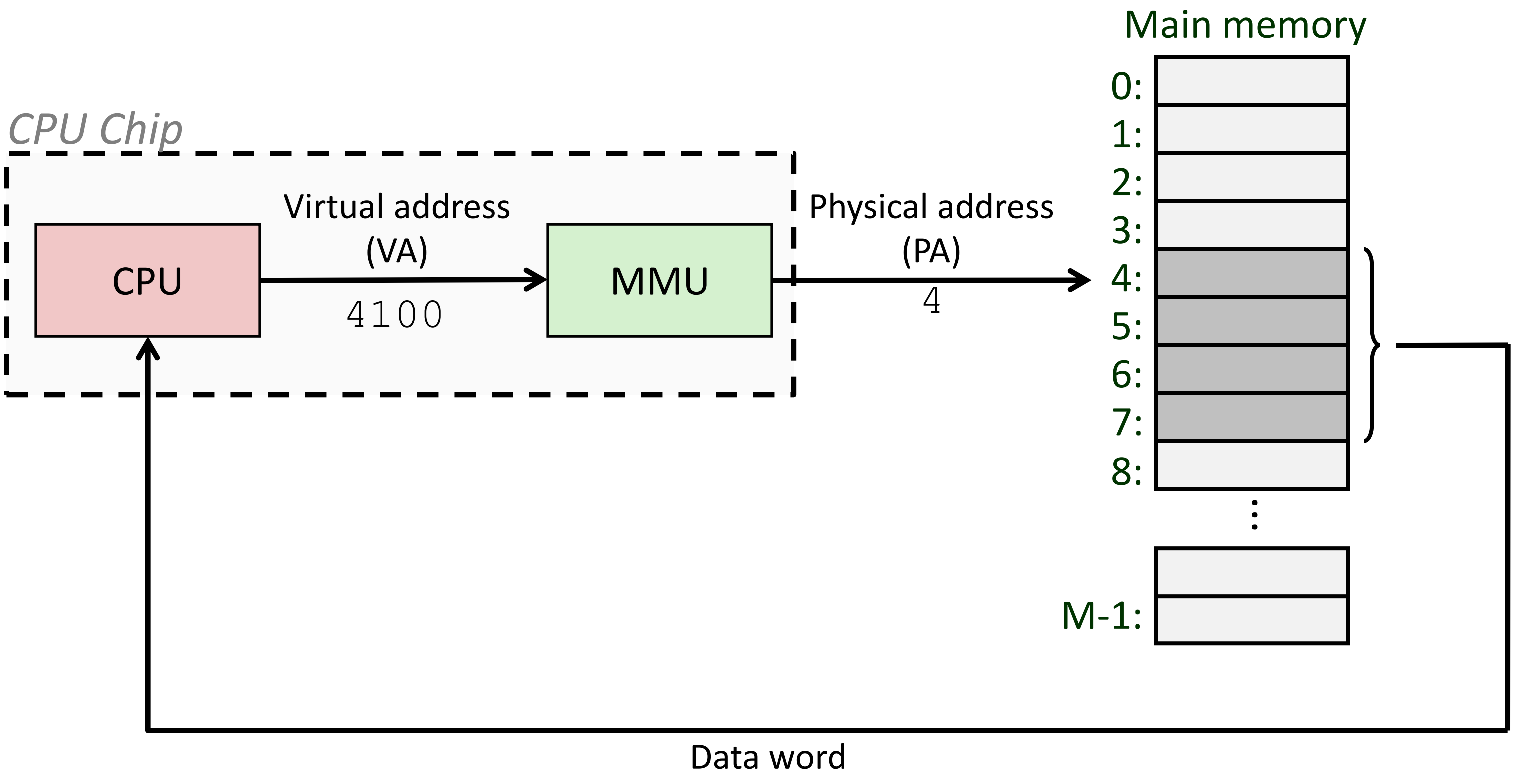
- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation

A System Using Physical Addressing



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

A System Using Virtual Addressing



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

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}

Why Virtual Memory (VM)?

- Uses main memory efficiently
  - Use DRAM as a cache for parts of a virtual address space

- Simplifies memory management
  - Each process gets the same uniform linear address space

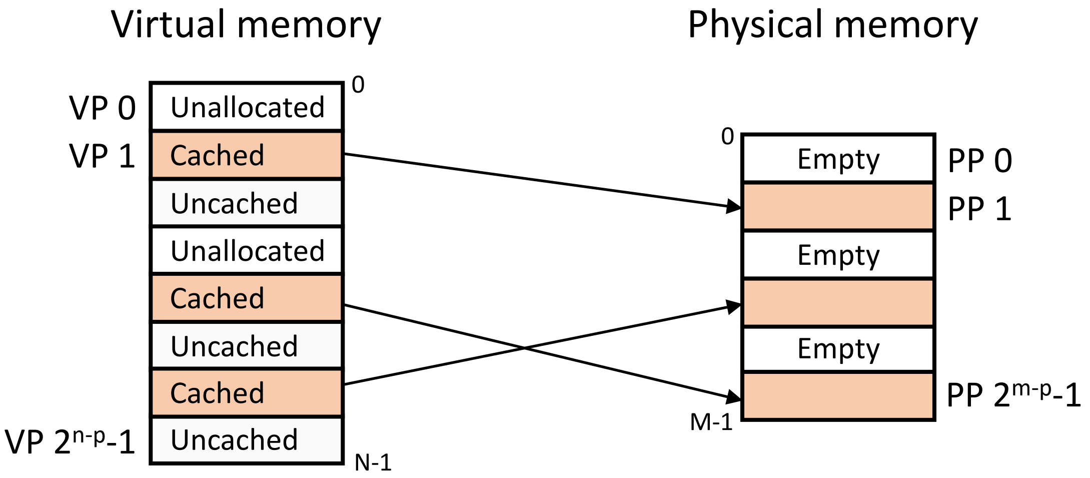
- Isolates address spaces
  - One process can’t interfere with another’s memory
  - User program cannot access privileged kernel information and code

Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation

VM as a Tool for Caching

- Conceptually, *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)



Virtual pages (VPs) stored on disk

Physical pages (PPs) cached in DRAM

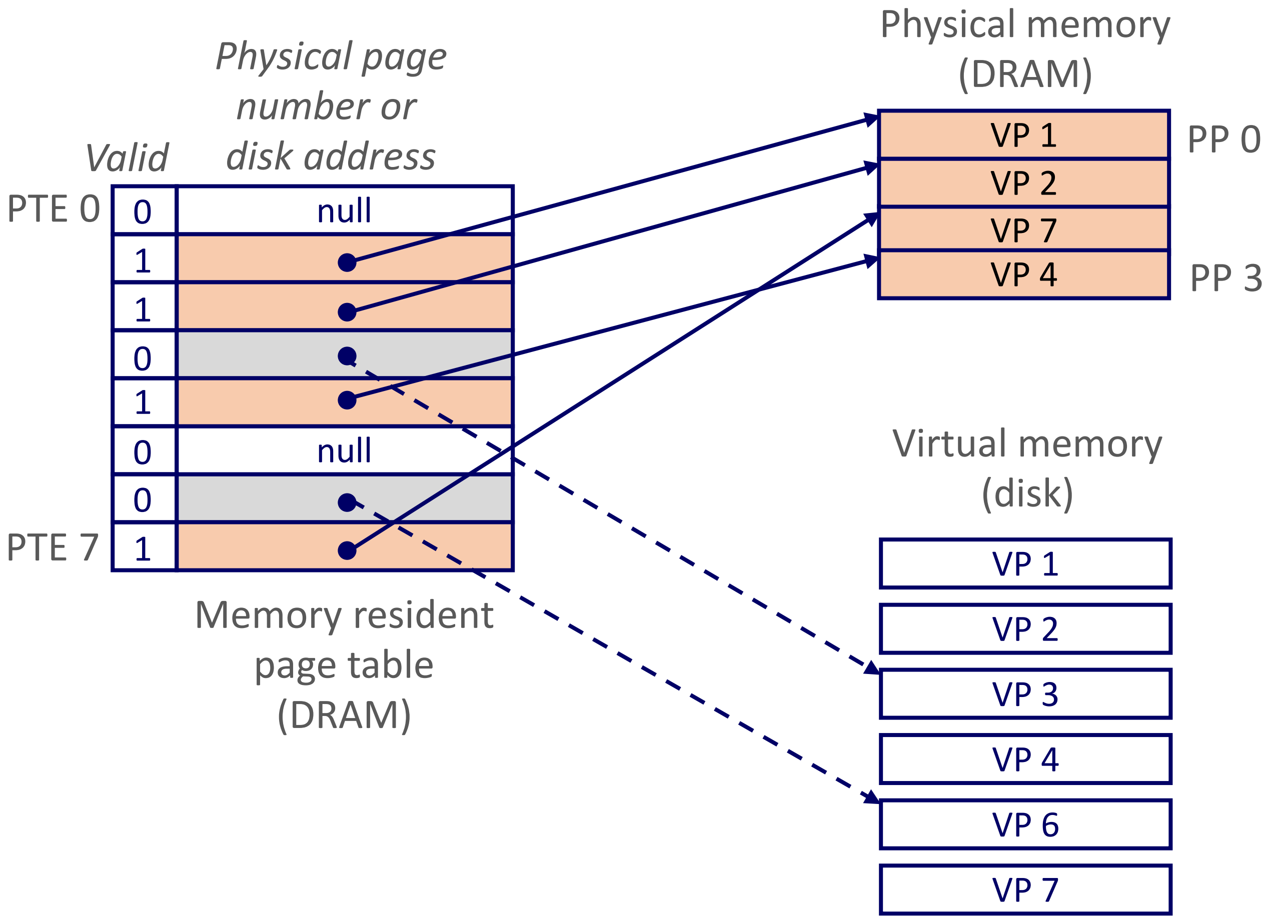
DRAM Cache Organization

- 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 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a “large” mapping function – different from cache memories
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through

