

Virtual Memory

Fall 2012

Instructors:

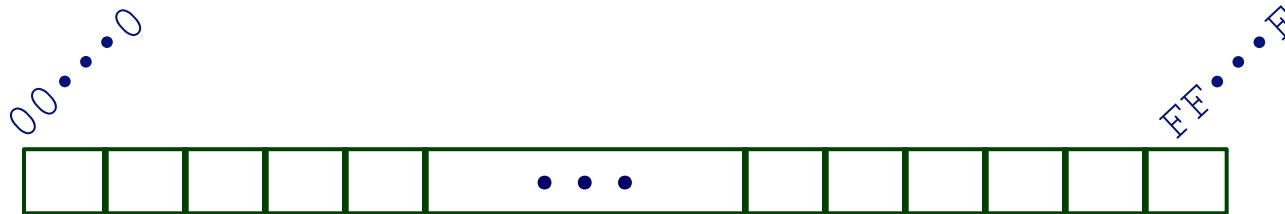
Aykut and Erkut Erdem

Acknowledgement: The course slides are adapted from the slides prepared by R.E. Bryant, D.R. O'Hallaron, G. Kesden and Markus Püschel of Carnegie-Mellon Univ.

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

Recall: Byte-Oriented Memory Organization



- **Programs refer to data by address**
 - Conceptually, envision it as a very large array of bytes
 - In reality, it's not, but can think of it that way
 - An address is like an index into that array
 - and, a pointer variable stores an address
- **Note: system provides private address spaces to each “process”**
 - Think of a process as a program being executed
 - So, a program can clobber its own data, but not that of others

Recall: Simple Addressing Modes

■ Normal (R) Mem[Reg[R]]

- Register R specifies memory address

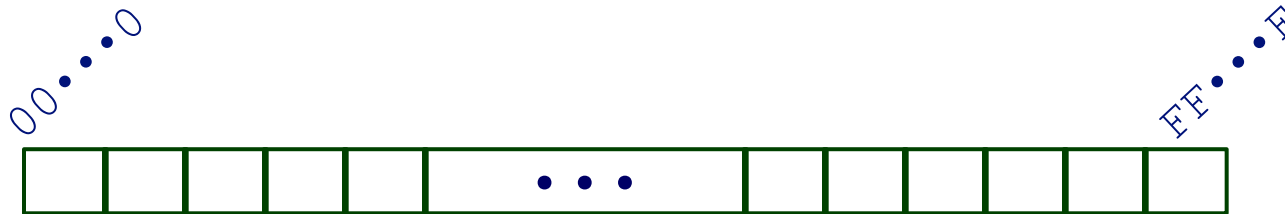
```
movl (%ecx), %eax
```

■ Displacement D(R) Mem[Reg[R]+D]

- Register R specifies start of memory region
- Constant displacement D specifies offset

```
movl 8(%ebp), %edx
```

Lets think on this: physical memory?



- **How does everything fit?**
 - 32-bit addresses: ~4,000,000,000 (4 billion) bytes
 - 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes
- **How to decide which memory to use in your program?**
 - What about after a `fork()`?
- **What if another process stores data into your memory?**
 - How could you debug your program?

So, we add a level of indirection

- **One simple trick solves all three problems**
 - Each process gets its own private image of memory
 - appears to be a full-sized private memory range
 - This fixes “how to choose” and “others shouldn’t mess w/yours”
 - surprisingly, it also fixes “making everything fit”
 - Implementation: translate addresses transparently
 - add a mapping function
 - to map private addresses to physical addresses
 - do the mapping on every load or store
- **This mapping trick is the heart of *virtual memory***

Address Spaces

- **Linear address space:** Ordered set of contiguous non-negative integer addresses:

$\{0, 1, 2, 3 \dots \}$

- **Virtual address space:** Set of $N = 2^n$ virtual addresses

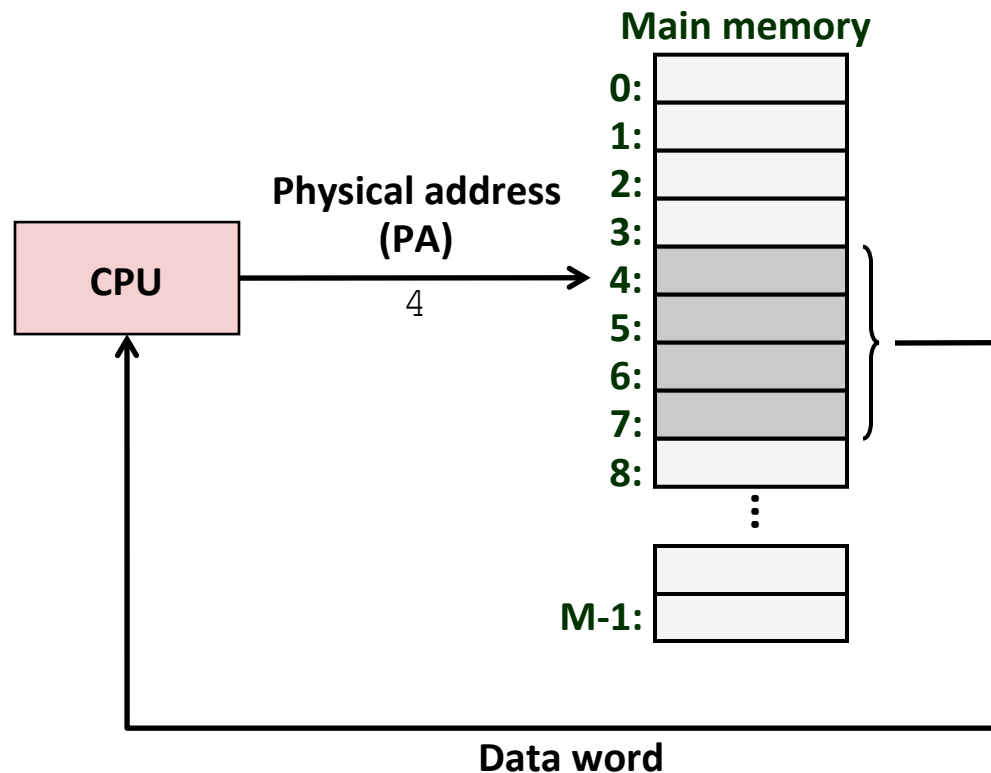
$\{0, 1, 2, 3, \dots, N-1\}$

- **Physical address space:** Set of $M = 2^m$ physical addresses

$\{0, 1, 2, 3, \dots, M-1\}$

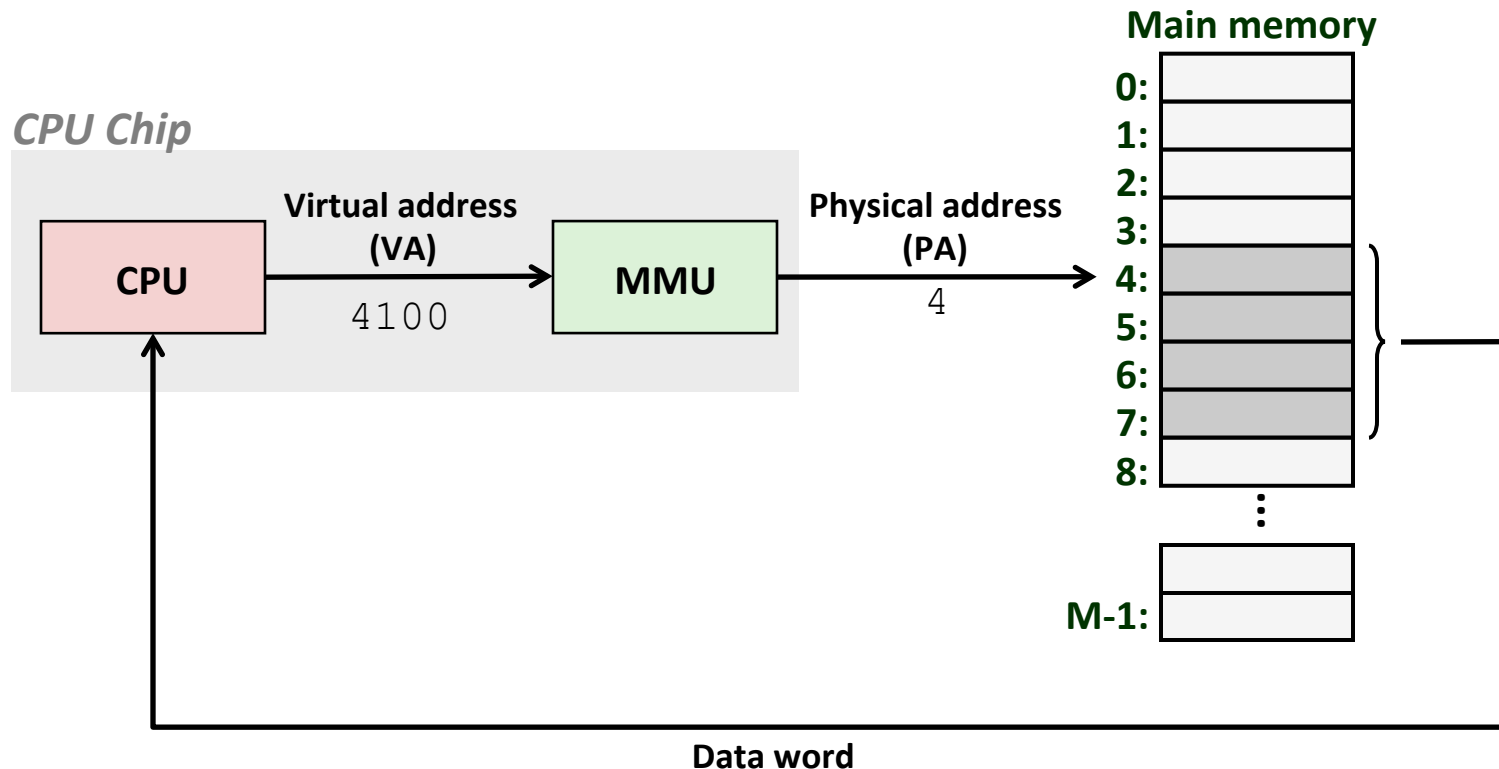
- Clean distinction between data (bytes) and their attributes (addresses)
- Each datum can now have multiple addresses
- Every byte in main memory:
one physical address, one (or more) virtual addresses

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

Why Virtual Memory?

(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

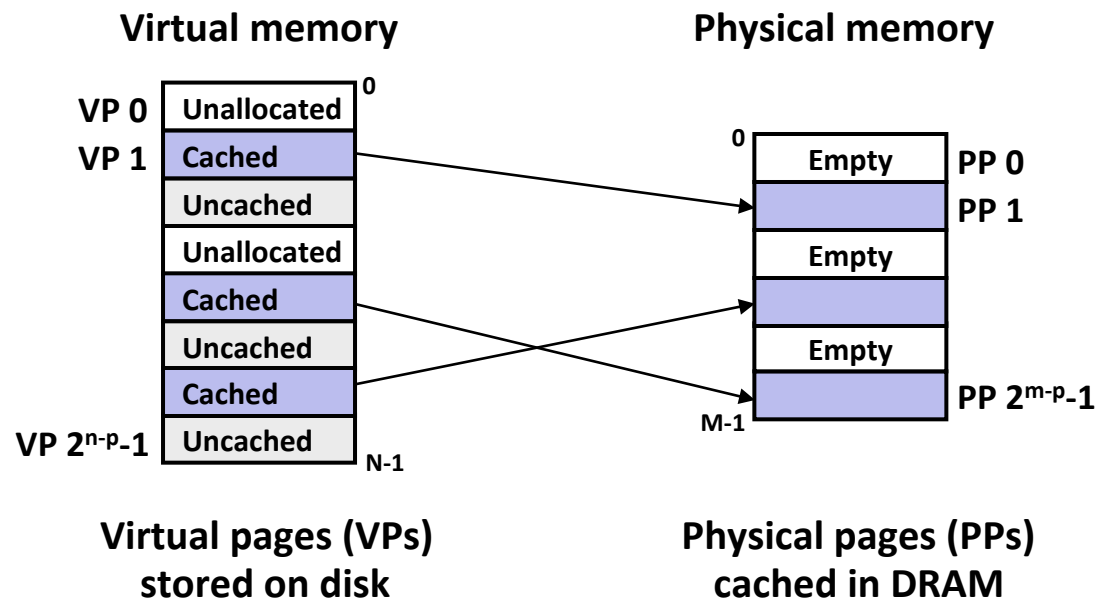
- One process can't interfere with another's memory
 - because they operate in different address spaces
- User process cannot access privileged information
 - different sections of address spaces have different permissions

Today

- Address spaces
- **(1) VM as a tool for caching**
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- Address translation
- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping

(1) 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)



