

# Virtual Memory: Systems

15-213: Introduction to Computer Systems  
18<sup>th</sup> Lecture, March 22, 2016

## **Instructors:**

Franz Franchetti & Seth Copen Goldstein, Ralf Brown, and Brian Railing

# Cheating: Description

## ■ What is cheating?

- **Sharing code:** by copying, retyping, **looking at**, or supplying a file
- **Describing:** verbal description of code from one person to another.
- **Coaching:** helping your friend to write a lab, line by line
- **Searching the Web** for solutions
- **Copying code** from a previous course or online solution
  - You are only allowed to use code we supply, or from the CS:APP website

## ■ What is NOT cheating?

- Explaining how to use systems or tools
- Helping others with high-level design issues

## ■ See the course syllabus for details.

- Ignorance is not an excuse

# Cheating: Consequences

## ■ Penalty for cheating:

- **If you do cheat – come clean asap!**
- Removal from course with failing grade (no exceptions!)
- Permanent mark on your record
- Your instructors' personal contempt

## ■ Detection of cheating:

- We have sophisticated tools for detecting code plagiarism
- Last Fall, 20 students were caught cheating and failed the course.
- Some were expelled from the University

## ■ Don't do it!

- Start early
- Ask the staff for help when you get stuck

**We found 10+ cheating cases in CacheLab**

# Regrading and Style Grading

- CacheLab Style Grades about to be released
- Regrade request deadline: 3/22, 11:59:00 pm EDT  
*tonight one minute before midnight*
- **Need one hardcopy request and one email**  
Gave hardcopy to TA after exam or to instructor,  
slide hardcopy under Seth's or Franz' door
- **Result *may* show up in exam server**  
unless you are a "special case"
- **Final score will be uploaded to autolab later this week**

# MallocLab

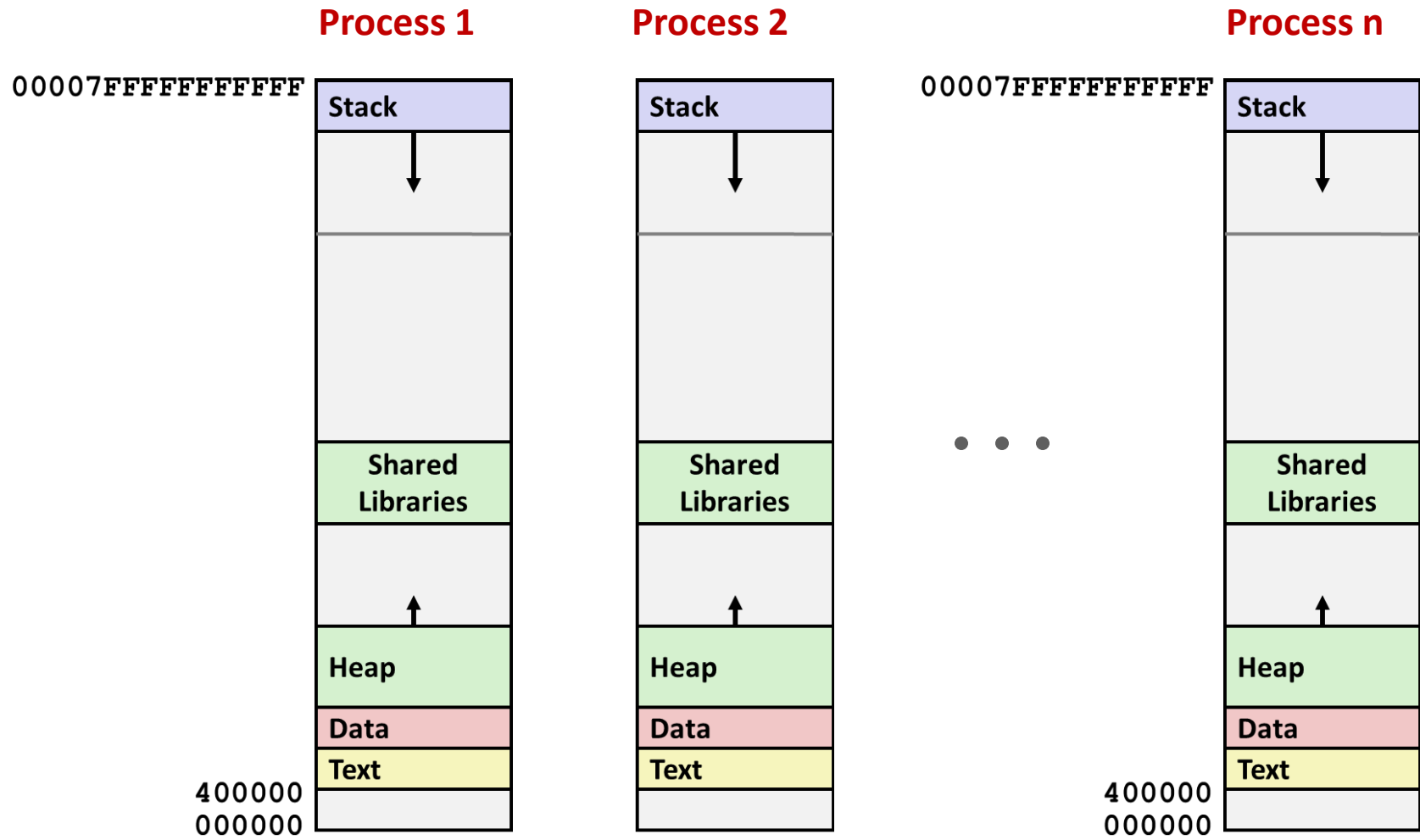
- **Out tonight**
- **Due April 8**
- **Checkpoint due March 31**

# Office Hours

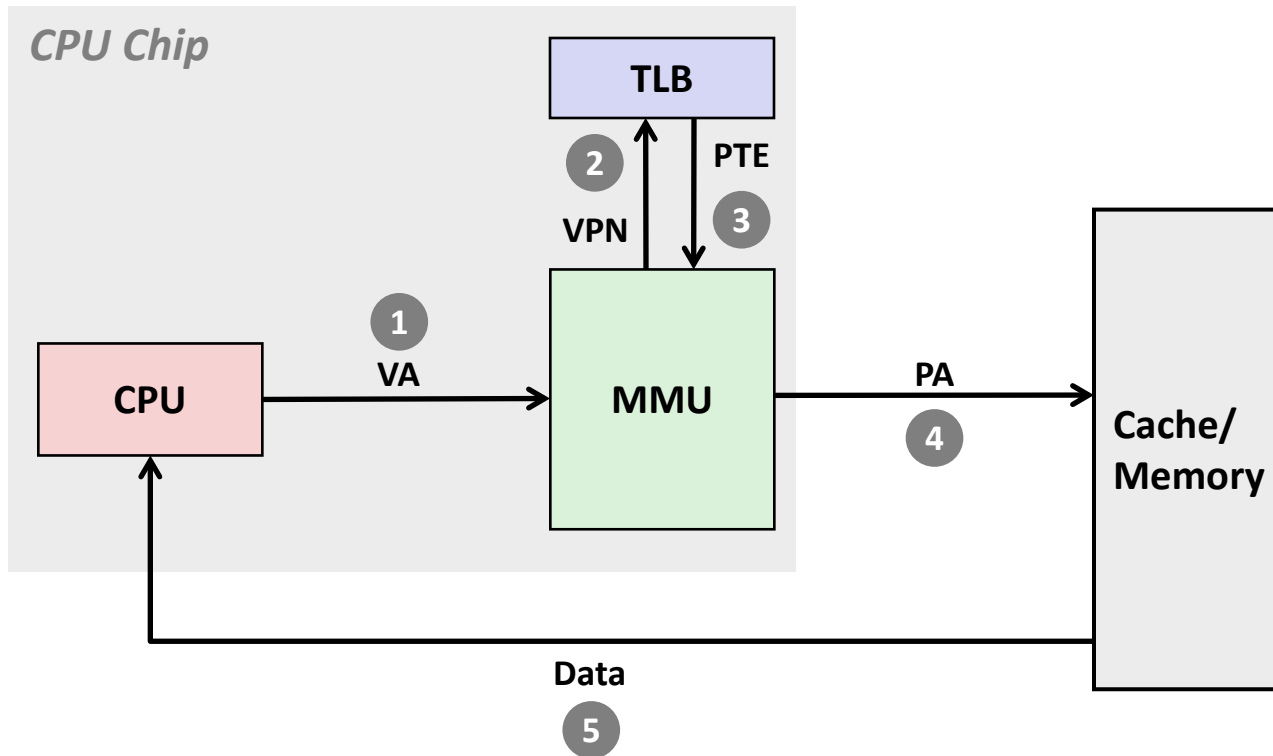
- I am there every week...
- It is usually not very crowded
- Just come by to check in even if you do not have a question