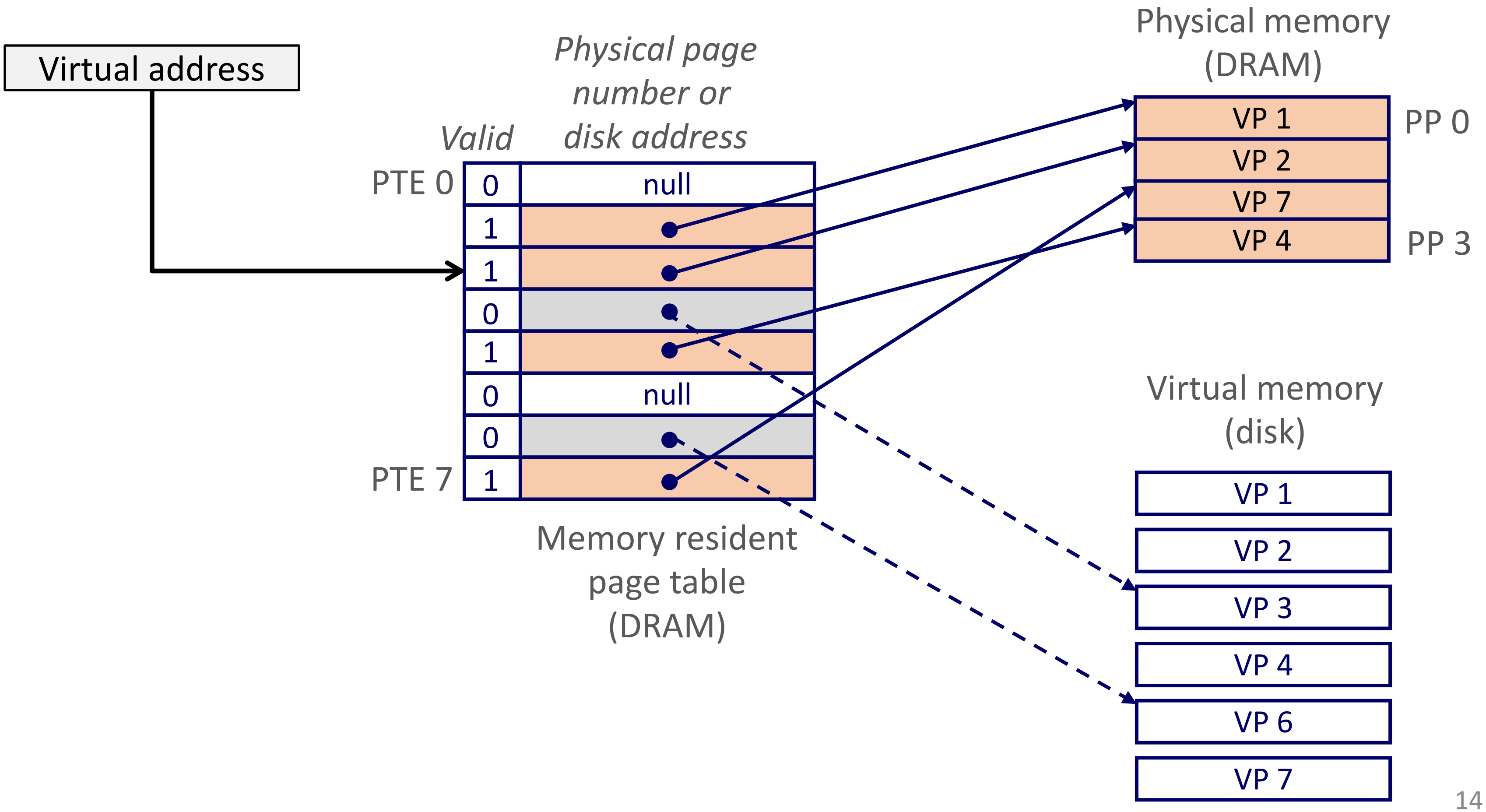
Enabling Data Structure: Page Table

- 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



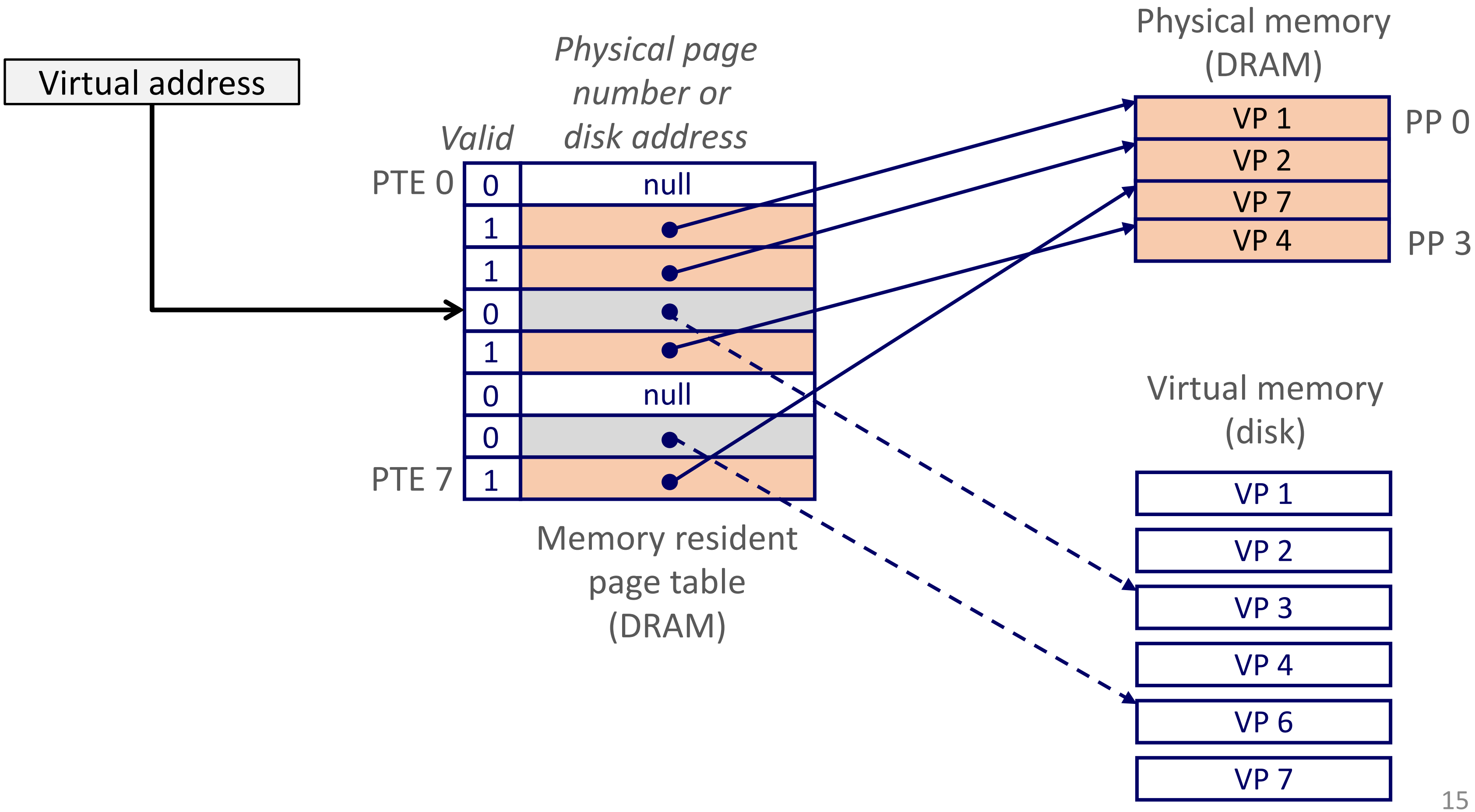
Page Hit

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



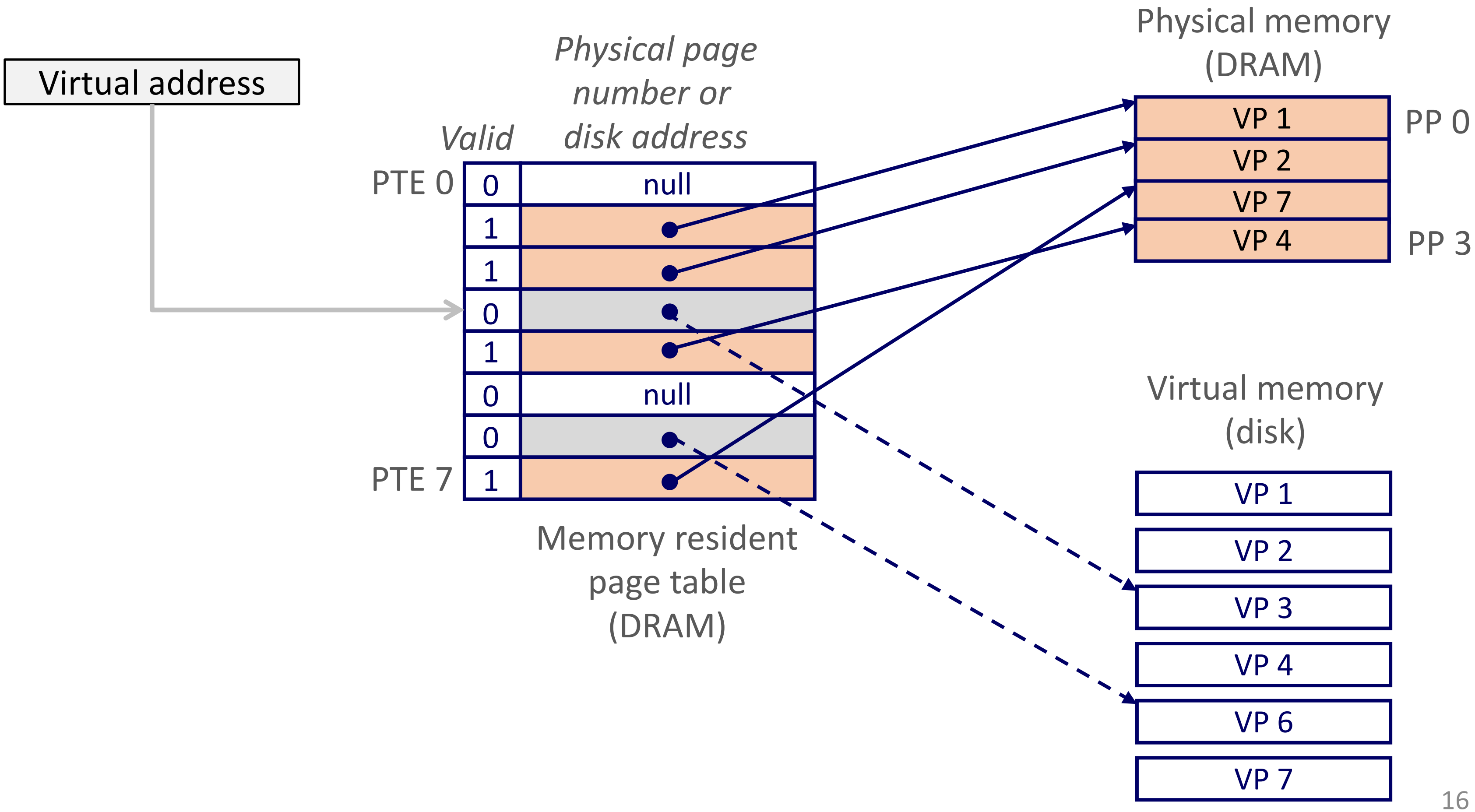
Page Fault

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



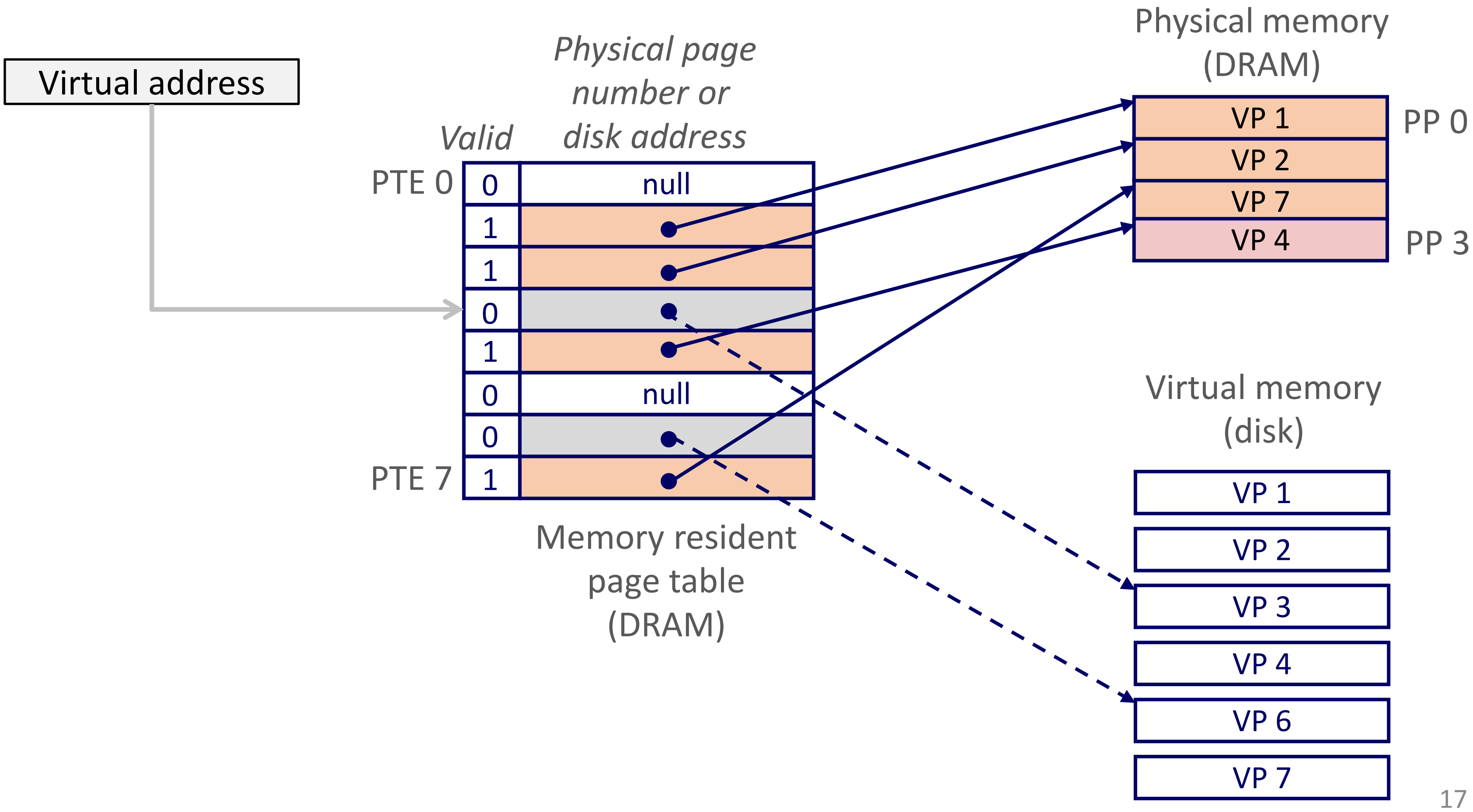
Handling Page Fault

- Page miss causes page fault (an exception)



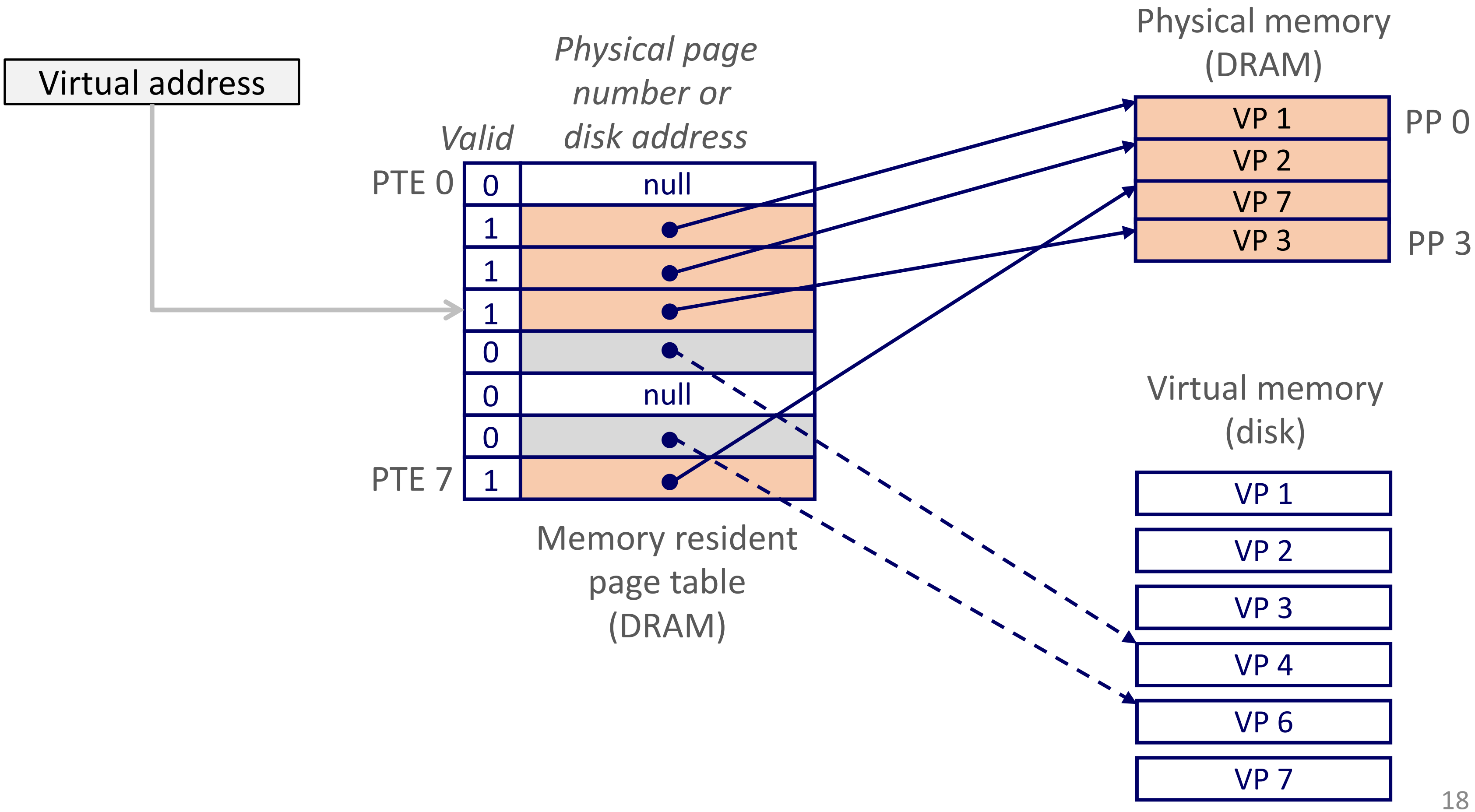
Handling Page Fault

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



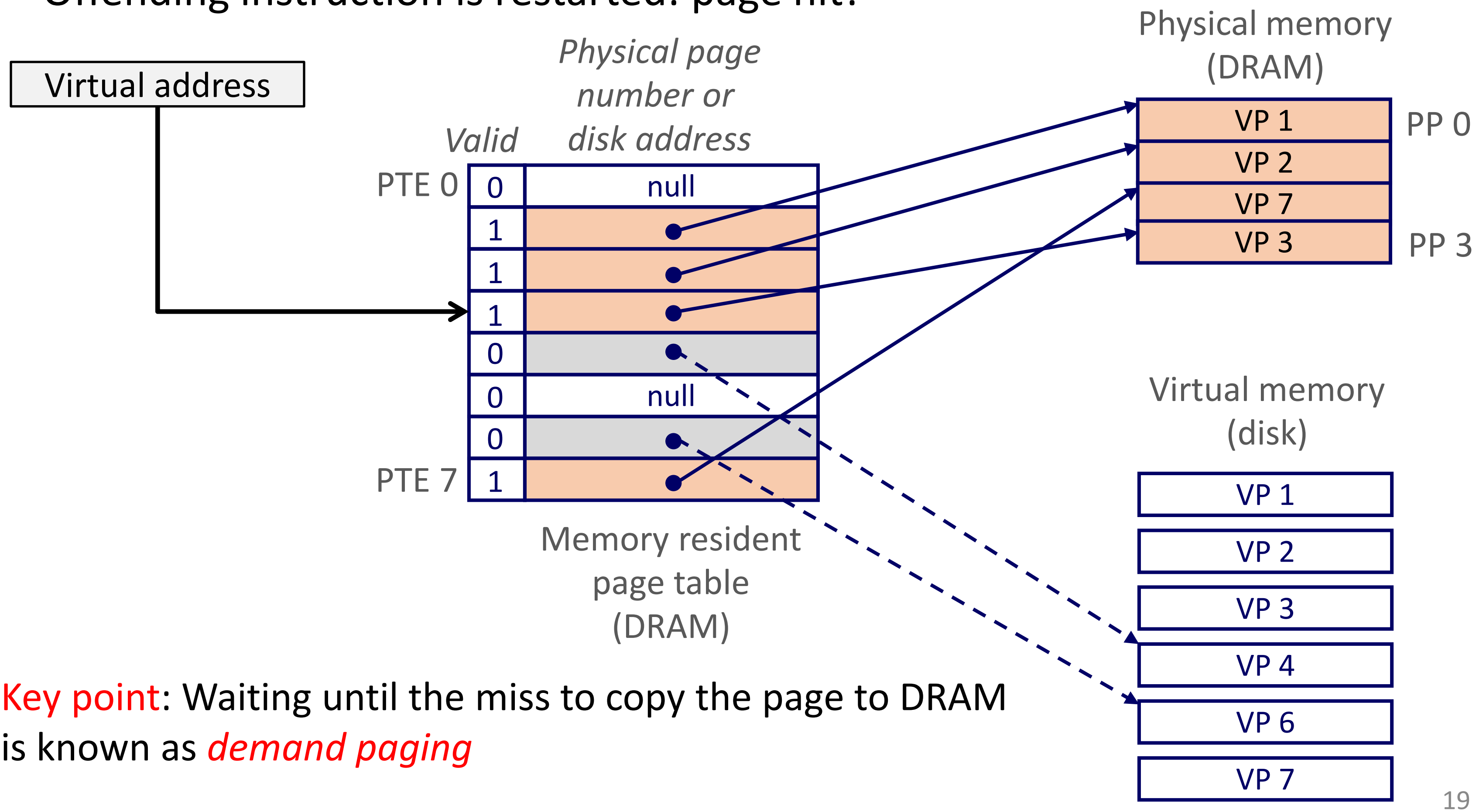
Handling Page Fault

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



Handling Page Fault

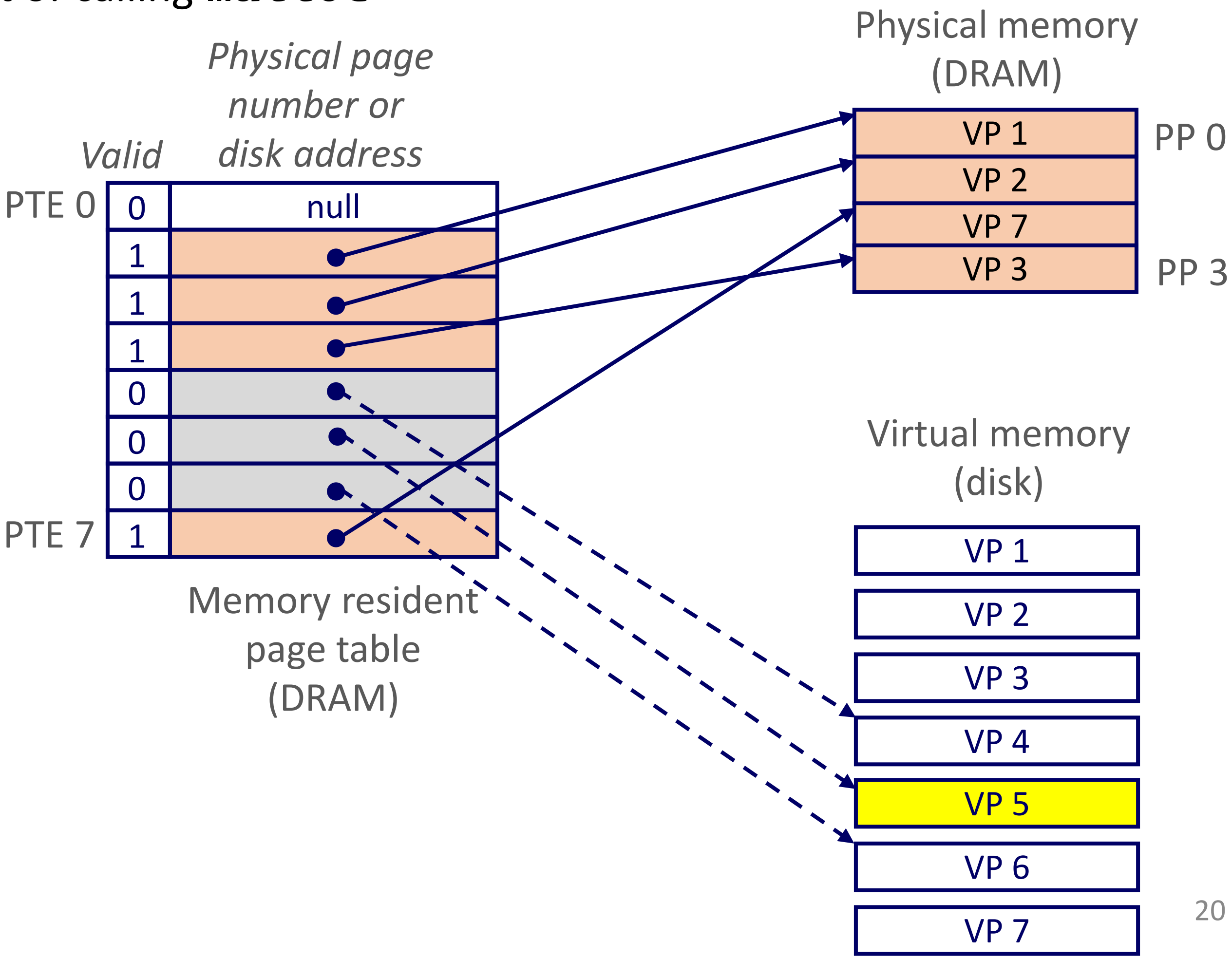
- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!



Key point: Waiting until the miss to copy the page to DRAM is known as *demand paging*

Allocating Pages

- Allocating a new page (VP 5) of virtual memory.
  - E.g., as a result of calling **`malloc`**



Locality to the Rescue Again!