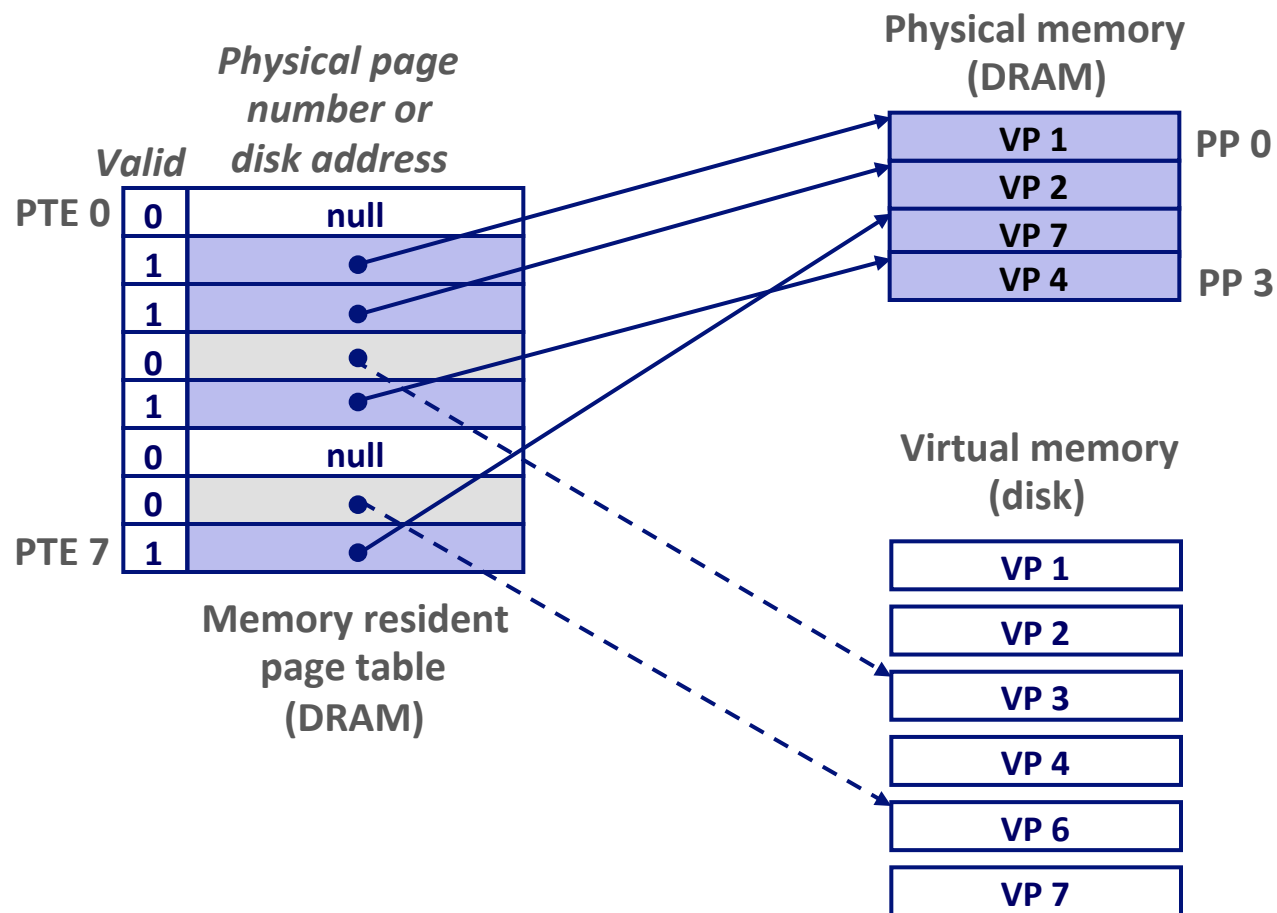
DRAM Cache Organization

- **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

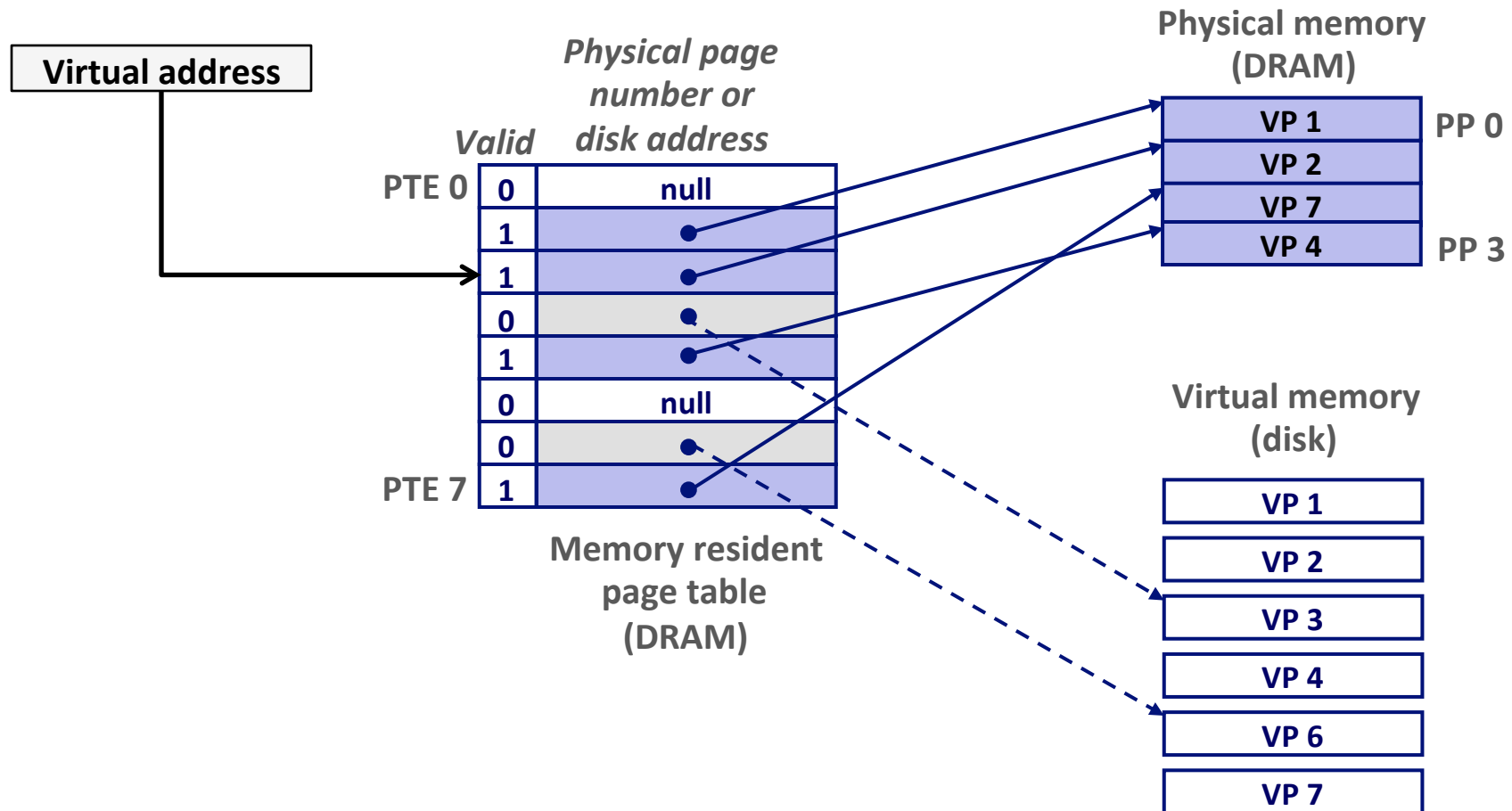
Enabling data structure: Page Table

- 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



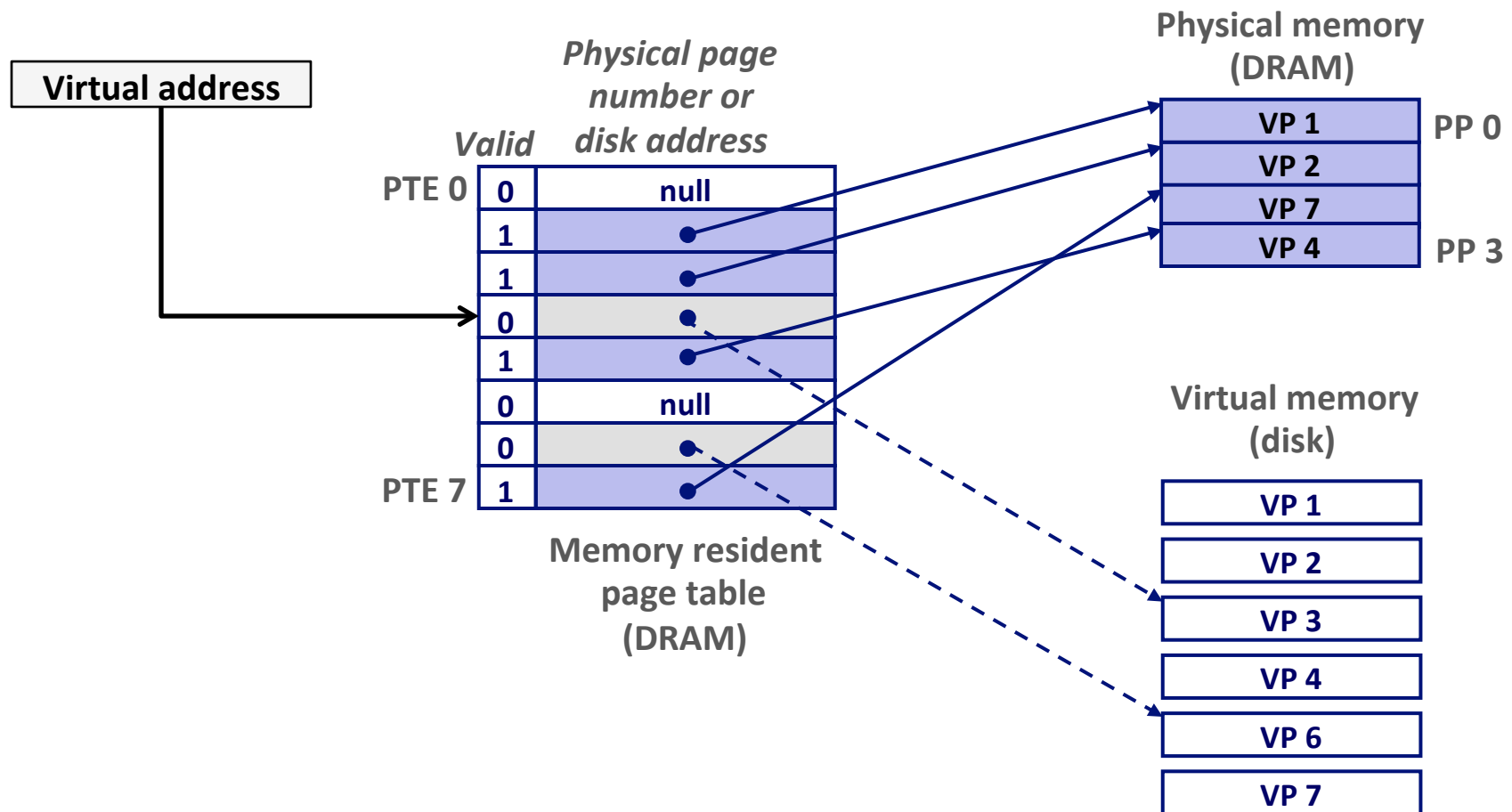
Page Hit

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



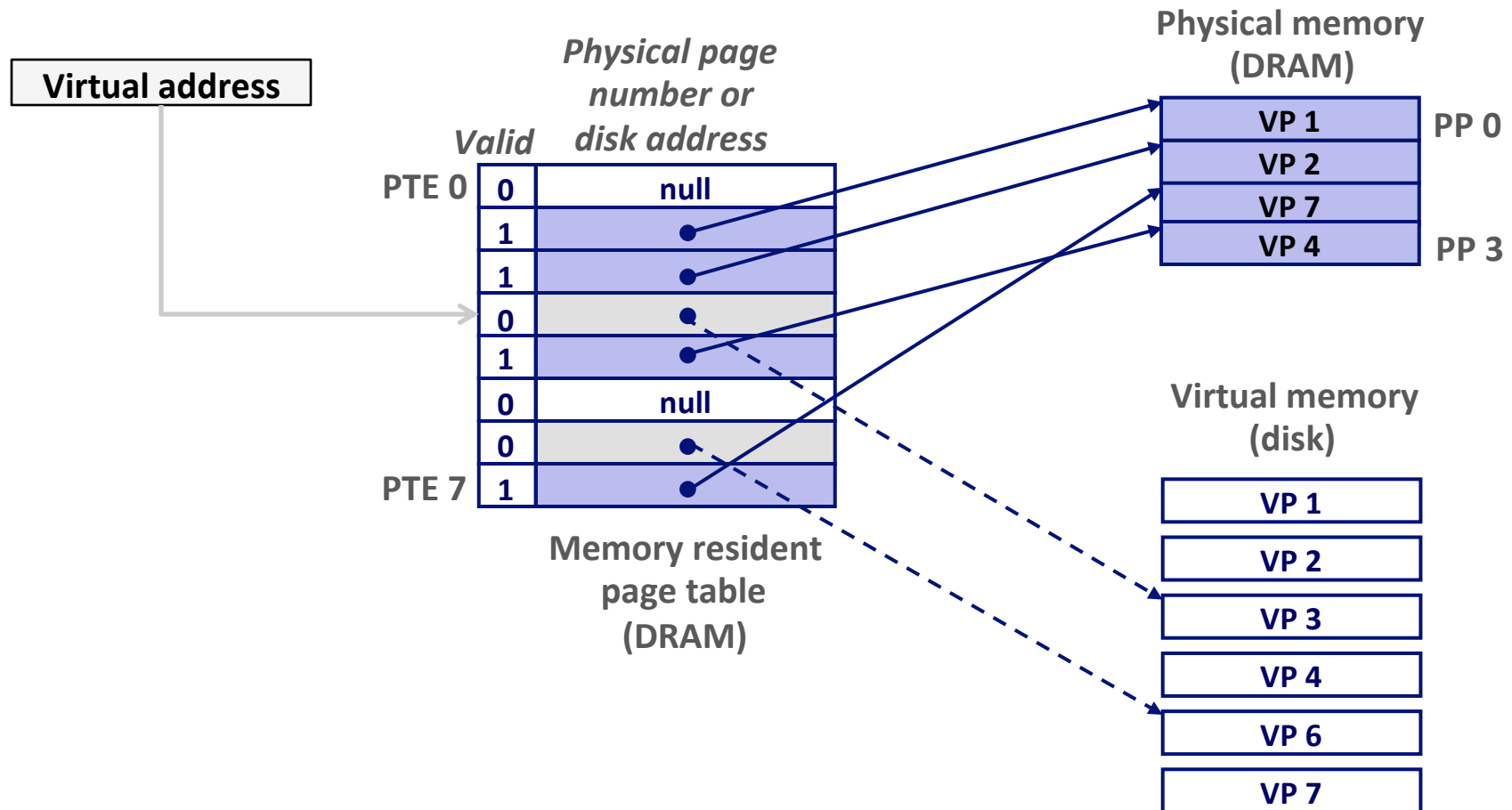
Page Fault

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



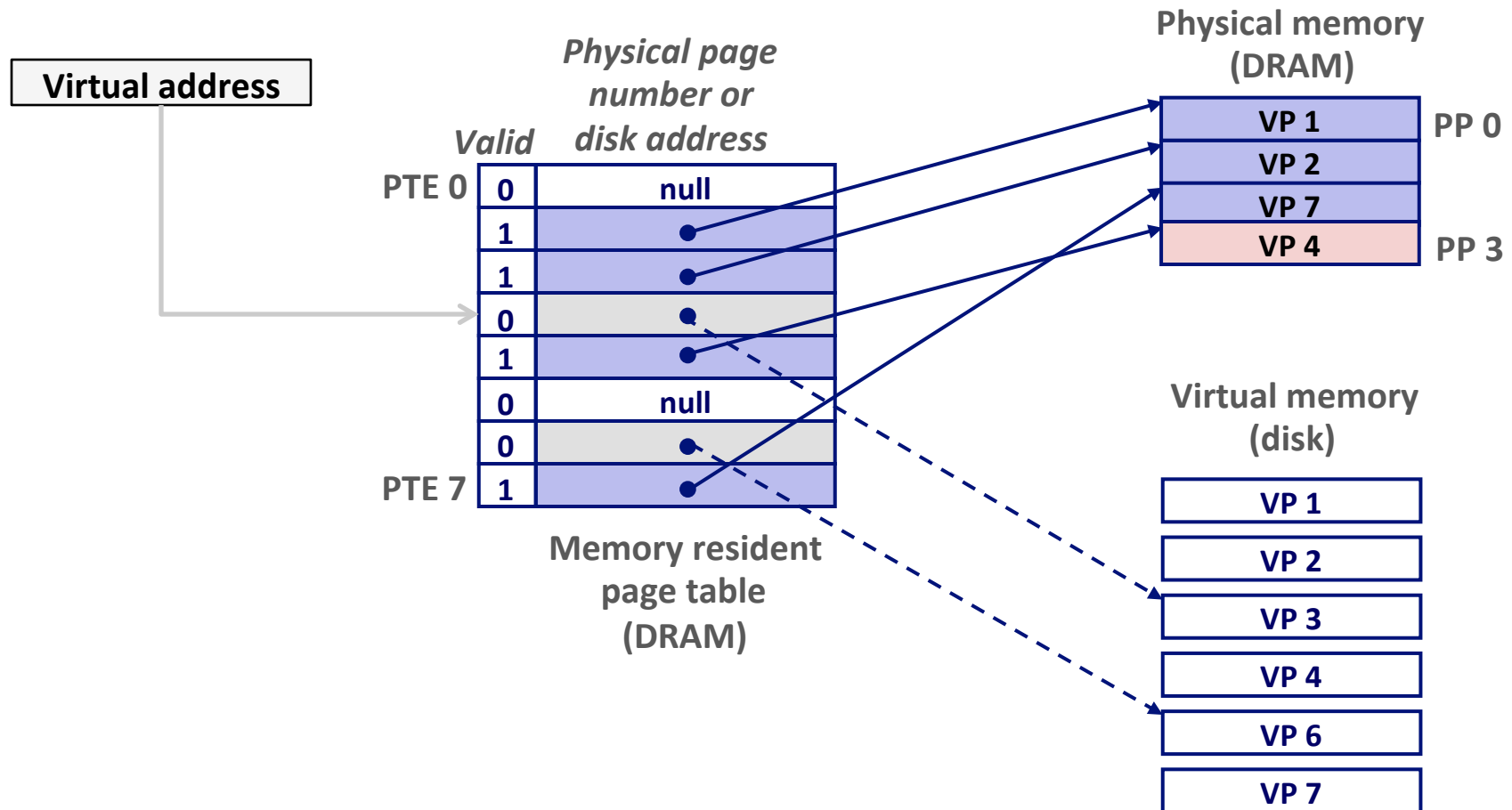
Handling Page Fault

- Page miss causes page fault (an exception)