# Recap: Hmmm, How Does This Work?!



# Transaction Lookaside Buffer (TLB)

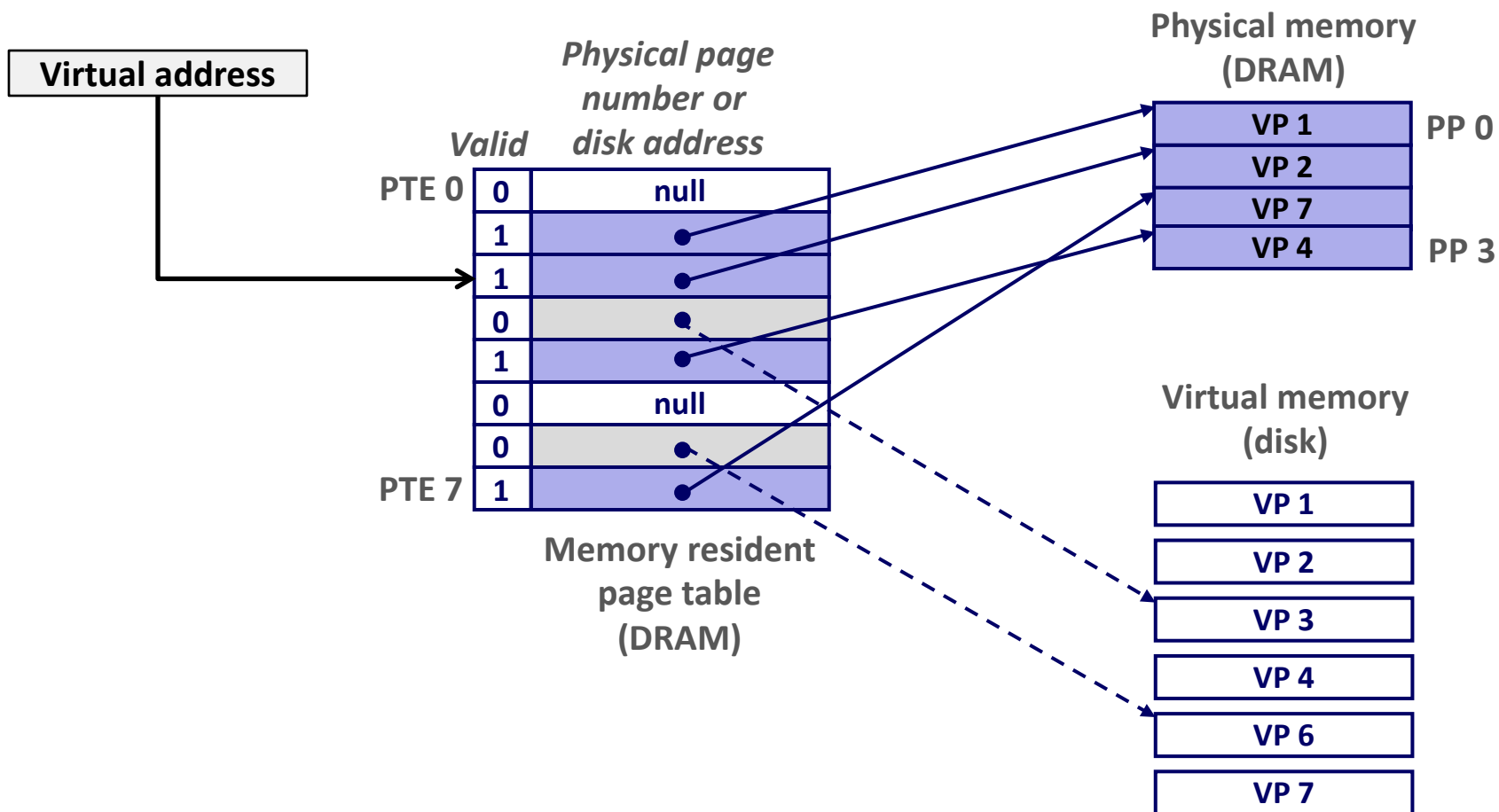


**A TLB hit eliminates a memory access**



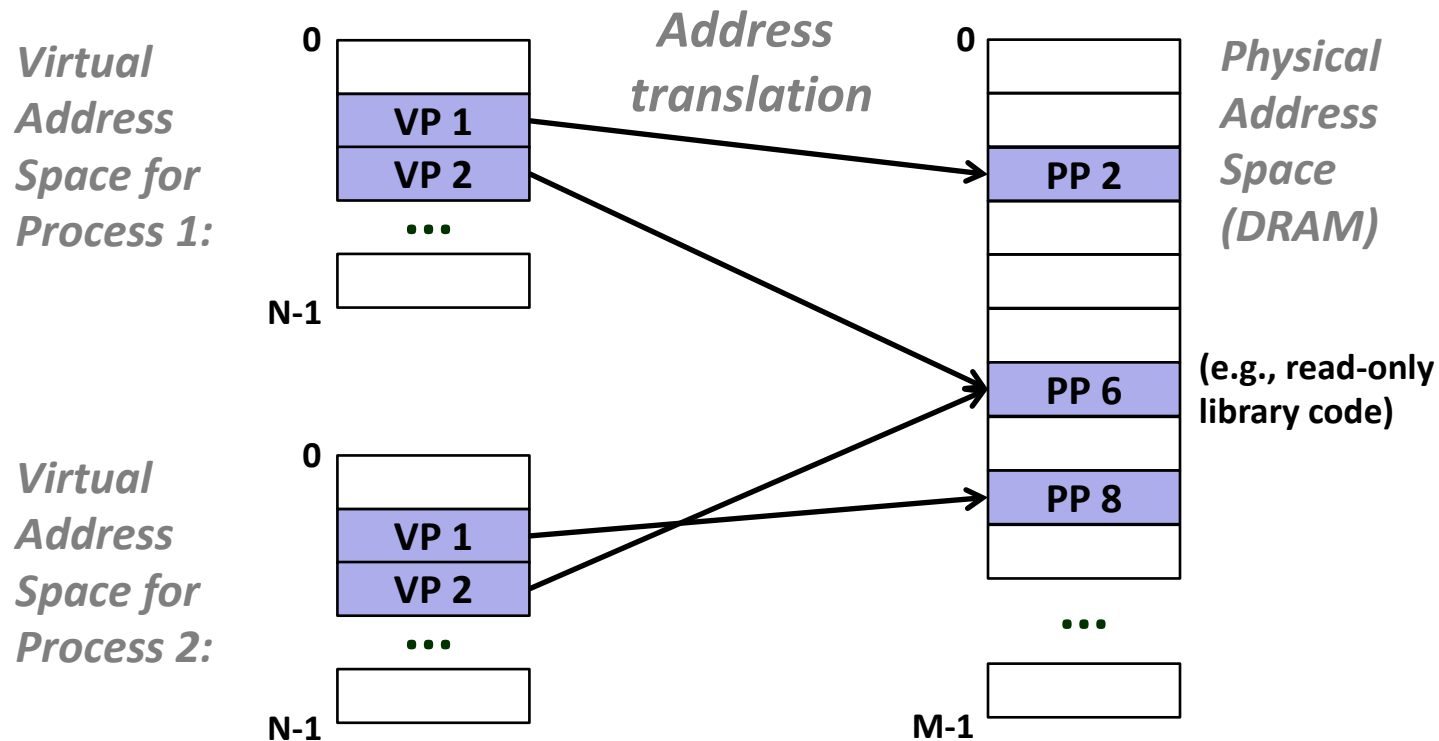
# Virtual Memory and Physical Memory

