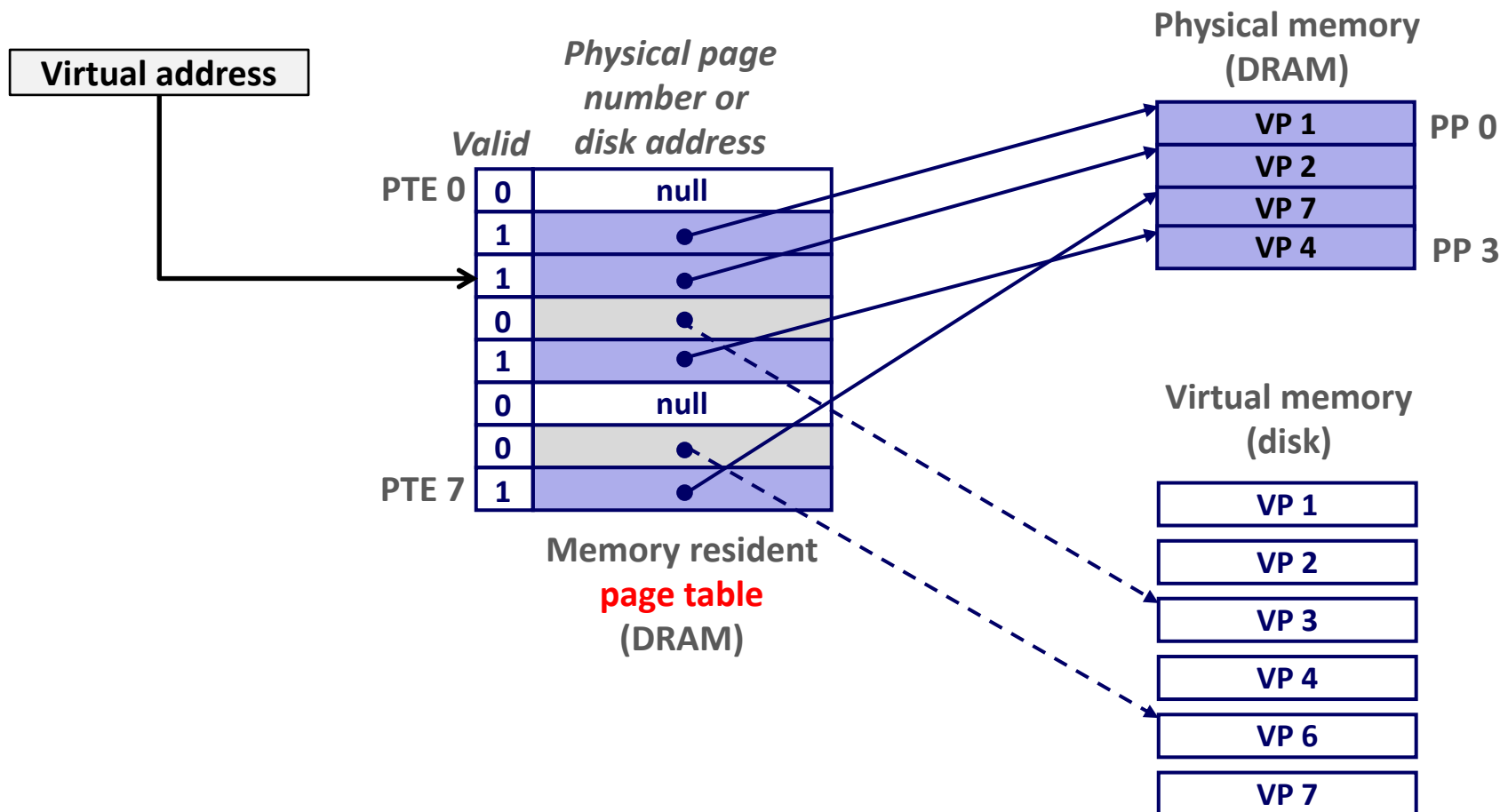




# Virtual Memory: Systems

18-213/18-613: Introduction to Computer Systems  
13<sup>th</sup> Lecture, June 10, 2024

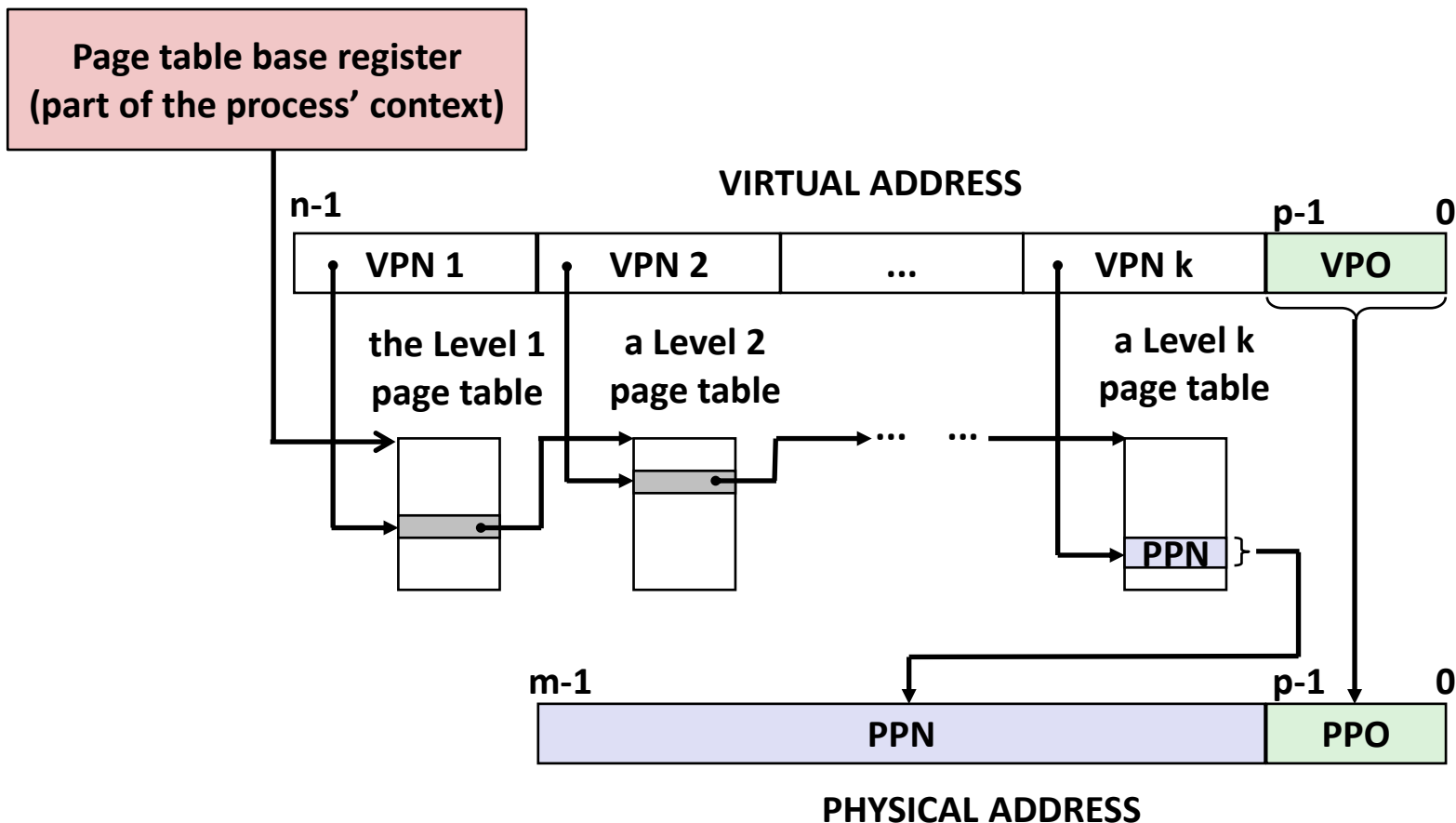
# Review: Virtual Memory & Physical Memory



- A **page table** contains page table entries (PTEs) that map virtual pages to physical pages.

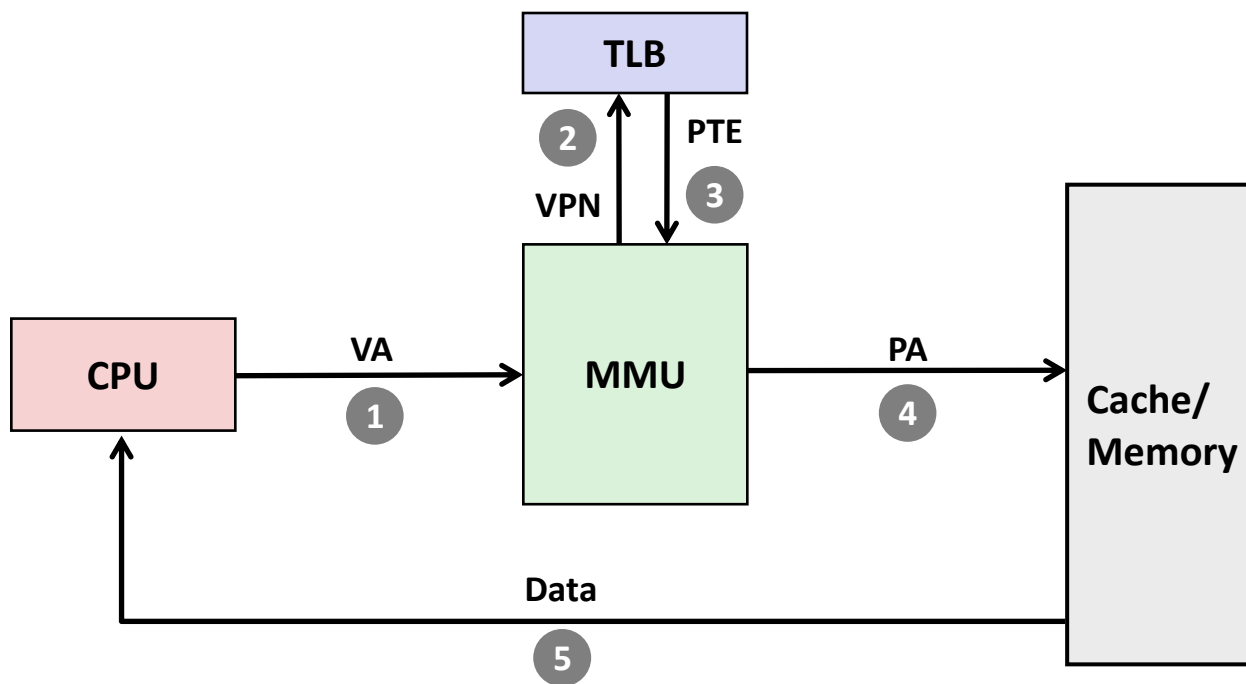
# Review: Translating with a k-level Page Table