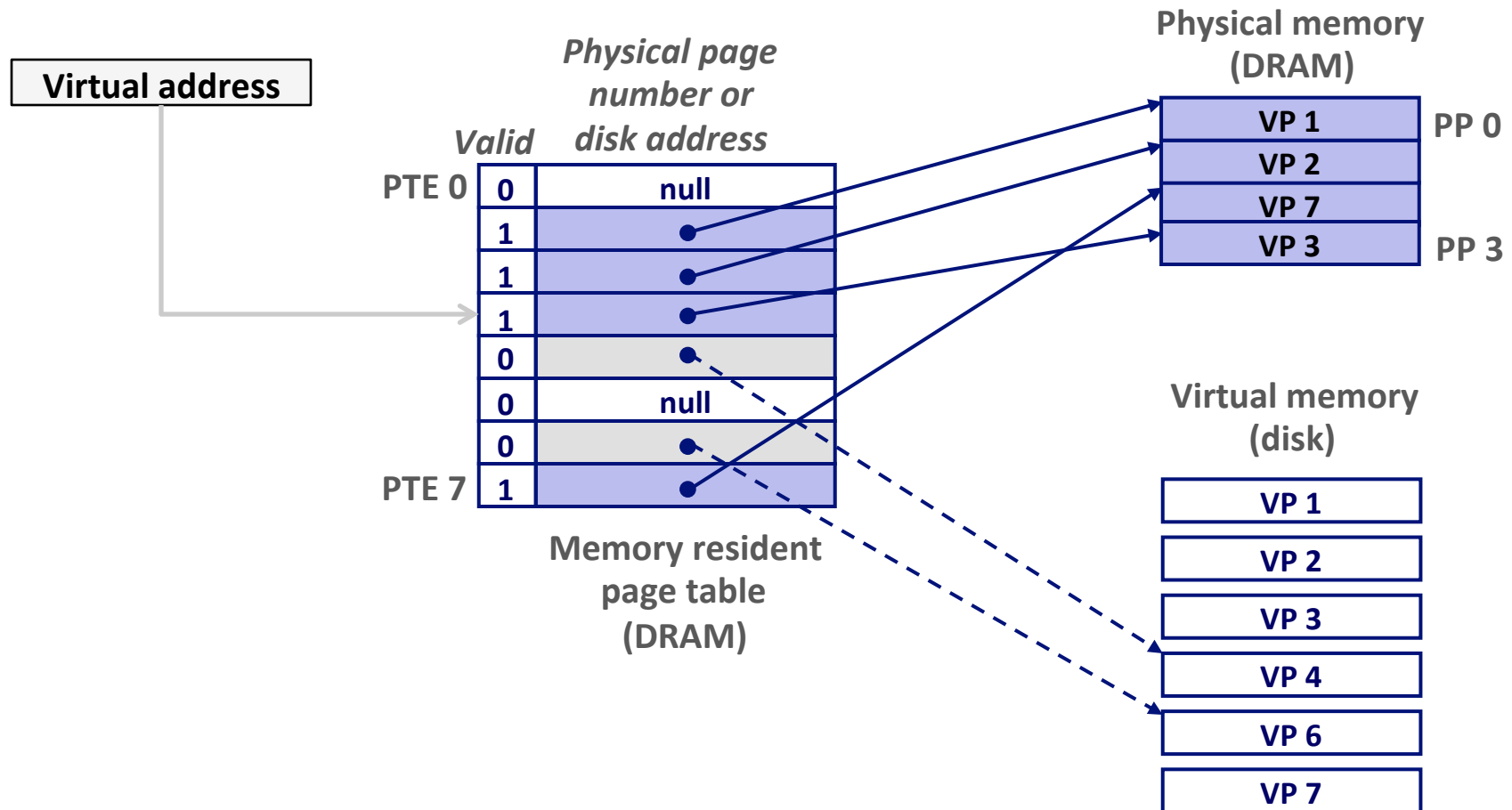
Handling Page Fault

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



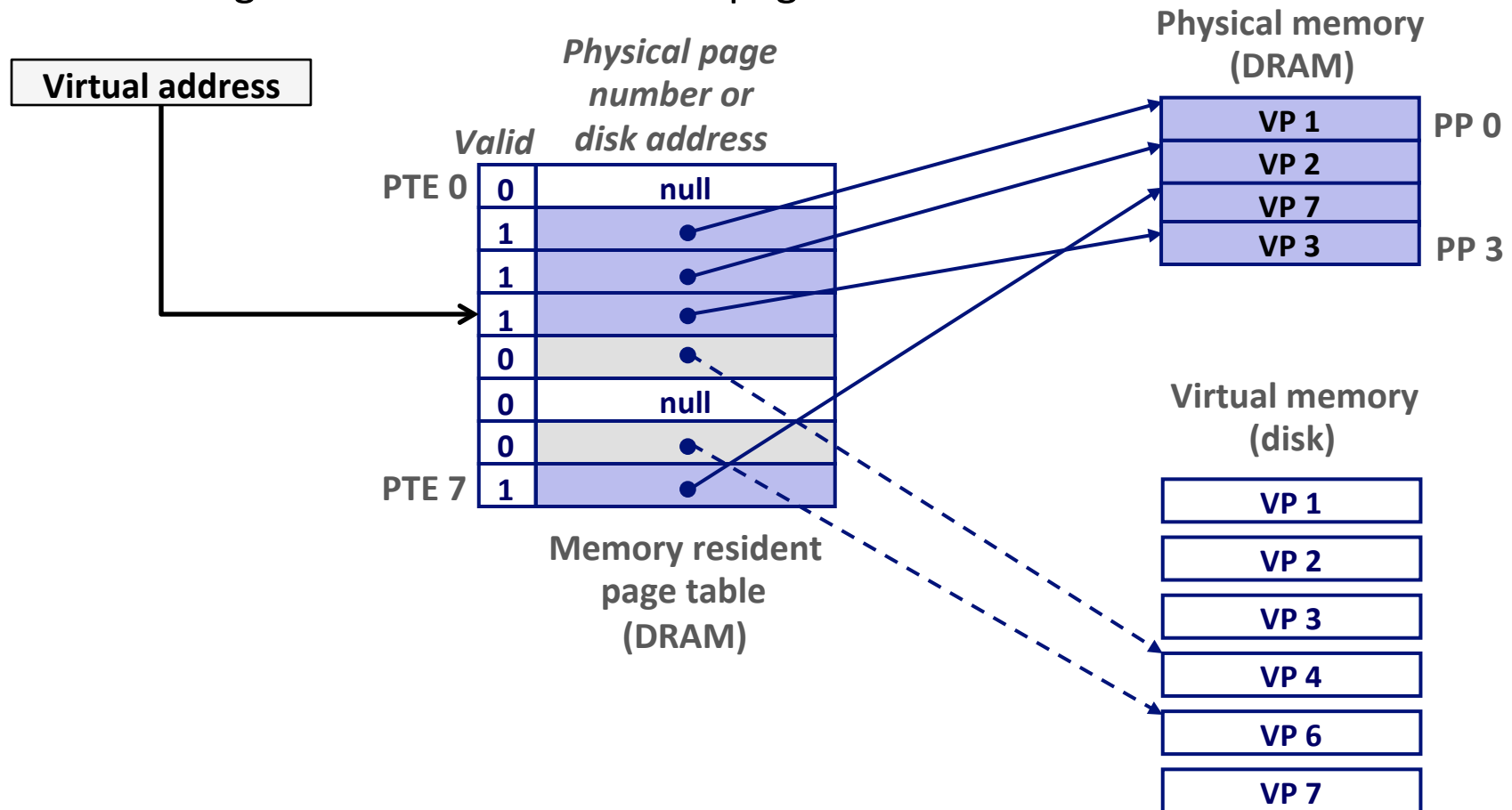
Handling Page Fault

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



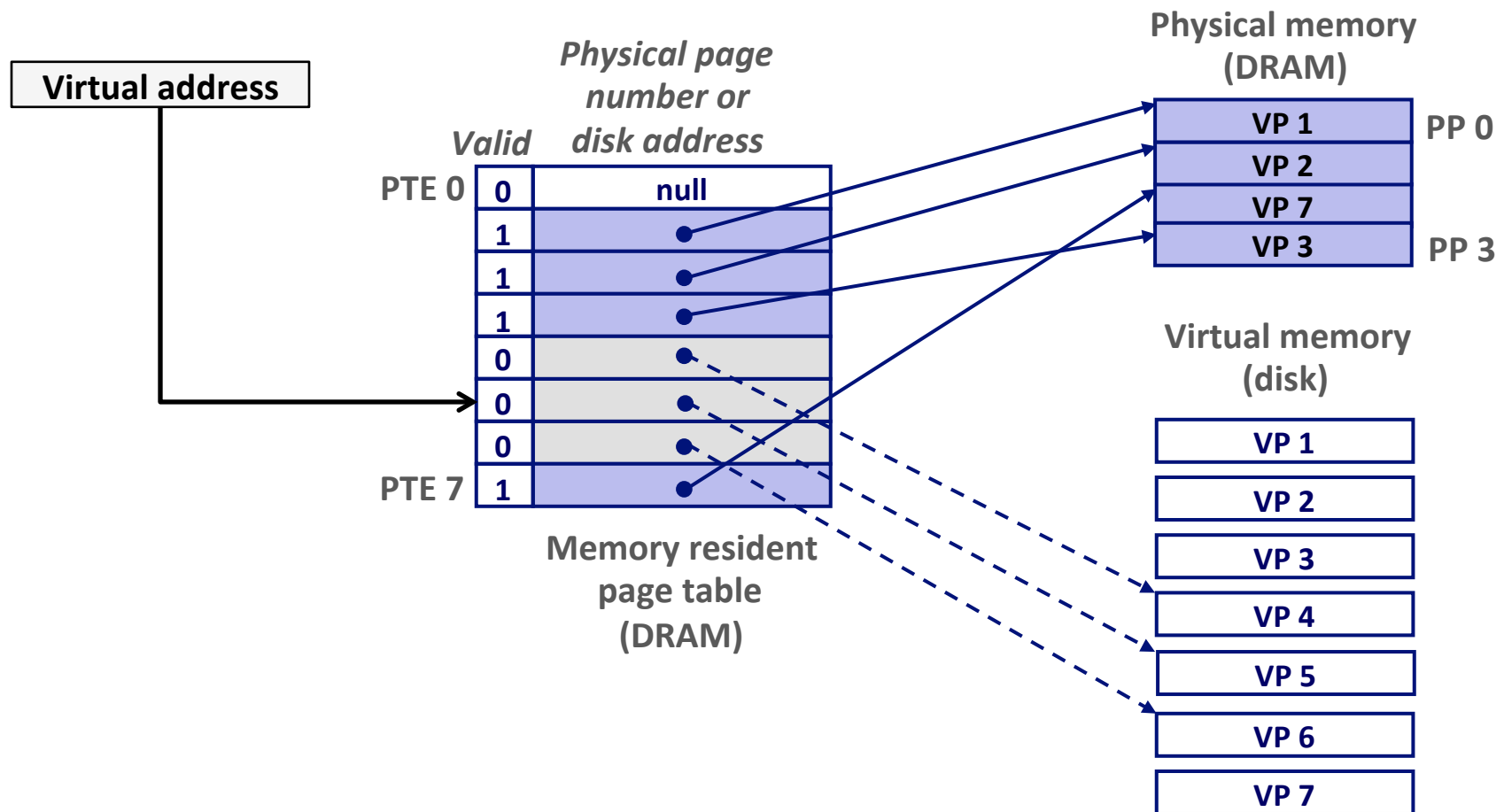
Handling Page Fault

- 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!



Allocating Pages

- Operating system allocates a new page of virtual memory, for example, as a result of calling `malloc`.
- In the example, VP 5 is allocated by creating room on disk and updating PTE 5 to point to the newly created page on disk.



Locality to the Rescue Again!

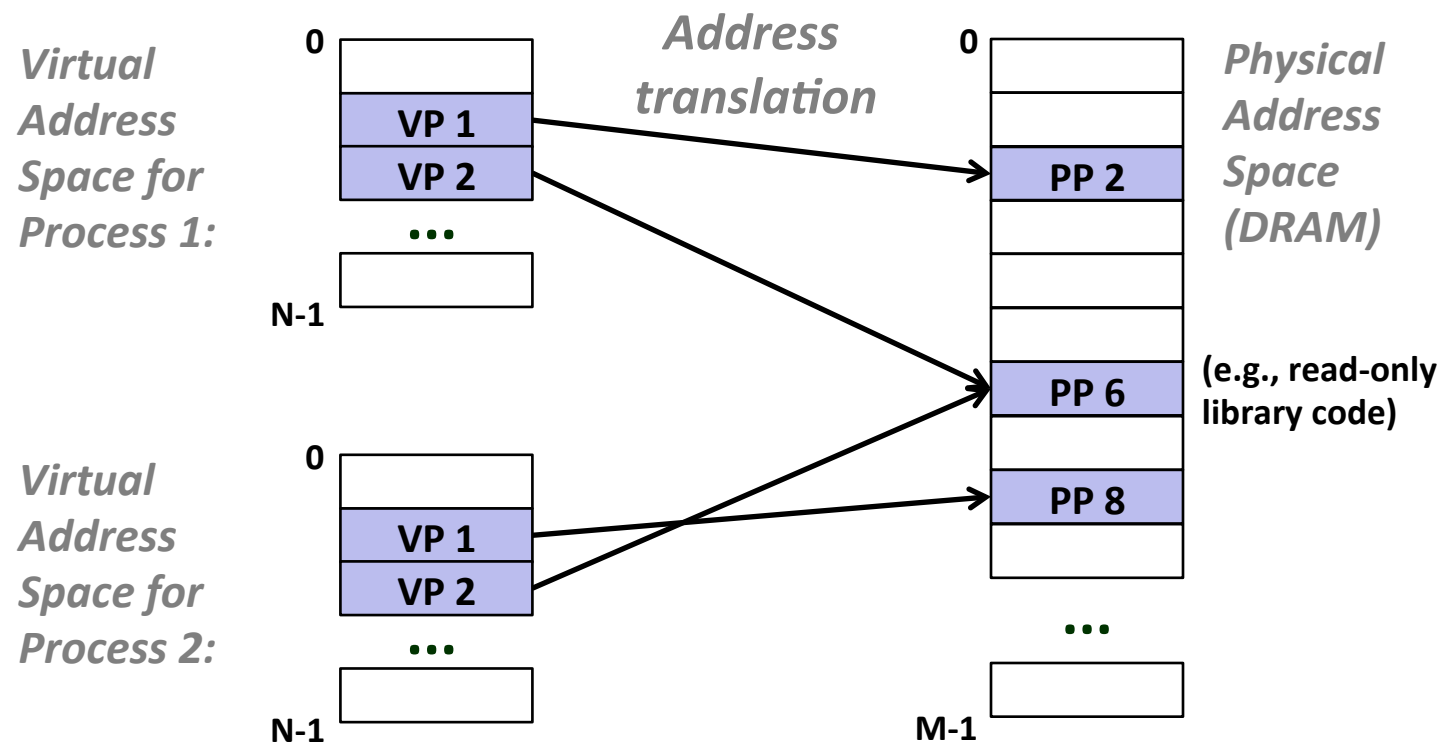
- Virtual memory works because of locality
- At any point in time, programs tend to access a set of active virtual pages called the *working set*
 - Programs with better temporal locality will have smaller working sets
- If (working set size < main memory size)
 - Good performance for one process after compulsory misses
- If (SUM(working set sizes) > main memory size)
 - *Thrashing*: Performance meltdown where pages are moved (copied) in and out continuously

Today

- Address spaces
- (1) VM as a tool for caching
- **(2) VM as a tool for memory management**
- (3) VM as a tool for memory protection
- Address translation
- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping

(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



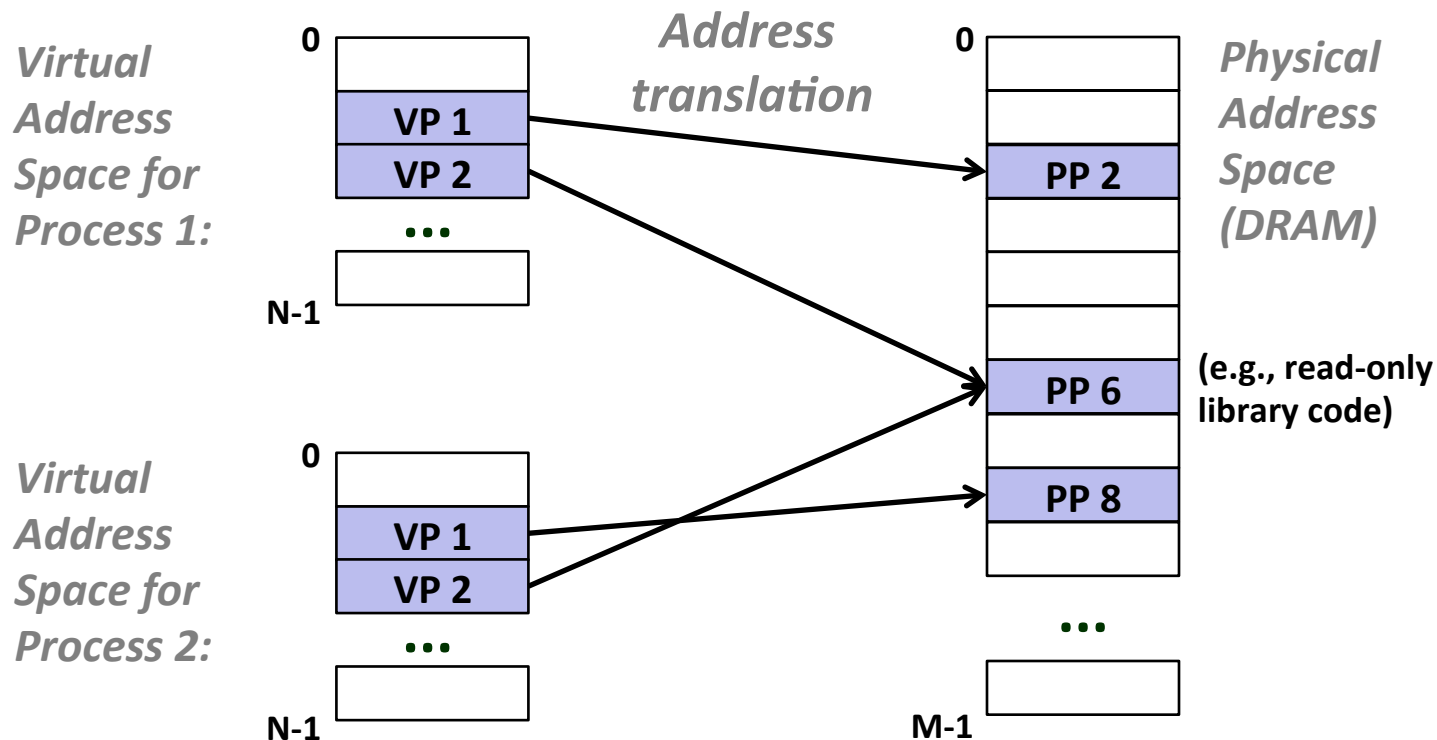
Simplifying allocation and sharing

■ 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)



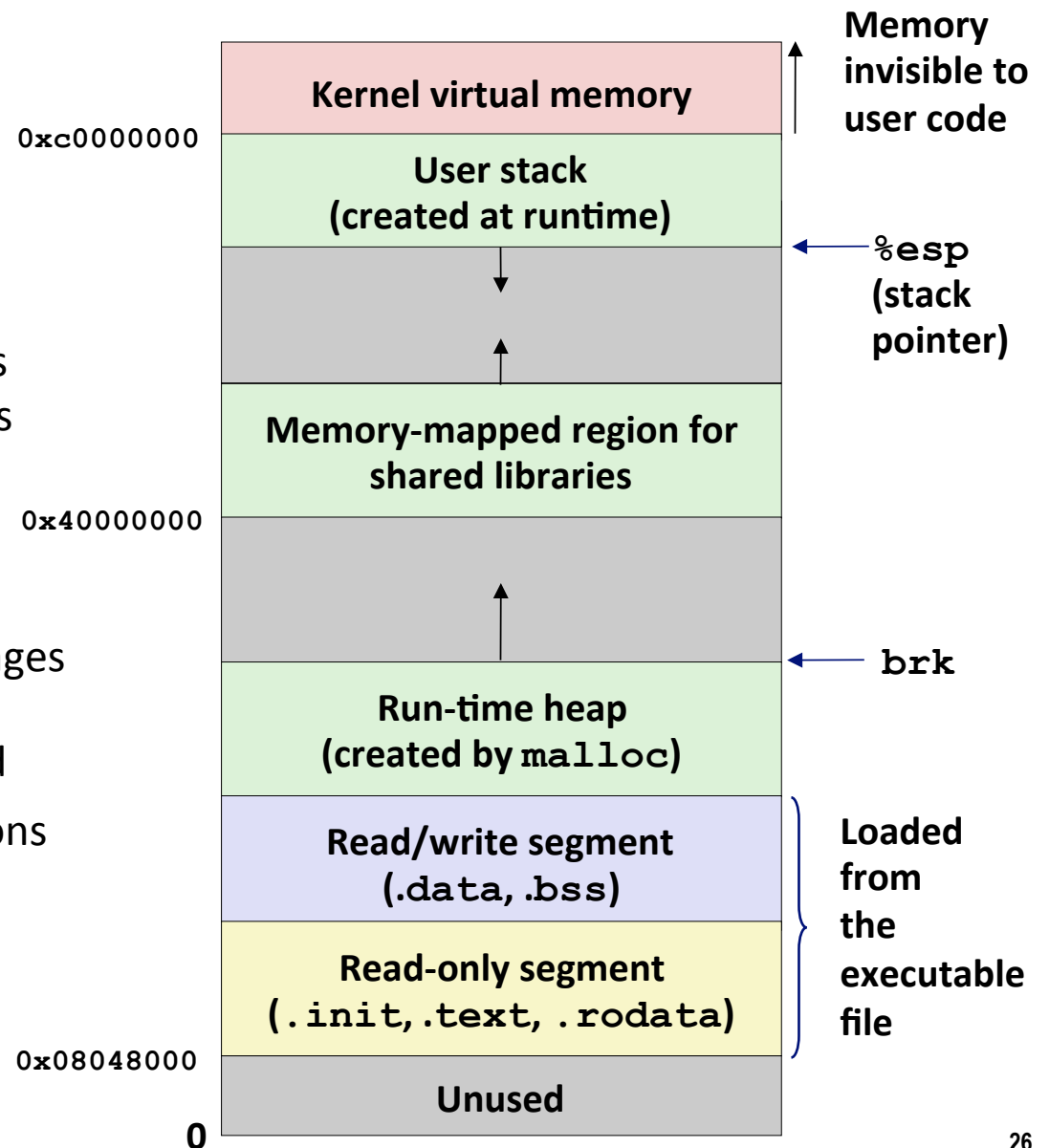
Simplifying Linking and Loading

■ 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

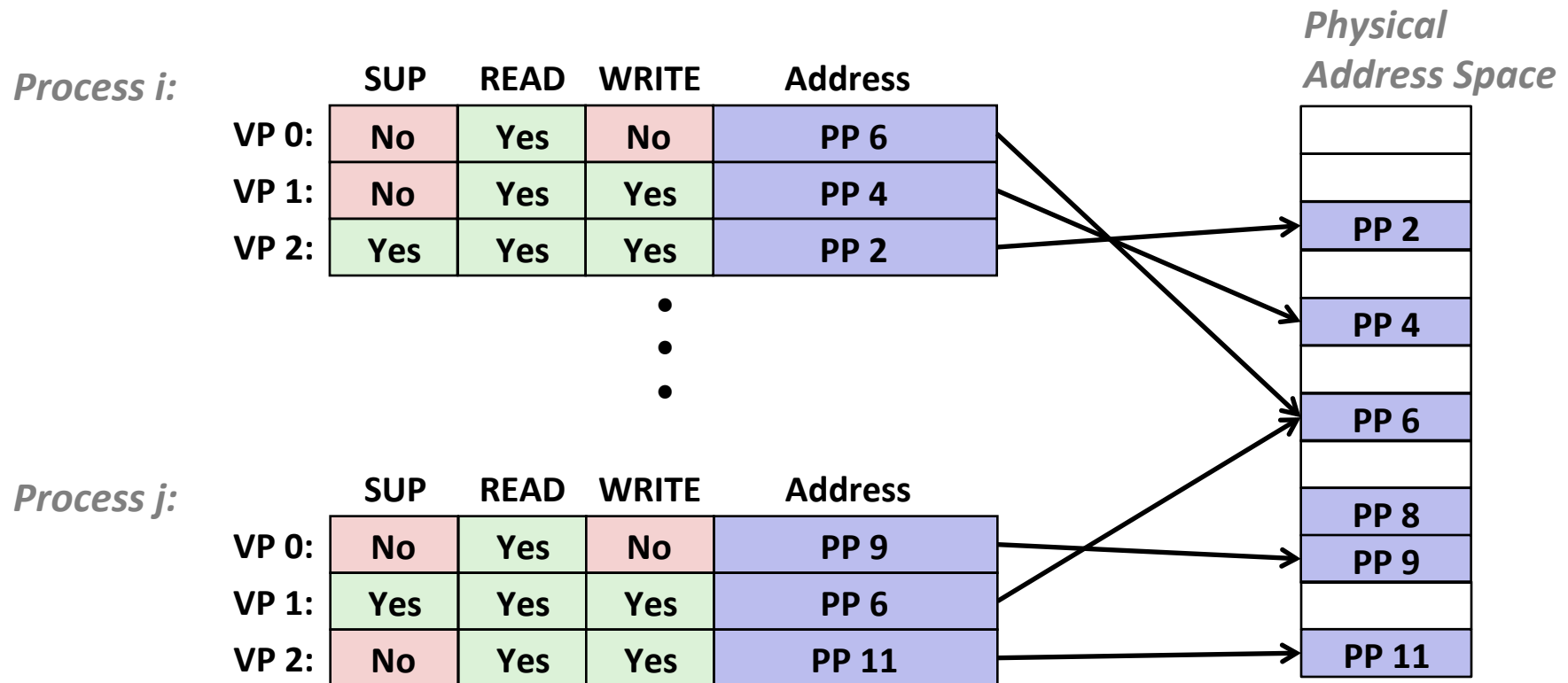


Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- **(3) VM as a tool for memory protection**
- Address translation
- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping