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

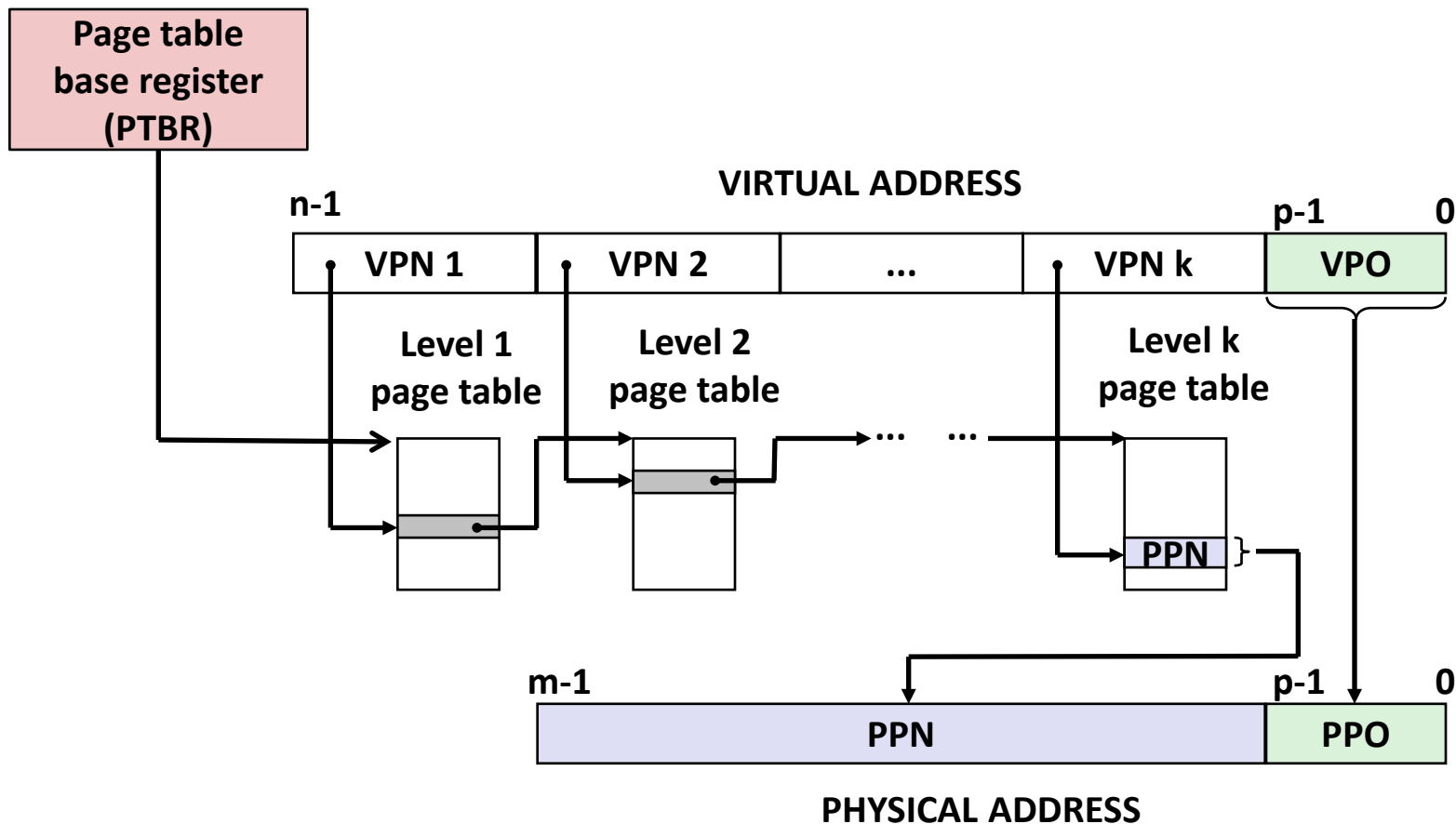


# VM as a Tool for Memory Management

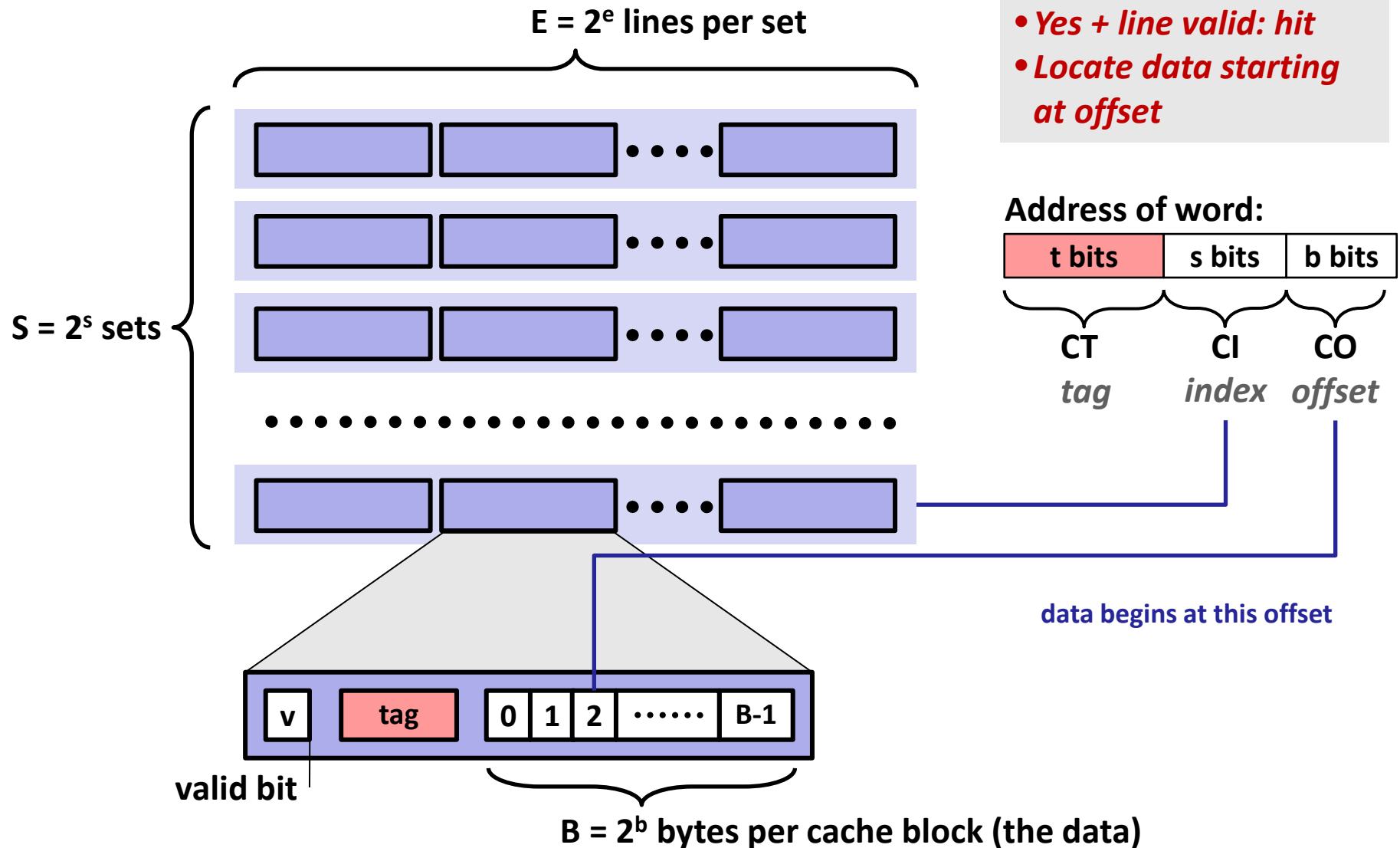
- Simplifying memory allocation
- Sharing code and data among processes