- Having multiple levels greatly reduces total page table size



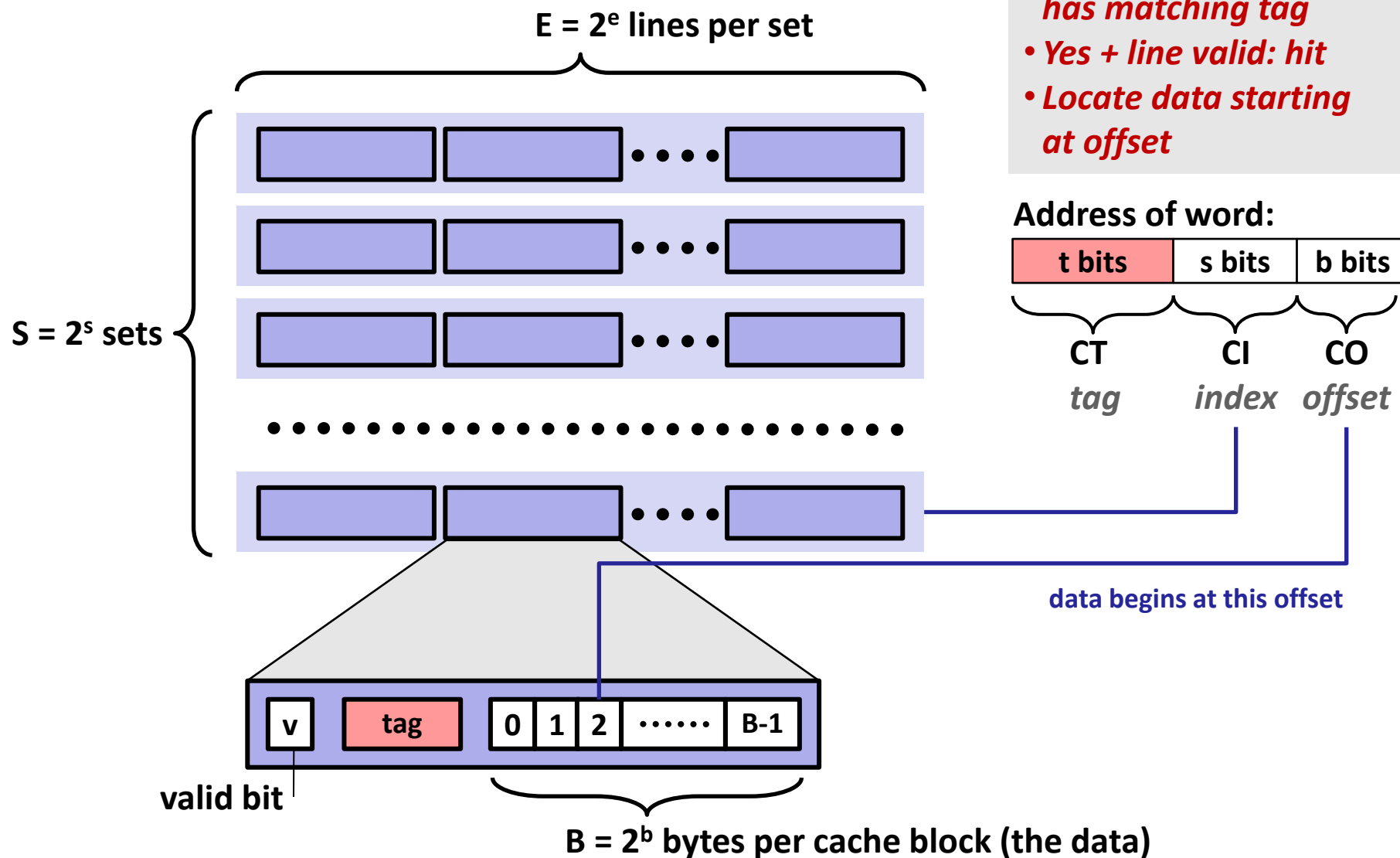
# Review: Translation Lookaside Buffer (TLB)

- A small cache of page table entries with fast access by MMU



Typically, a **TLB hit** eliminates the  $k$  memory accesses required to do a page table lookup.

# Recall: Set Associative Cache



# Review of Symbols

## Basic Parameters

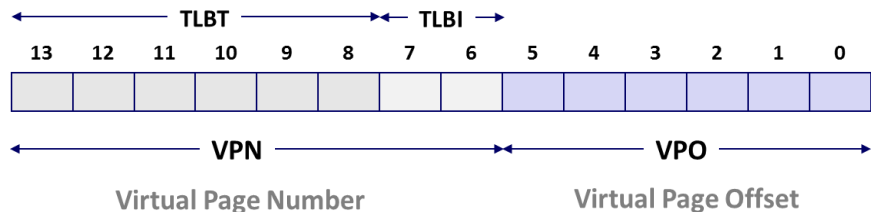
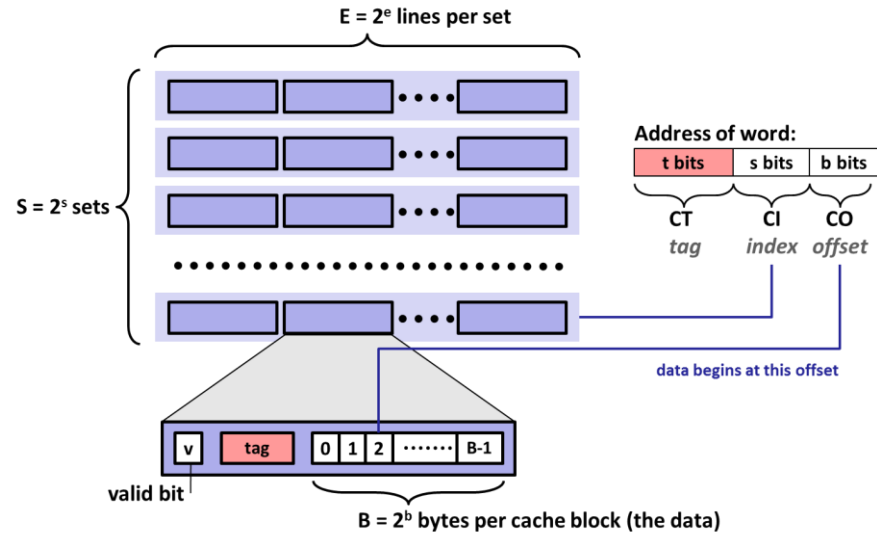
- $N = 2^n$ : Number of addresses in virtual address space
- $M = 2^m$ : Number of addresses in physical address space
- $P = 2^p$ : Page size (bytes)

## Components of the *virtual address* (VA)

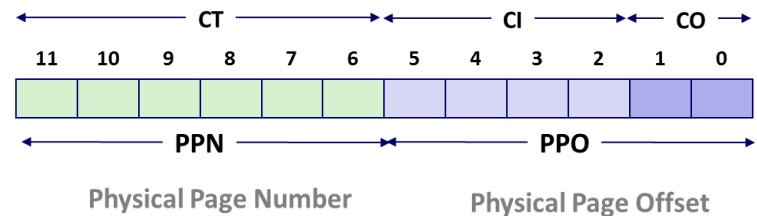
- TLBI: TLB index
- TLBT: TLB tag
- VPO: Virtual page offset
- VPN: Virtual page number

## 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



(bits per field for our simple example)



# Today

- **Simple memory system example** CSAPP 9.6.4
- **Case study: Core i7/Linux memory system** CSAPP 9.7
- **Memory mapping** CSAPP 9.8



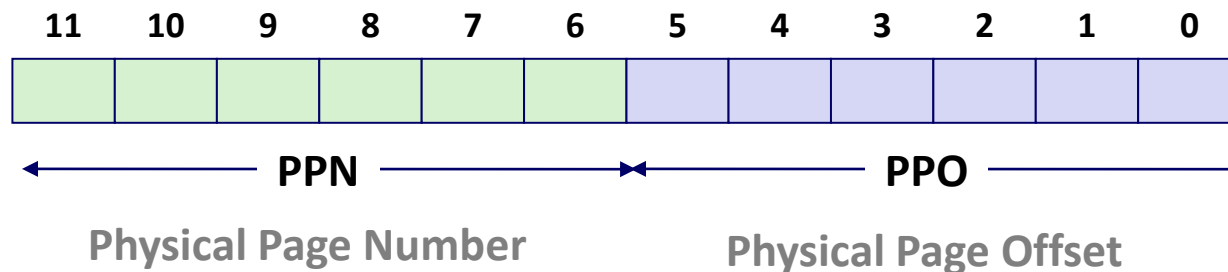
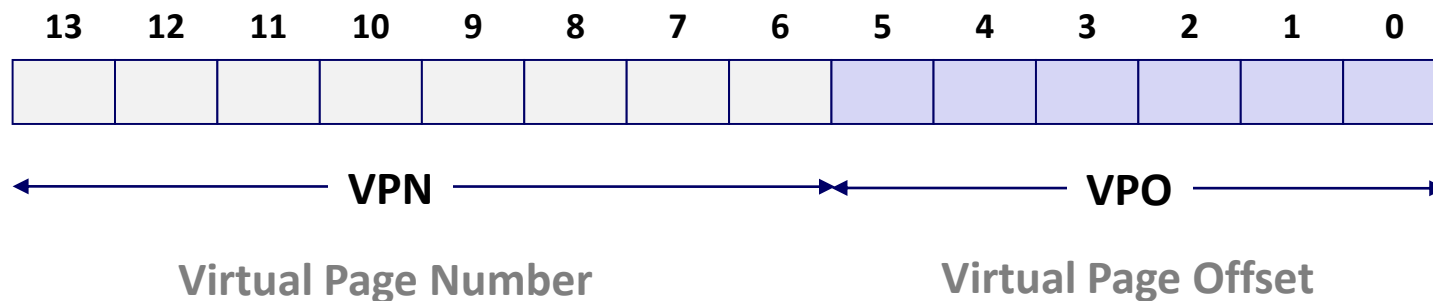
# Simple Memory System Example

## ■ Addressing

- 14-bit virtual addresses
- 12-bit physical address
- Page size = 64 bytes

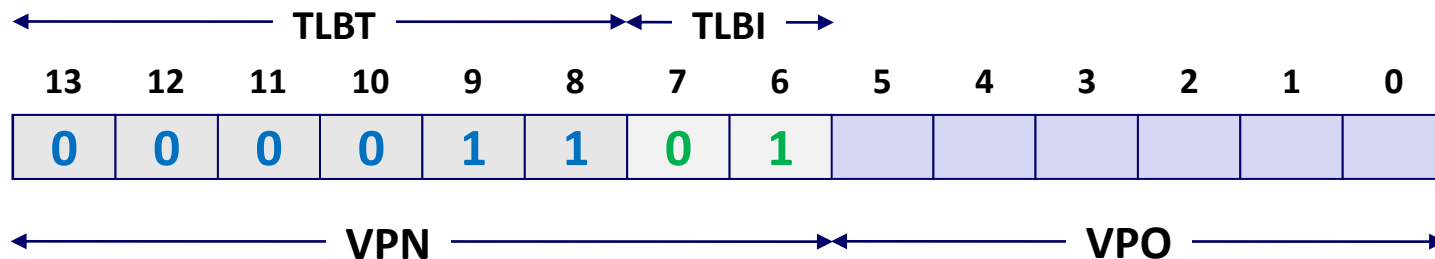
Why is the  
VPO 6 bits?

Why is the  
VPN 8 bits?



# Simple Memory System TLB

- 16 entries
- 4-way associative



$$\text{VPN} = 0b1101 = 0x0D$$

## Translation Lookaside Buffer (TLB)

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

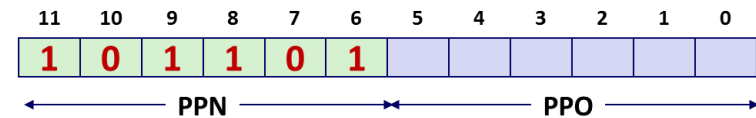
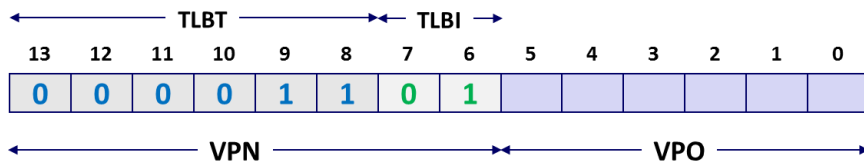
# Simple Memory System Page Table

Only showing the 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

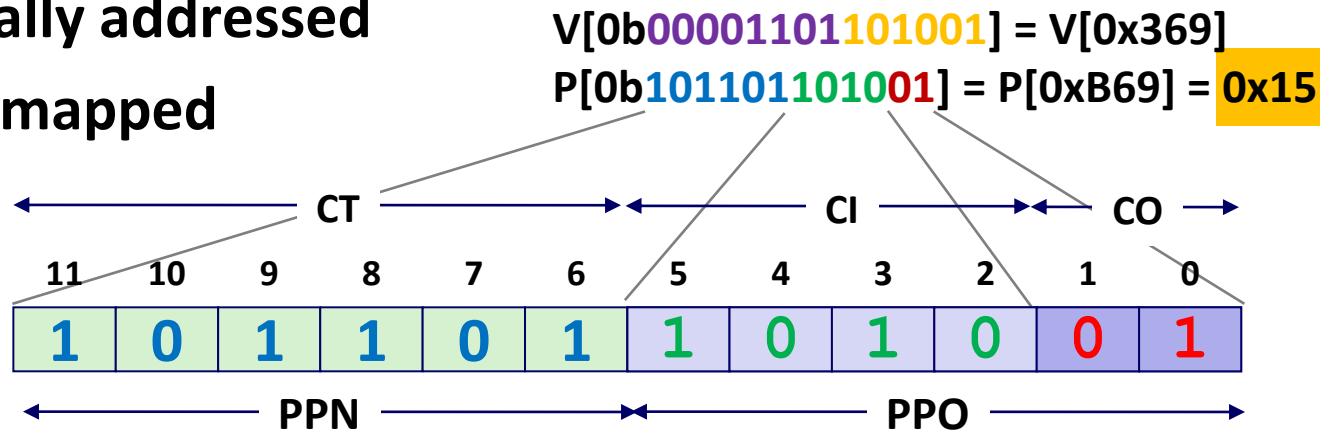
<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