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)



Virtual memory review

■ 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

Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- **Address translation**
- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping

VM Address Translation

■ Virtual Address Space

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

■ Physical Address Space

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

■ Address Translation

- $MAP: V \rightarrow P \cup \{\emptyset\}$

- For virtual address a :

- $MAP(a) = a'$ if data at virtual address a is at physical address a' in P

- $MAP(a) = \emptyset$ 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)

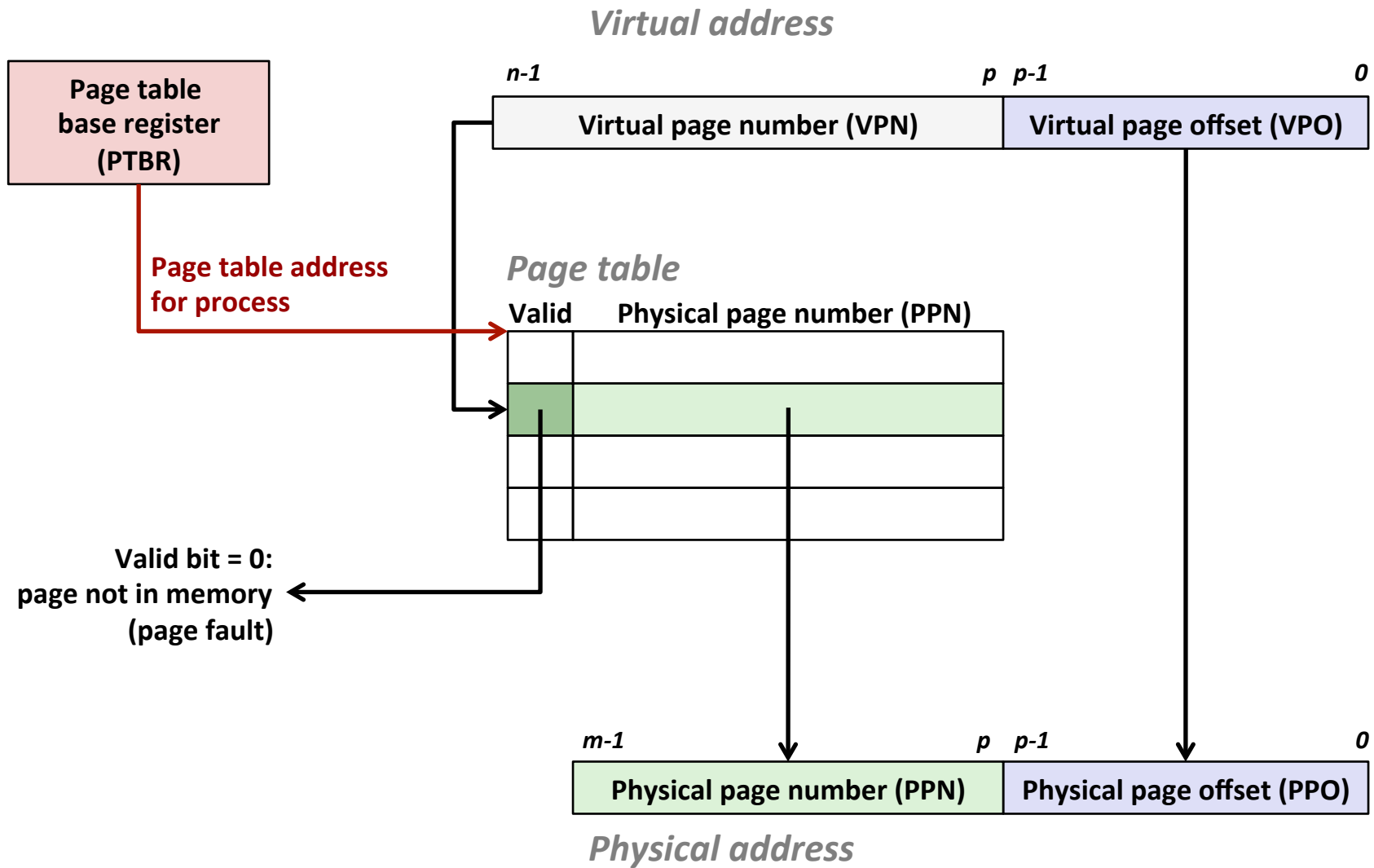
■ Components of the virtual address (VA)

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

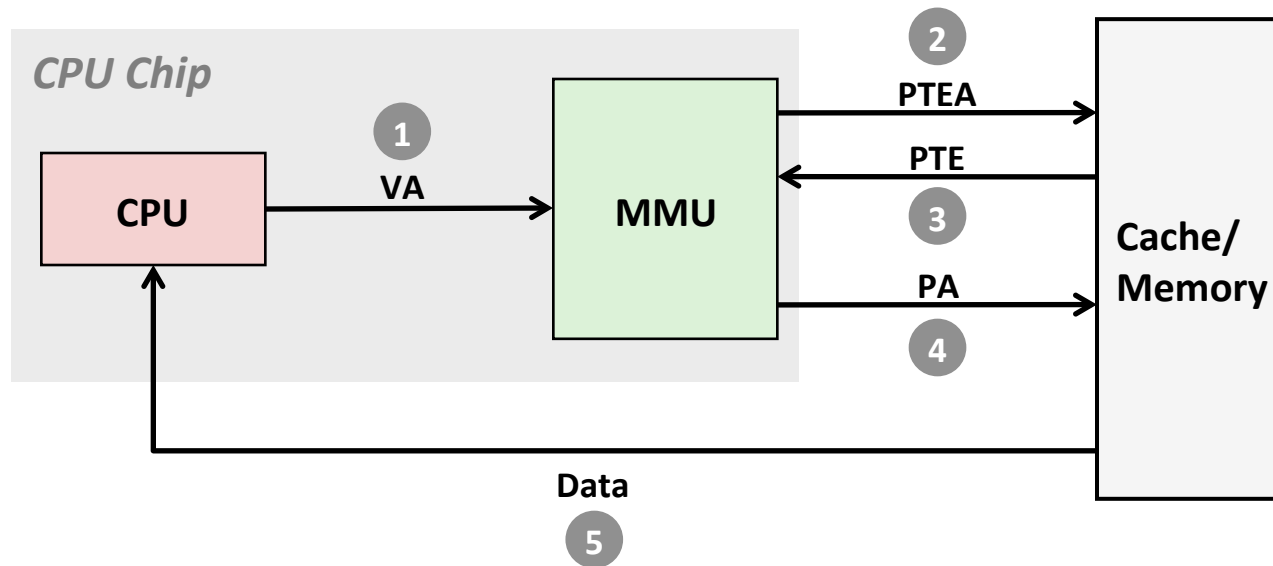
■ Components of the physical address (PA)

- **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



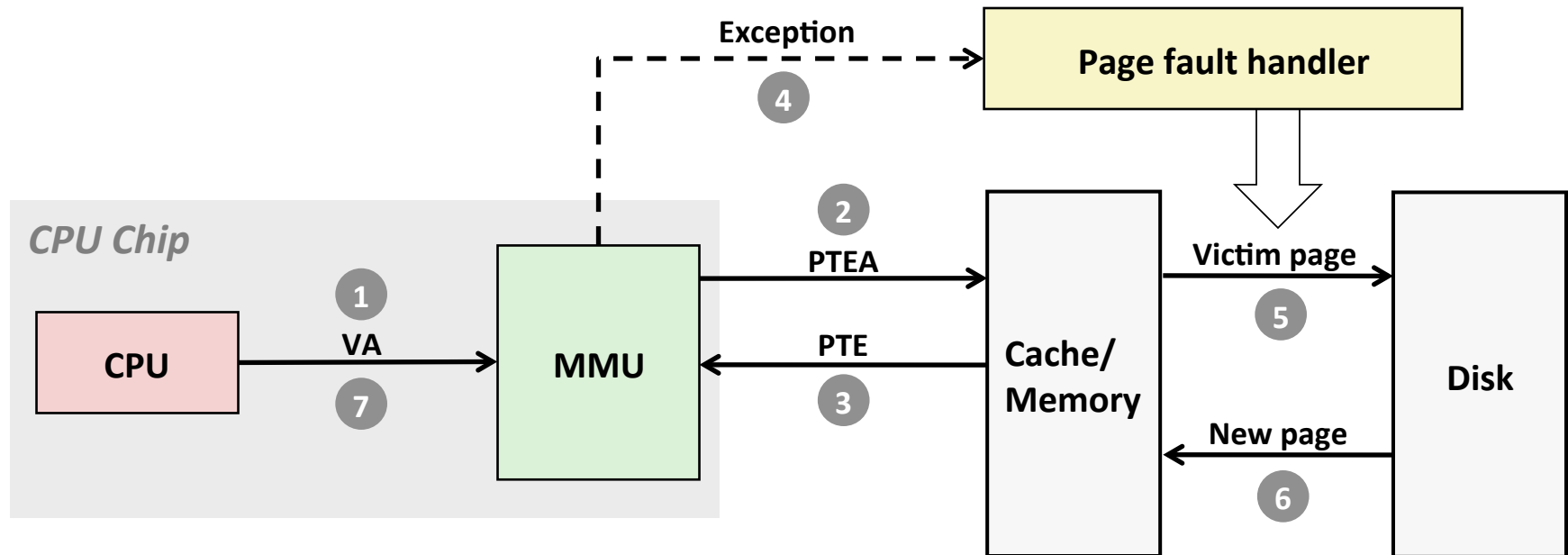
Address Translation: Page Hit



- 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

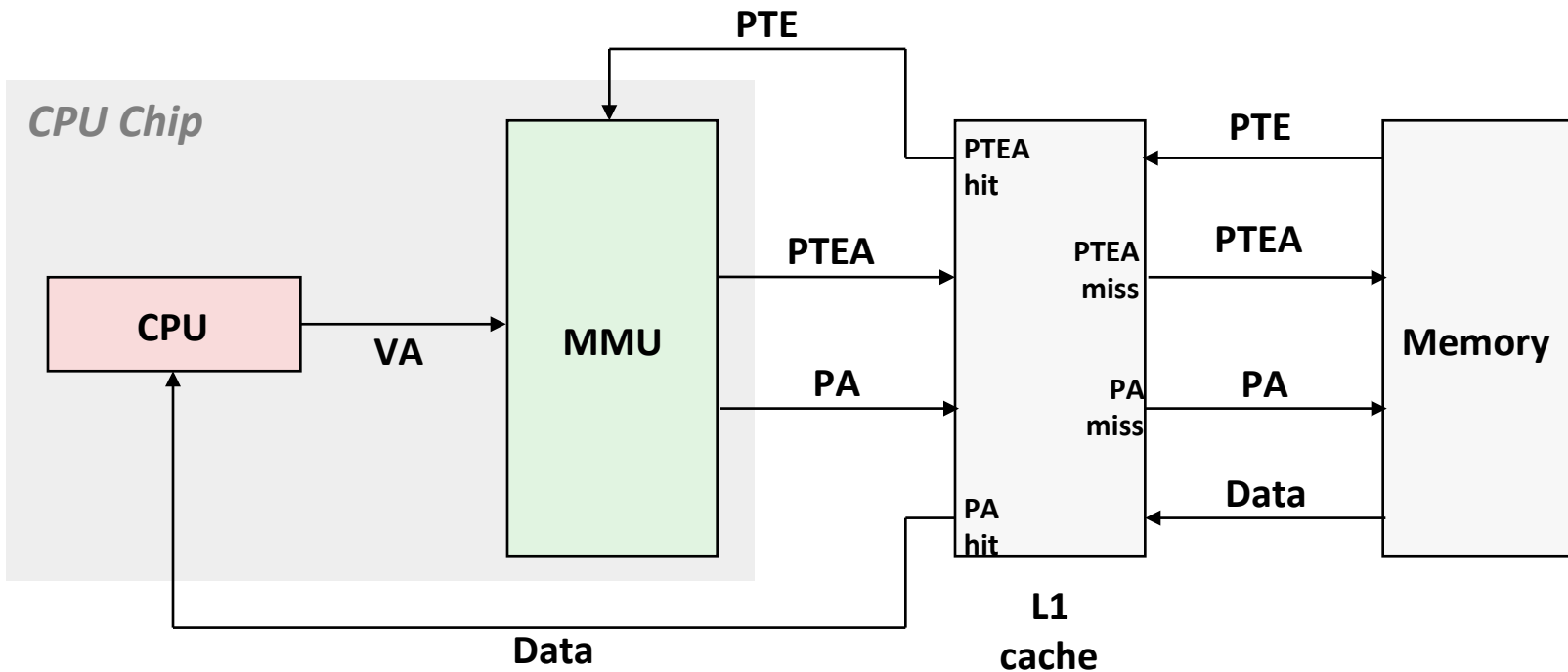
VA: virtual address.
PTEA: page table entry address.
PTE: page table entry.
PA: physical address.

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

Integrating VM and Cache



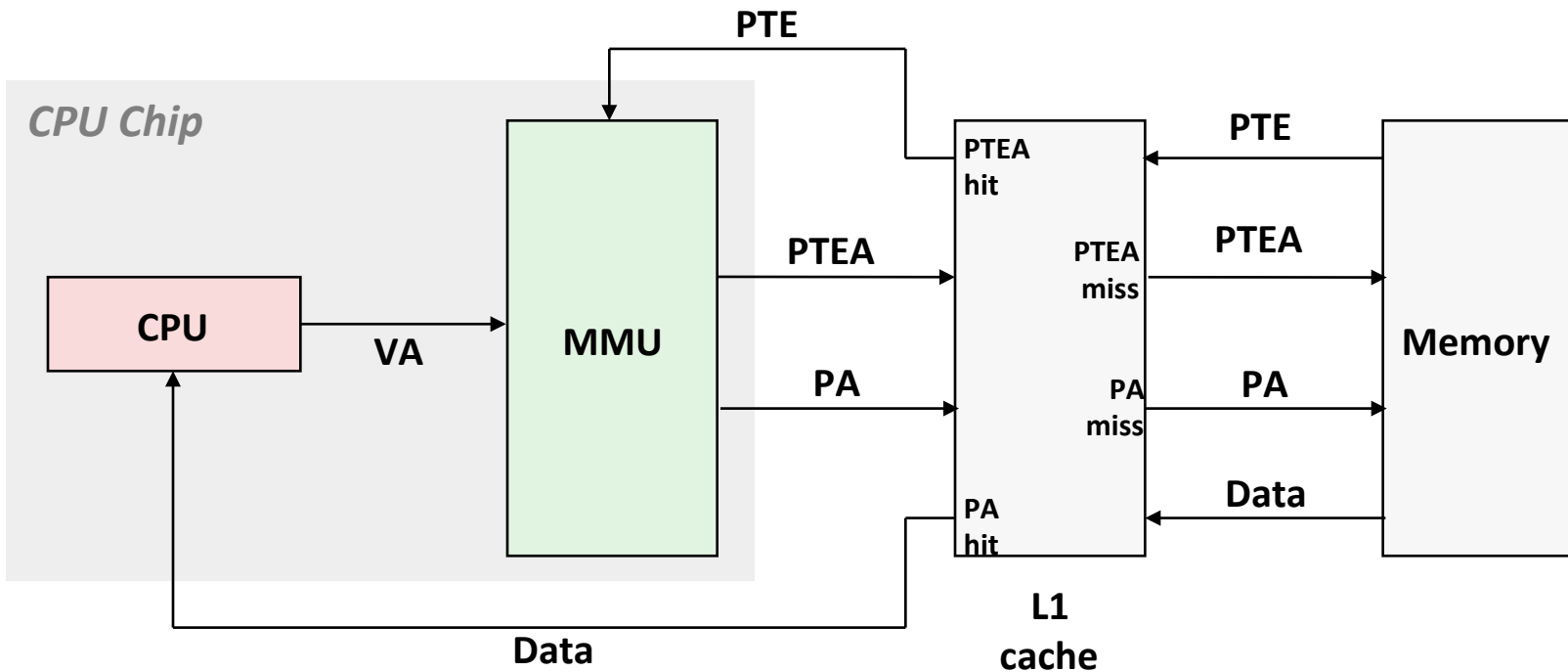
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Question #1

- Are the PTEs cached like other memory accesses?