# Translating with a k-level Page Table



# Set Associative Cache: Read



- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*

# Today

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

# Review of Symbols

## Basic Parameters

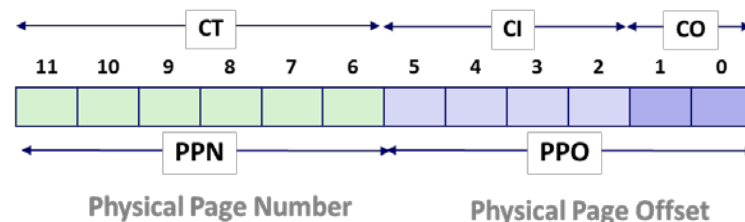
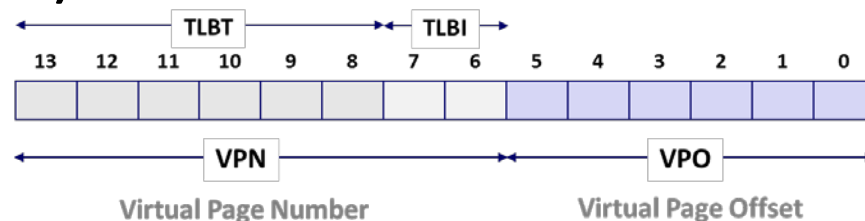
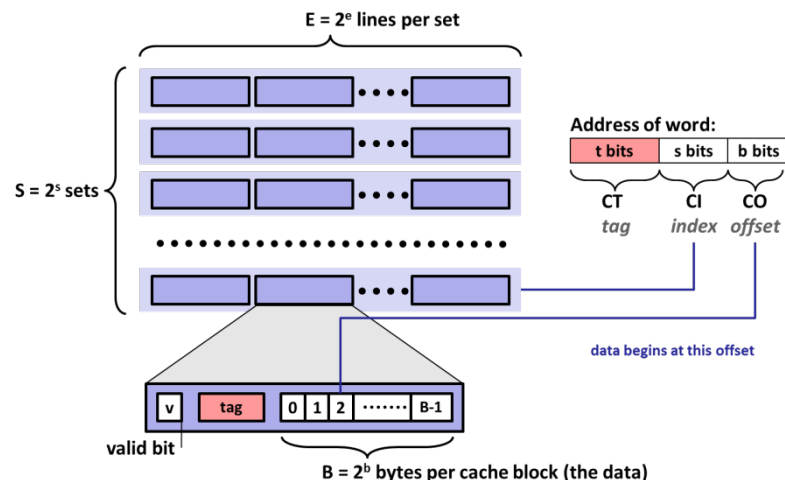
- $N = 2^n$ : Number of addresses in virtual address space
- $M = 2^m$ : Number of addresses in physical address space
- $P = 2^p$ : Page size (bytes)

## 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)

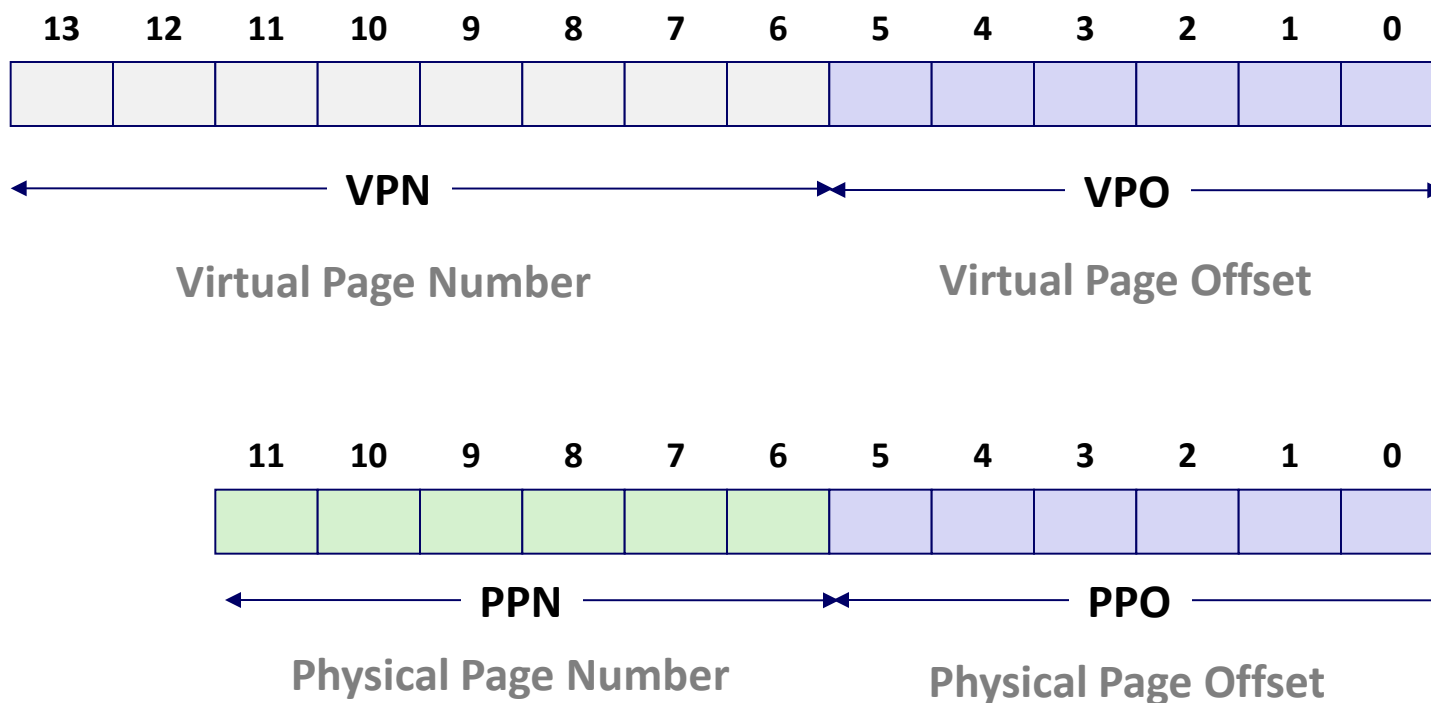
- PPO: Physical page offset (same as VPO)
- PPN: Physical page number
- CO: Byte offset within cache line
- CI: Cache index
- CT: Cache tag



# Simple Memory System Example

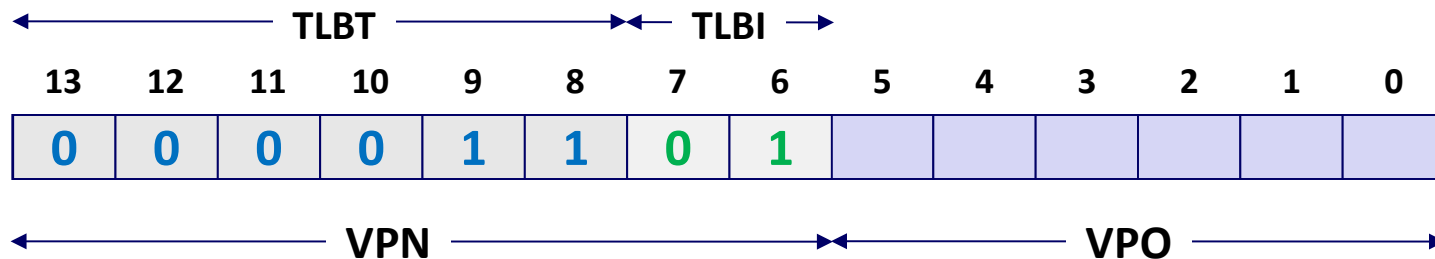
## ■ Addressing

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



# Simple Memory System TLB

- 16 entries
- 4-way associative



$$\text{VPN} = 0\text{b}1101 = 0\text{x}0\text{D}$$

## Transaction 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



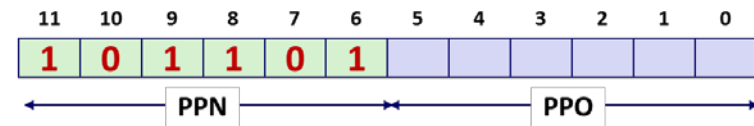
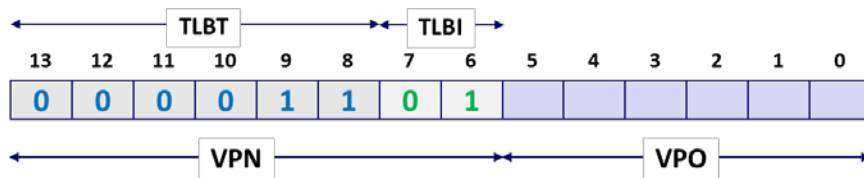
# Simple Memory System Page Table