0x0D → 0x2D



# Simple Memory System Cache

- 16 lines, 4-byte cache line size
- Physically addressed
- Direct mapped

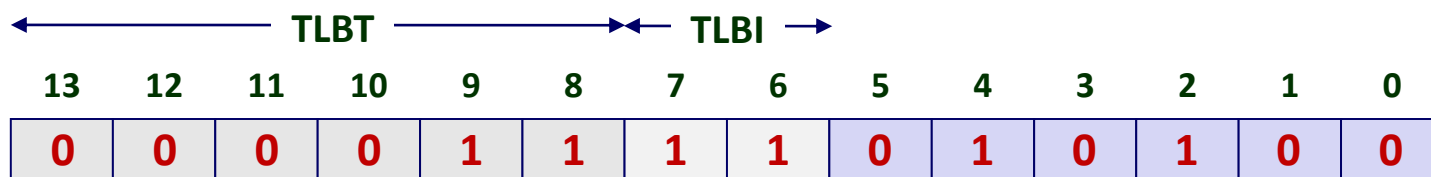


<i>Idx</i>	<i>Tag</i>	<i>Valid</i>	<i>B0</i>	<i>B1</i>	<i>B2</i>	<i>B3</i>
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

<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

Virtual Address: 0x03D4

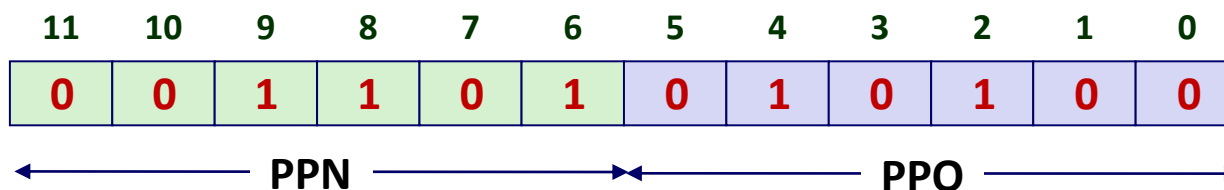


VPN 0x0F TLBI 0x3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

TLB

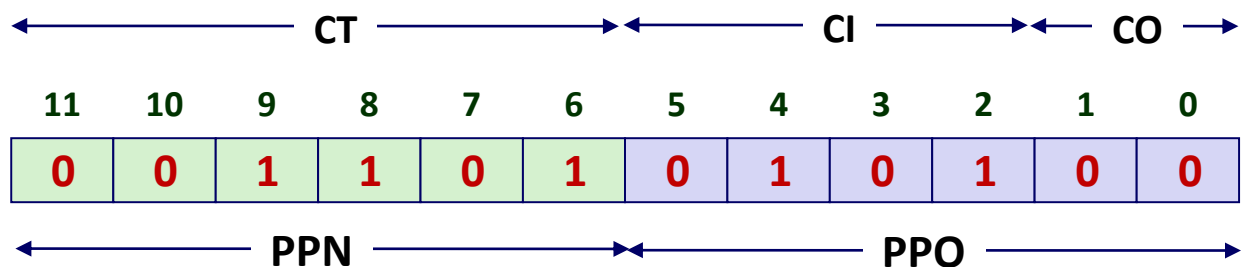
Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	-	0	09	0D	1	00	-	0	07	02	1
1	03	2D	1	02	-	0	04	-	0	0A	-	0
2	02	-	0	08	-	0	06	-	0	03	-	0
3	07	-	0	03	0D	1	0A	34	1	02	-	0

Physical Address



# Address Translation Example

## Physical Address



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

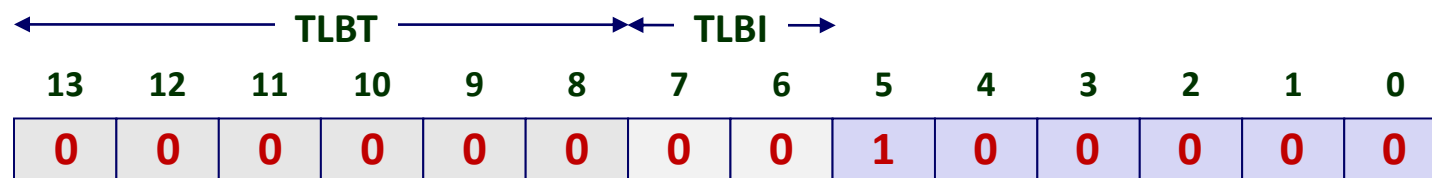
## Cache

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	-	-	-	-

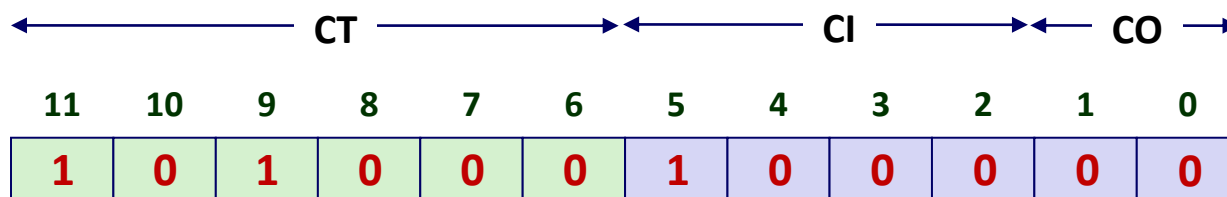
# Address Translation Example: TLB/Cache Miss

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? \_\_ Byte: \_\_\_\_\_

Page table

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

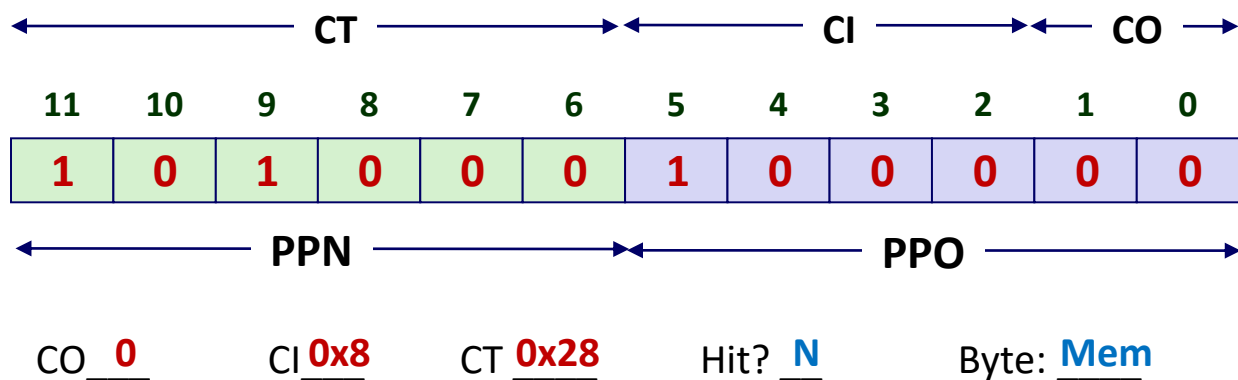
# Address Translation Example: TLB/Cache Miss