- Yes (and no: see next question)

Page tables in memory, like other data



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

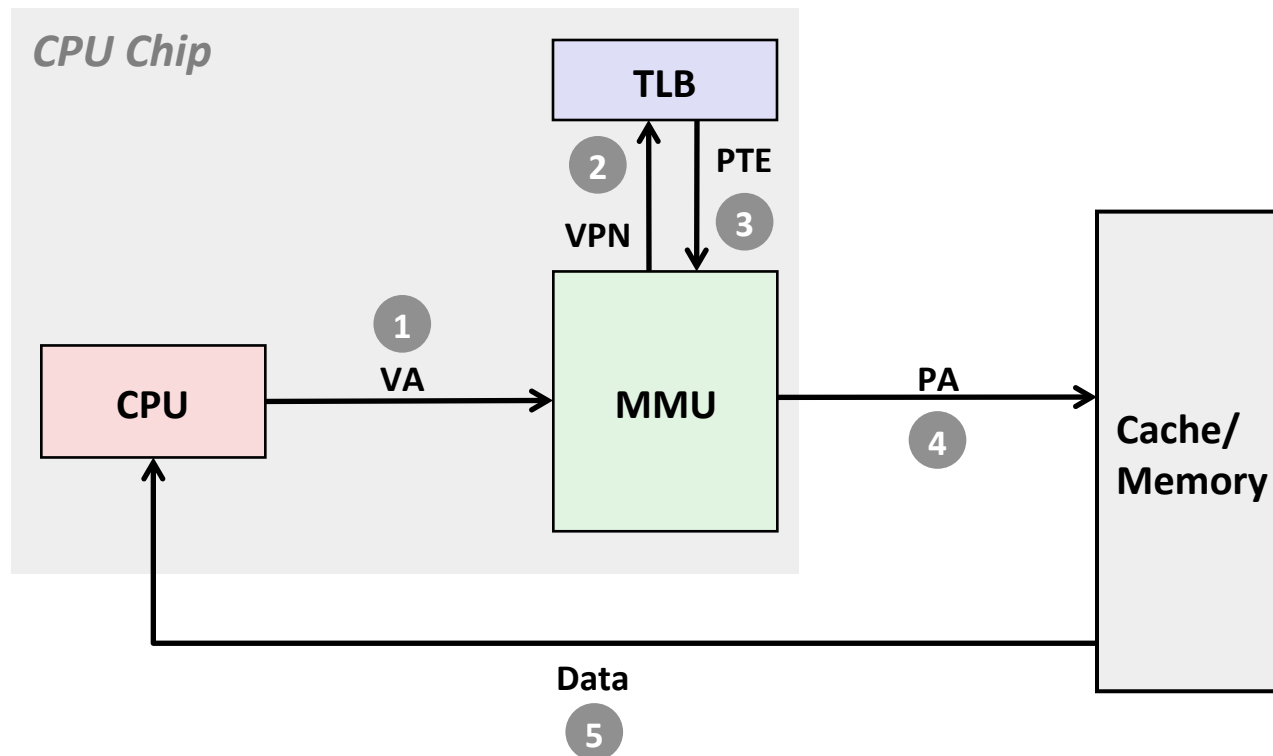
Question #2

- 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

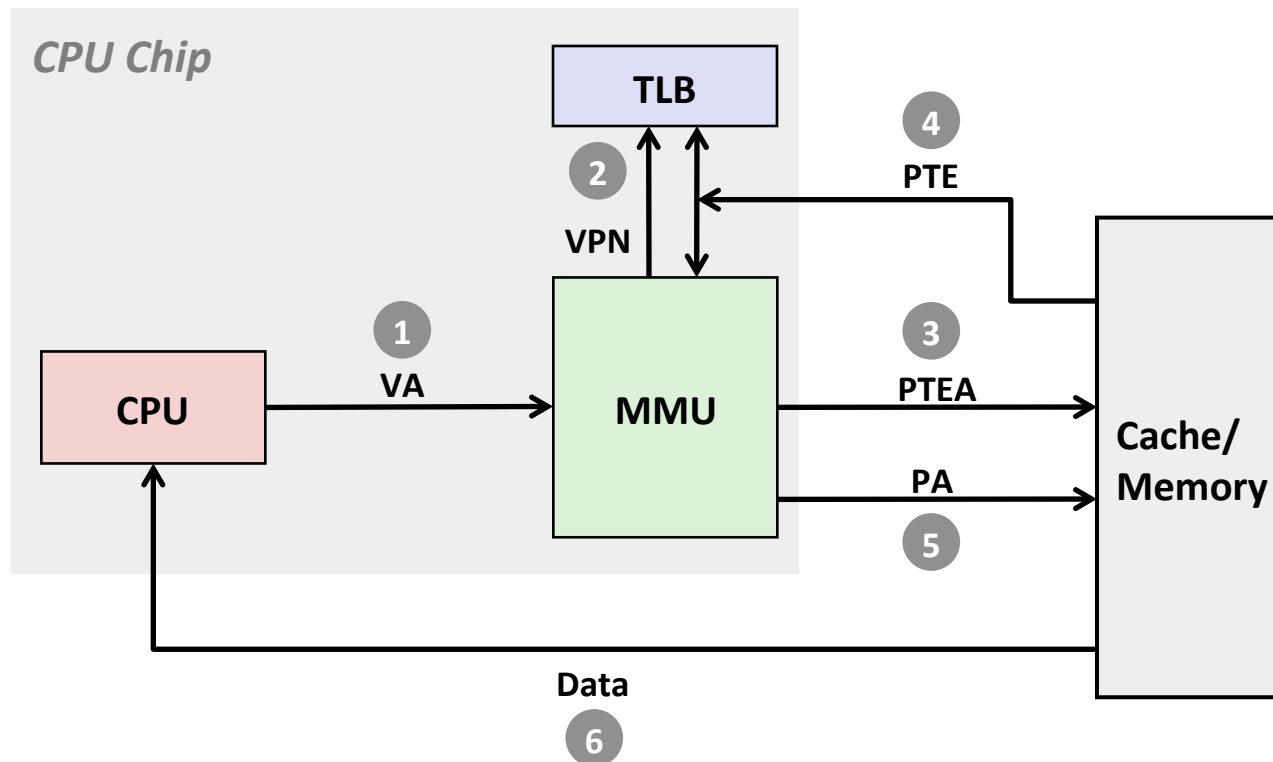
- **Page table entries (PTEs) are cached in L1 like any other memory word**
 - PTEs may be evicted by other data references
 - PTE hit still requires a small L1 delay
- **Solution: *Translation Lookaside Buffer* (TLB)**
 - Small, dedicated, super-fast hardware cache of PTEs in MMU
 - Contains complete 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?

Question #3

- 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:

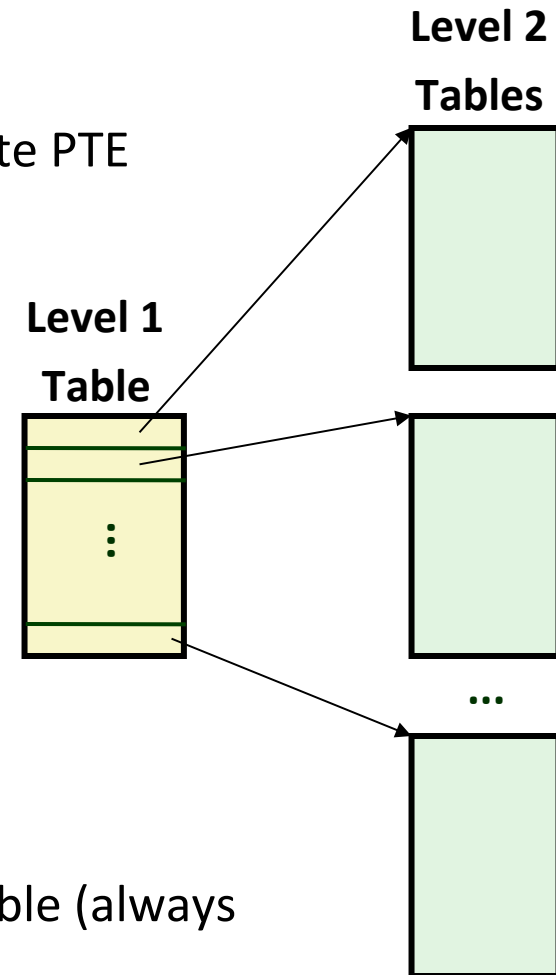
- 4KB (2^{12}) page size, 64-bit address space, 8-byte PTE

■ Problem:

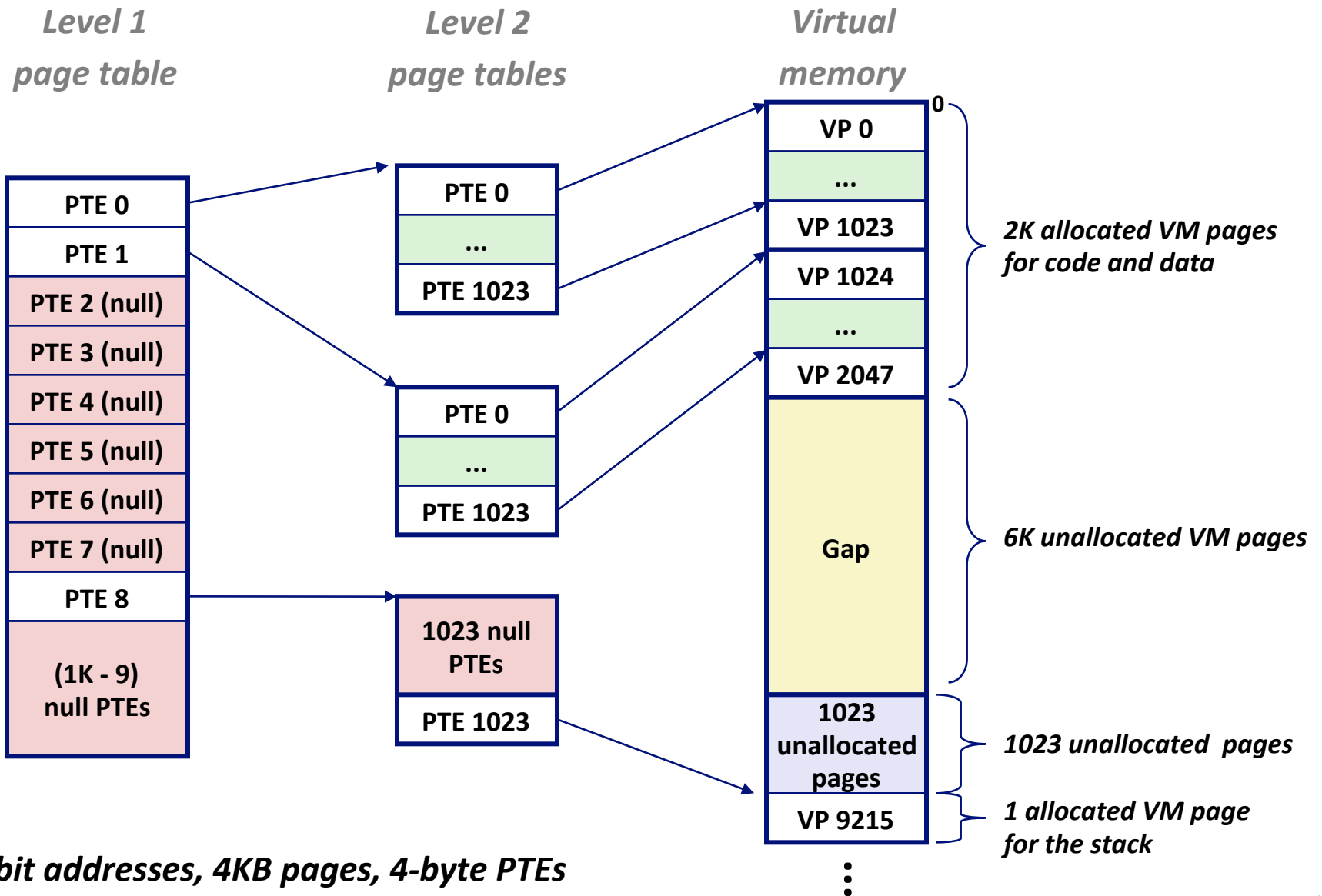
- Would need a 32,000 TB page table!
 - $2^{64} * 2^{-12} * 2^3 = 2^{55}$ bytes

■ Common solution:

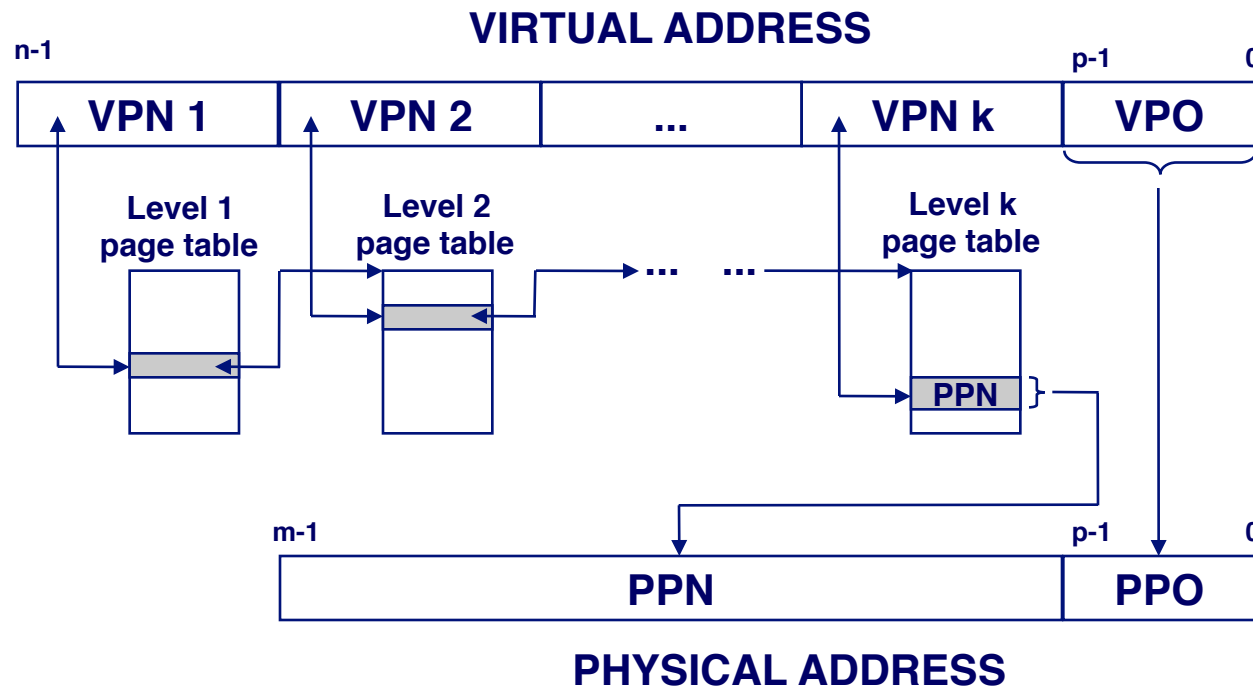
- Multi-level page tables
- Example: 2-level page table
 - Level 1 table: each PTE points to a page table (always memory resident)
 - Level 2 table: each PTE points to a page (paged in and out like any other data)



A Two-Level Page Table Hierarchy



Translating with a k-level Page Table

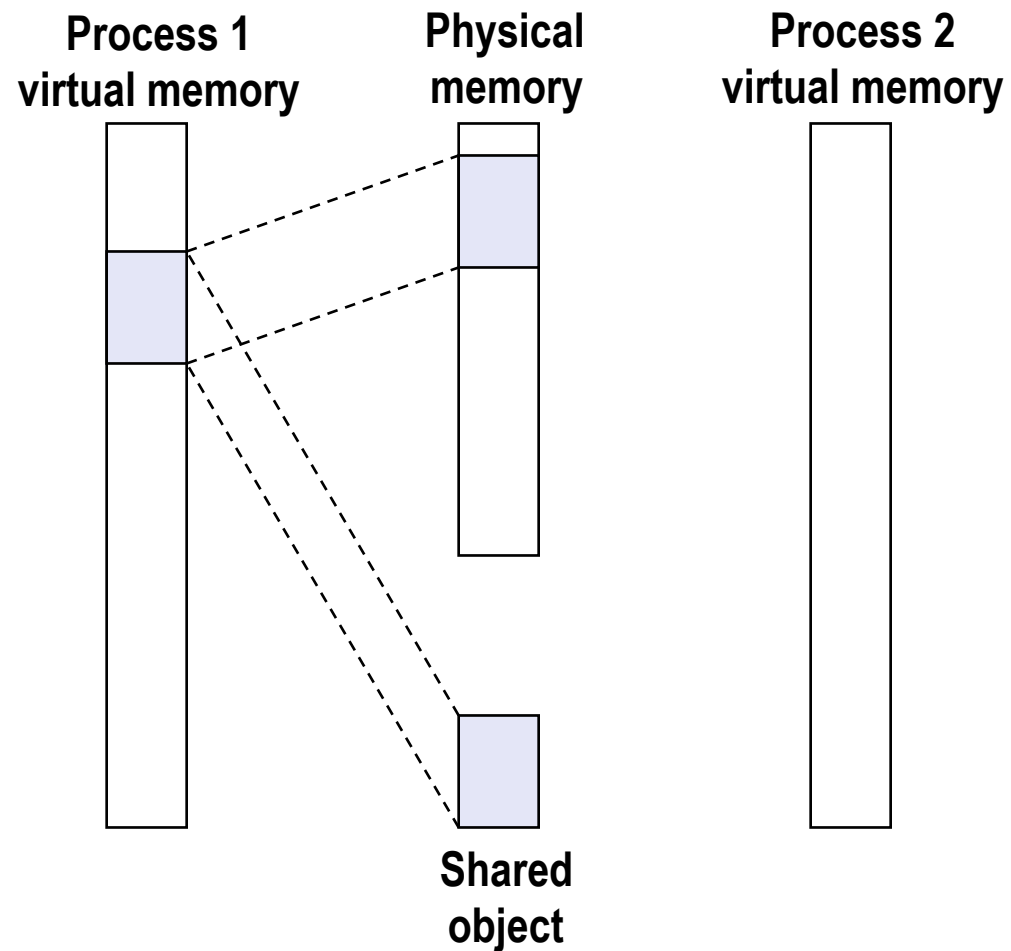


Question #4

- **Shouldn't fork() be really slow, since the child needs a copy of the parent's address space?**

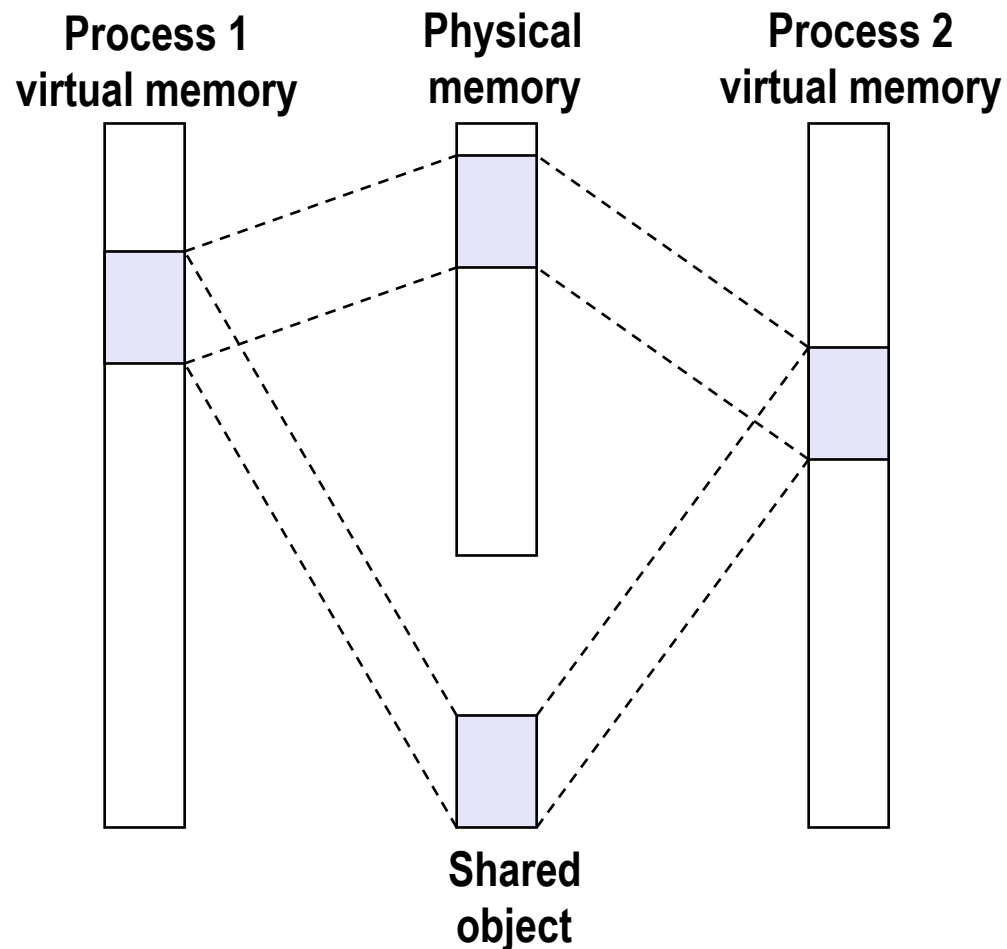
- **Yes, it would be... so, fork() doesn't really work that way**