Only 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

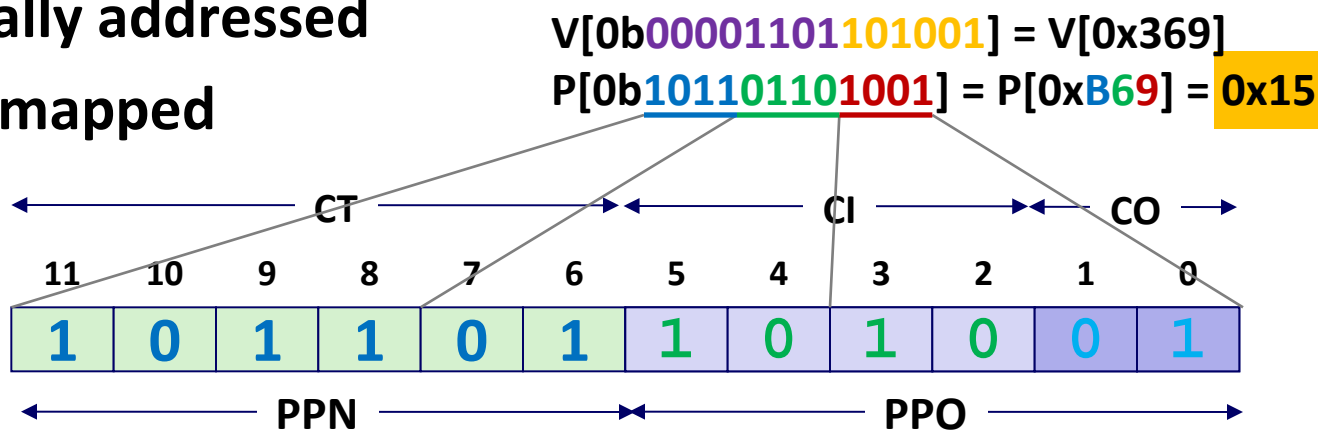
<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

0x0D → 0x2D



# 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

<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	–	–	–	–

## Transaction Lookaside Buffer (TLB)

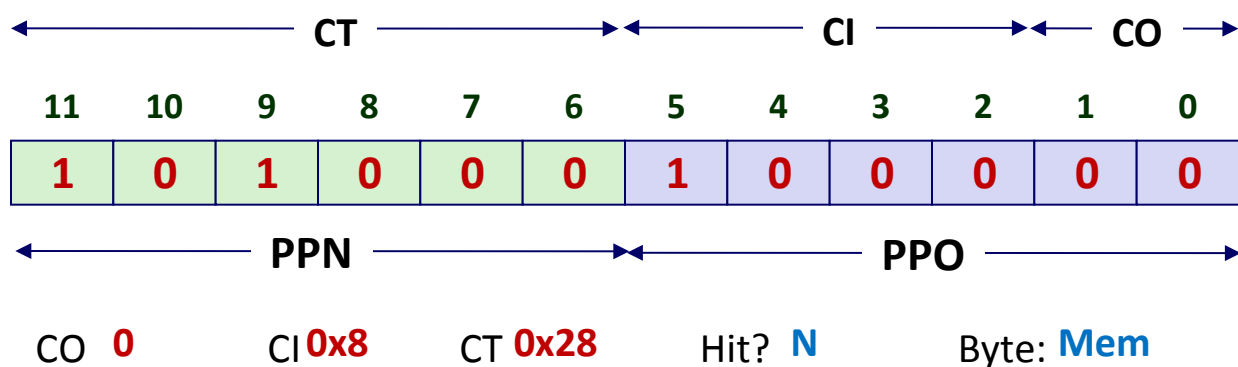
<i>Set</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

# Address Translation Example: TLB/Cache Miss

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



## 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

# Virtual Memory Exam Question

## Problem 5. (10 points):

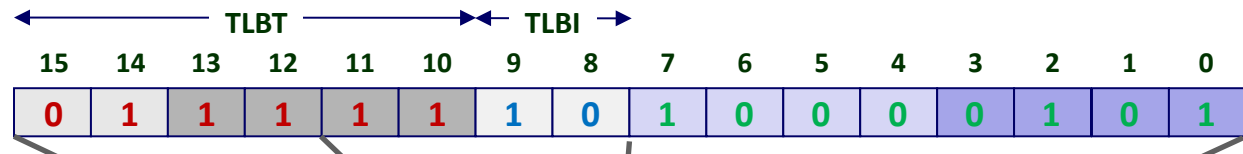
Assume a System that has

1. A two way set associative TLB
2. A TLB with 8 total entries
3.  $2^8$  byte page size
4.  $2^{16}$  bytes of virtual memory
5. one (or more) boats



Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

TLB			
Index	Tag	Frame Number	Valid
0	0x13	0x30	1
	0x34	0x58	0
1	0x1F	0x80	0
	0x2A	0x72	1
	0x1F	0x95	1
3	0x20	0xAA	0
	0x3F	0x20	1
	0x3E	0xFF	0



A. Use the TLB to fill in the table. Strike out anything that you don't have enough information to fill in.

Virtual Address	Physical Address
0x7E85	0x9585
0xD301	-----
0x4C20	0x3020
0xD040	-----
-----	0x5830

$$0x7E85 = 0x0111111010000101$$

$$CI = 0x2$$

$$CT = 0x1F$$

$$0x7E85 \rightarrow 0x9585$$

Exam: <http://www.cs.cmu.edu/~213/oldexams/exam2b-s11.pdf> (solution)