- Virtual memory seems terribly inefficient, but it 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

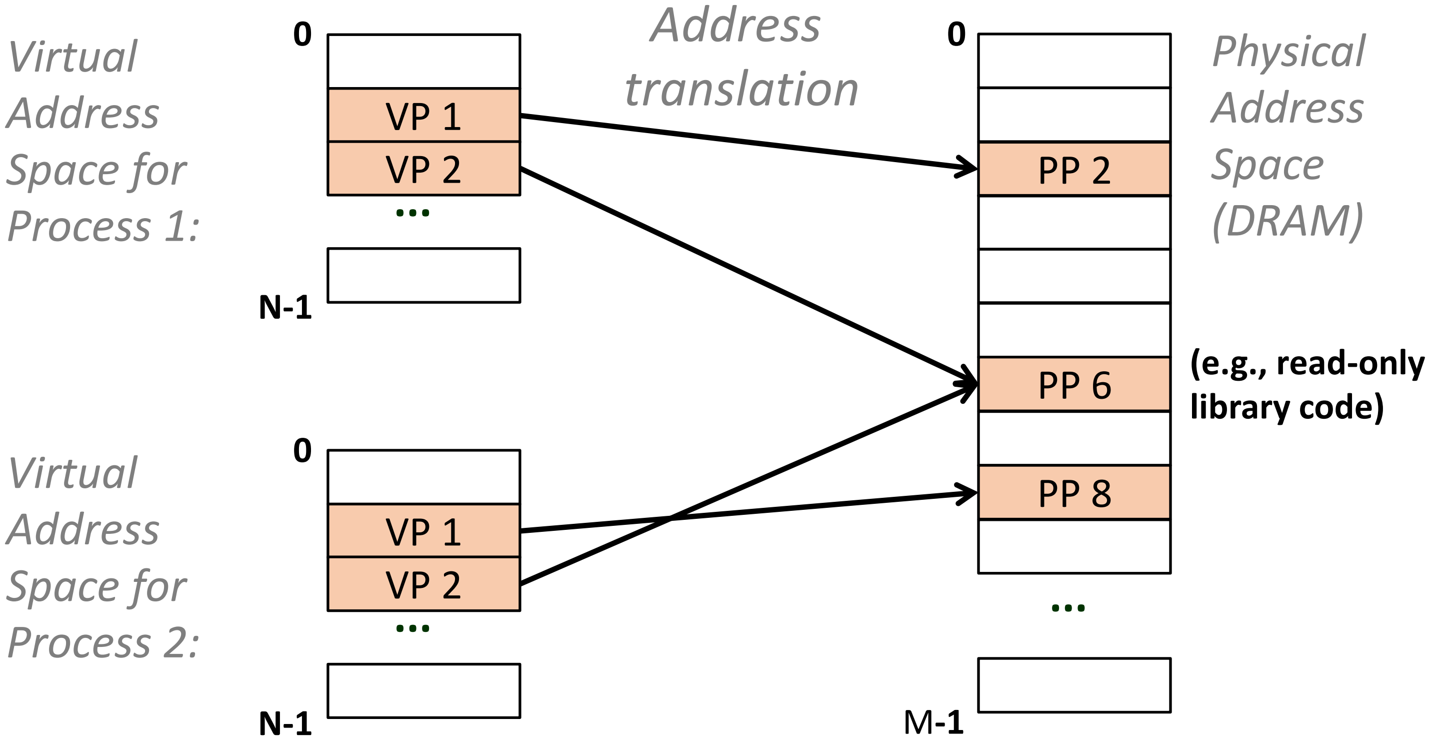
- If (SUM(working set sizes) > main memory size)
  - *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously

Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation

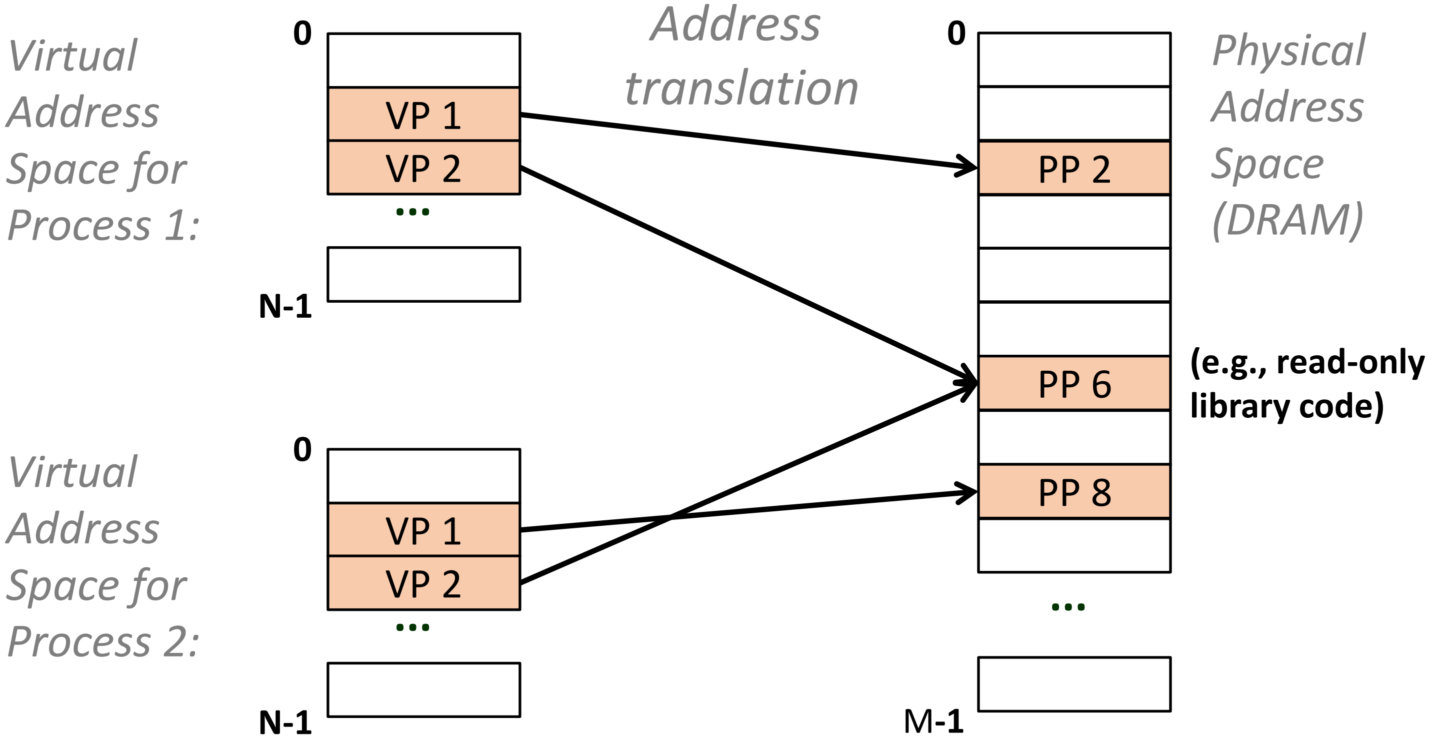
VM as a Tool for Memory Management

- 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 can improve locality


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

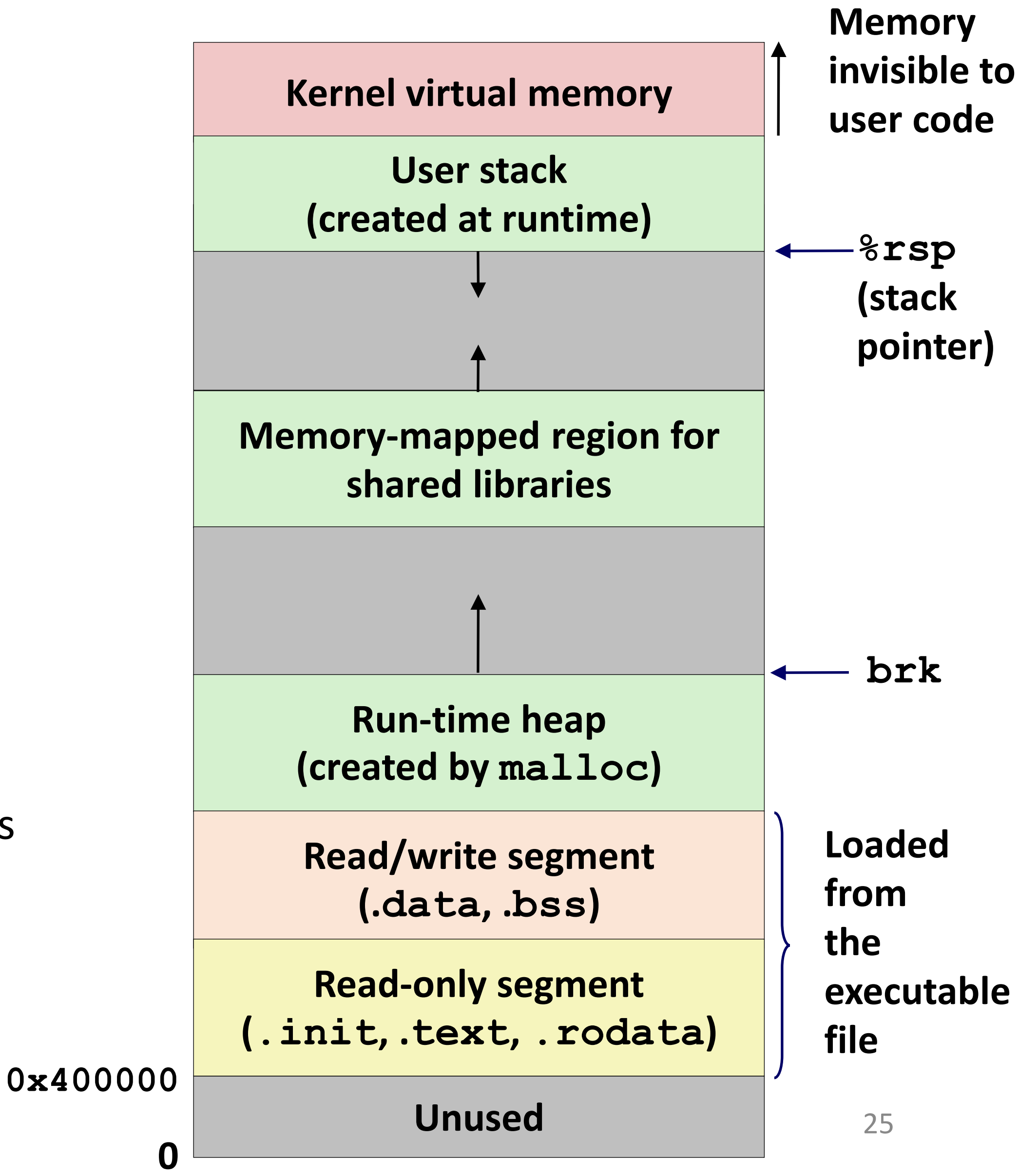
VM as a Tool for Memory Management

- 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)


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

Simplifying Linking and Loading

- Linking
  - Each program has similar virtual address space
  - Code, data, and heap always start at the same addresses.
- 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

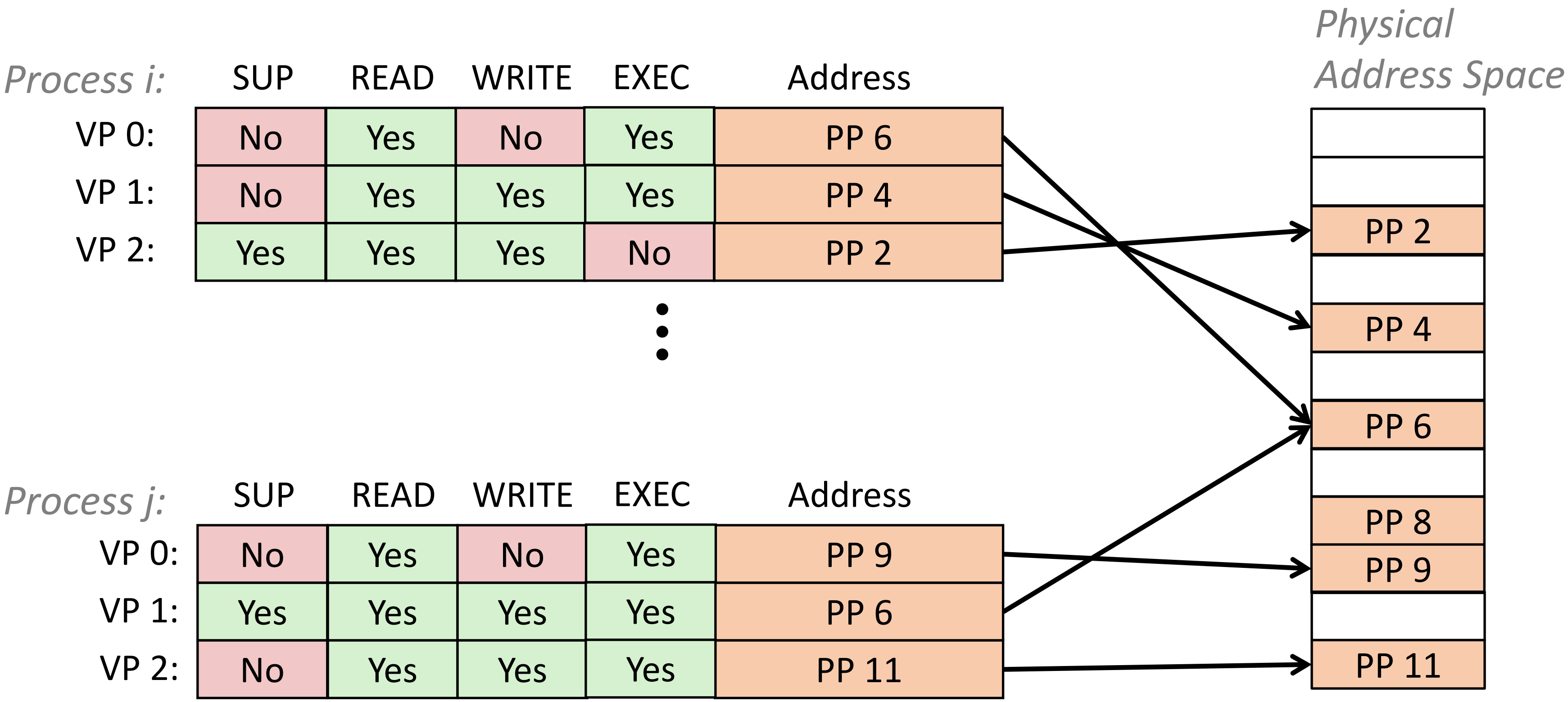

Kernel virtual memory
Memory invisible to user code
User stack (created at runtime)
%rsp (stack pointer)
Memory-mapped region for shared libraries
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Read-only segment (.init, .text, .rodata)
Loaded from the executable file
0x400000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

Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation

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

Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation

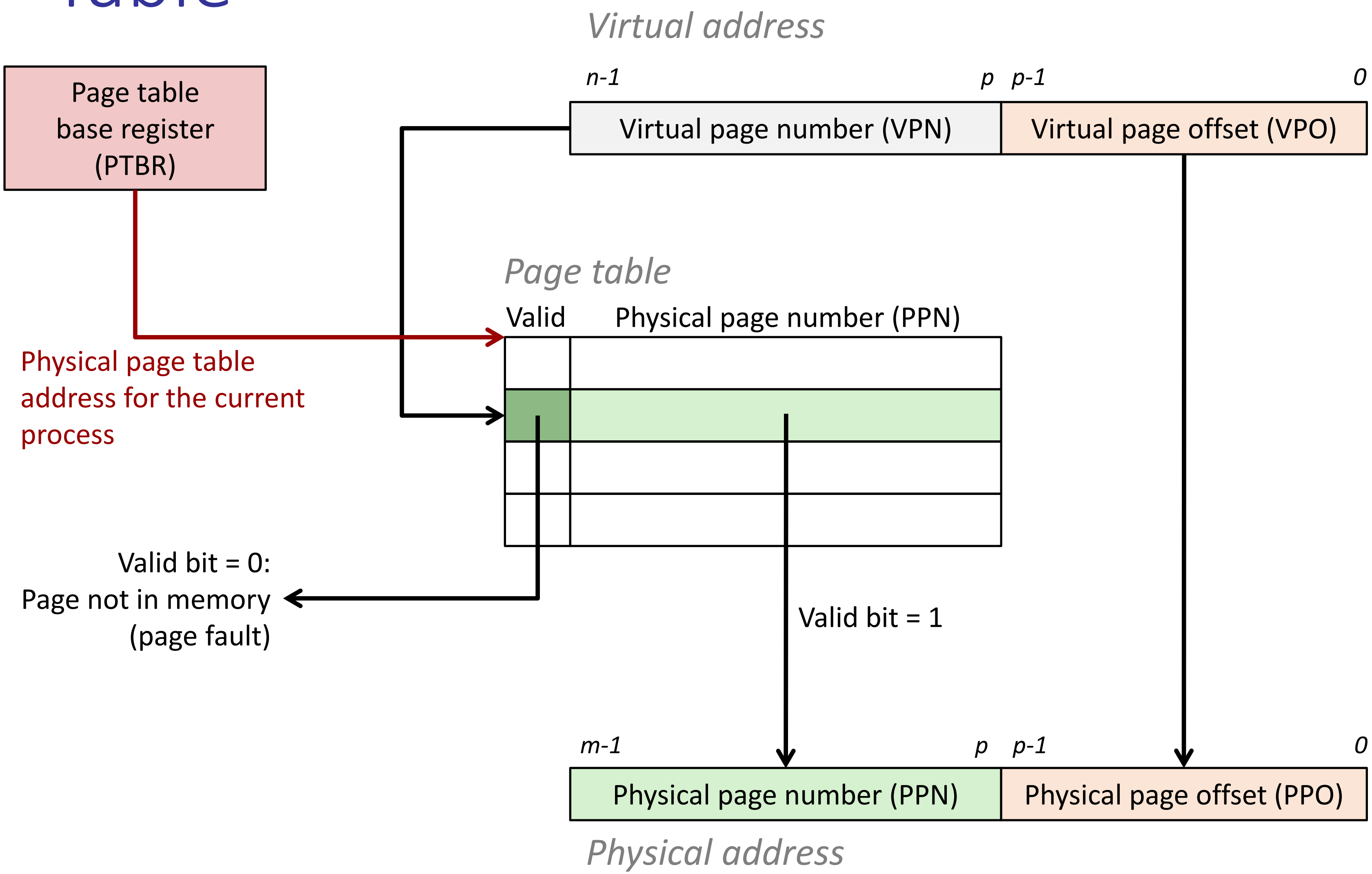
VM Address Translation

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

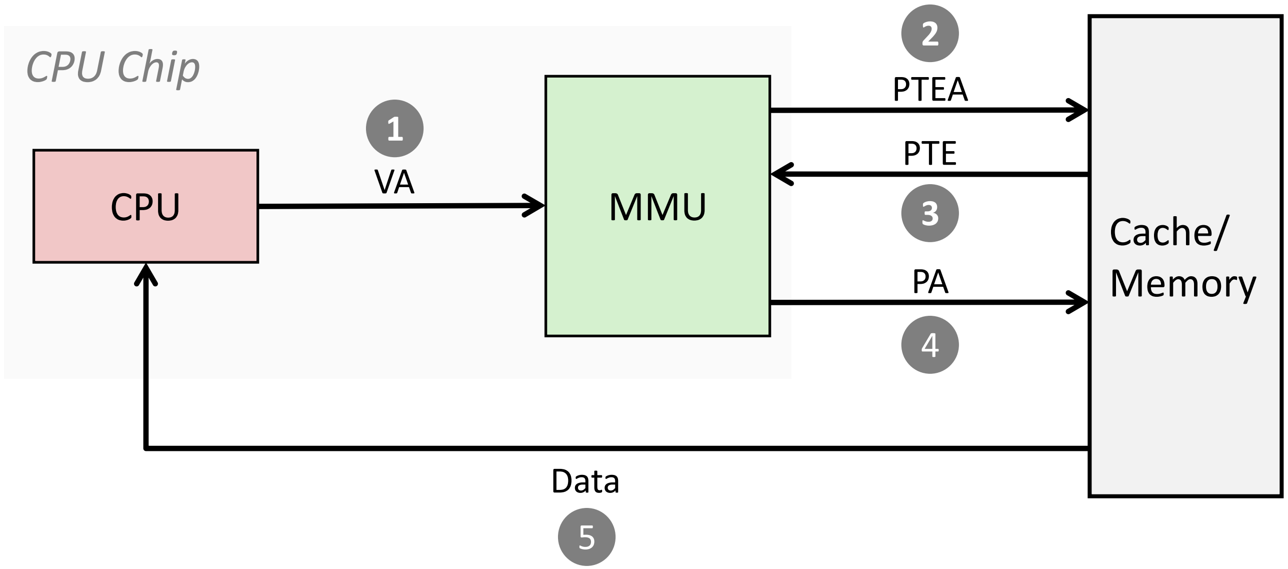
Summary of Address Translation Symbols

- 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)
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
- Components of the physical address (PA)
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number