Cache

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	–	–	–	–

## Physical Address





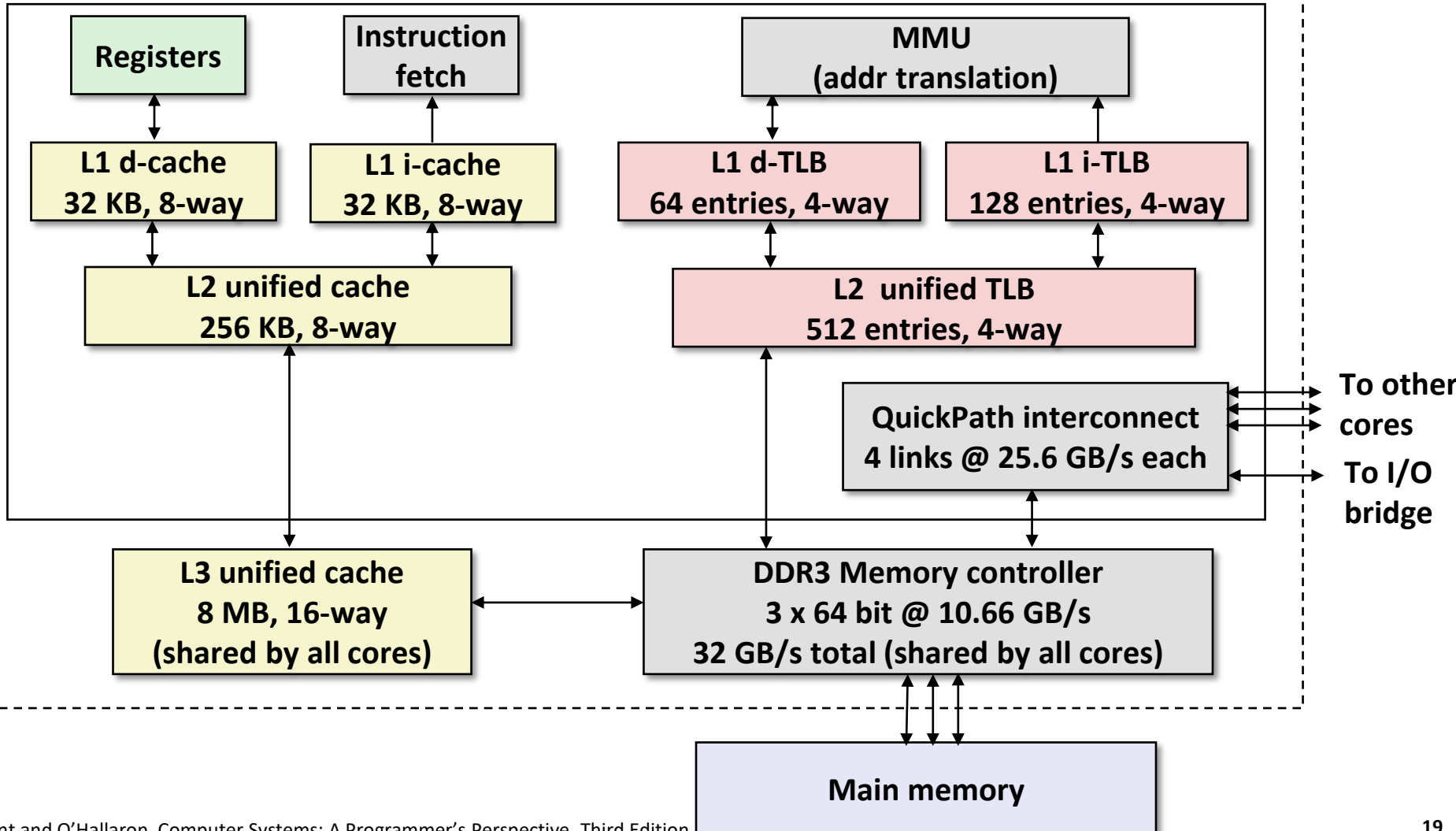
# Today

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

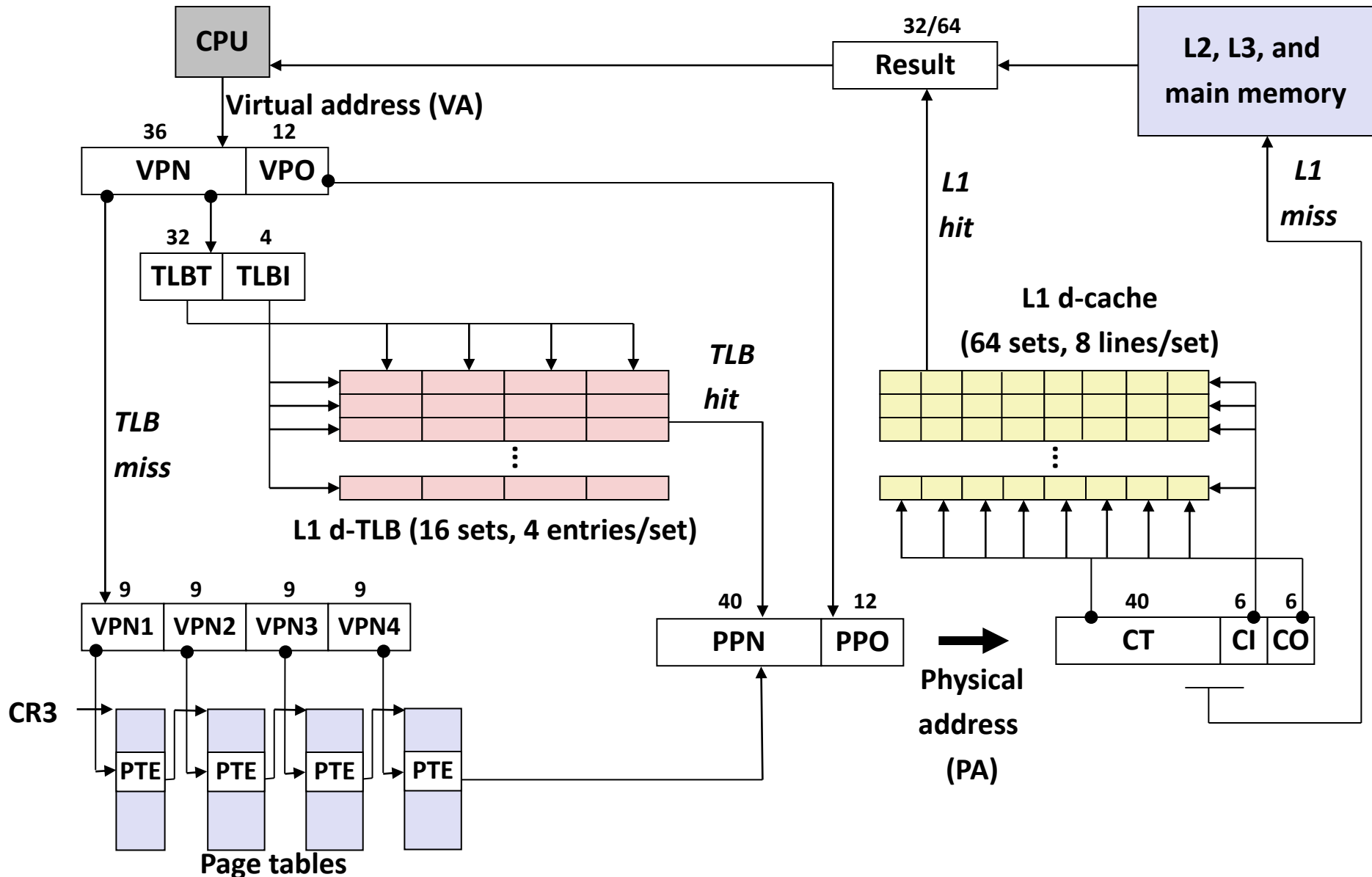
# Intel Core i7 Memory System

Processor package

Core x4



# 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)

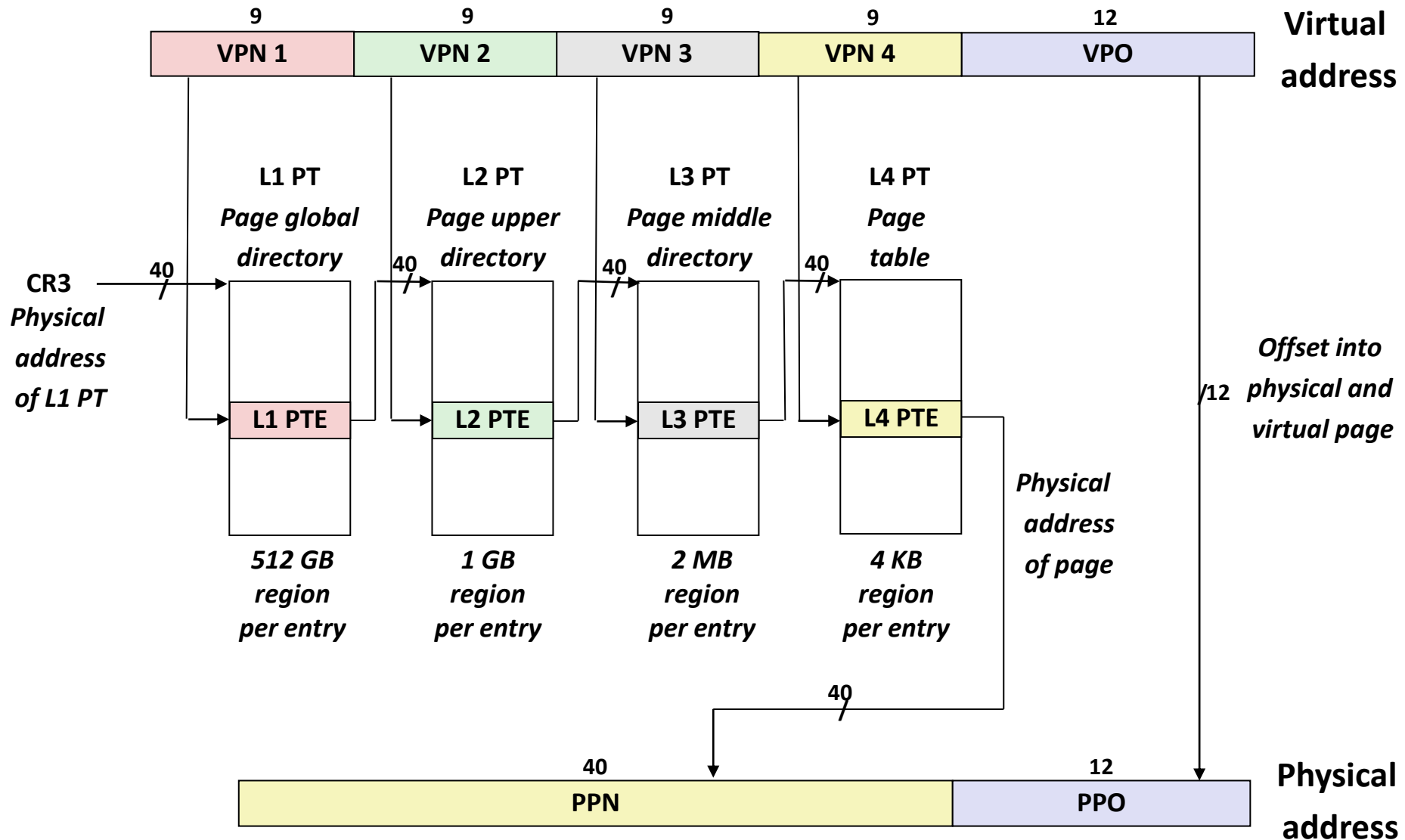
**D:** Dirty bit (set by MMU on writes, cleared by software)

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

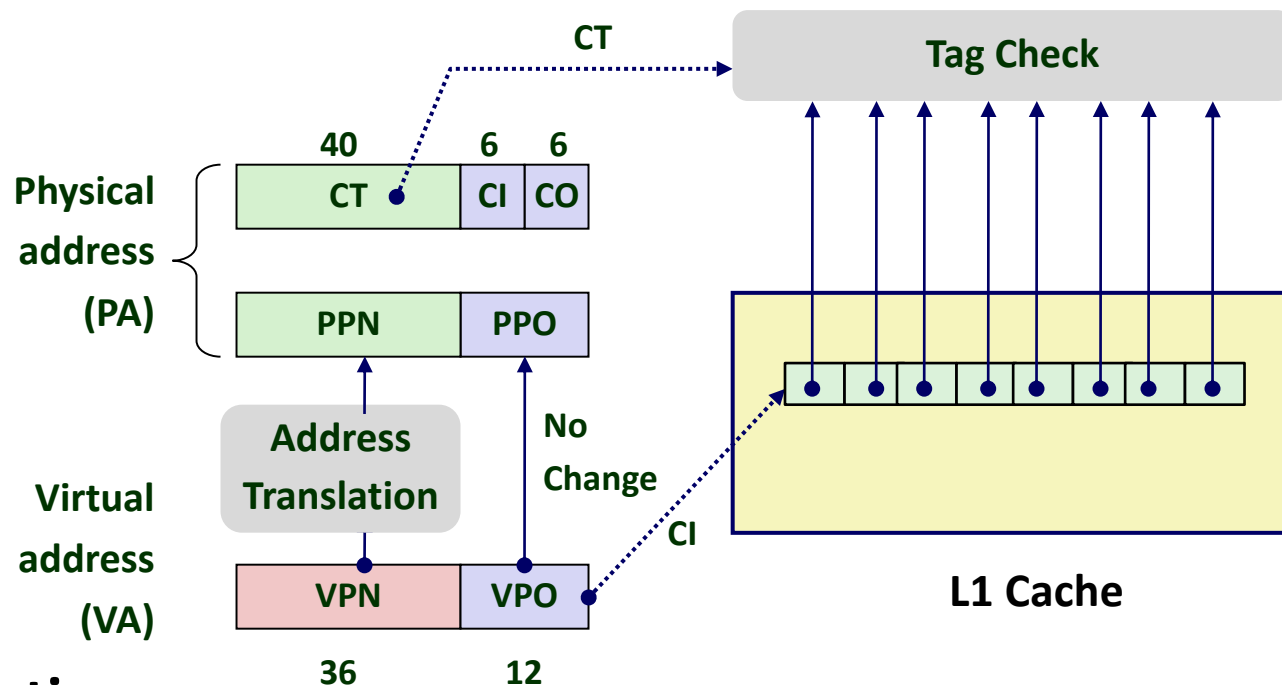
**Page physical base address:** 40 most significant bits of physical page address  
(forces pages to be 4KB aligned)