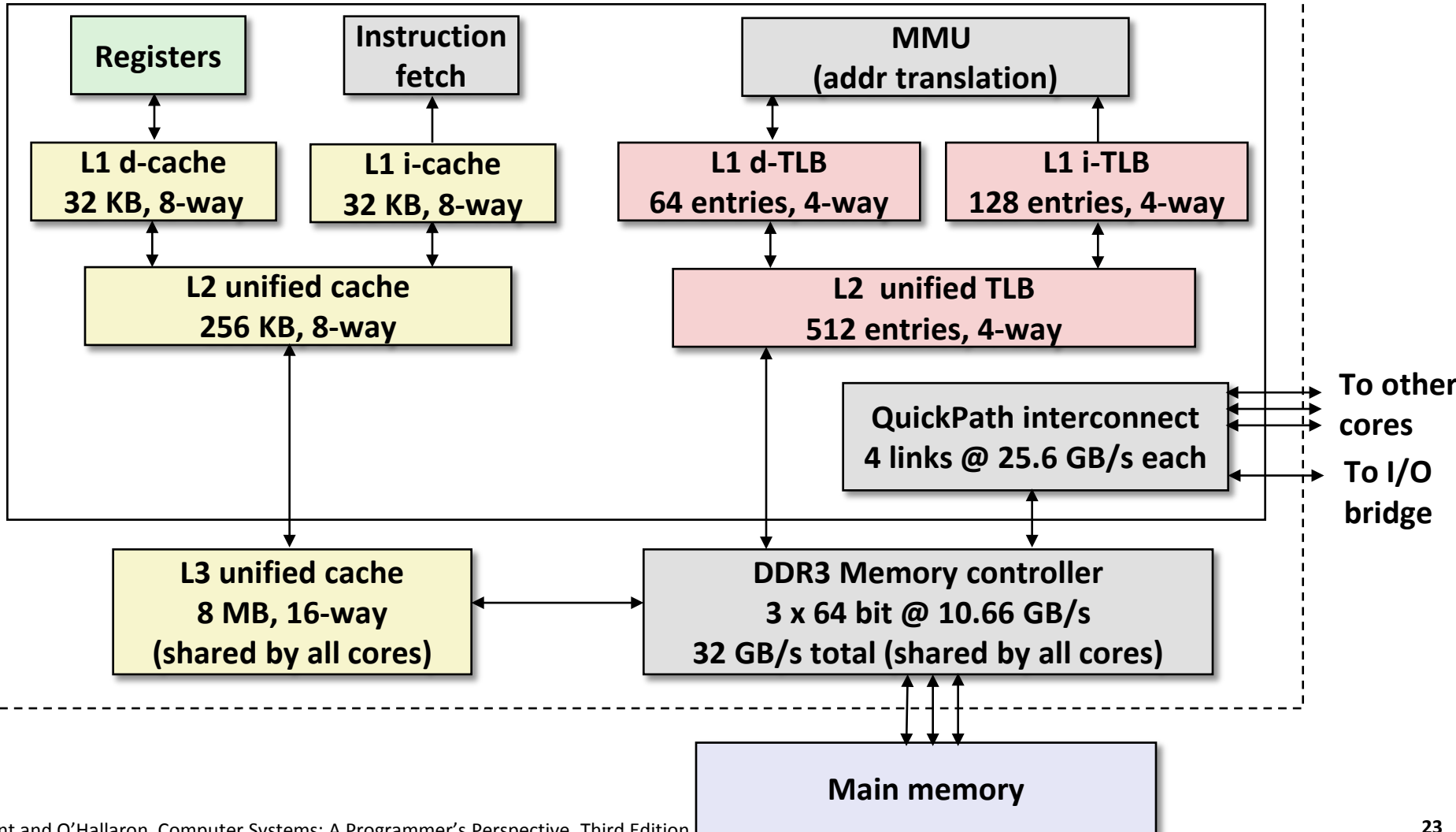
# Today

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

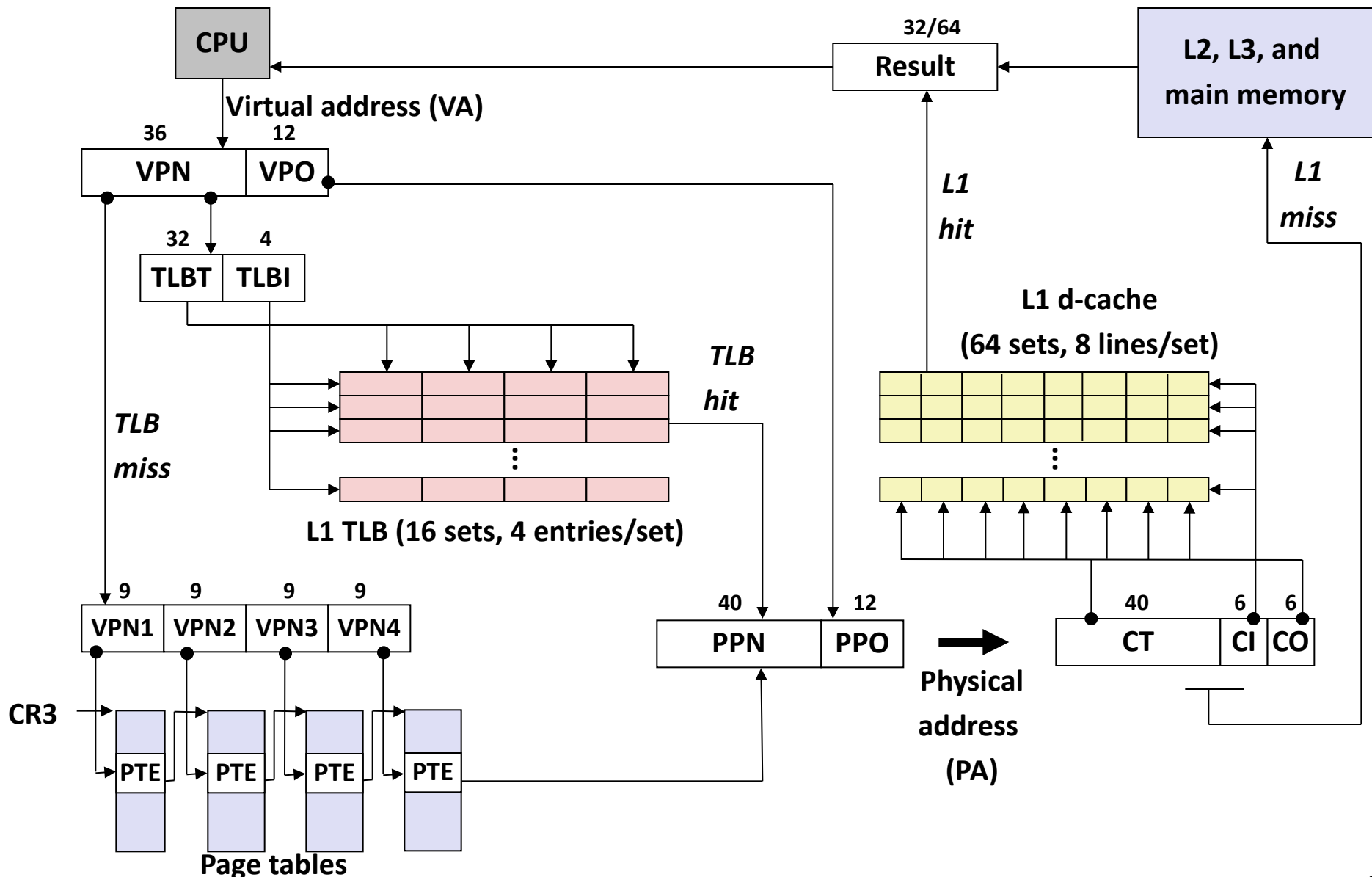
# Intel Core i7 Memory System

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

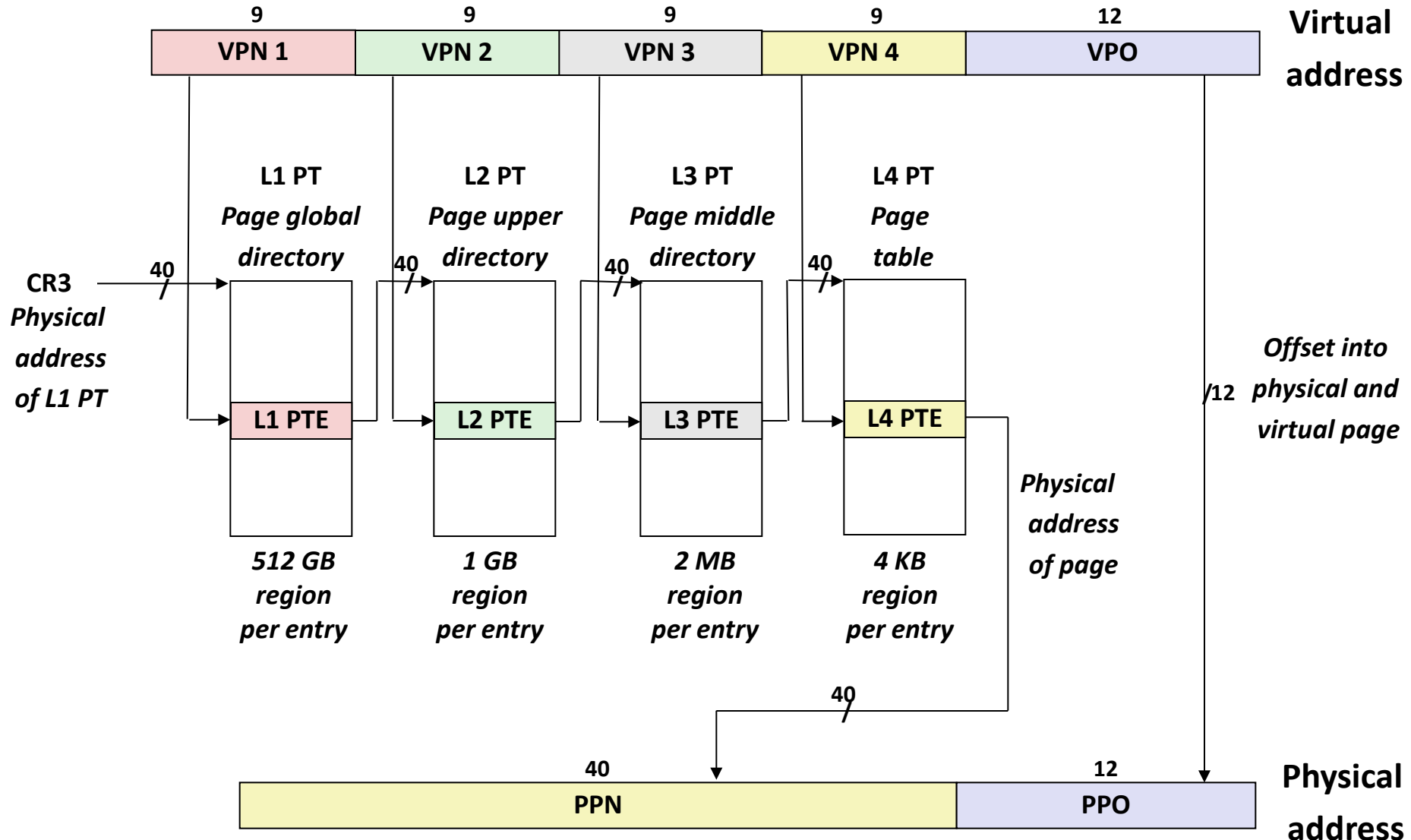
**A:** Reference bit (set by MMU on reads and writes, cleared by software)