Address Translation With a Page Table



Address Translation: Page Hit



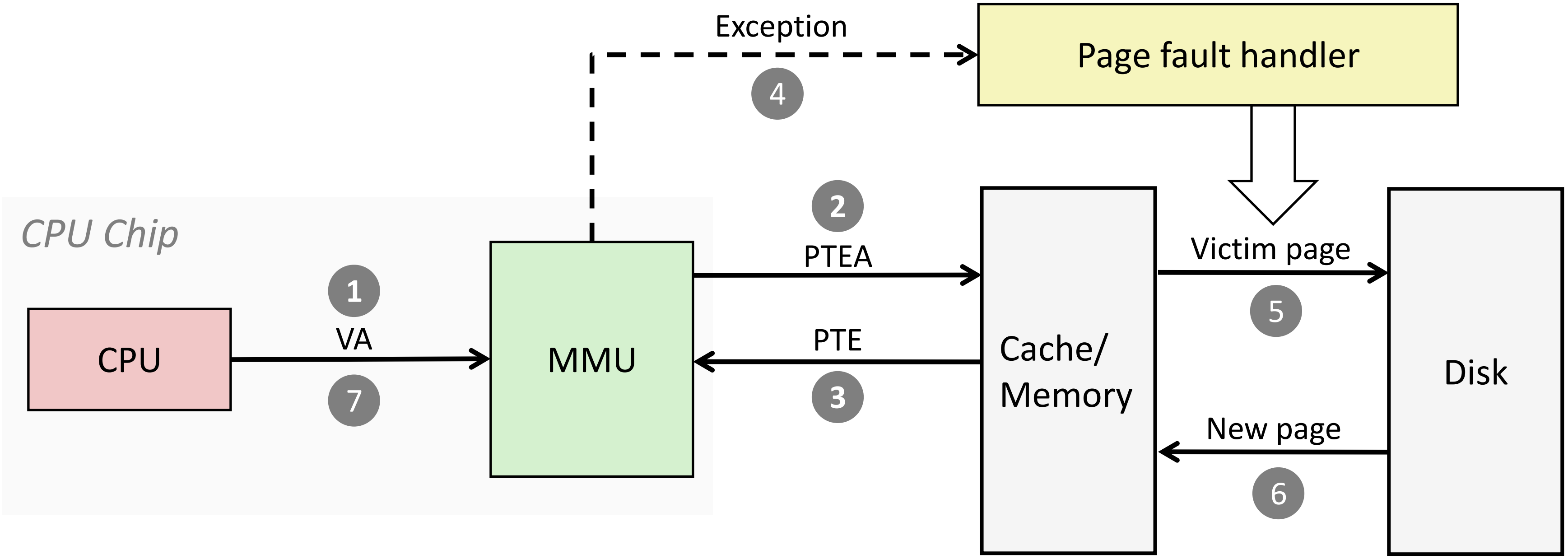
1) Processor sends virtual address to MMU

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

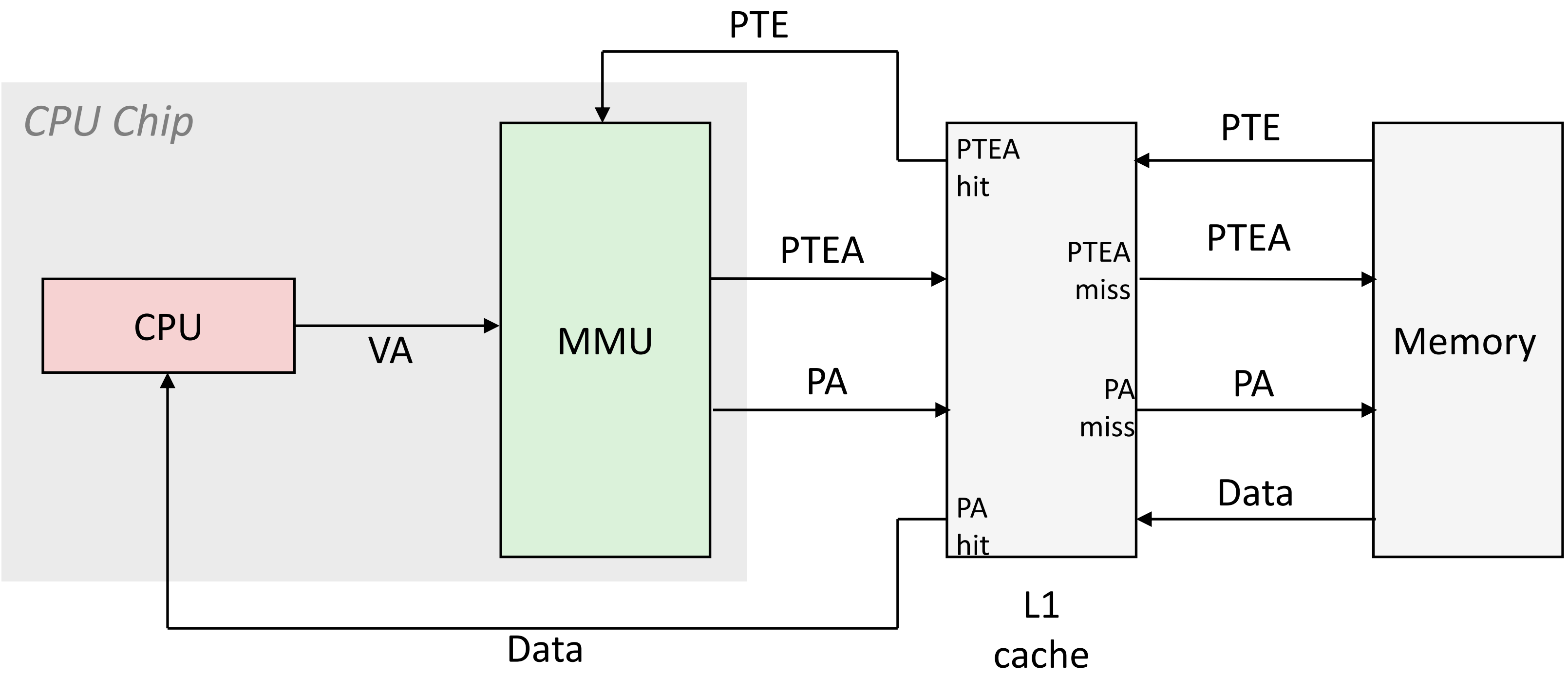
4) Valid bit is zero, so MMU triggers page fault exception

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

Integrating VM and Cache



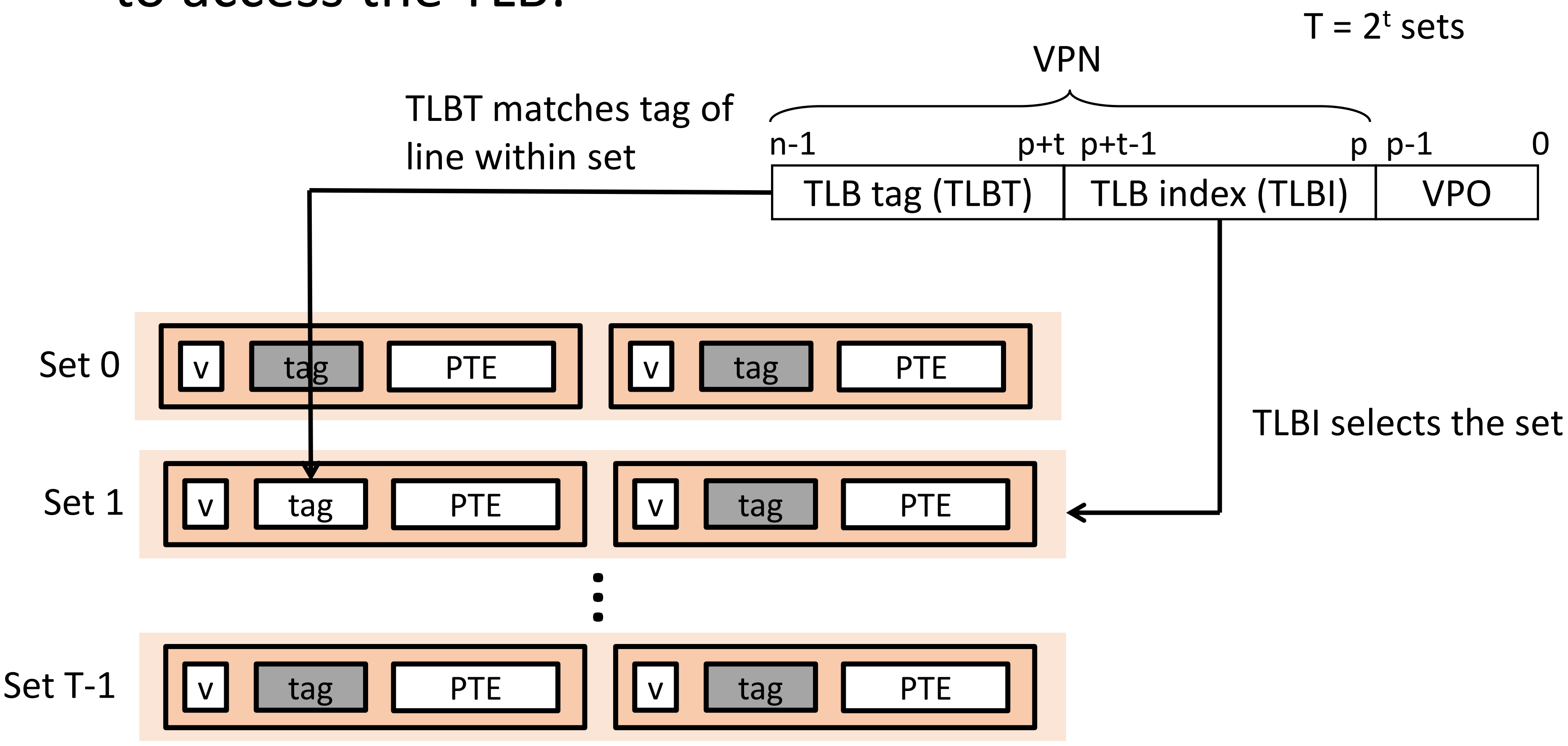
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Speeding up Translation with a TLB

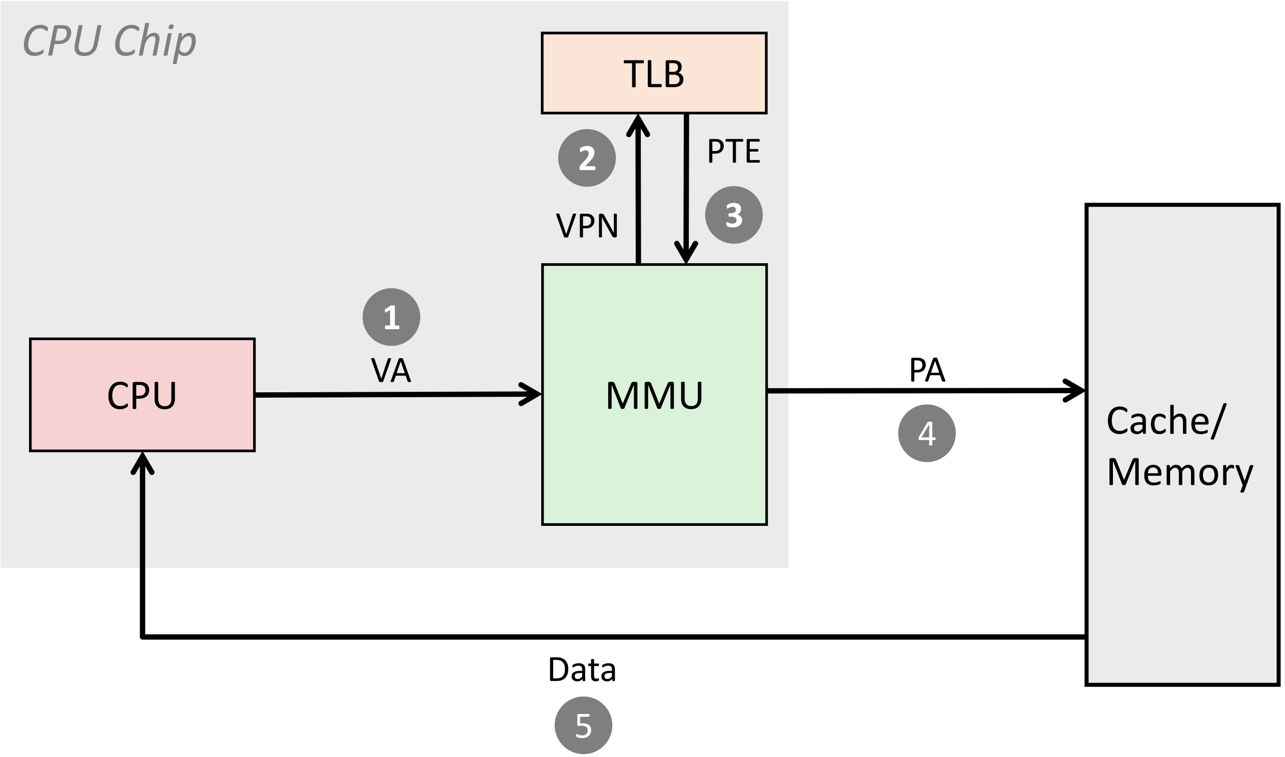
- Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay
- Solution: *Translation Lookaside Buffer* (TLB)
  - Small set-associative hardware cache in MMU
  - Maps virtual page numbers to physical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

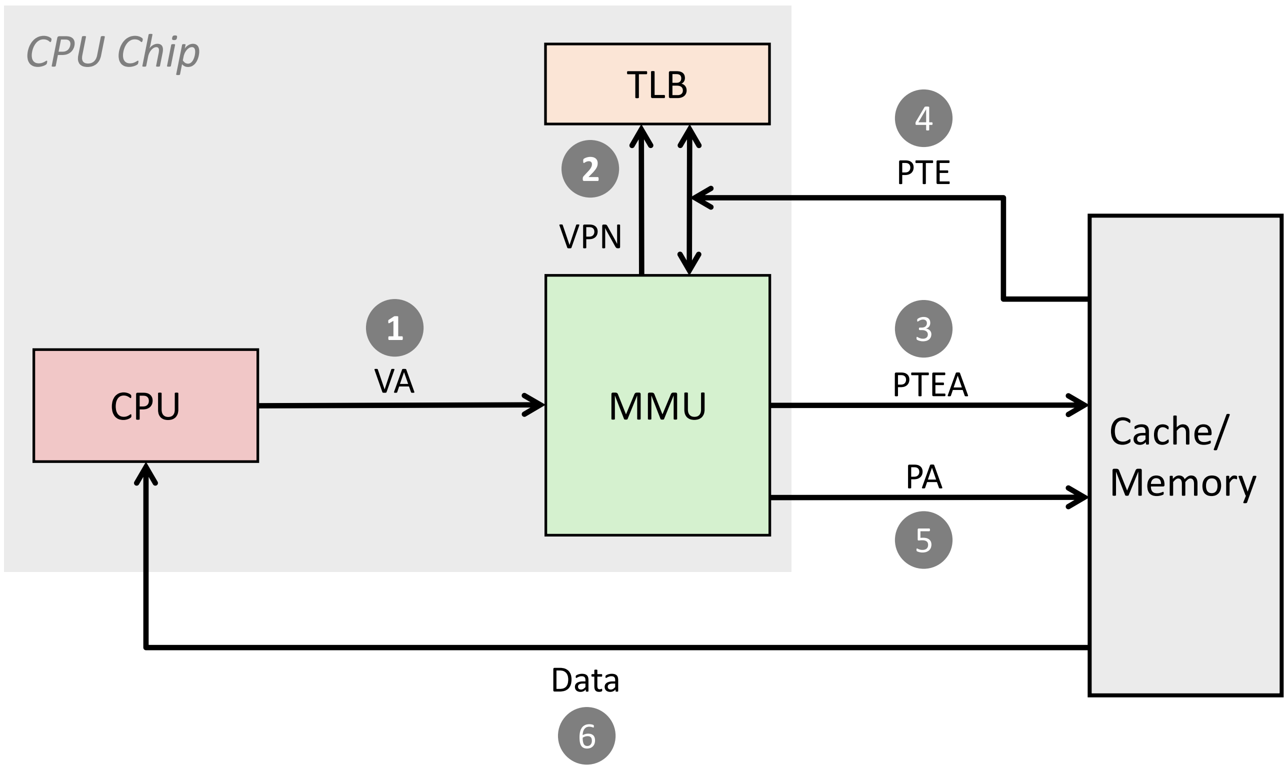


TLB Hit



A TLB hit eliminates a memory access

TLB Miss

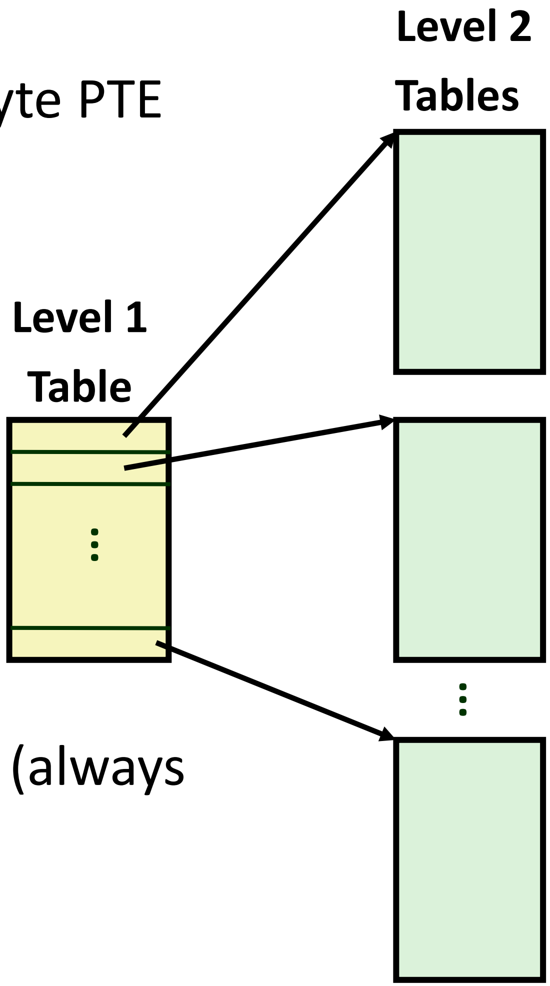


A TLB miss incurs an additional memory access (the PTE)

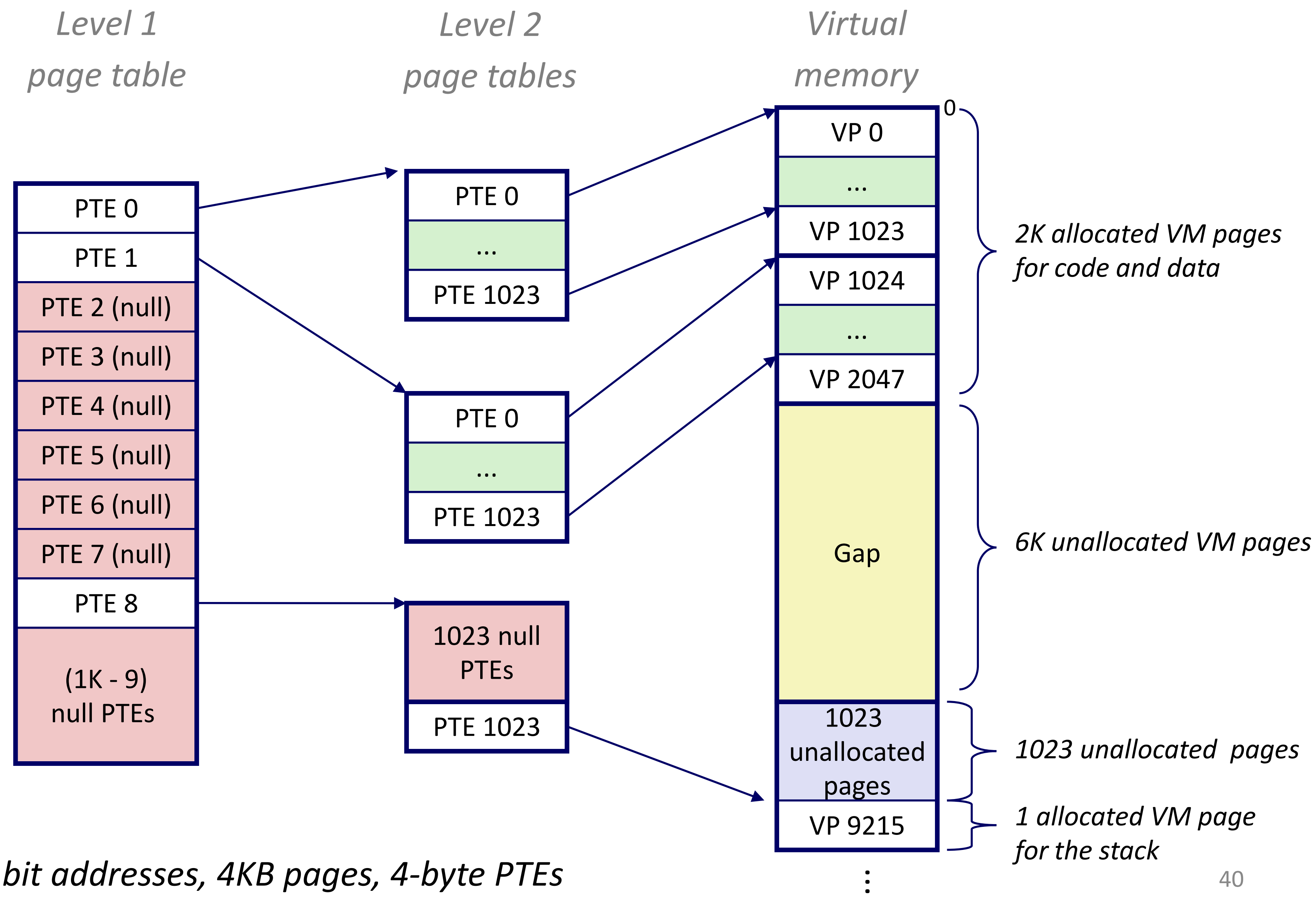
Fortunately, TLB misses are rare. Why?