**XD:** Disable or enable instruction fetches from this page.

# Core i7 Page Table Translation



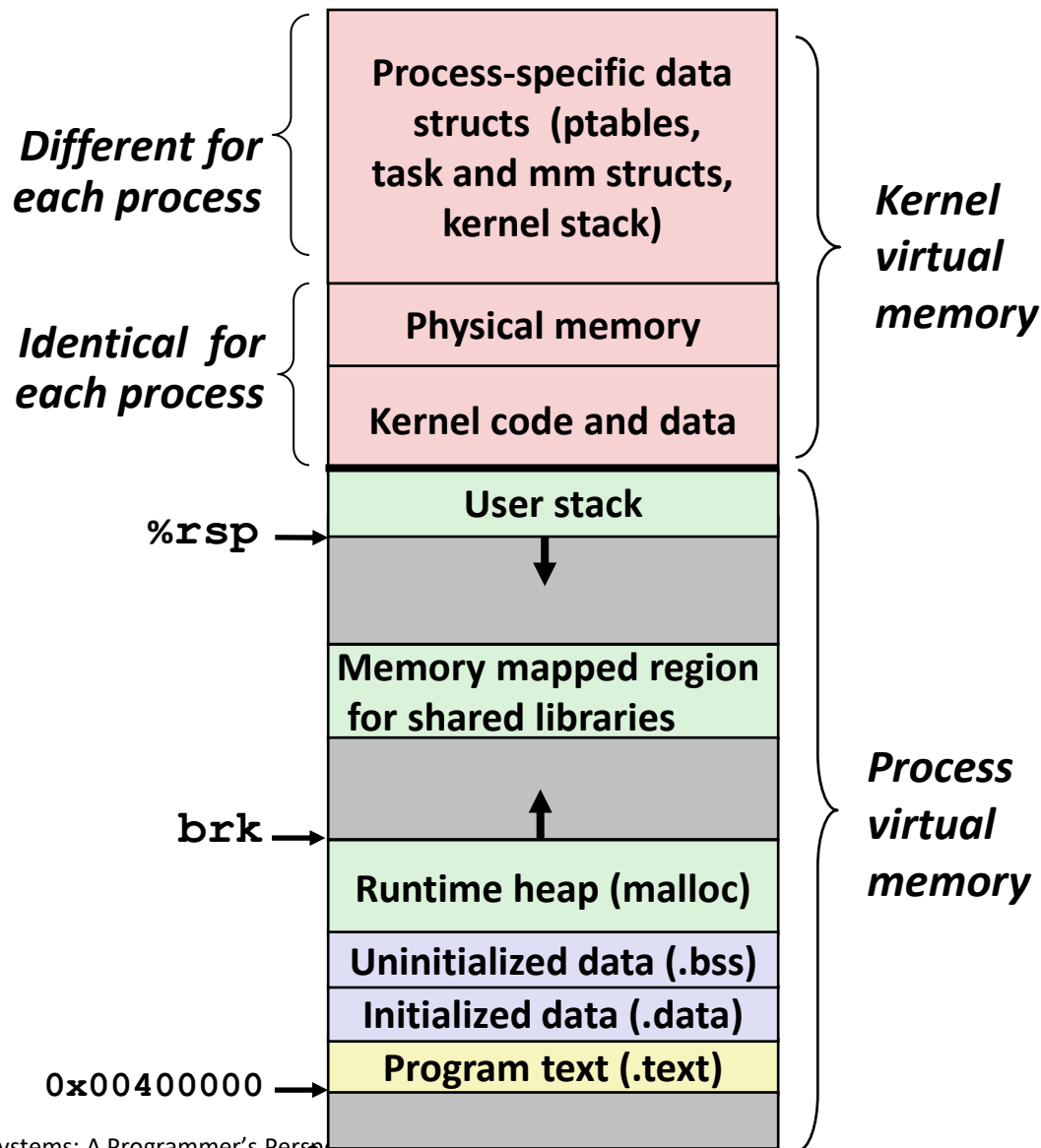
# Cute Trick for Speeding Up L1 Access



## ■ Observation

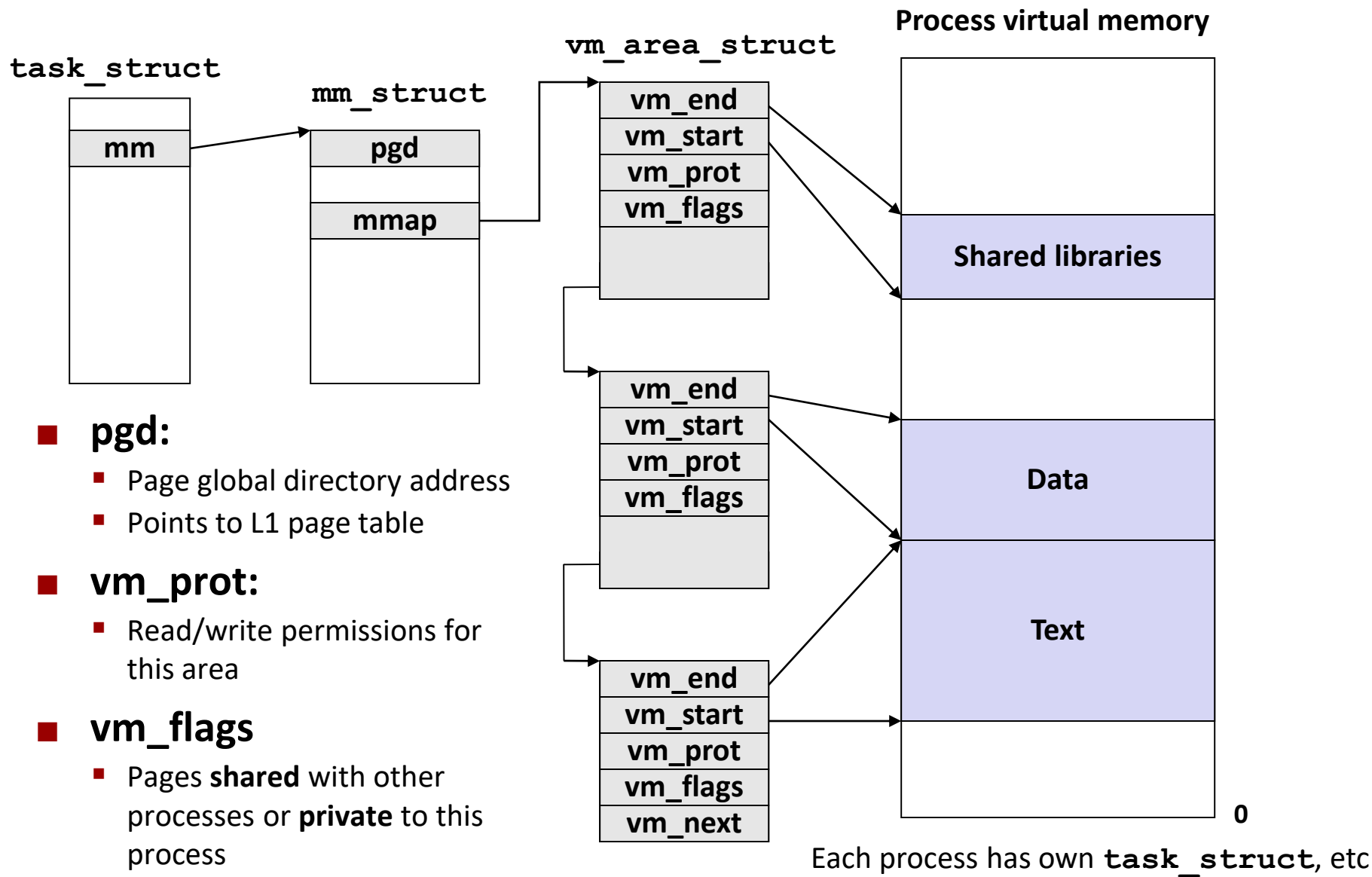
- Bits that determine CI identical in virtual and physical address
- Can index into cache while address translation taking place
- Generally we hit in TLB, so PPN bits (CT bits) available quickly
- ***“Virtually indexed, physically tagged”***
- Cache carefully sized to make this possible

# Virtual Address Space of a Linux Process

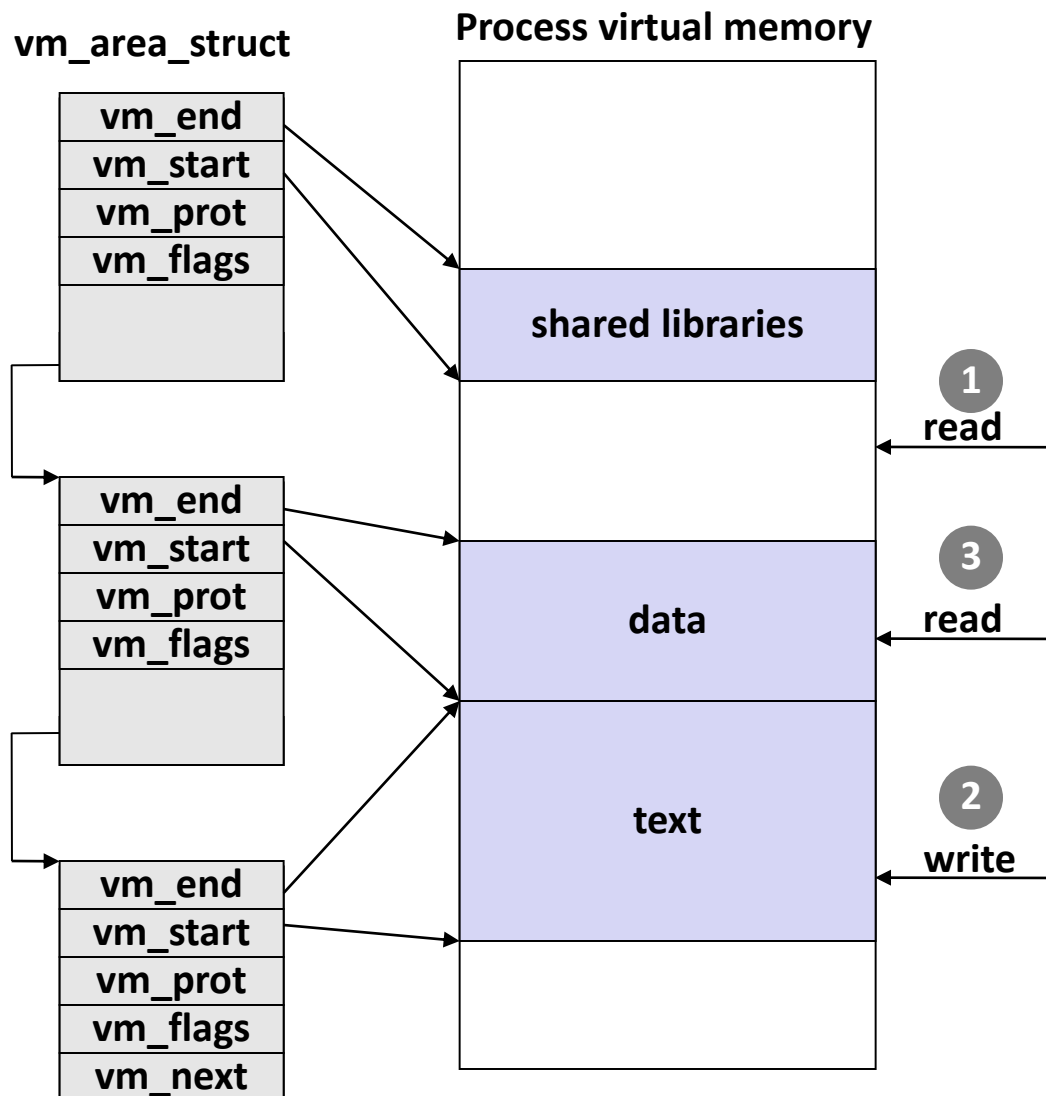




# Linux Organizes VM as Collection of “Areas”



# 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)

# Today

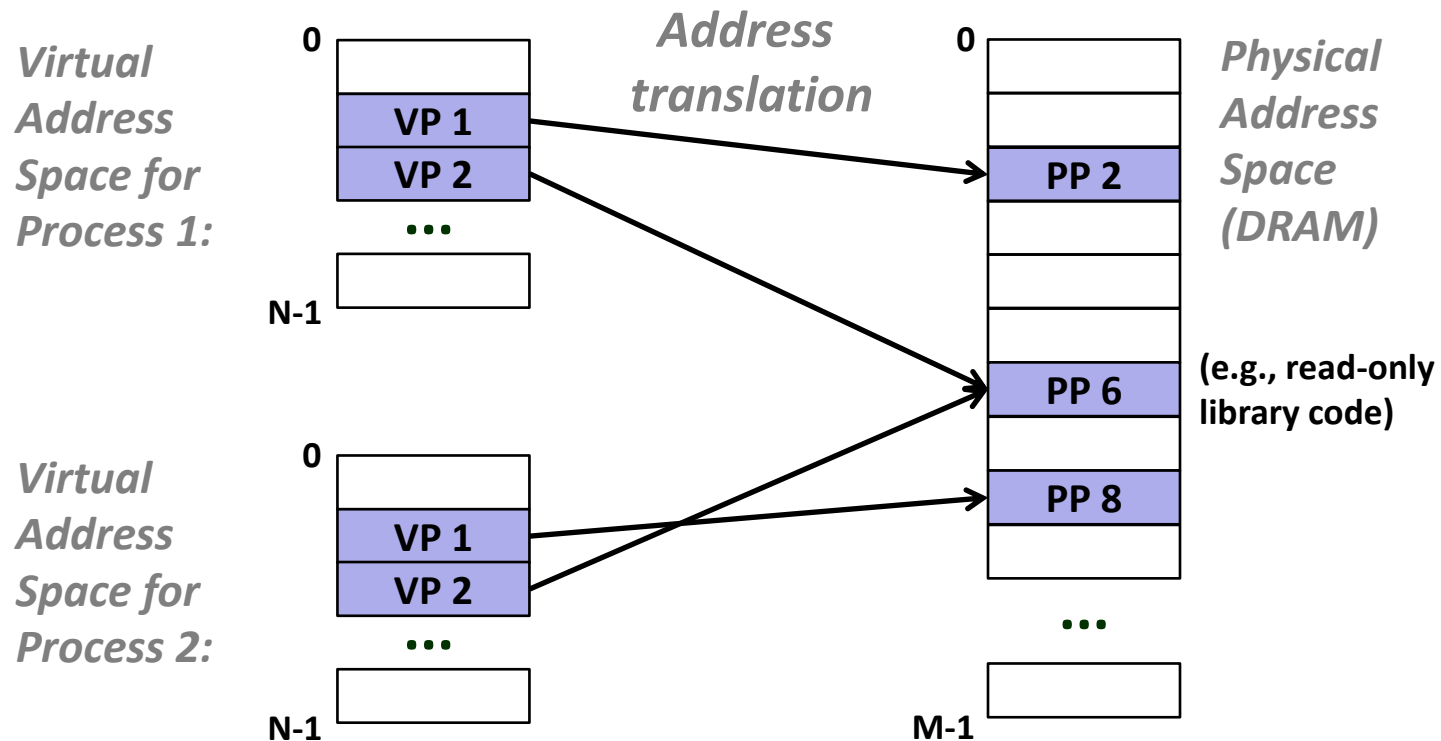
- Simple memory system example
- Case study: Core i7/Linux memory system
- **Memory mapping**