Sharing Revisited: Shared Objects



- **Process 1 maps the shared object.**

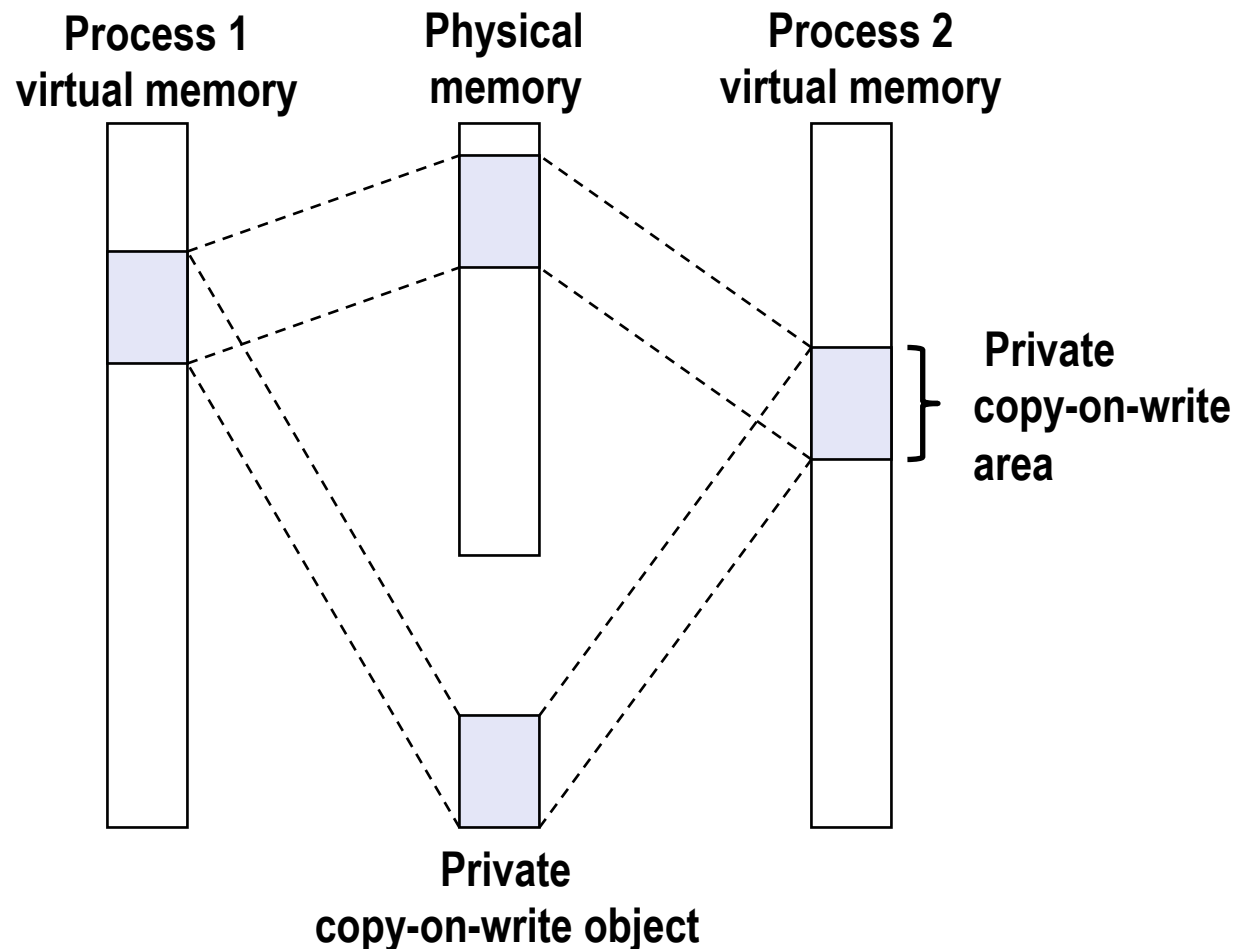
Sharing Revisited: Shared Objects



- **Process 2 maps the shared object.**
- **Notice how the virtual addresses can be different.**

Sharing Revisited:

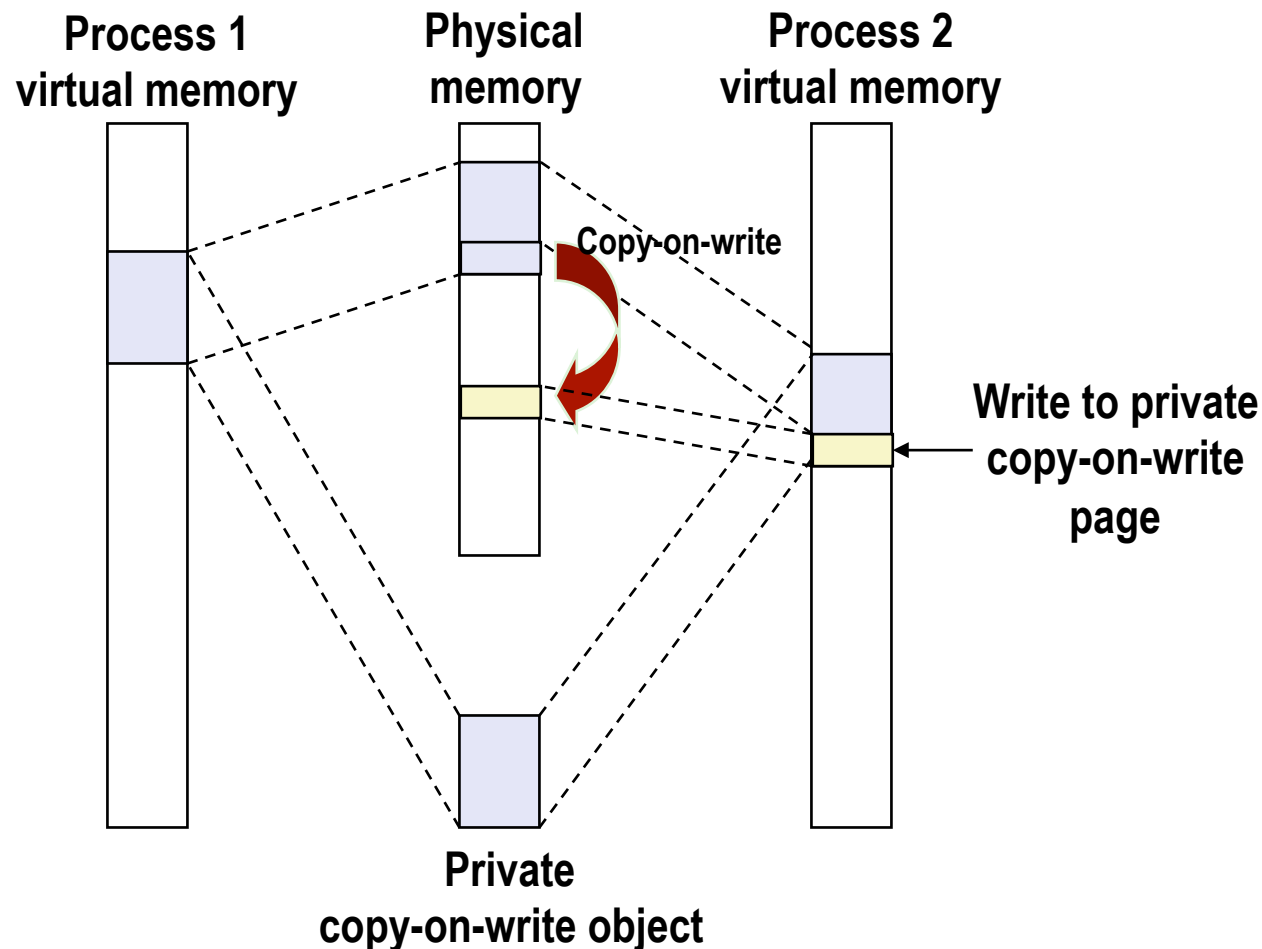
Private Copy-on-write (COW) Objects



- Two processes mapping a *private copy-on-write (COW)* object.
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only

Sharing Revisited:

Private Copy-on-write (COW) Objects



- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

The `fork` Function Revisited

- `fork` provides private address space for each process
- To create virtual address for new process
 - Create exact copies of parent page tables
 - Flag each page in both processes (parent and child) as read-only
 - Flag writeable areas in both processes as private COW
- On return, each process has exact copy of virtual memory
- Subsequent writes create new physical pages using COW mechanism
- Perfect approach for common case of `fork()` followed by `exec()`
 - Why?

Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- Address translation
- **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)

- VPO: Virtual page offset
- VPN: Virtual page number
- TLBI: TLB index
- TLBT: TLB tag

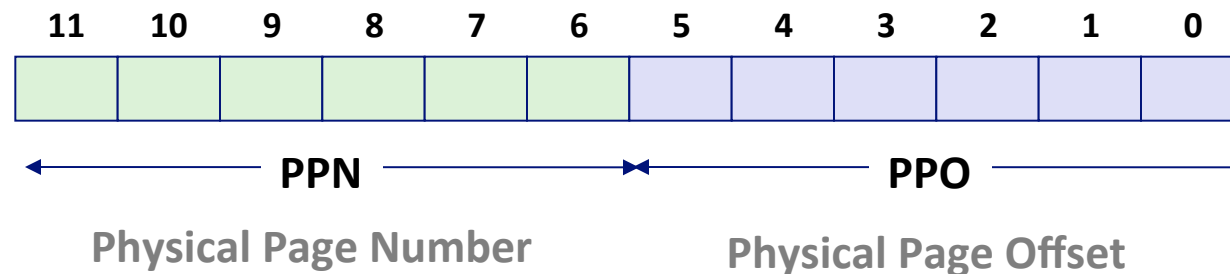
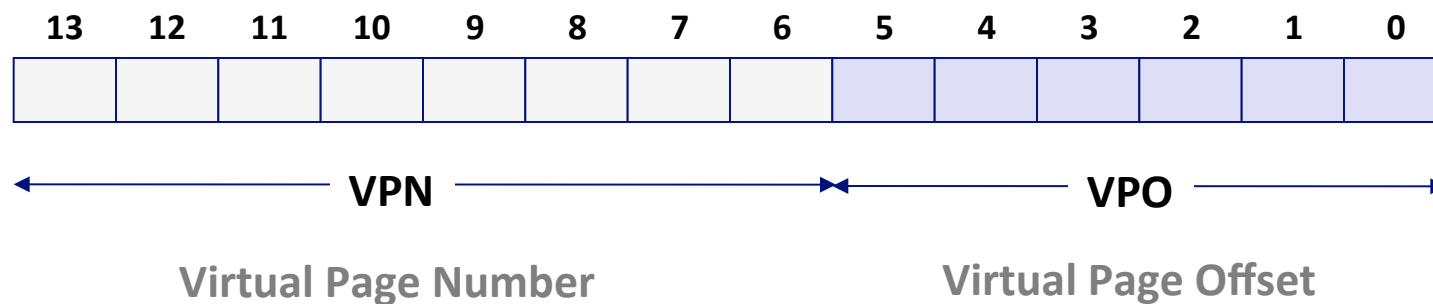
■ Components of the physical address (PA)

- 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

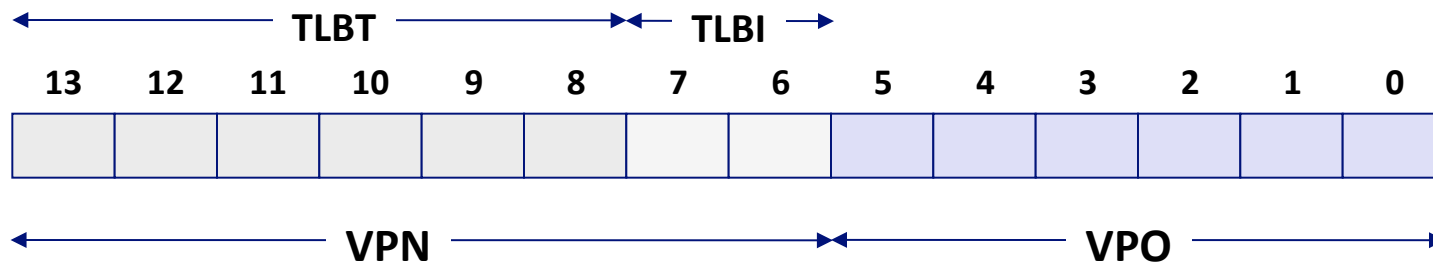
Only show first 16 entries (out of 256)

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

<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
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

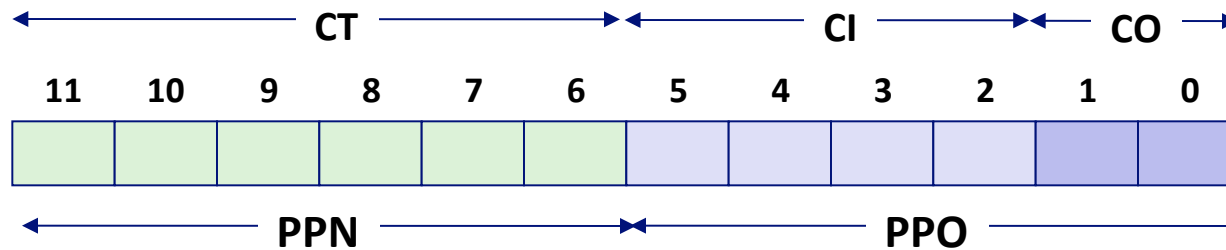
- 16 entries
- 4-way associative



<i>Set</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>
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

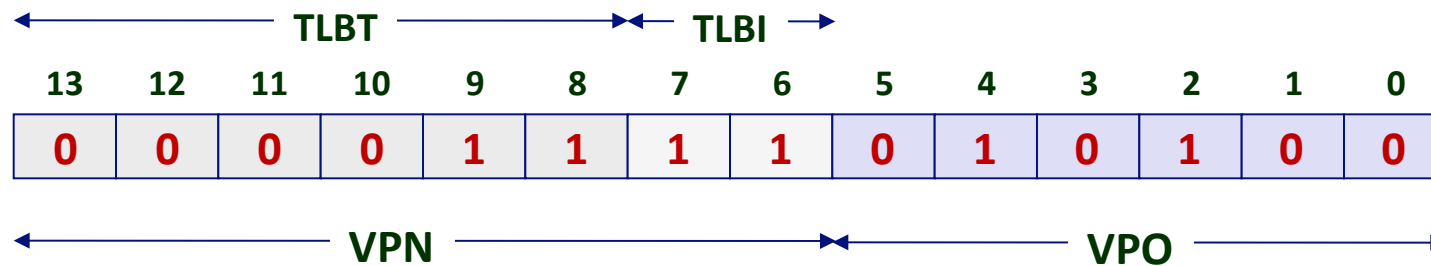


<i>Idx</i>	<i>Tag</i>	<i>Valid</i>	<i>B0</i>	<i>B1</i>	<i>B2</i>	<i>B3</i>
0	19	1	99	11	23	11
1	15	0	-	-	-	-
2	1B	1	00	02	04	08
3	36	0	-	-	-	-
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	-	-	-	-
7	16	1	11	C2	DF	03

<i>Idx</i>	<i>Tag</i>	<i>Valid</i>	<i>B0</i>	<i>B1</i>	<i>B2</i>	<i>B3</i>
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	-	-	-	-

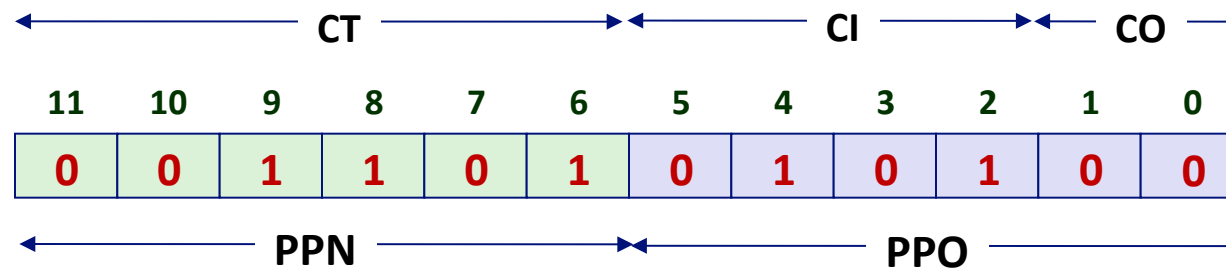
Address Translation Example #1

Virtual Address: 0x03D4



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

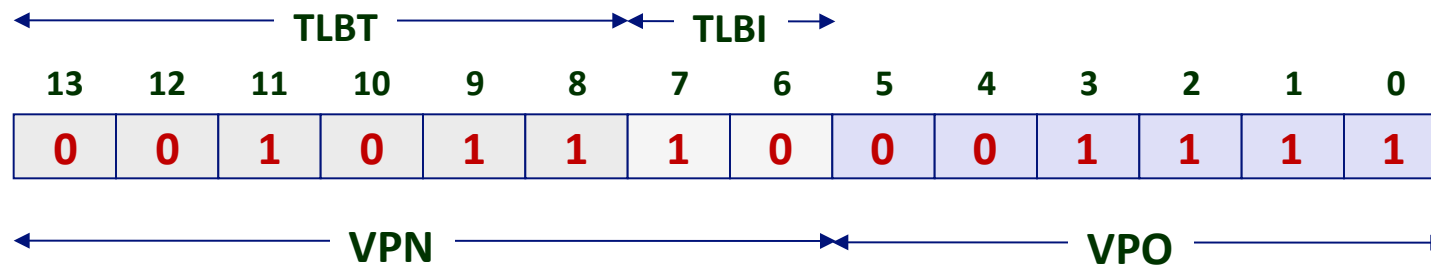
Physical Address



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

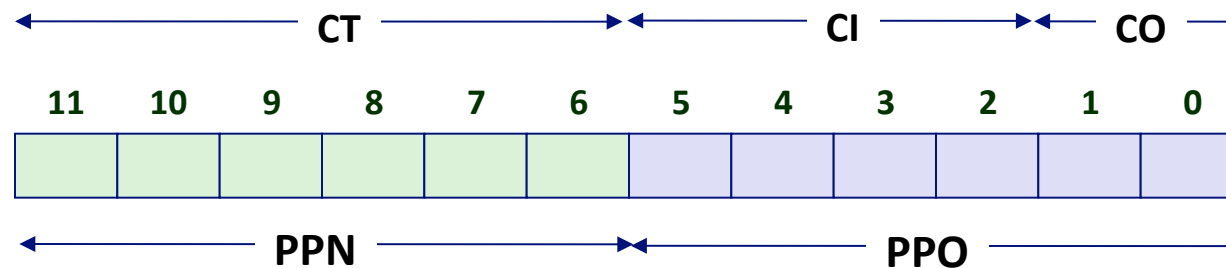
Address Translation Example #2

Virtual Address: 0x0B8F



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

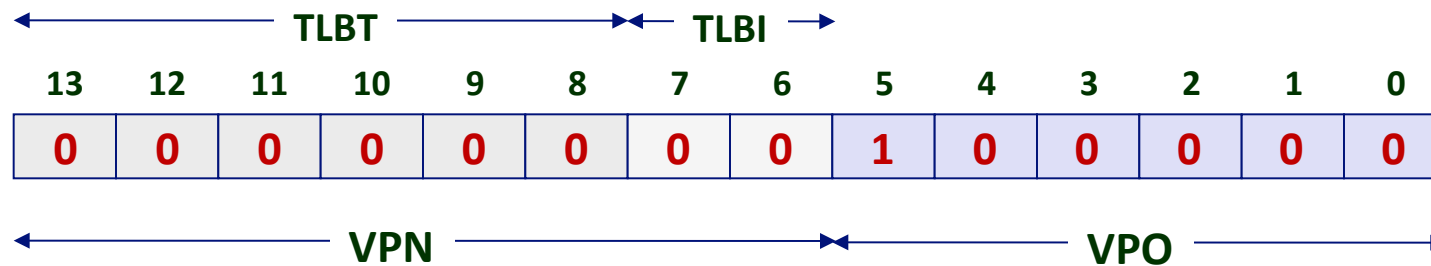
Physical Address



CO ___ CI ___ CT ___ Hit? ___ Byte: ___

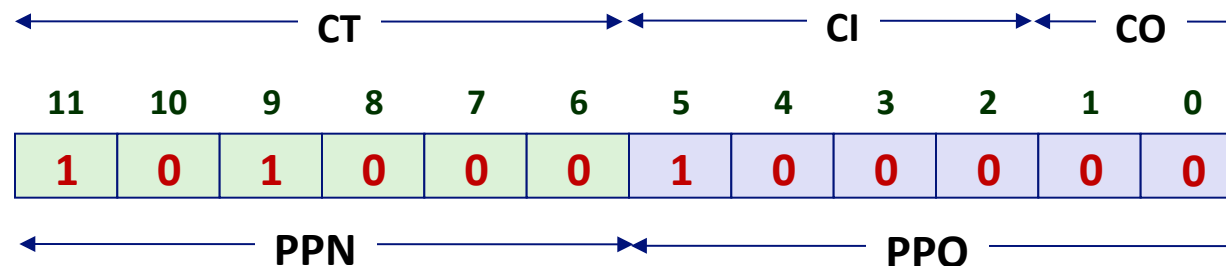
Address Translation Example #3

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

Reading Assignment

Diverse workloads and a wide range of hardware configurations compound the complexity of an operating system's memory management policies.