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)

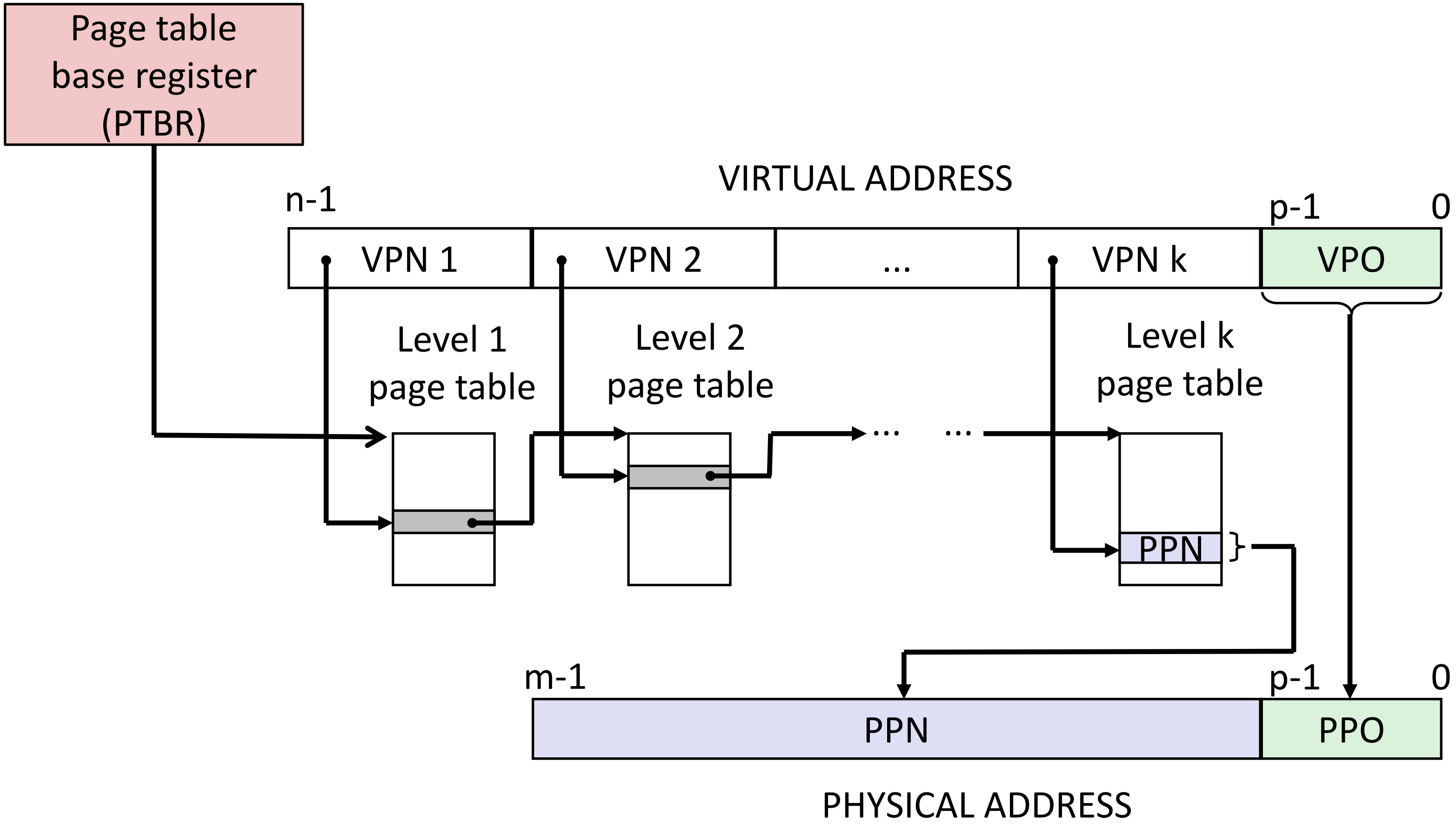


A Two-Level Page Table Hierarchy



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

Translating with a k-level Page Table



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

Virtual Memory: Systems

The content of this part is mainly from:
Randal E. Bryant and David R. O'Hallaron, “Computer Systems: A Programmer's Perspective,” 3/e.

(本節內容改自Prof. Randal E. Bryant and David R. O’Hallaron 18th Lectures課程講義)

Today

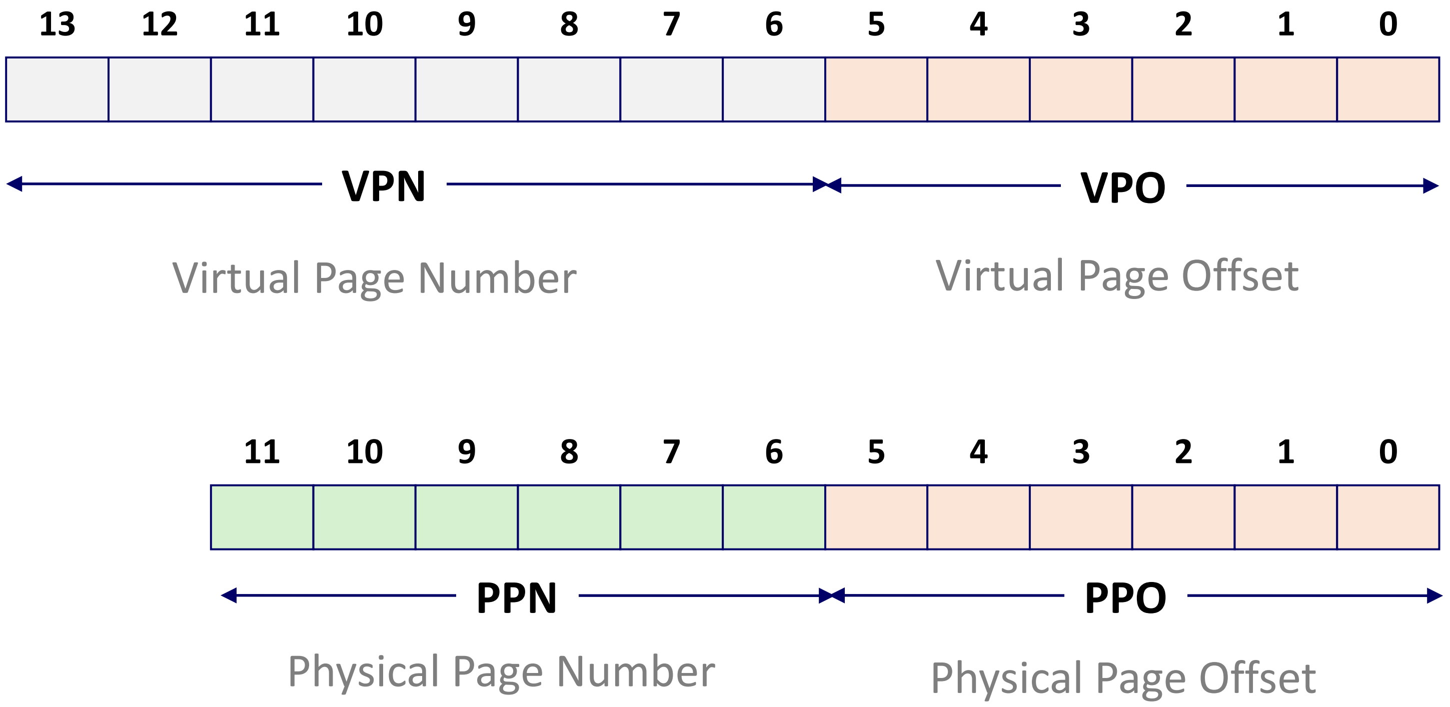
- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping

Review of 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)
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
- 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



Virtual Page Number

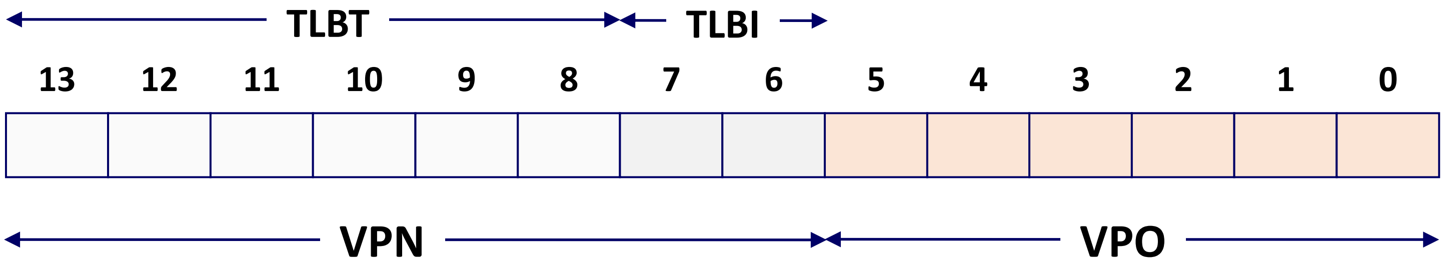
Virtual Page Offset

Physical Page Number

Physical Page Offset

1. Simple Memory System TLB

- 16 entries
- 4-way associative



TLBT

TLBI

13 12 11 10 9 8 7 6 5 4 3 2 1 0

VPN

VPO

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

2. Simple Memory System Page Table

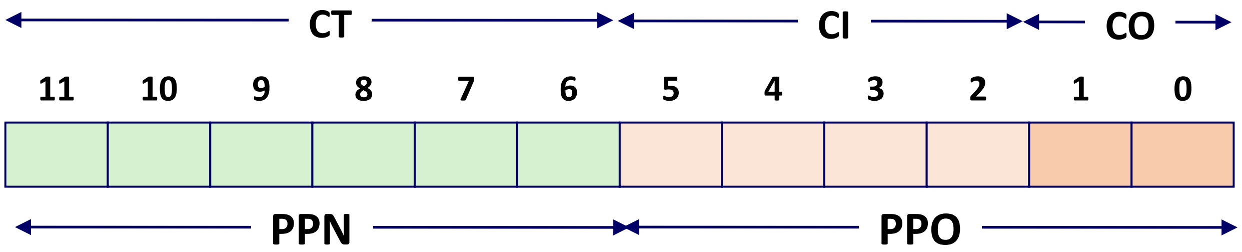
Only show first 16 entries (out of 256)

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

3. 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: 0x0020

TLBT | TLBI

13	12	11	10	9	8	7	6	5	4	3	2	1	0
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

CT | CI | CO

11	10	9	8	7	6	5	4	3	2	1	0
1	**0**	**1**	0	**0**	0	1	0	**0**	0	**0**	**0**

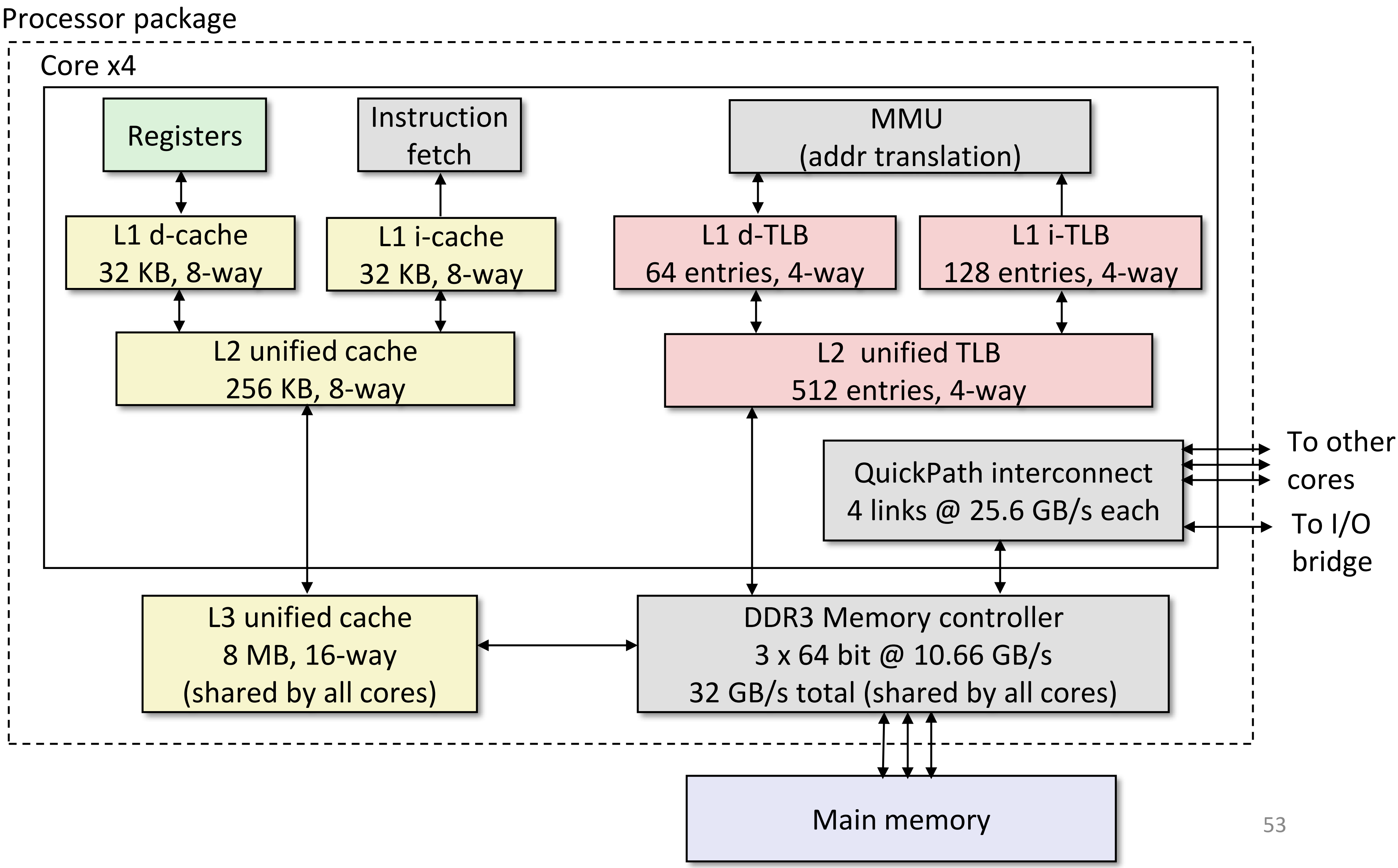
PPN | PPO

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

Today

- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping

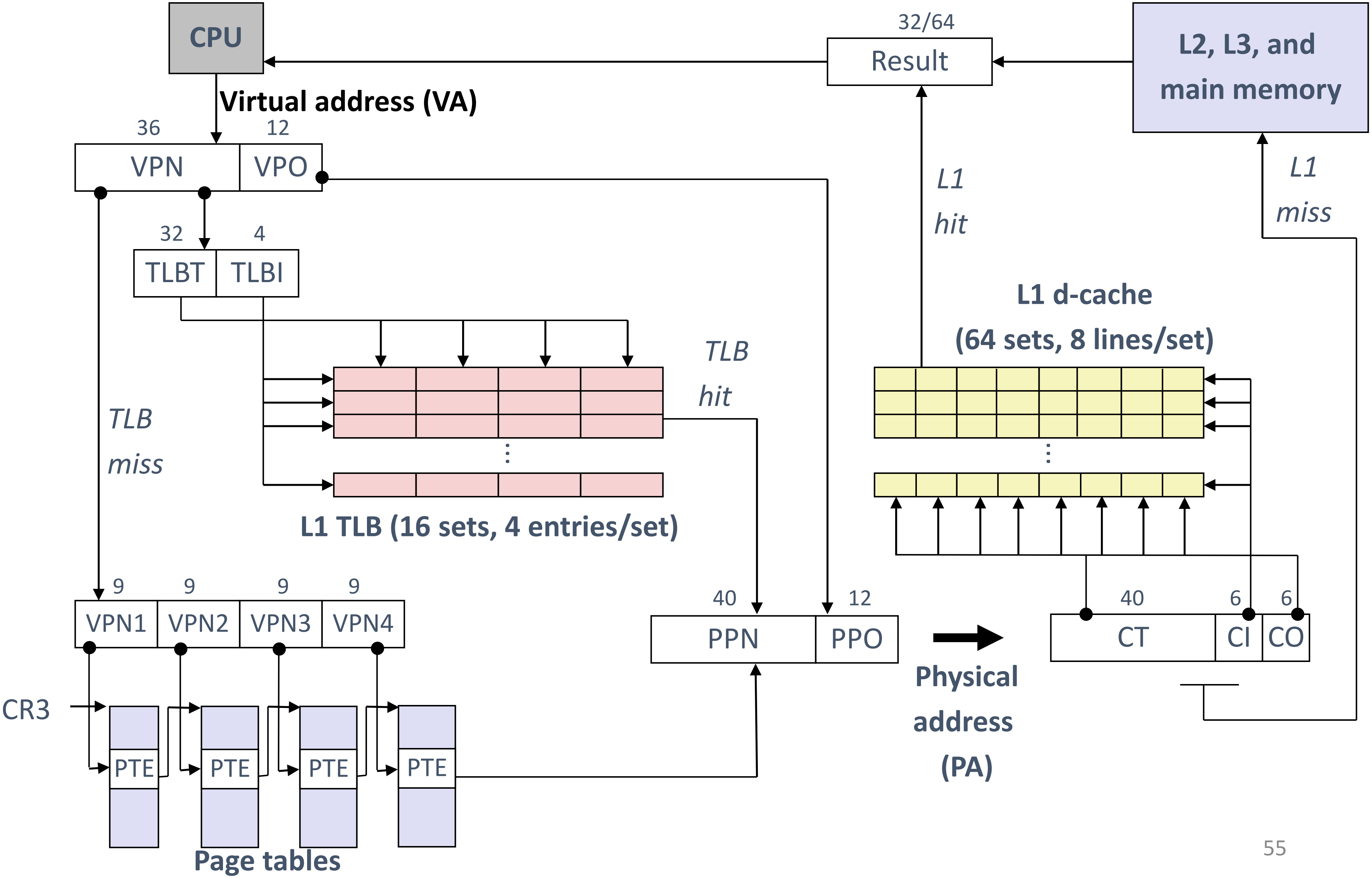
Intel Core i7 Memory System


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

Review of 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)
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
- 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

End-to-end Core i7 Address 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. Significant fields:

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.