# Memory Mapping

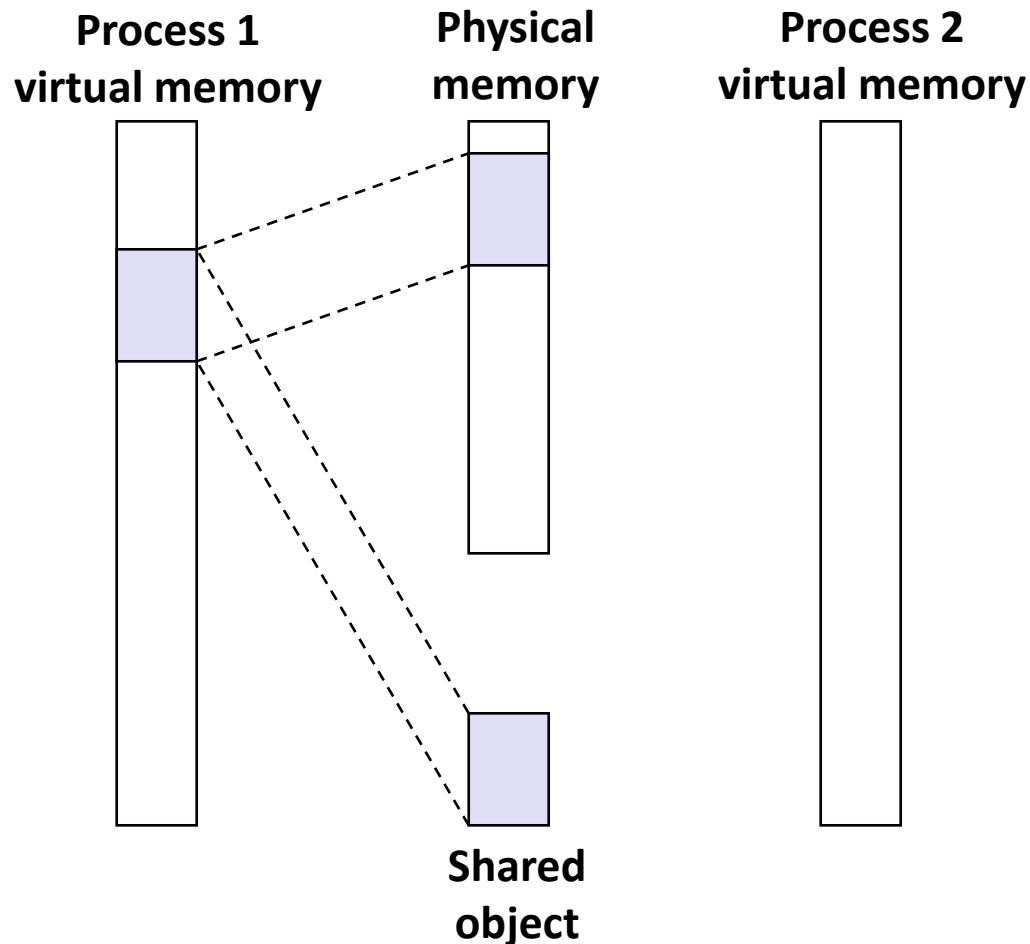
- VM areas initialized by associating them with disk objects.
  - Called *memory mapping*
- Area can be *backed by* (i.e., get its initial values from) :
  - *Regular file* on disk (e.g., an executable object file)
    - Initial page bytes come from a section of a file
  - *Anonymous file* (e.g., nothing)
    - 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*.

# Review: Memory Management & Protection

- Code and data can be isolated or shared among processes

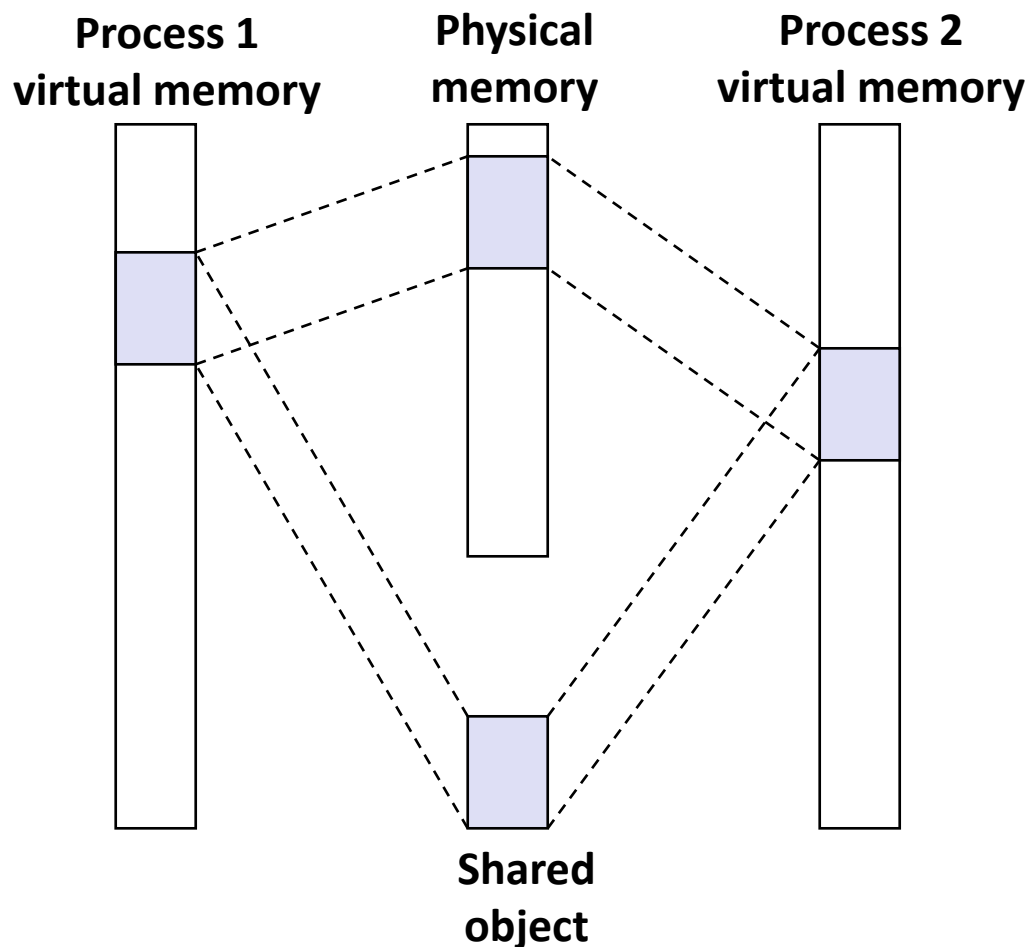


# Sharing Revisited: Shared Objects



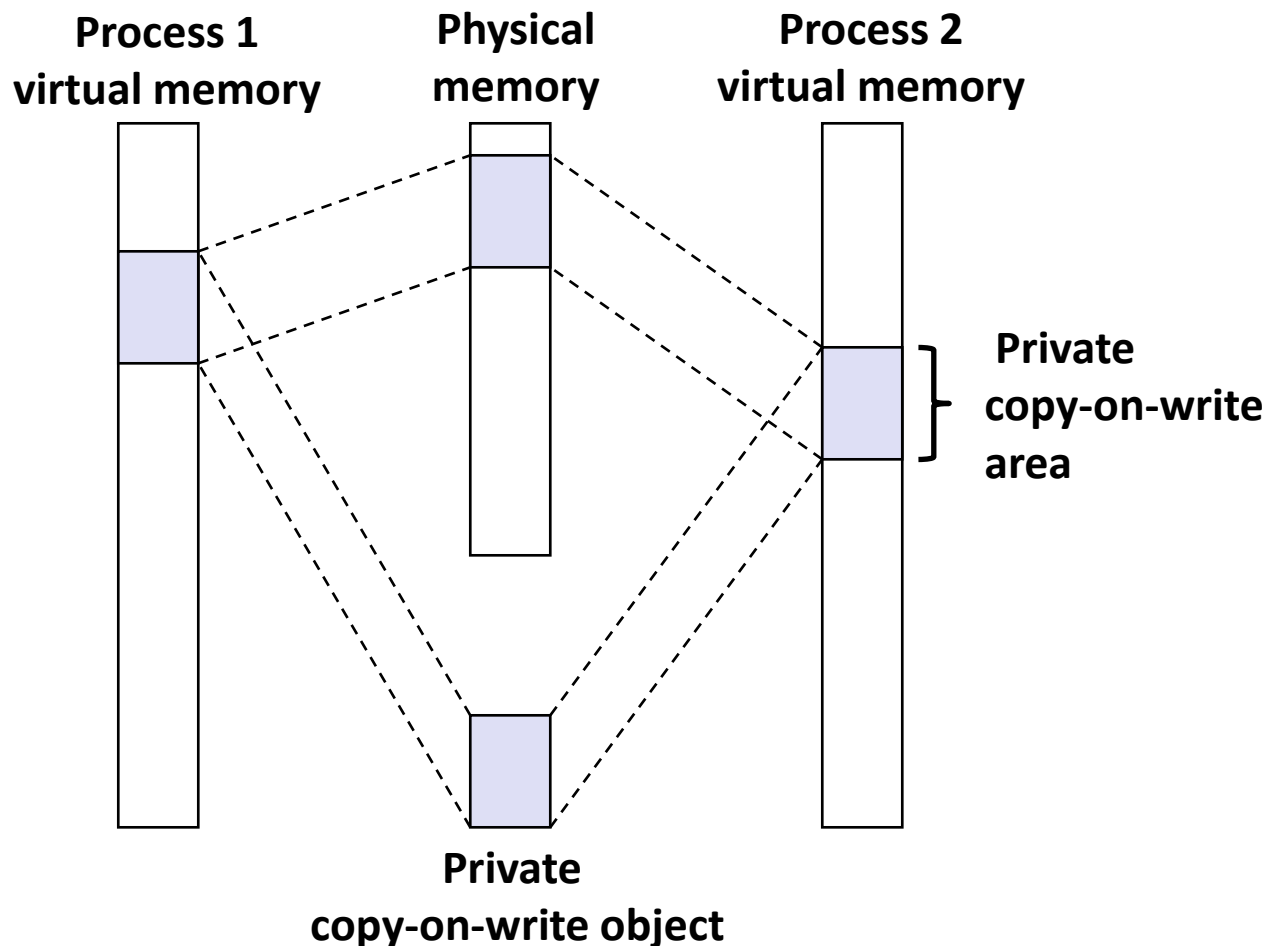
- **Process 1 maps the shared object (on disk).**

# Sharing Revisited: Shared Objects



- **Process 2 maps the same shared object.**
- **Notice how the virtual addresses can be different.**
- **But, difference must be multiple of page size.**