Virtual Memory Management in the VAX/VMS Operating System

Henry M. Levy and Peter H. Lipman, Digital Equipment Corporation

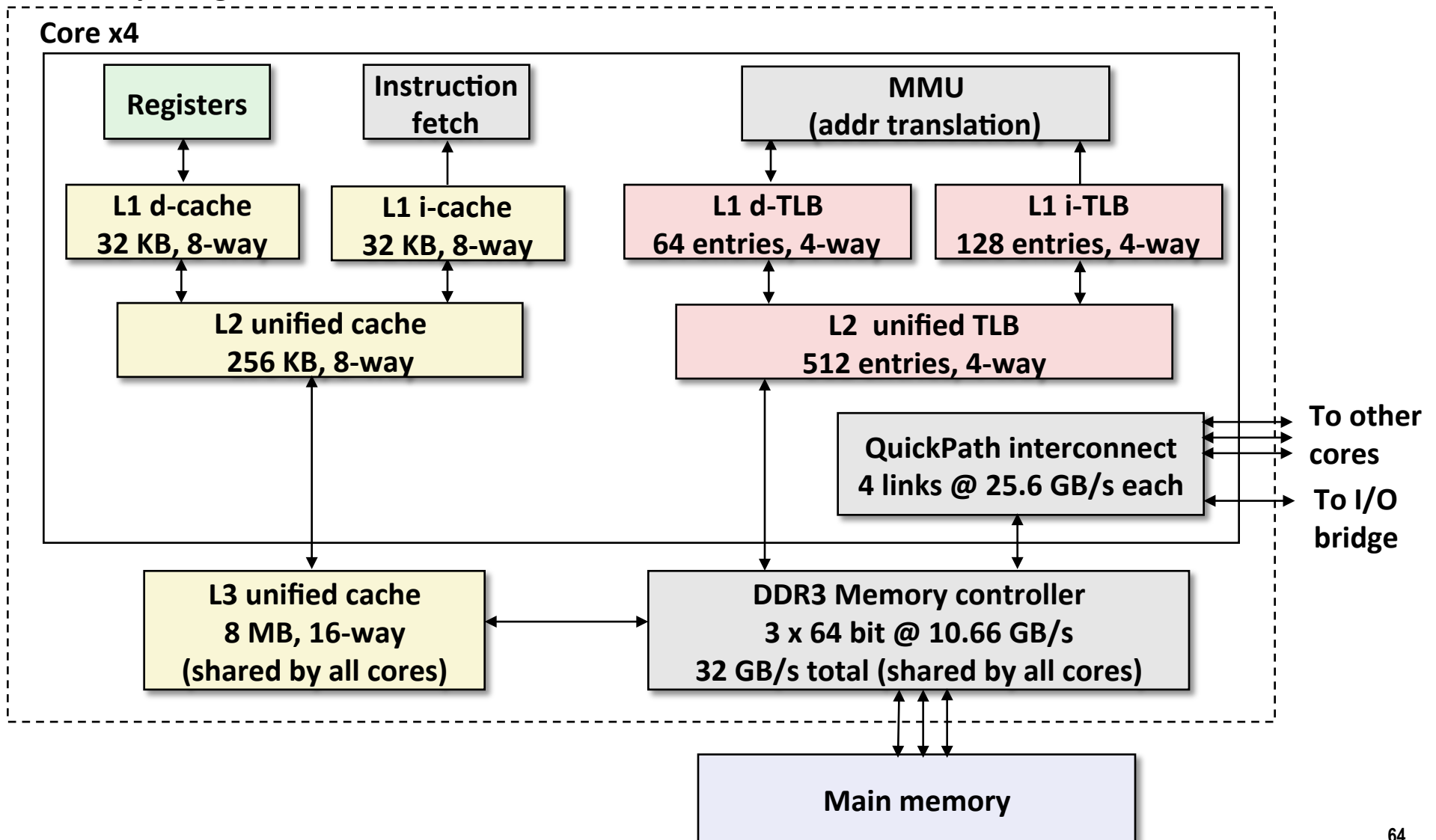
H. Levy and P. Lipman, Virtual Memory Management in the VAX/VMS Operating Systems, *Computing Surveys*, 4(3) September 1972, pp. 149-167

Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- Address translation
- Simple memory system example
- **Case study: Core i7/Linux memory system**
- Memory mapping

Intel Core i7 Memory System

Processor package



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)

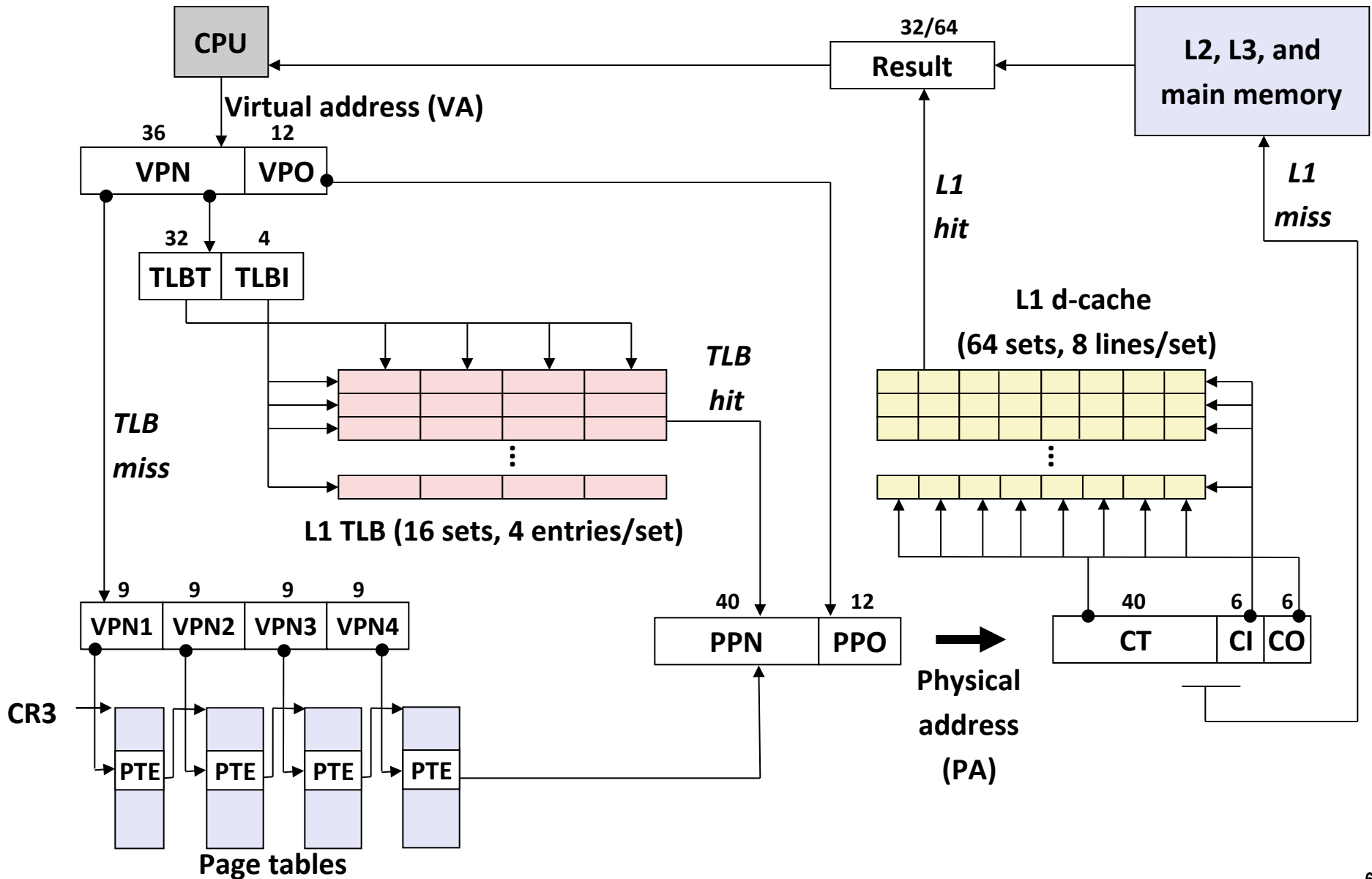
■ Components of the virtual address (VA)

- 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

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)

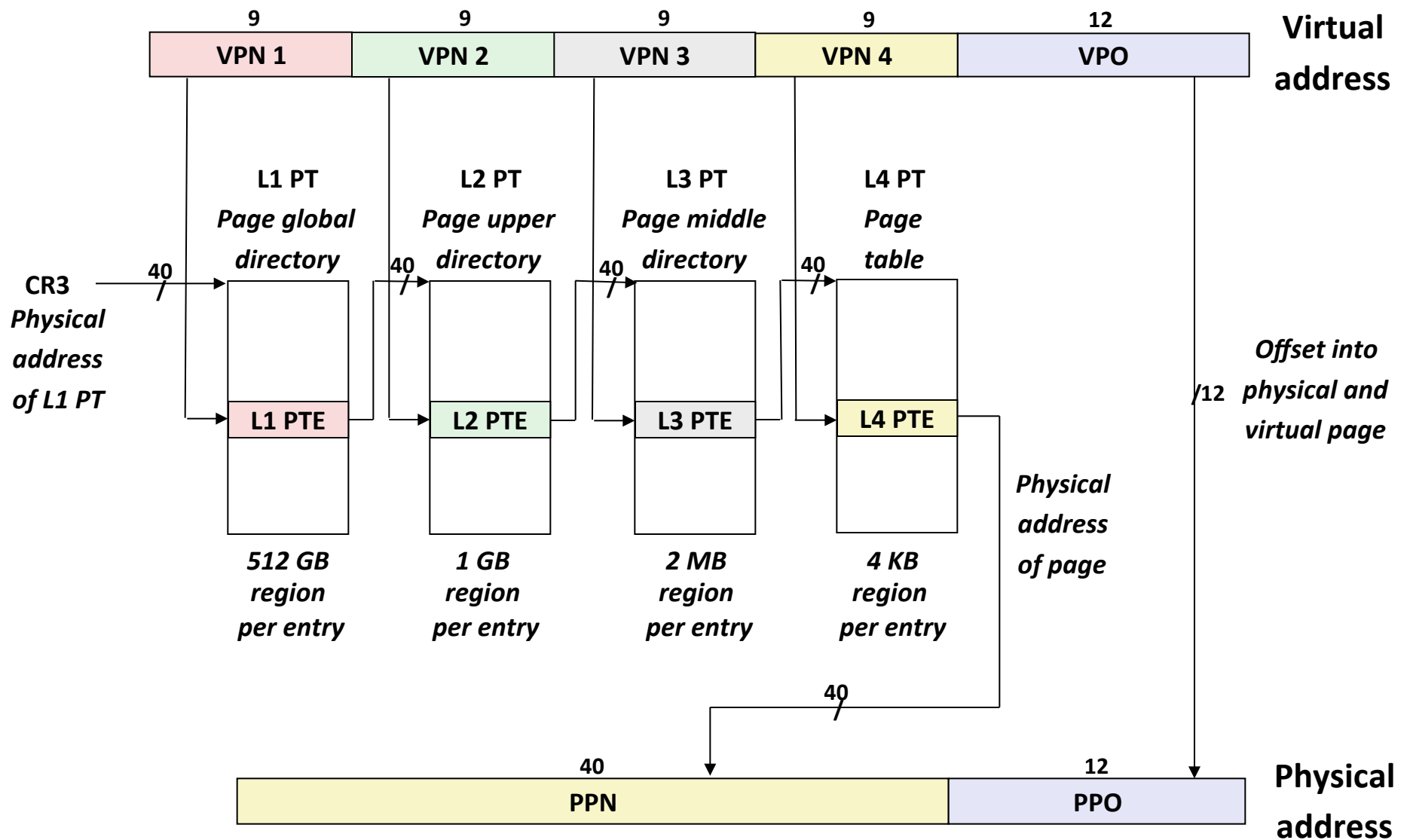
A: Reference bit (set by MMU on reads and writes, cleared by software)

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

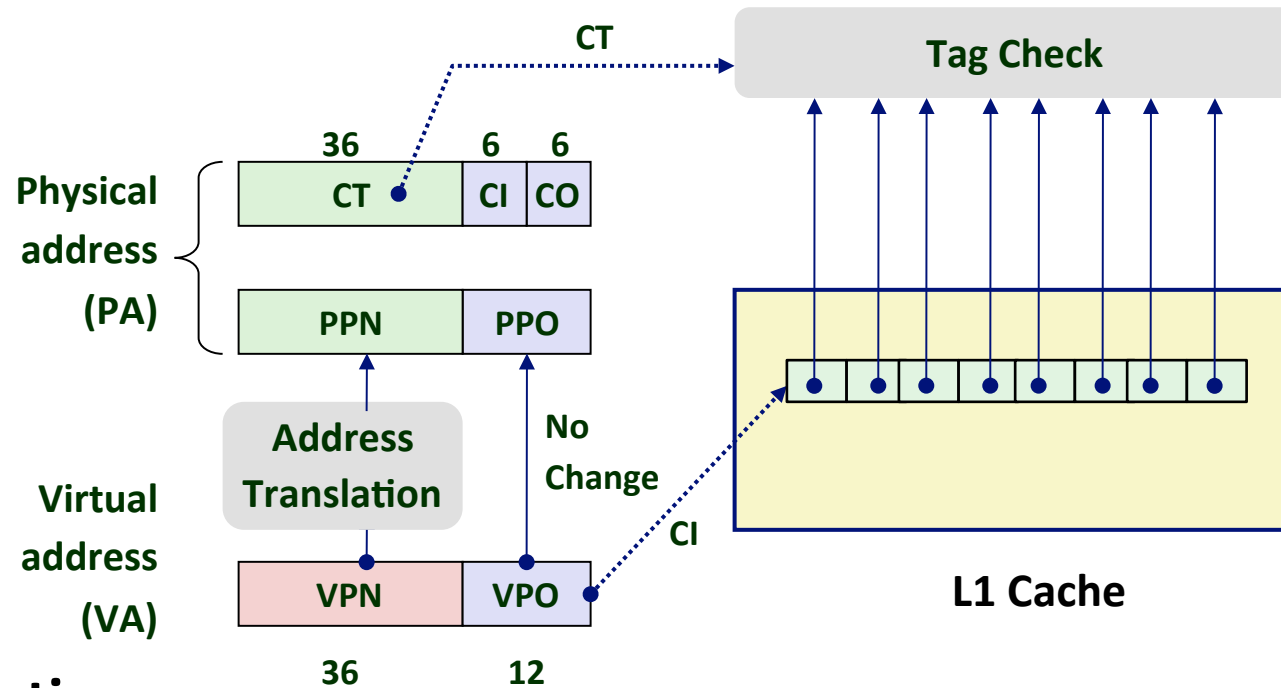
G: Global page (don't evict from TLB on task switch)

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

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

Today

- Address spaces
- (1) VM as a tool for caching
- (2) VM as a tool for memory management
- (3) VM as a tool for memory protection
- Address translation
- Simple memory system example
- Case study: Core i7/Linux memory system
- **Memory mapping**

Memory Mapping

- VM areas initialized by associating them with disk objects.
 - Process is known as *memory mapping*.

- Area can be backed by (i.e., get its initial values from) :
 - *Regular file* on disk (e.g., an executable object file)
 - Initial page bytes come from a section of a file
 - *Anonymous file* (e.g., nothing)
 - First fault will allocate a physical page full of 0's (*demand-zero page*)
 - Once the page is written to (*dirtied*), it is like any other page

- Dirty pages are copied back and forth between memory and a special *swap file*.