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).

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

XD: Disable or enable instruction fetches from all pages reachable from this PTE.

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

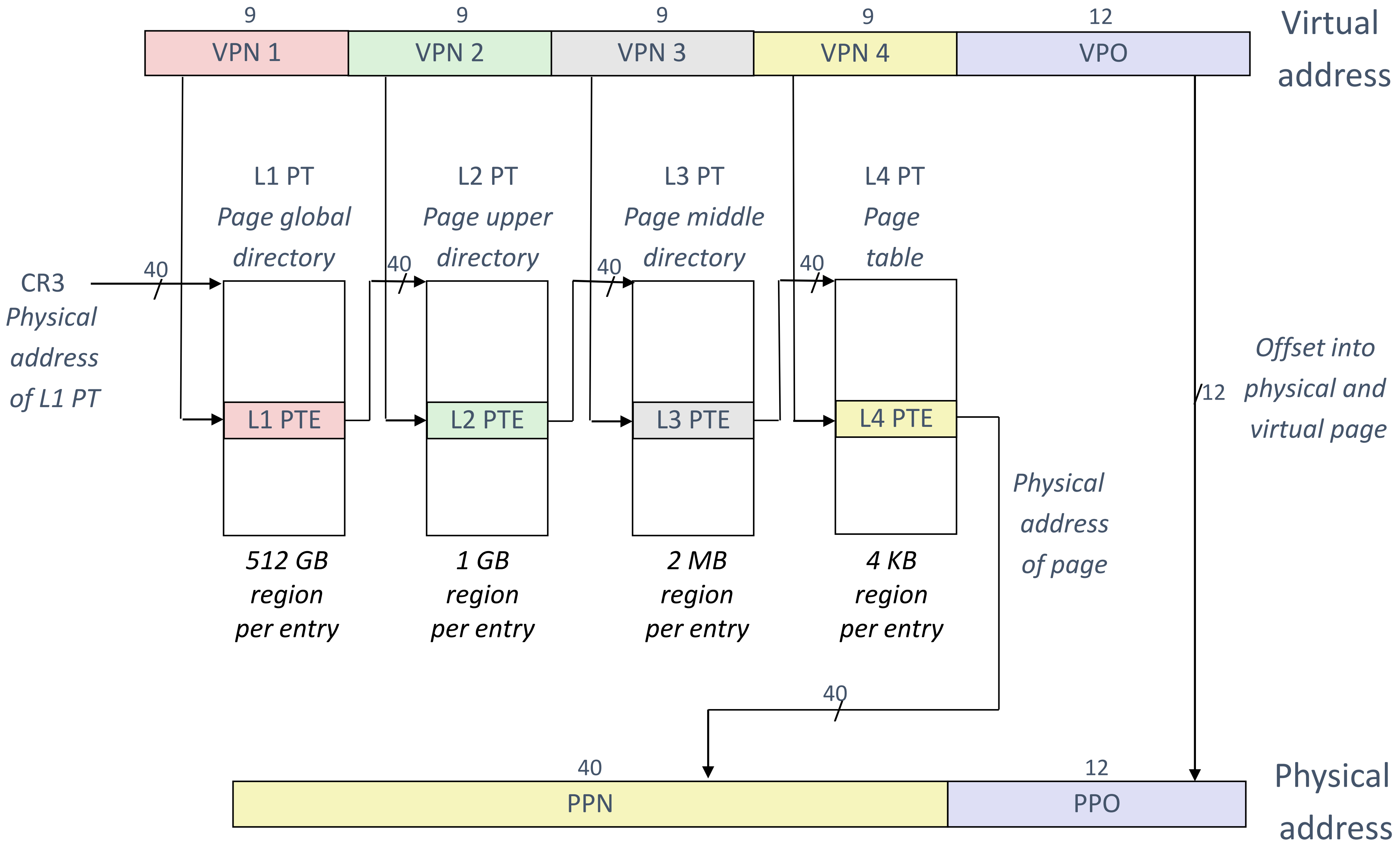
A: Reference bit (set by MMU on reads and writes, cleared by software)

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

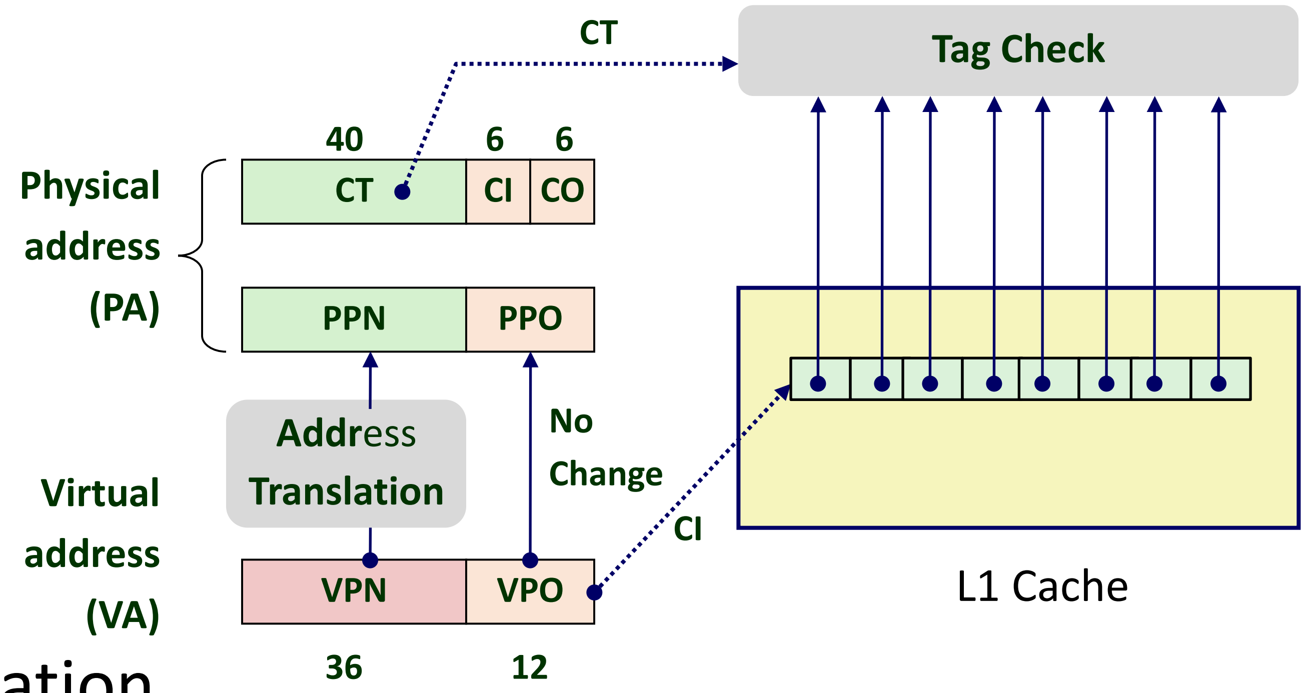
Page physical base address: 40 most significant bits of physical page address
(forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.

Core i7 Page Table Translation

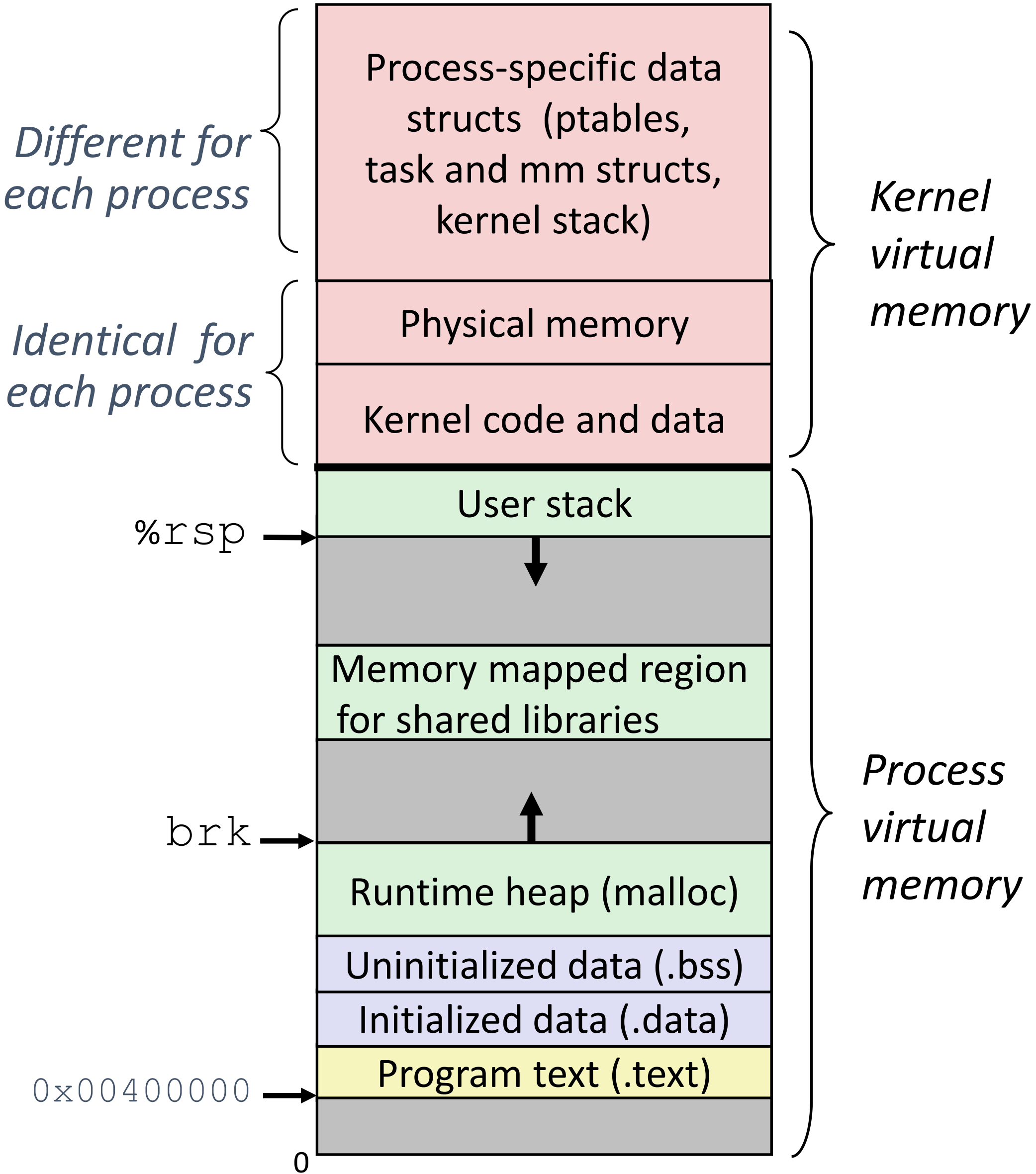


Cute Trick for Speeding Up L1 Access



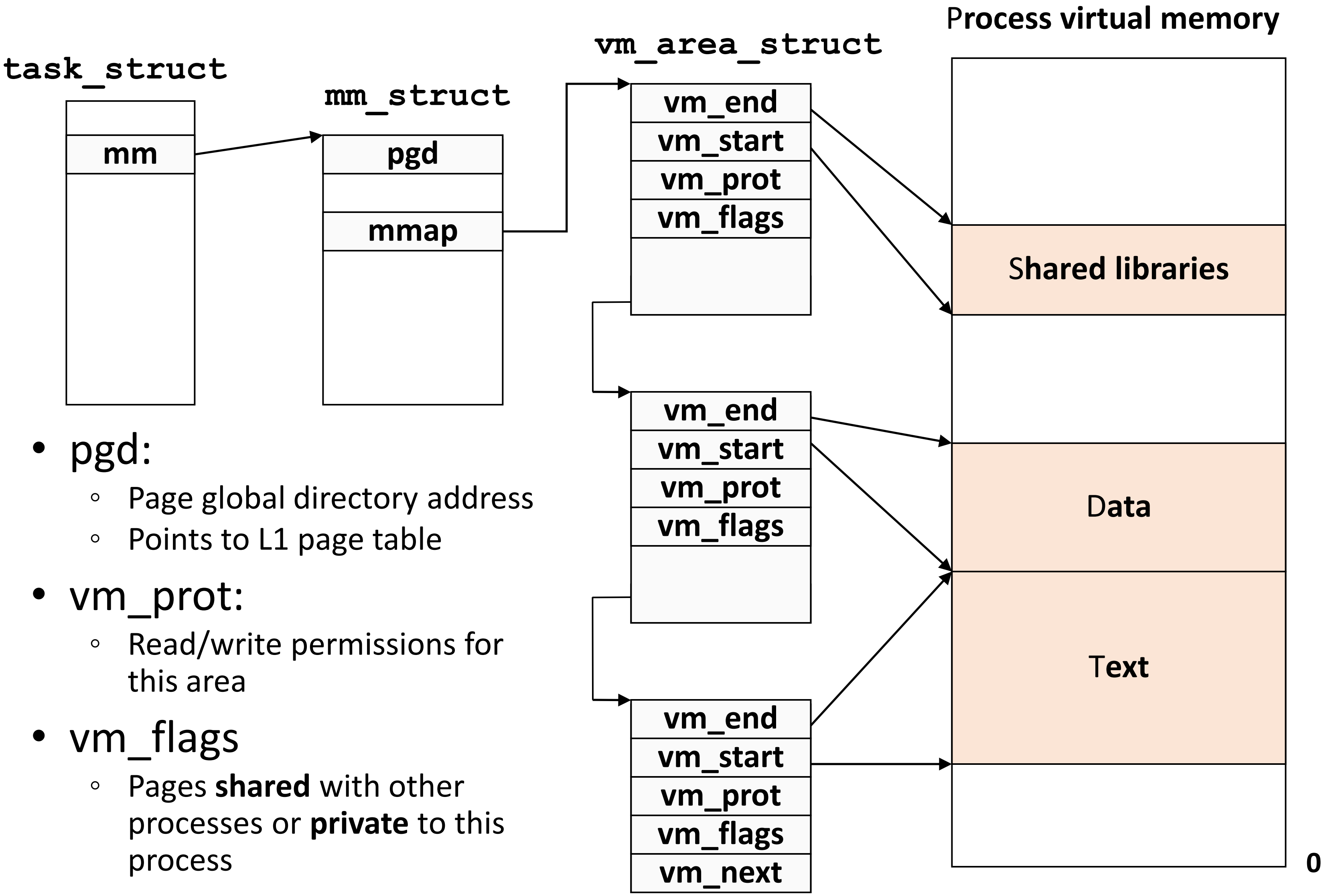
- Observation
  - Bits that determine CI identical in virtual and physical address
  - Can index into cache while address translation taking place
  - Generally we hit in TLB, so PPN bits (CT bits) available next
  - “Virtually indexed, physically tagged”
  - Cache carefully sized to make this possible

Virtual Address Space of a Linux Process



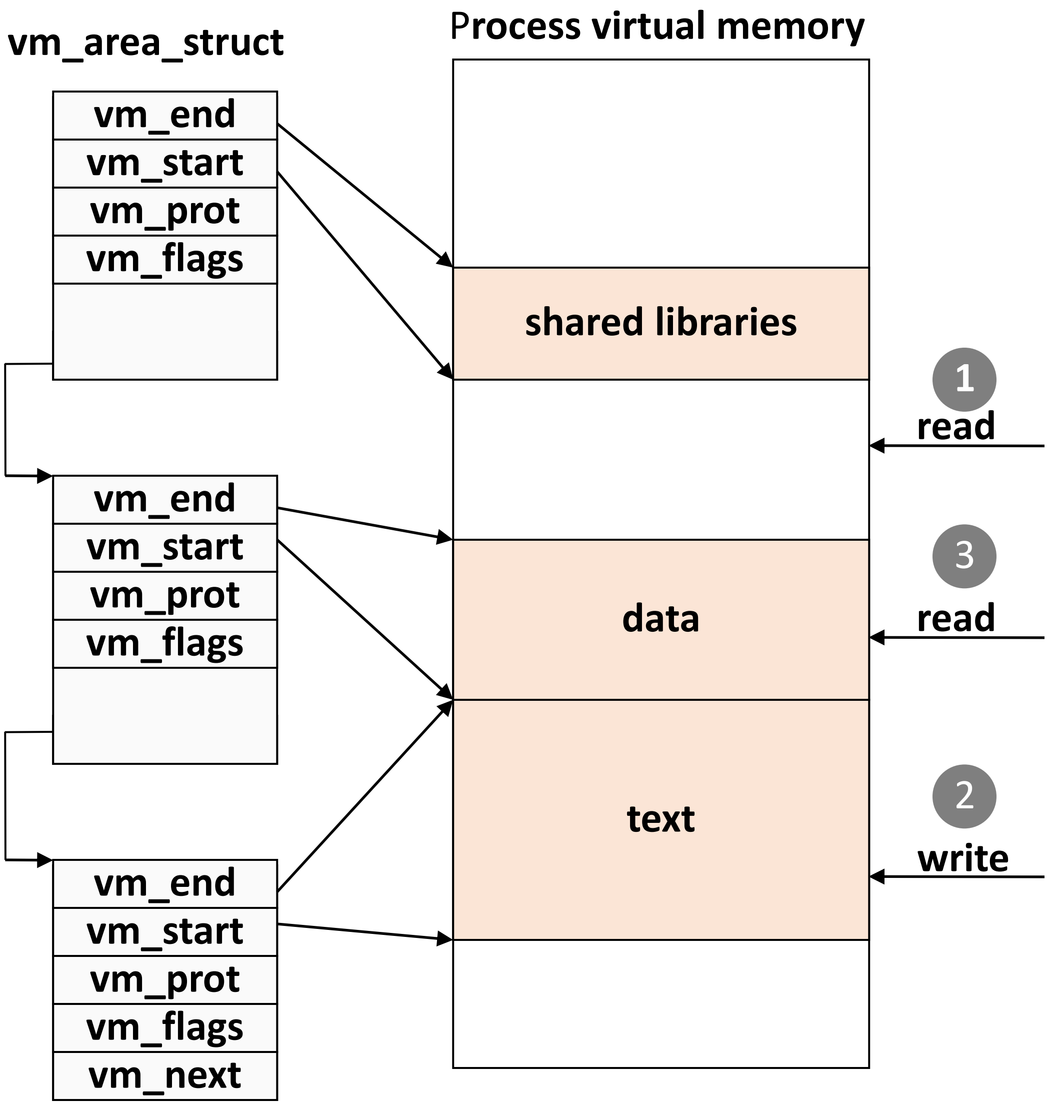
Different for each process
Process-specific data structs (ptables, task and mm structs, kernel stack)
Identical for each process
Physical memory
Kernel code and data
Kernel virtual memory
User stack
%rsp
Memory mapped region for shared libraries
brk
Runtime heap (malloc)
Uninitialized data (.bss)
Initialized data (.data)
Program text (.text)
0x00400000
0
Process virtual memory

Linux Organizes VM as Collection of “Areas”



- pgd:
  - Page global directory address
  - Points to L1 page table
- vm_prot:
  - Read/write permissions for this area
- vm_flags
  - Pages **shared** with other processes or **private** to this process

Linux Page Fault Handling



Segmentation fault:
accessing a non-existing page

Normal page fault

Protection exception:
e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)

Additional Information on Cache Memory and TLB

(本節內容改自 B. Jacob et al, "Memory Systems- Cache, DRAM and Disk", Morgan Kaufmann 2008.)