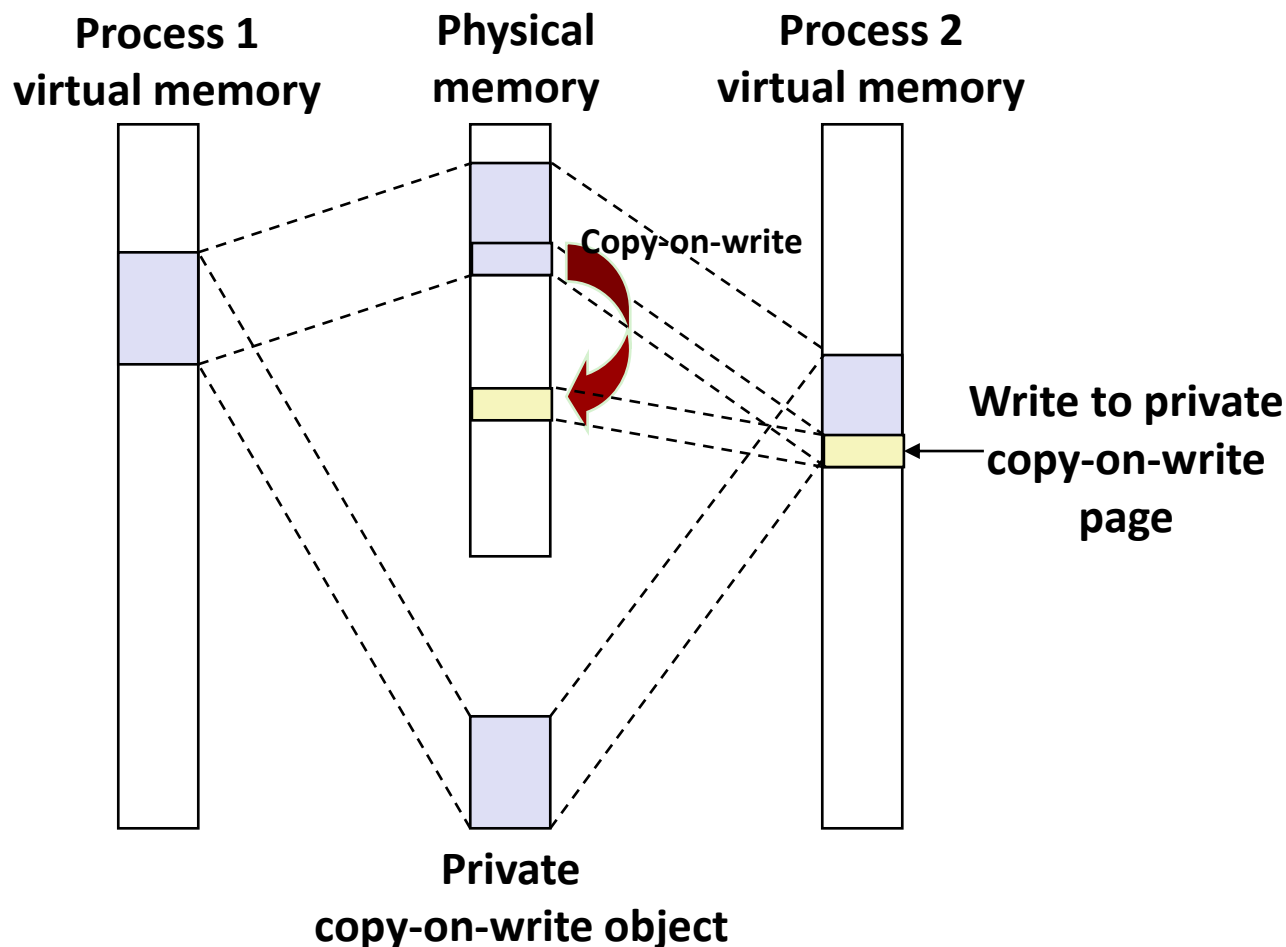
# Sharing Revisited: Private Copy-on-write (COW) Objects



- 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!

# Finding Shareable Pages

## ■ Kernel Same-Page Merging

- OS scans through all of physical memory, looking for duplicate pages
- When found, merge into single copy, marked as copy-on-write
- Implemented in Linux kernel in 2009
- Limited to pages marked as likely candidates
- Especially useful when processor running many virtual machines
  - A *virtual machine* is an abstraction for an entire computer, including its OS & I/O devices (beyond the scope of this course)

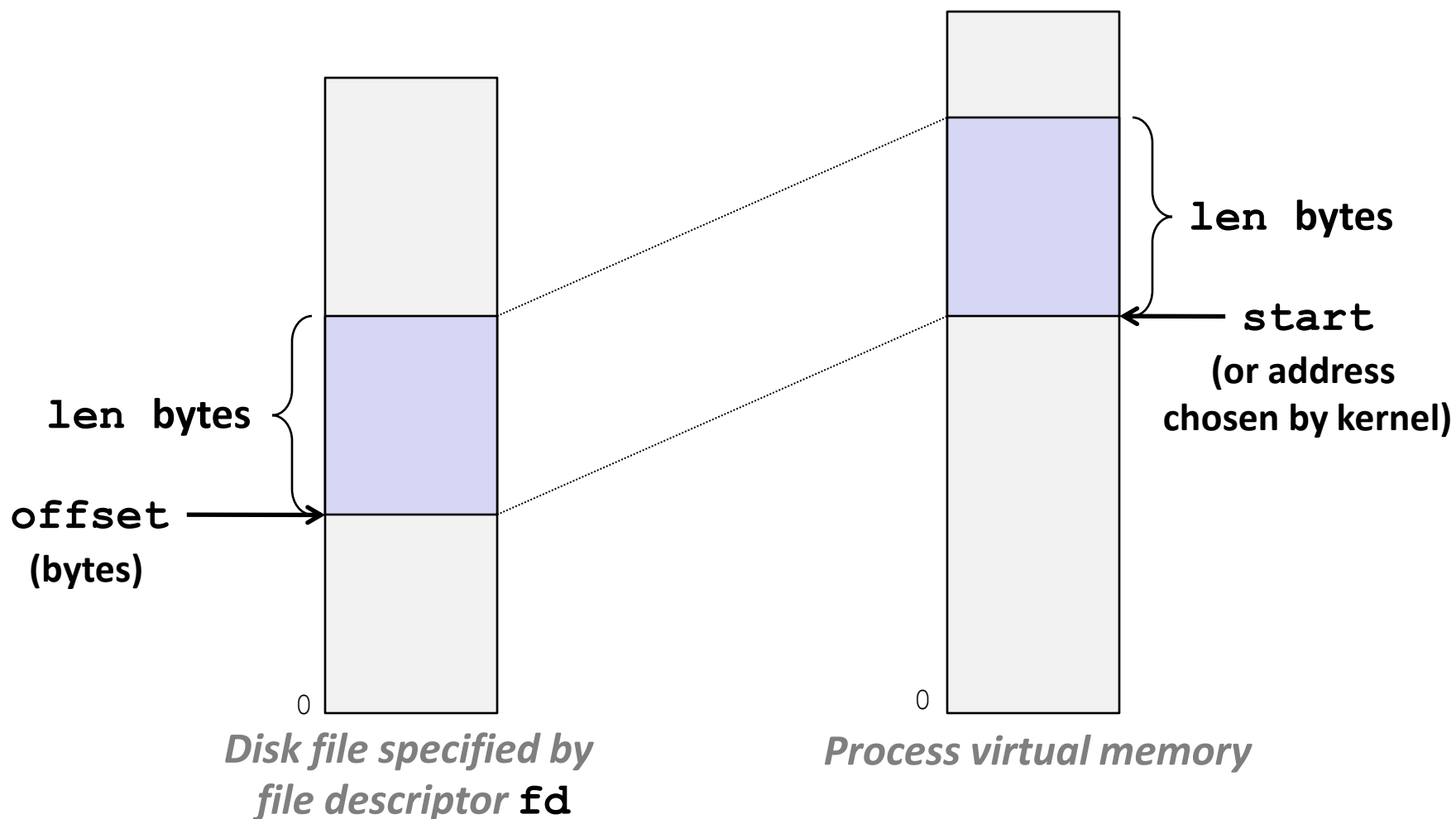
# User-Level Memory Mapping

```
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`, `PROT_EXEC`, ...
  - `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)
```



# Uses of mmap

## ■ Reading big files

- Uses paging mechanism to bring files into memory

## ■ Shared data structures

- When call with **MAP\_SHARED** flag
  - Multiple processes have access to same region of memory
  - Risky!

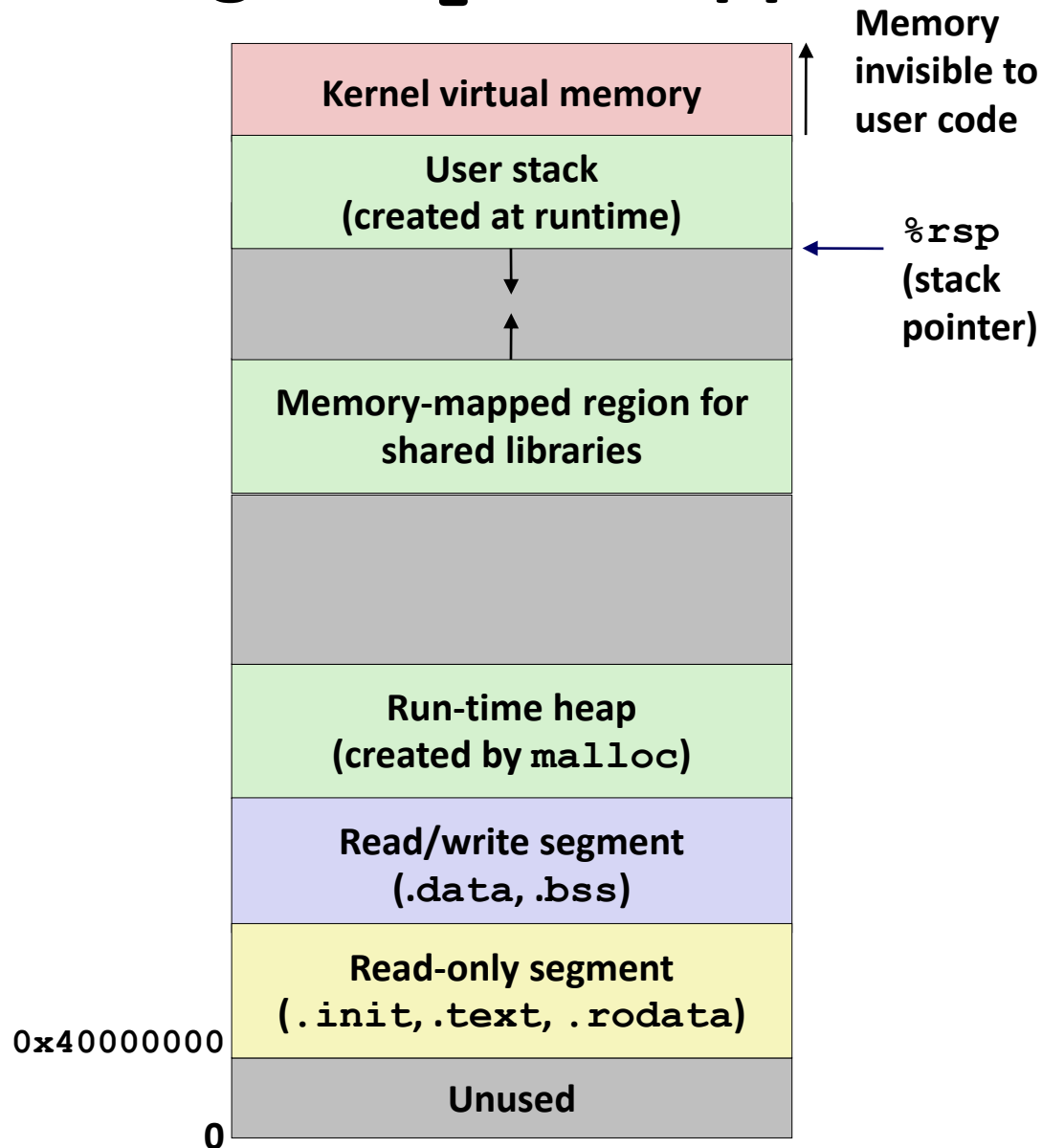
## ■ File-based data structures

- E.g., database
- Give `prot` argument **PROT\_READ** | **PROT\_WRITE**
- When unmap region, file will be updated via write-back
- Can implement load from file / update / write back to file