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

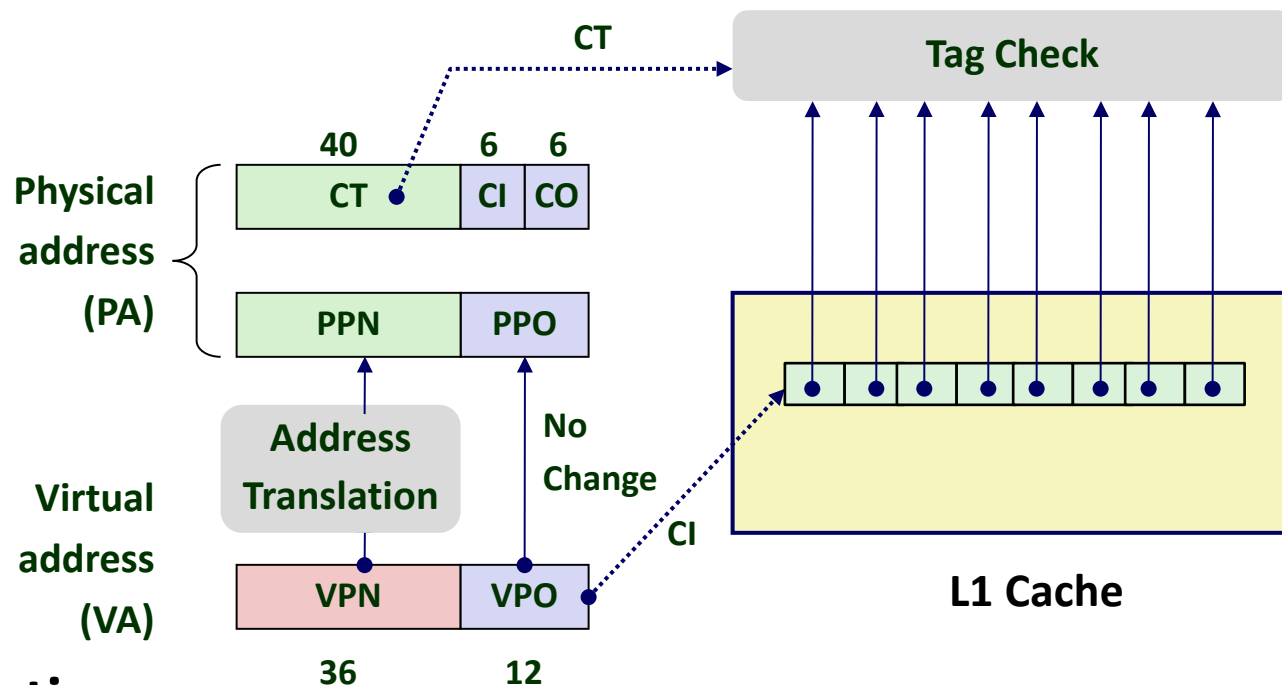
**Page physical base address:** 40 most significant bits of physical page address  
(forces pages to be 4KB aligned)