Different Hardware Caches

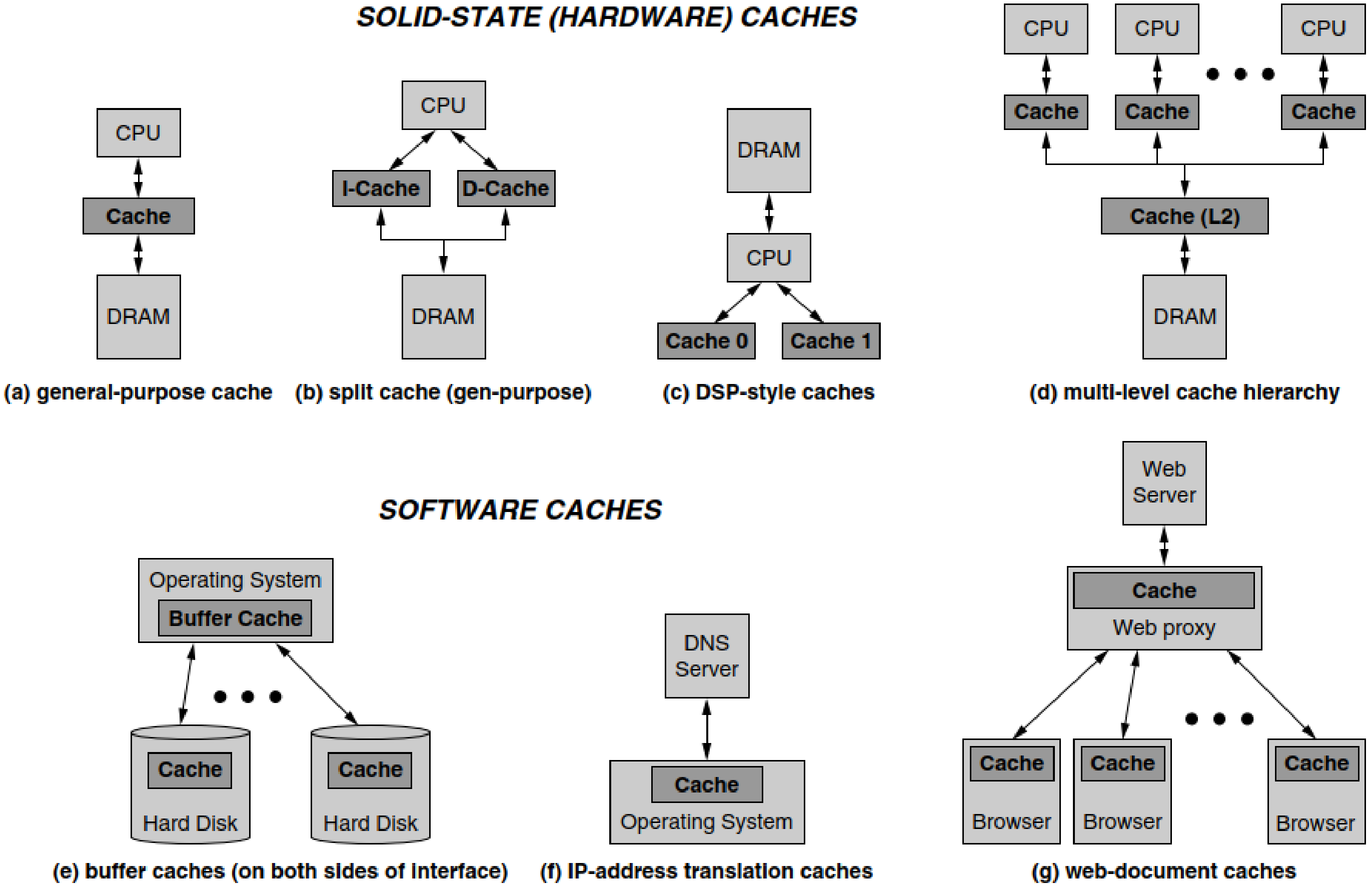


FIGURE 1.1: Examples of caches. The caches are divided into two main groups: solid-state caches (top), and those that are implemented by software mechanisms, typically storing the cached data in main memory (e.g., DRAM) or disk.

Classificatior of Hardware Caches

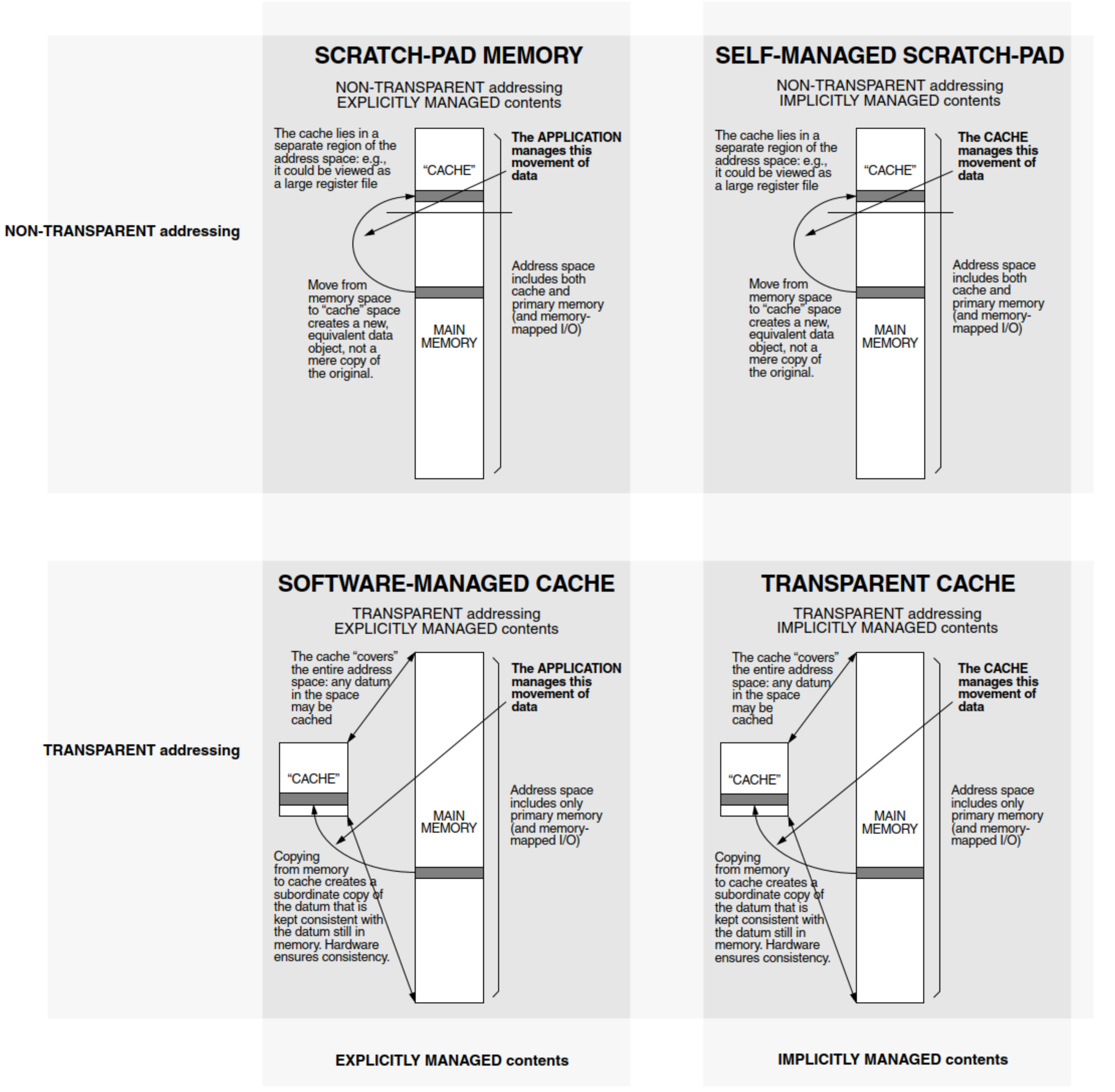

SCRATCH-PAD MEMORY
NON-TRANSPARENT addressing
EXPLICITLY MANAGED contents
The cache lies in a separate region of the address space: e.g., it could be viewed as a large register file
"CACHE"
The APPLICATION manages this movement of data
Move from memory space to "cache" space creates a new, equivalent data object, not a mere copy of the original.
MAIN MEMORY
Address space includes both cache and primary memory (and memory-mapped I/O)
SELF-MANAGED SCRATCH-PAD
NON-TRANSPARENT addressing
IMPLICITLY MANAGED contents
The cache lies in a separate region of the address space: e.g., it could be viewed as a large register file
"CACHE"
The CACHE manages this movement of data
Move from memory space to "cache" space creates a new, equivalent data object, not a mere copy of the original.
MAIN MEMORY
Address space includes both cache and primary memory (and memory-mapped I/O)
NON-TRANSPARENT addressing
TRANSPARENT addressing
SOFTWARE-MANAGED CACHE
TRANSPARENT addressing
EXPLICITLY MANAGED contents
The cache "covers" the entire address space: any datum in the space may be cached
The APPLICATION manages this movement of data
"CACHE"
MAIN MEMORY
Address space includes only primary memory (and memory-mapped I/O)
Copying from memory to cache creates a subordinate copy of the datum that is kept consistent with the datum still in memory. Hardware ensures consistency.
TRANSPARENT CACHE
TRANSPARENT addressing
IMPLICITLY MANAGED contents
The cache "covers" the entire address space: any datum in the space may be cached
The CACHE manages this movement of data
"CACHE"
MAIN MEMORY
Address space includes only primary memory (and memory-mapped I/O)
Copying from memory to cache creates a subordinate copy of the datum that is kept consistent with the datum still in memory. Hardware ensures consistency.
EXPLICITLY MANAGED contents
IMPLICITLY MANAGED contents

FIGURE 2.1: Examples of several different cache organizations. Though this illustrates "cache" in the context of solid-state memories (e.g., SRAM cache and DRAM main memory), the concepts are applicable to all manners of storage technologies, including SRAM, DRAM, disk, and even tertiary (backup) storage such as optical disk and magnetic tape.

Building Direct-mapped Cache

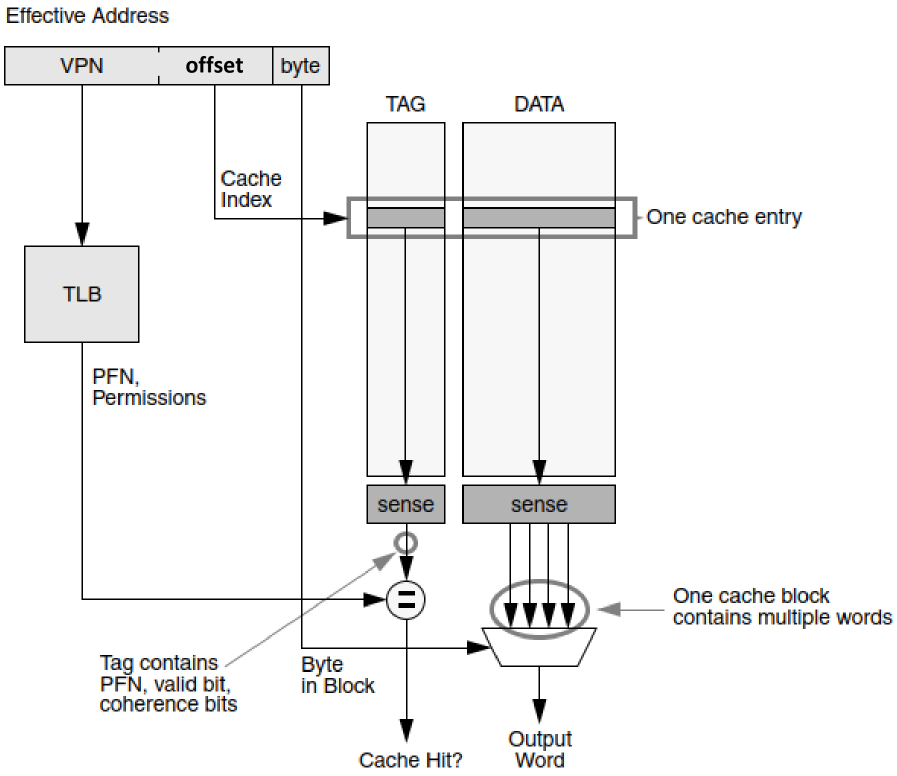


FIGURE 2.3: Block diagram for a direct-mapped cache. Note that data read-out is controlled by tag comparison with the TLB output as well as the block-valid bit (part of the tag or metadata entry) and the page permissions (part of the TLB entry). Data is not read from the cache if the valid bit is not set or if the permissions indicate an invalid access (e.g., writing a read-only block). Note also that the cache size is equal to the virtual memory page size (the cache index does not use bits from the VPN).

Building Fully-Associative Cache with Content-Addressable Memory (CAM)

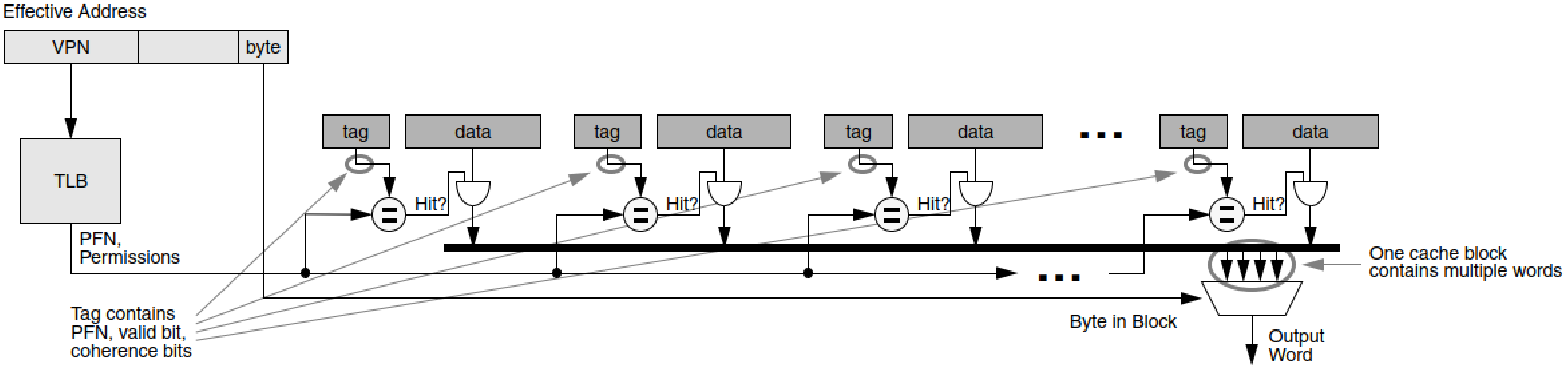
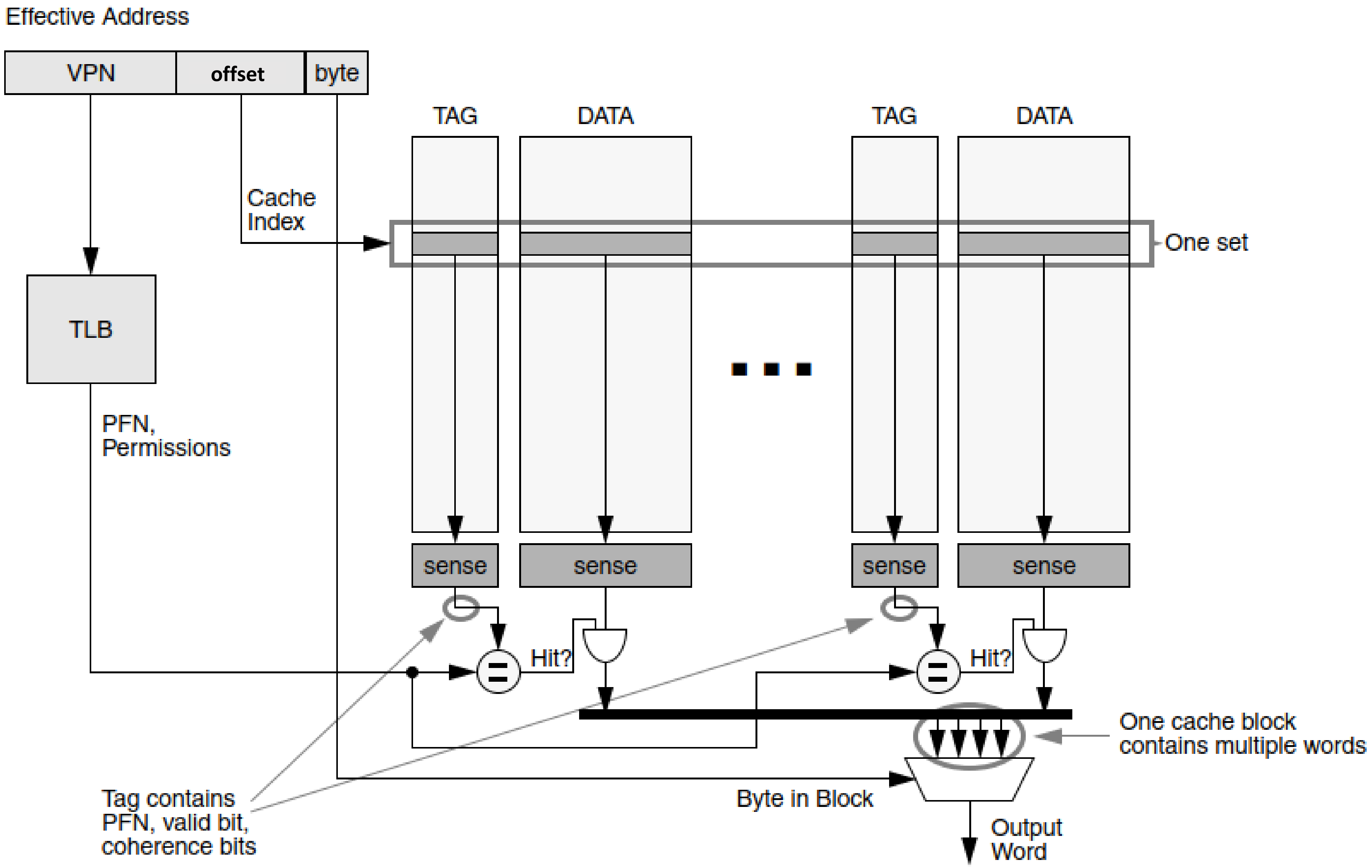


FIGURE 2.4: Fully associative lookup mechanism. This organization is also called a CAM, for content-addressable memory. It is similar to a database in that any entry that has the same tag as the lookup address matches, no matter where it is in the cache. This organization reduces cache contention, but the lookup can be expensive, since the tag of every entry is matched against the lookup address.