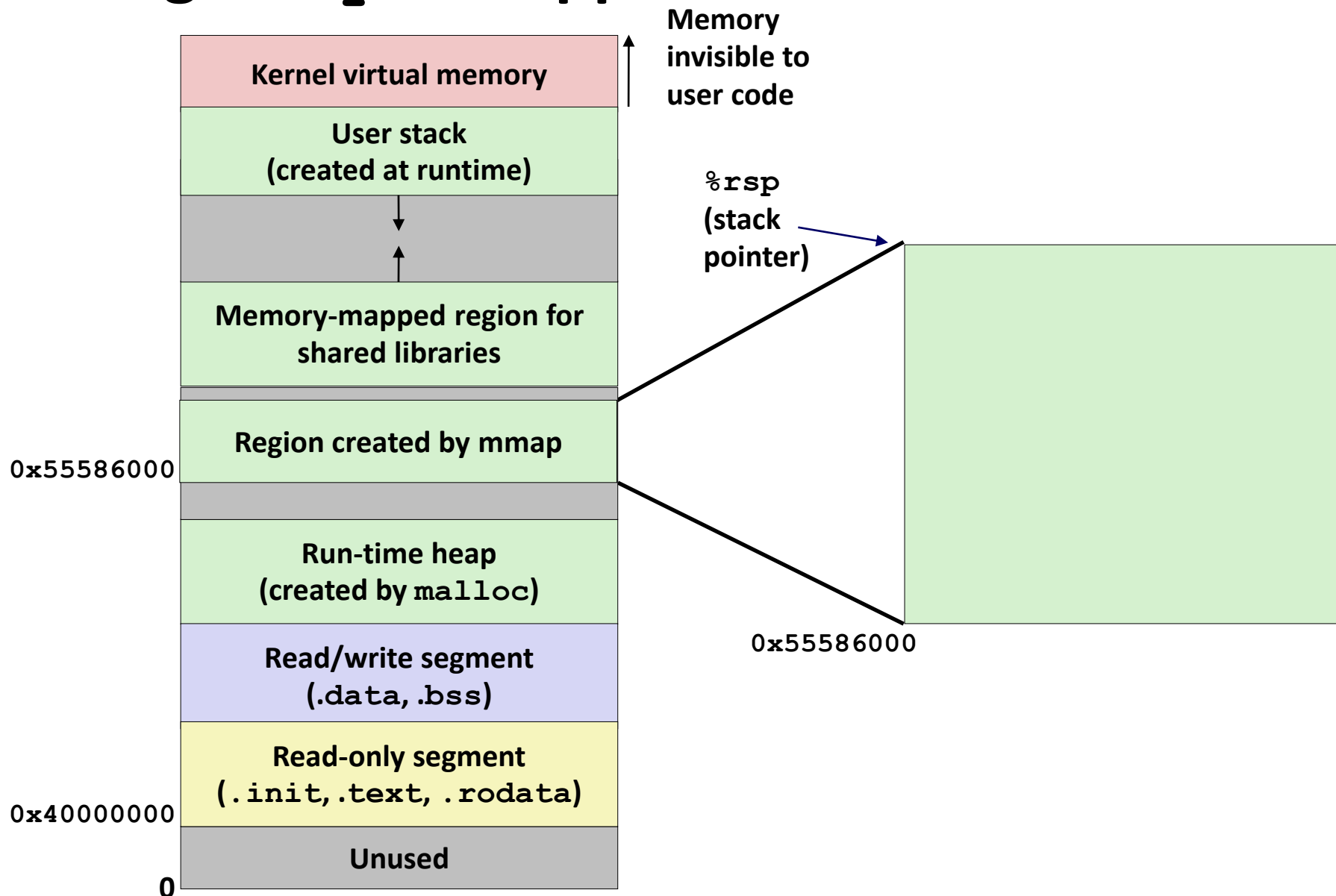
# Example: Using `mmap` to Support Attack Lab

- **Problem**
  - Want students to be able to perform code injection attacks
  - Shark machine stacks are not executable
- **Solution**
  - Suggested by Sam King (now at UC Davis)
  - Use `mmap` to allocate region of memory marked executable
  - Divert stack to new region
  - Execute student attack code
  - Restore back to original stack
  - Use `munmap` to remove mapped region

# Using mmap to Support Attack Lab

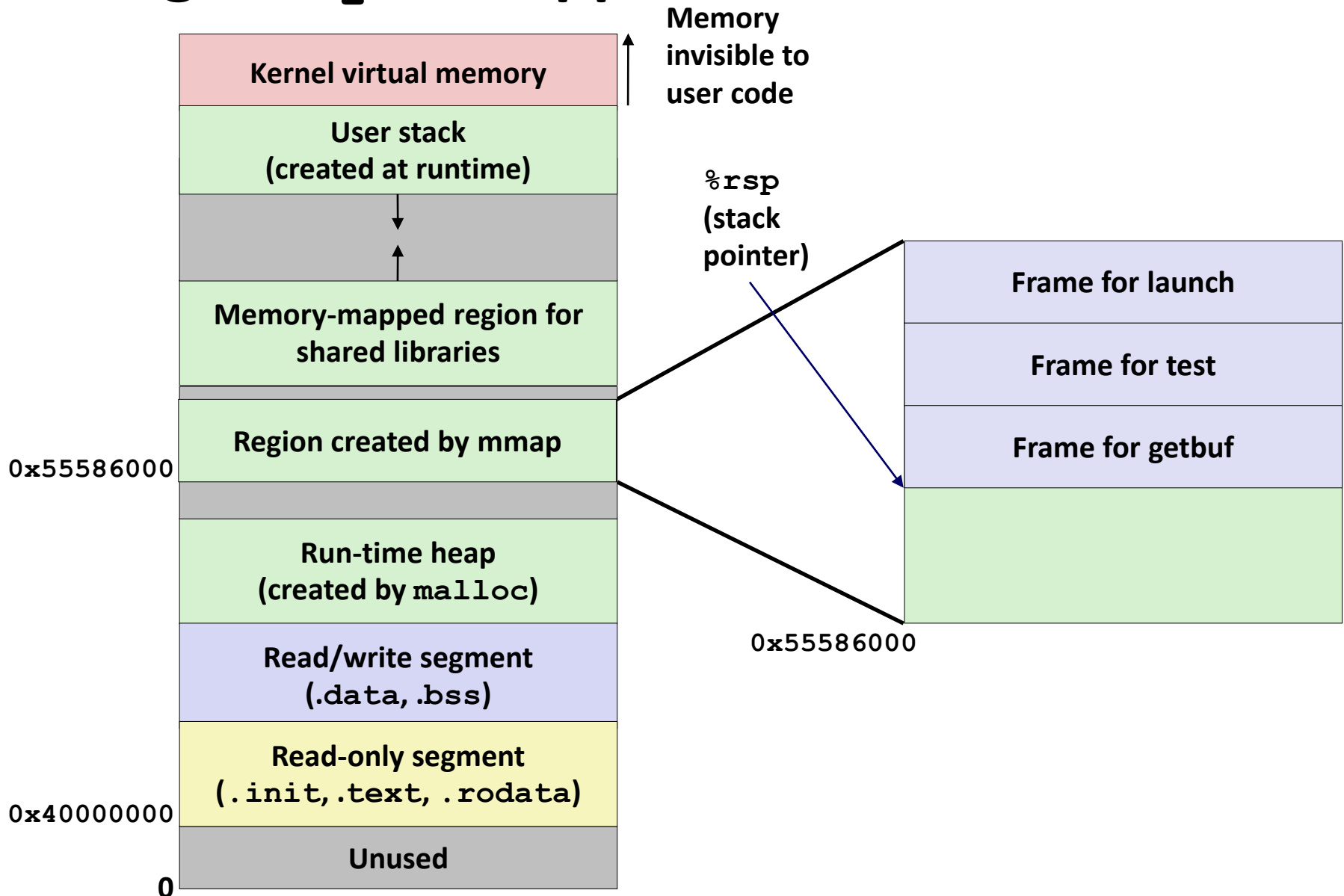


# Using mmap to Support Attack Lab

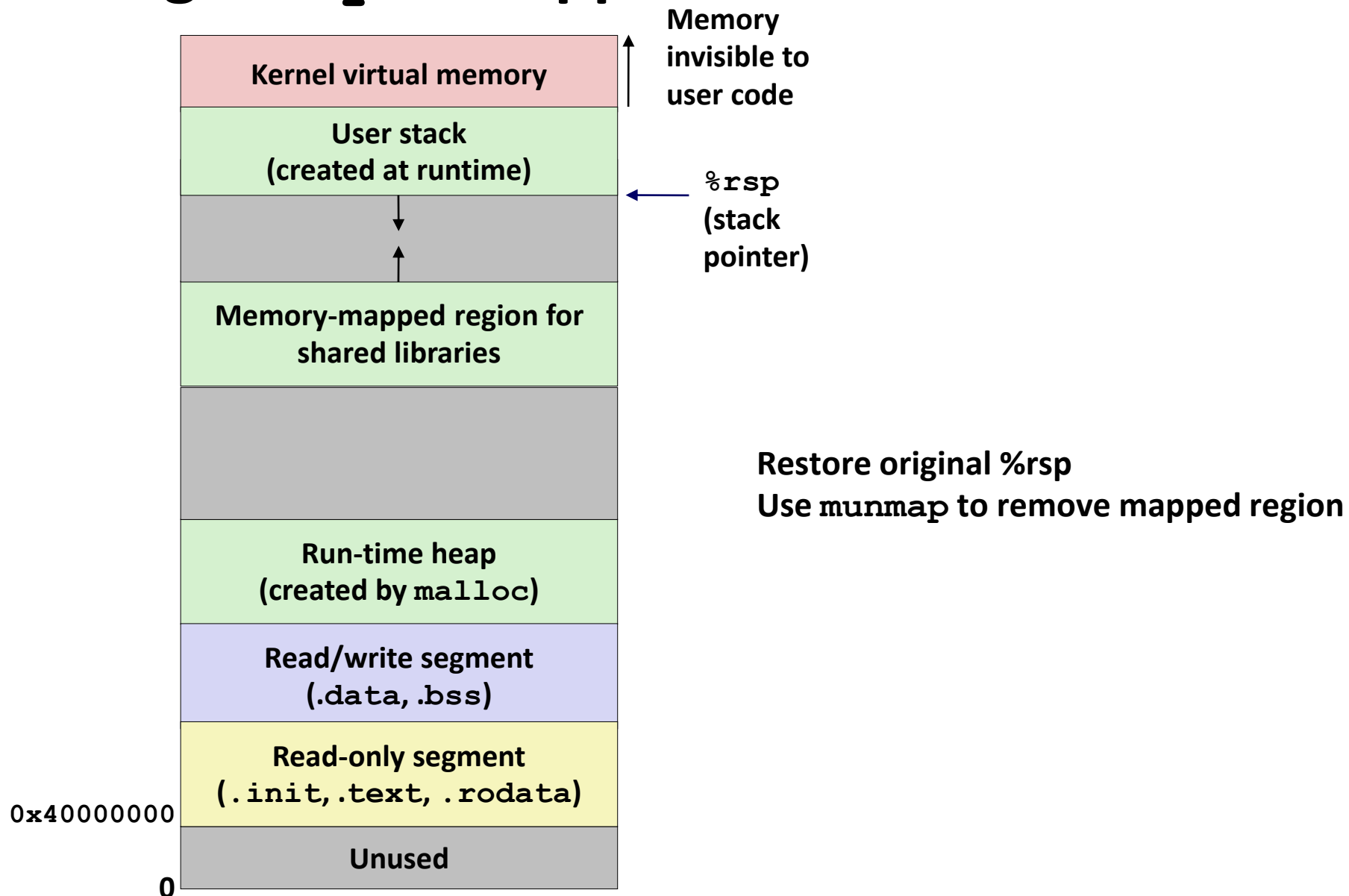




# Using mmap to Support Attack Lab



# Using mmap to Support Attack Lab



# Summary

## ■ VM requires hardware support

- Exception handling mechanism
- TLB
- Various control registers

## ■ VM requires OS support

- Managing page tables
- Implementing page replacement policies
- Managing file system

## ■ VM enables many capabilities

- Loading programs from memory
- Providing memory protection

# Using mmap to Support Attack Lab

## Allocate new region

```
void *new_stack = mmap(START_ADDR, STACK_SIZE, PROT_EXEC|PROT_READ|PROT_WRITE,
                      MAP_PRIVATE | MAP_GROWSDOWN | MAP_ANONYMOUS | MAP_FIXED,
                      0, 0);
if (new_stack != START_ADDR) {
    munmap(new_stack, STACK_SIZE);
    exit(1);
}
```

## Divert stack to new region & execute attack code

```
stack_top = new_stack + STACK_SIZE - 8;
asm("movq %%rsp,%%rax ; movq %1,%%rsp ;
    movq %%rax,%0"
    : "=r" (global_save_stack) // %0
    : "r" (stack_top) // %1
    );

launch(global_offset);
```

## Restore stack and remove region

```
asm("movq %0,%%rsp"
    :
    : "r" (global_save_stack) // %0
    );

munmap(new_stack, STACK_SIZE);
```