Demand paging

- ***Key point:*** no virtual pages are copied into physical memory until they are referenced!
 - Known as ***demand paging***
- **Crucial for time and space efficiency**

User-Level Memory Mapping

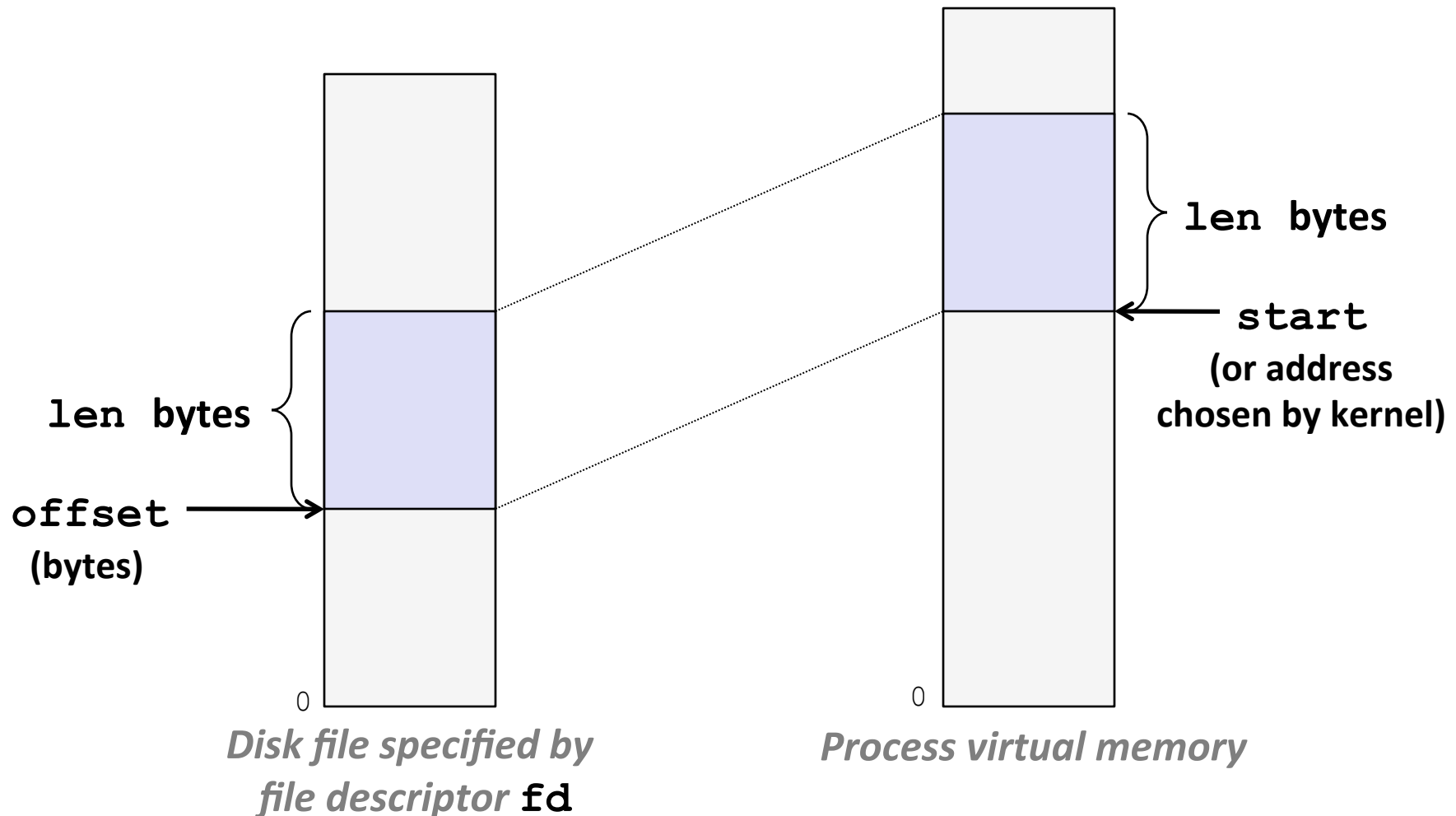
```
void *mmap(void *start, int len,  
           int prot, int flags, int fd, int offset)
```

- **Map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start`**
 - **`start`**: may be 0 for “pick an address”
 - **`prot`**: `PROT_READ`, `PROT_WRITE`, ...
 - **`flags`**: `MAP_ANON`, `MAP_PRIVATE`, `MAP_SHARED`, ...

- **Return a pointer to start of mapped area (may not be `start`)**

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```



Using mmap to Copy Files

- Copying without transferring data to user space .

```
#include "csapp.h"

/*
 * mmapcopy - uses mmap to copy
 *            file fd to stdout
 */
void mmapcopy(int fd, int size)
{
    /* Ptr to mem-mapped VM area */
    char *bufp;

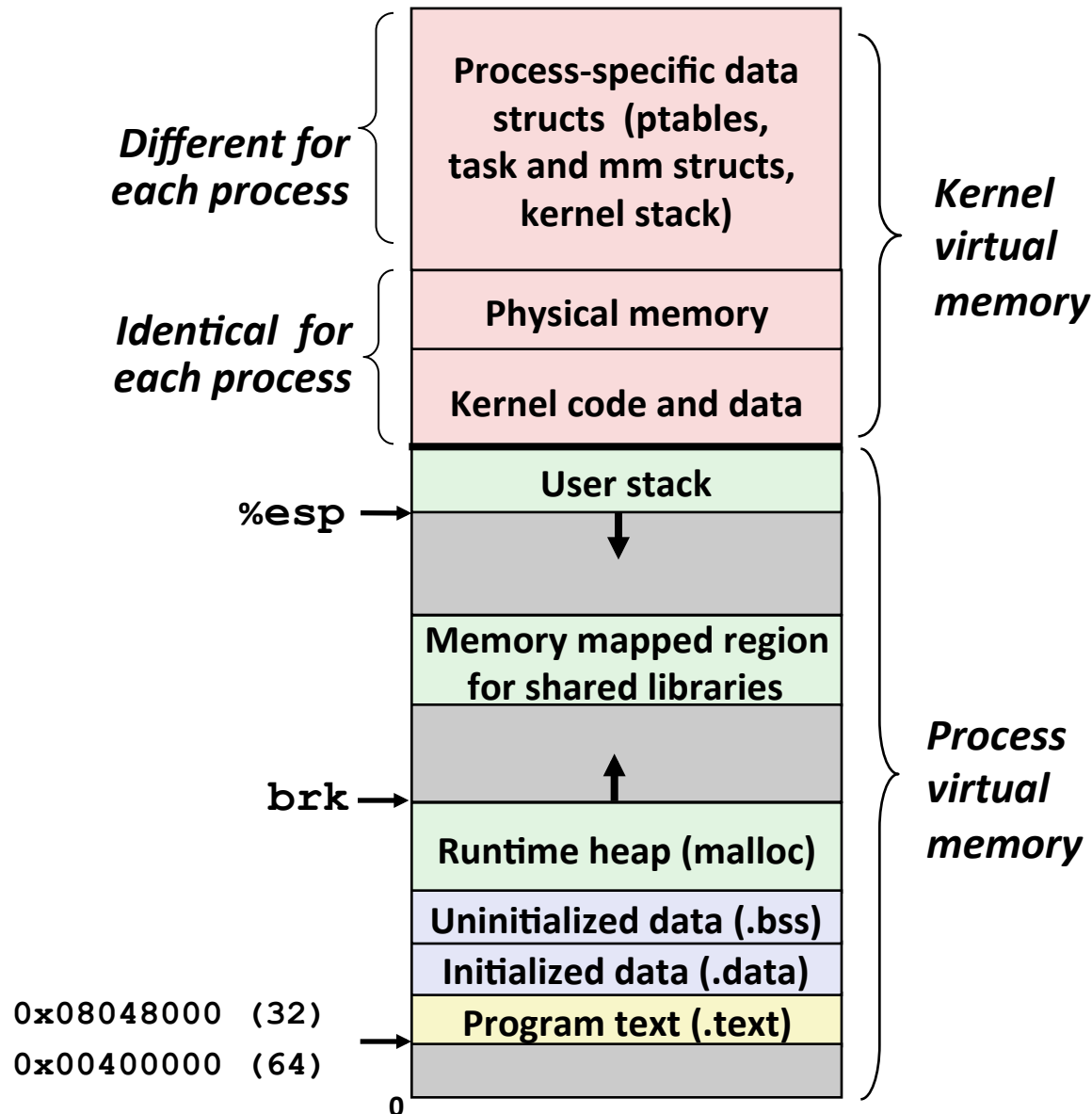
    bufp = Mmap(NULL, size,
                PROT_READ,
                MAP_PRIVATE, fd, 0);
    Write(1, bufp, size);
    return;
}
```

```
/* mmapcopy driver */
int main(int argc, char **argv)
{
    struct stat stat;
    int fd;

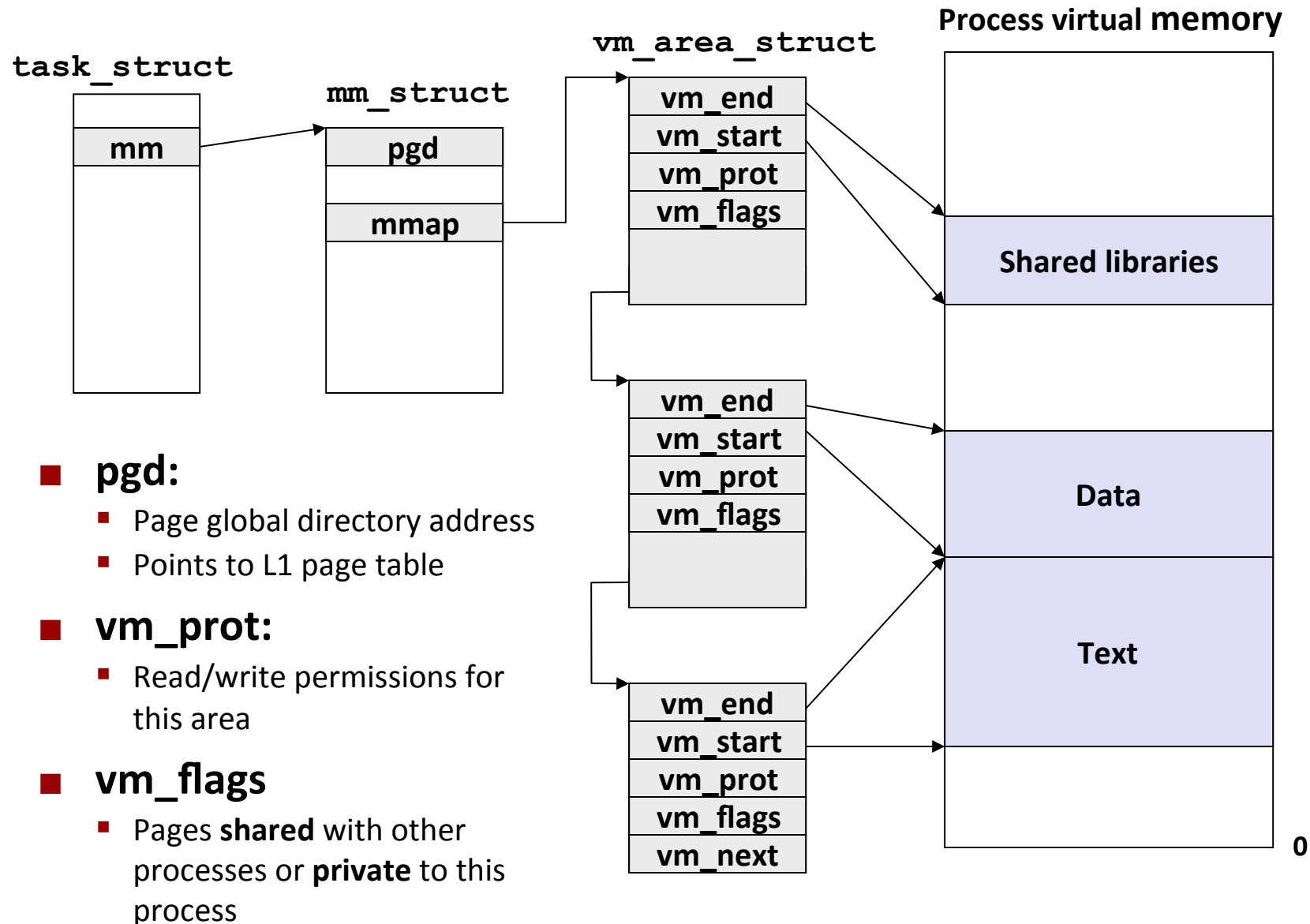
    /* Check for required cmdline arg */
    if (argc != 2) {
        printf("usage: %s <filename>\n",
              argv[0]);
        exit(0);
    }

    /* Copy the input arg to stdout */
    fd = Open(argv[1], O_RDONLY, 0);
    Fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
    exit(0);
}
```

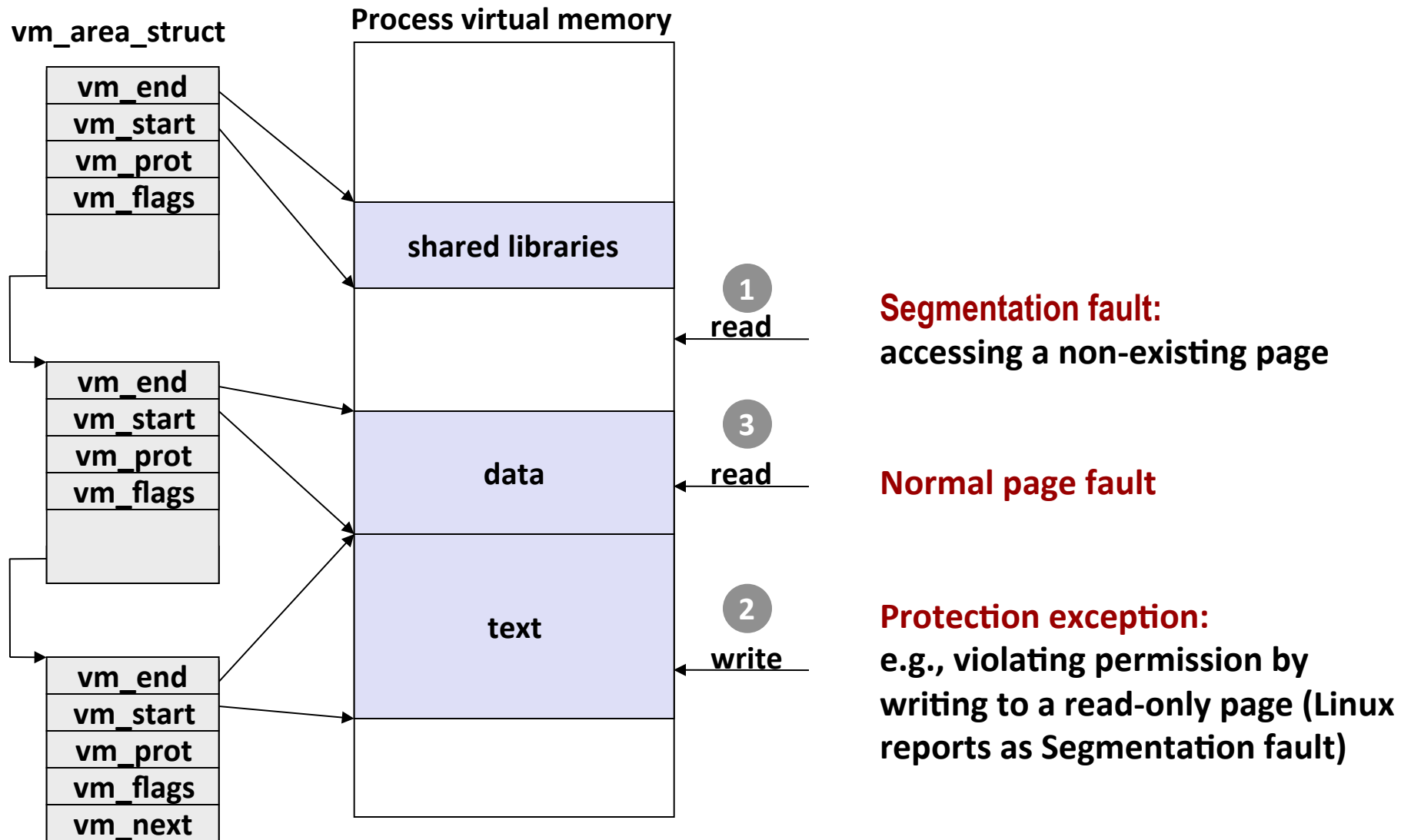
Virtual Memory of a Linux Process



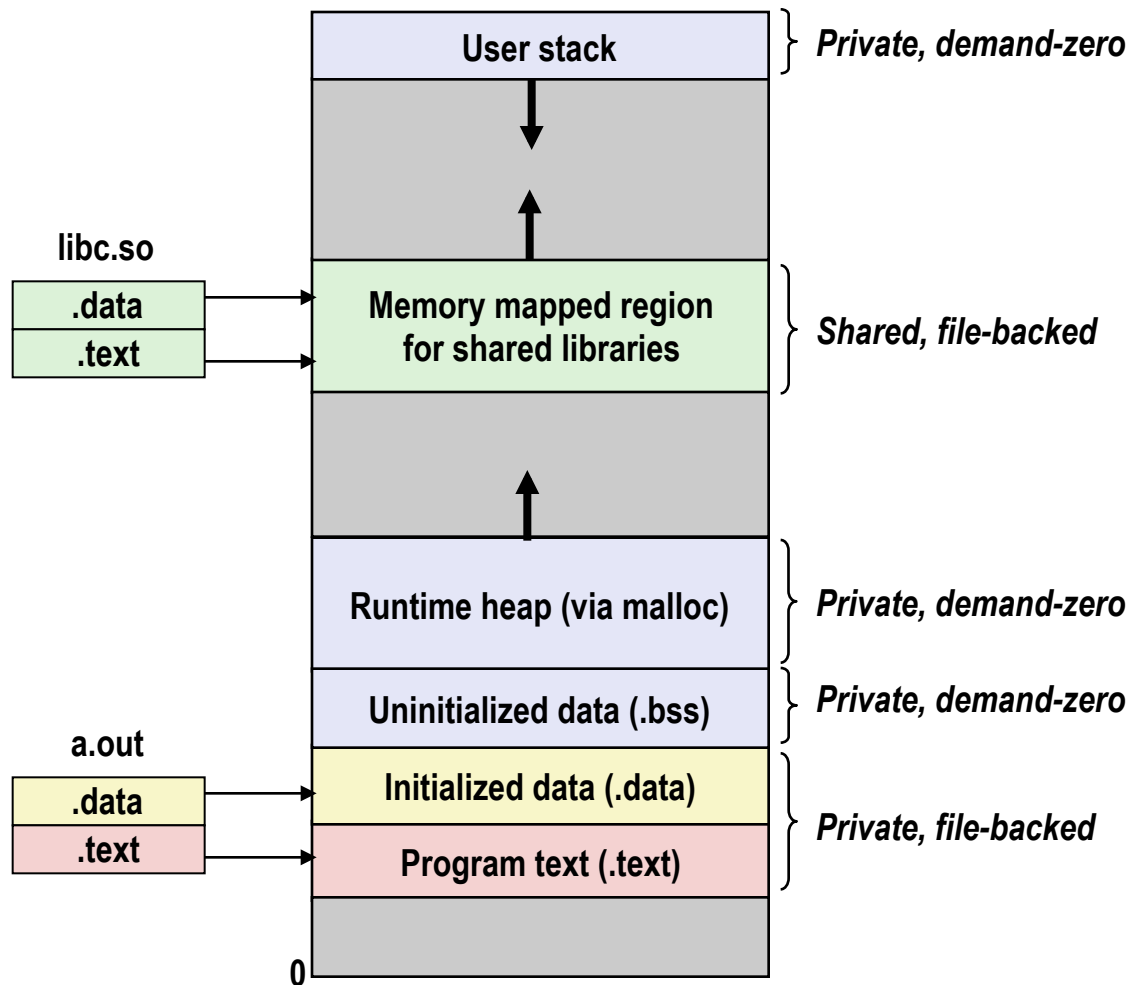
Linux Organizes VM as Collection of “Areas”



Linux Page Fault Handling



The `execve` Function Revisited



- To load and run a new program `a.out` in the current process using `execve`:
- Free `vm_area_struct`'s and page tables for old areas
- Create `vm_area_struct`'s and page tables for new areas
 - Programs and initialized data backed by object files.
 - `.bss` and stack backed by anonymous files.
- Set PC to entry point in `.text`
 - Linux will fault in code and data pages as needed.