Building Set-associative Cache



Building Set-associative Cache with Content-Addressable Memory (CAM)

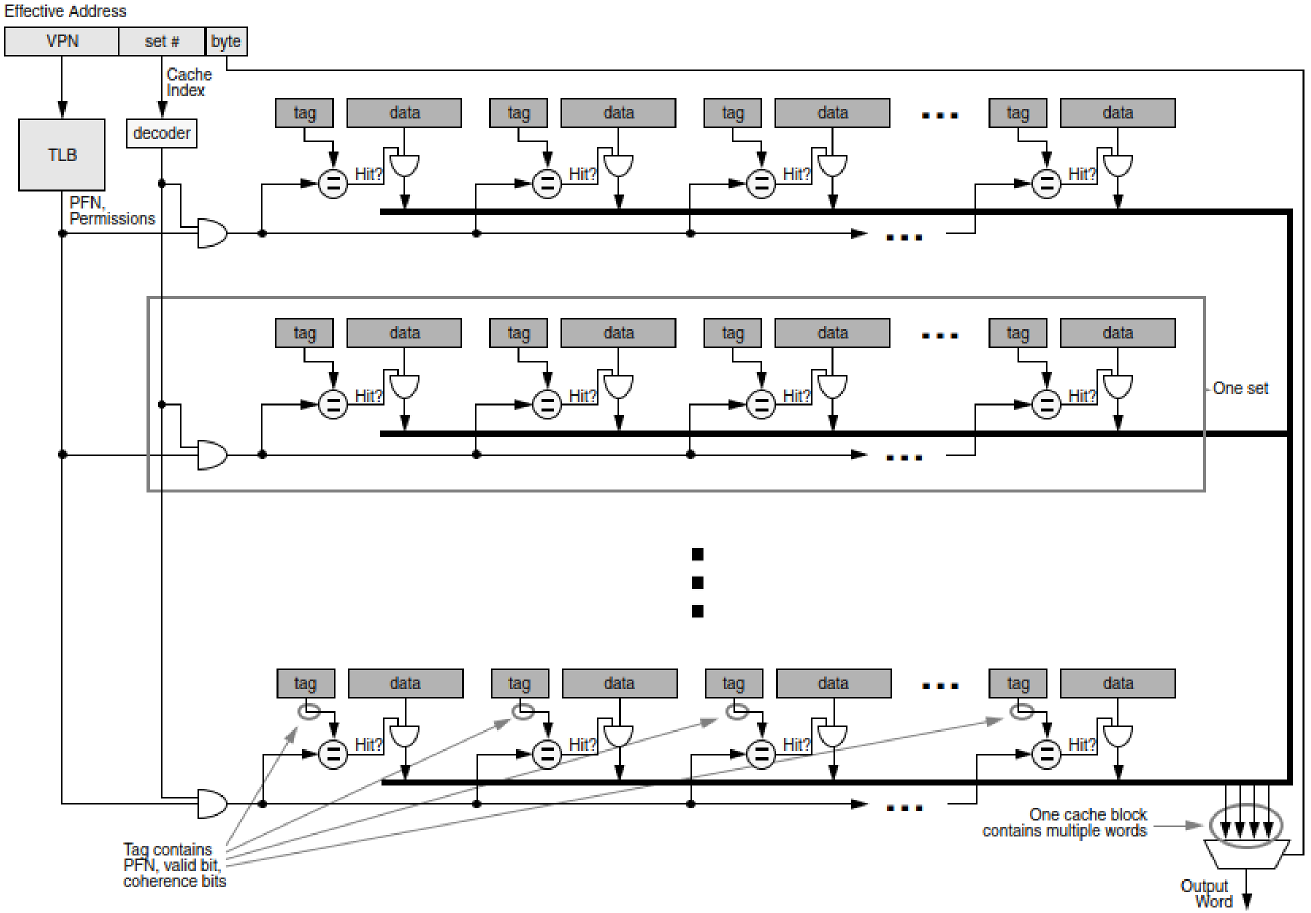


FIGURE 2.6: Set-associative cache built from CAMs. The previous figure showed a set-associative cache built of several direct-mapped caches, which is appropriate when the degree of associativity is low. When the degree of associativity is high, it is more appropriate to build the cache out of CAMs (content-addressable memories, i.e., fully associative caches), where each CAM represents one data set or equivalence class. For example, the Strong ARM's 32-way set-associative cache is built this way.

Realizing TLB CAM with Array Structure

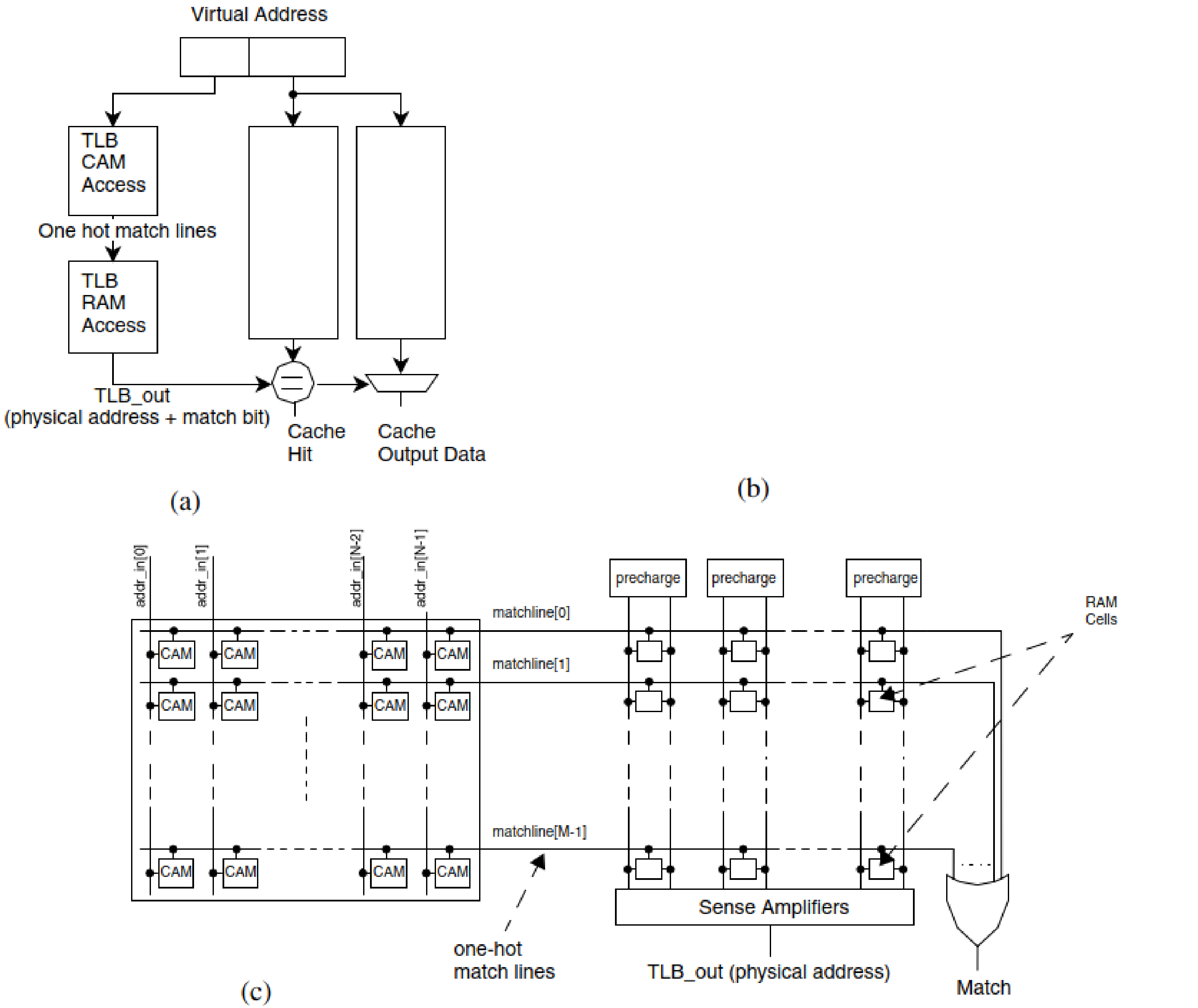


FIGURE 6.12: A conventional TLB and cache read access. (a) Shows a block diagram of the cache access, including the TLB access (both the CAM and RAM parts). (b) Shows internals of a CAM cell showing the inputs being compared to the internal state of the CAM cell and discharging a "match line" during a match. (c) Shows internals of the TLB CAM and RAM access. The left part shows the N-bit address input being matched to an M-entry CAM producing M-bit one-hot match signals which are then used as the wordlines of a conventional RAM array, which produces the actual data and a match signal signifying a TLB hit.

Putting Everything Together

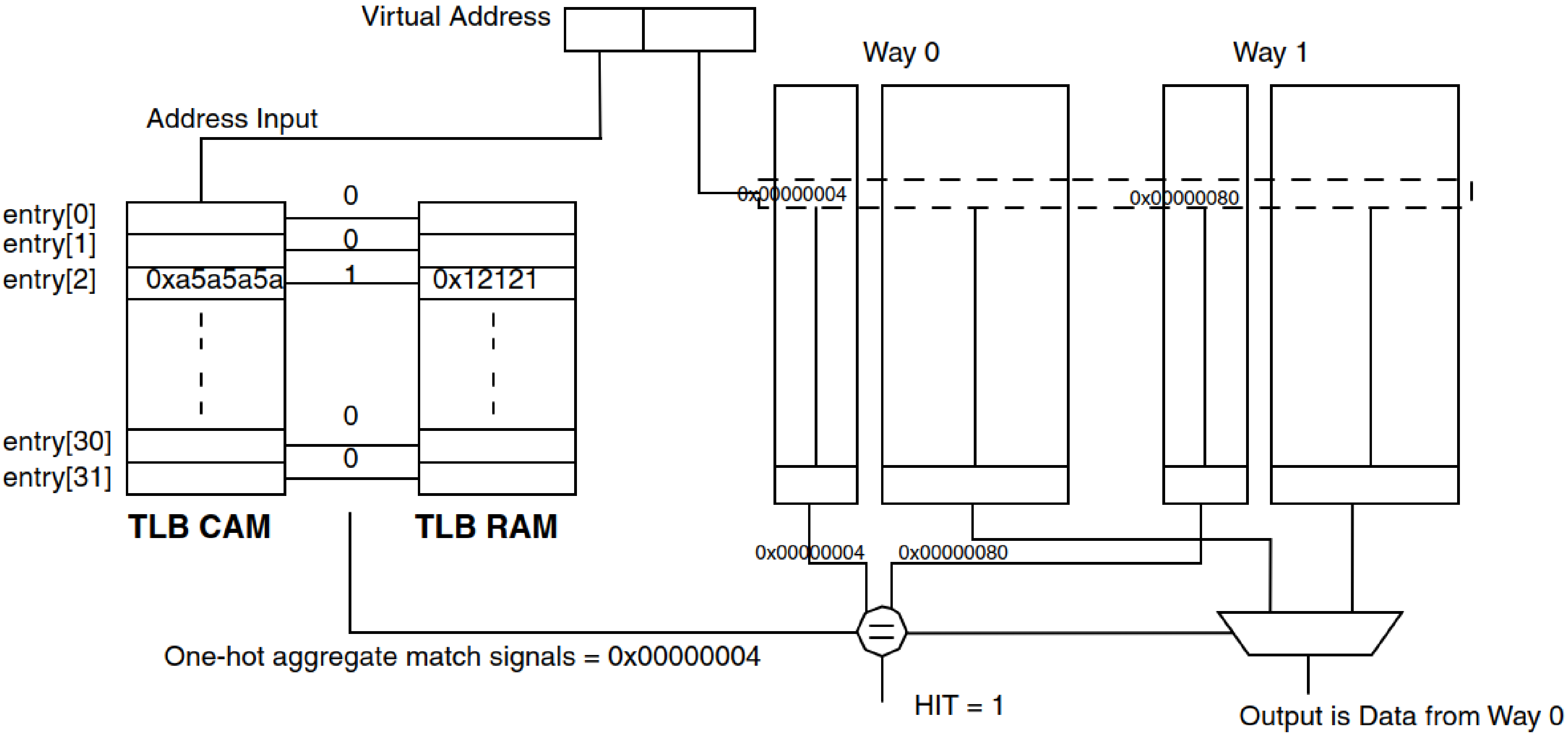
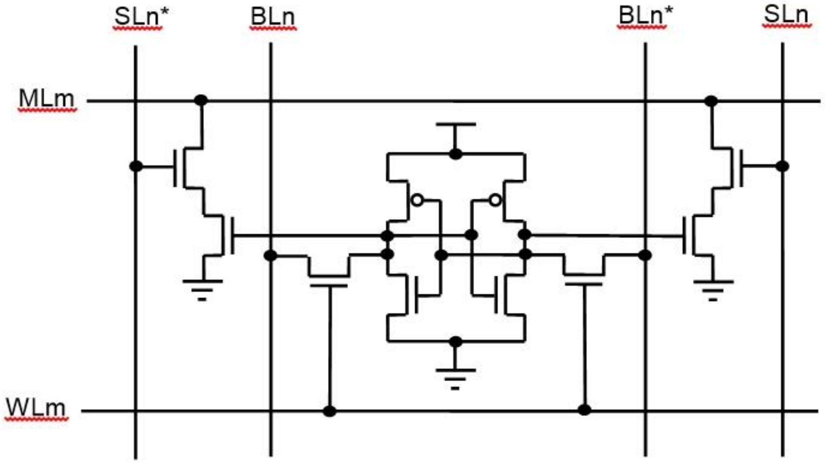


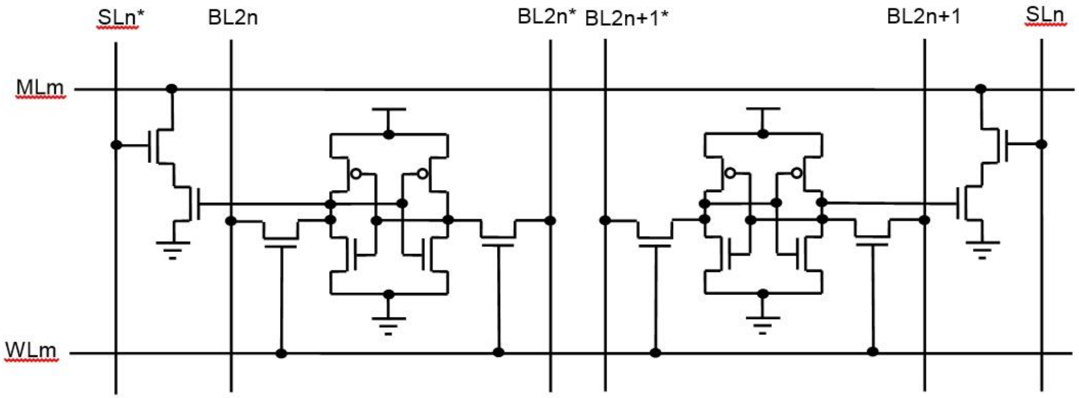
FIGURE 6.13: An Intel Itanium-2 prevalidated-tag microarchitecture (way-0 match). The cache tag arrays store a one-hot pointer to the TLB entry instead of the entire physical tag (here, Way 0 points to TLB entry 2, while Way 1 points to TLB entry 7). The tag comparison can be performed right after the access to the TLB CAM cells when the match lines are produced, instead of waiting for the actual address output from the TLB RAM. This example shows the prevalidated-tag microarchitecture for two ways, but the concept is readily applicable to caches with higher associativity.

Basic CMOS CAM Cell

Binary CAM



Ternary CAM (TCAM)



[Picture Source] https://en.wikipedia.org/wiki/Content-addressable_memory

A Modern Processor Architecture

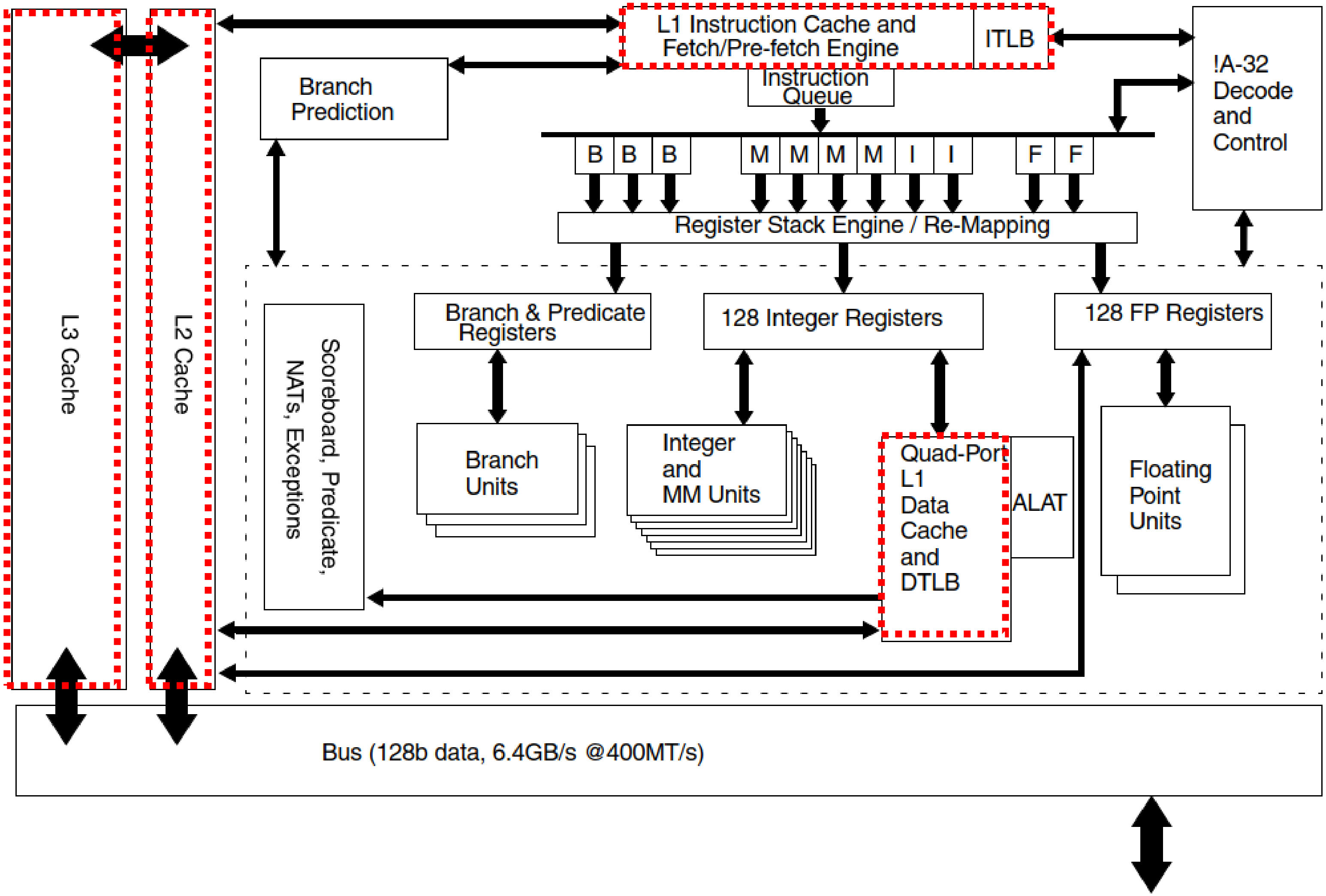

L1 Instruction Cache and Fetch/Pre-fetch Engine
ITLB
Branch Prediction
Instruction Queue
!A-32 Decode and Control
B
B
B
M
M
M
M
I
I
F
F
Register Stack Engine / Re-Mapping
L3 Cache
L2 Cache
Scoreboard, Predicate, NATs, Exceptions
Branch & Predicate Registers
128 Integer Registers
128 FP Registers
Branch Units
Integer and MM Units
Quad-Port L1 Data Cache and DTLB
ALAT
Floating Point Units
Bus (128b data, 6.4GB/s @400MT/s)

FIGURE 6.5: The Intel Itanium-2 processor block diagram.