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

# Core i7 Page Table Translation



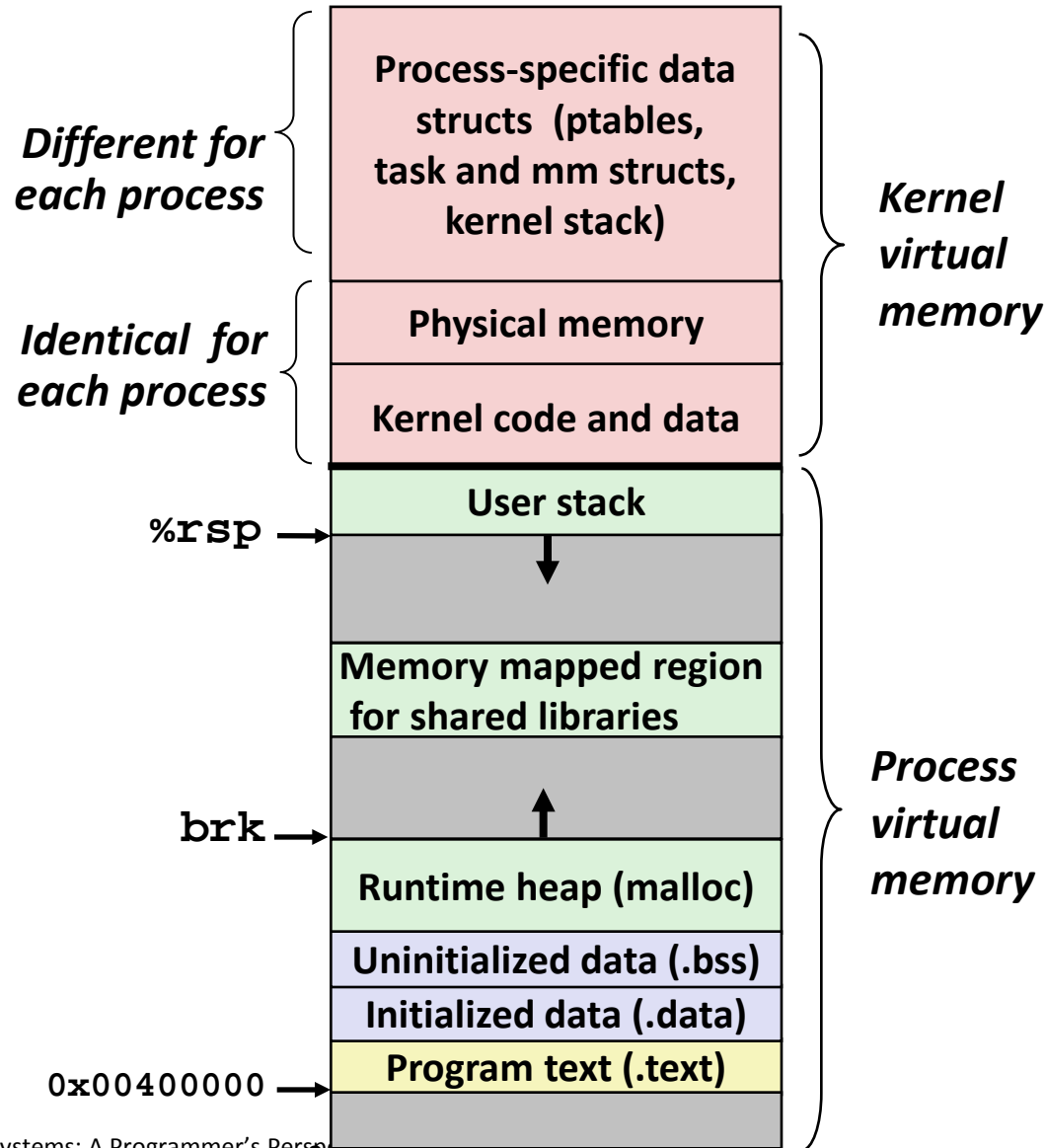
# Cute Trick for Speeding Up L1 Access



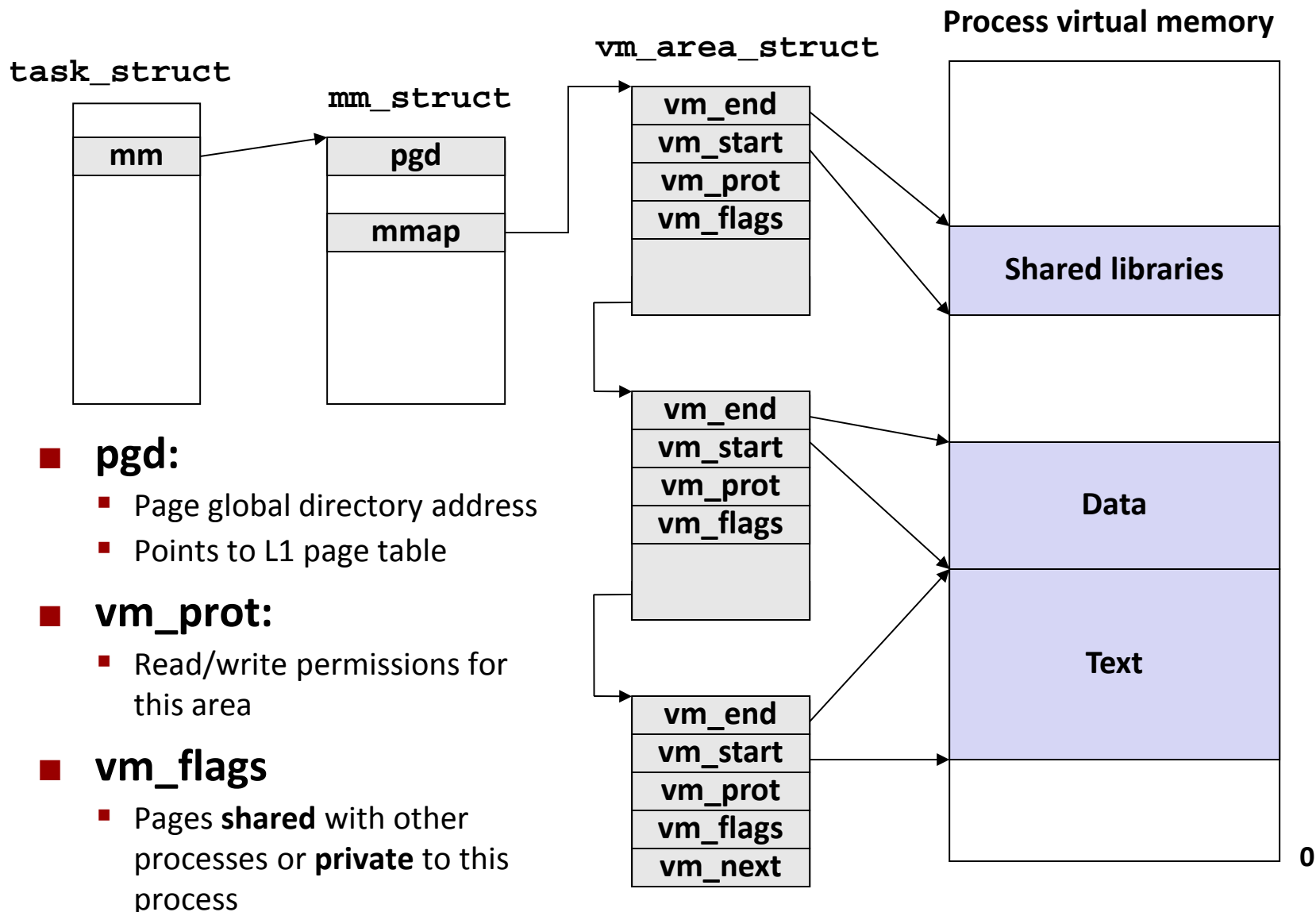
## ■ Observation

- 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

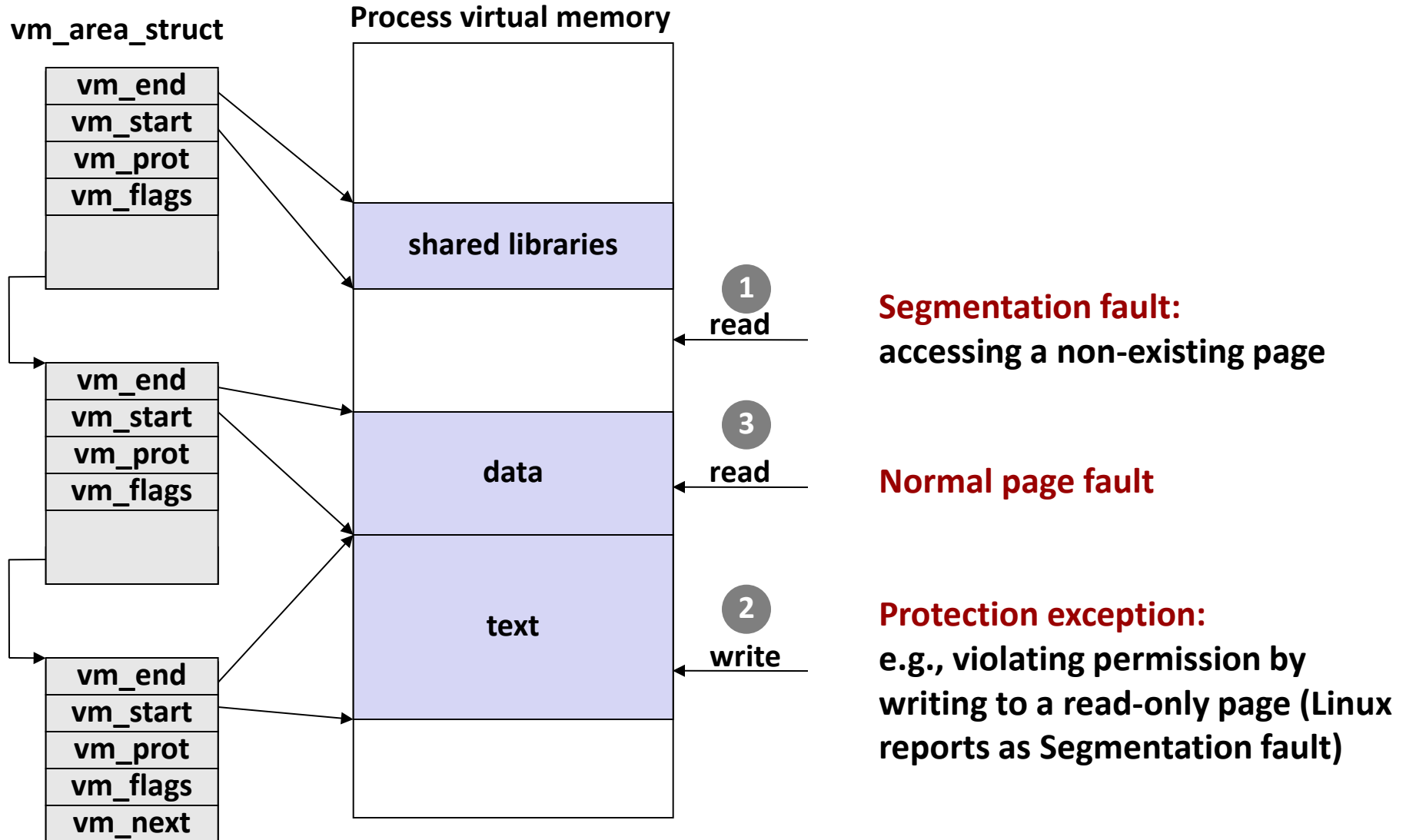
# Virtual Address Space of a Linux Process



# Linux Organizes VM as Collection of “Areas”



# Linux Page Fault Handling



# Today

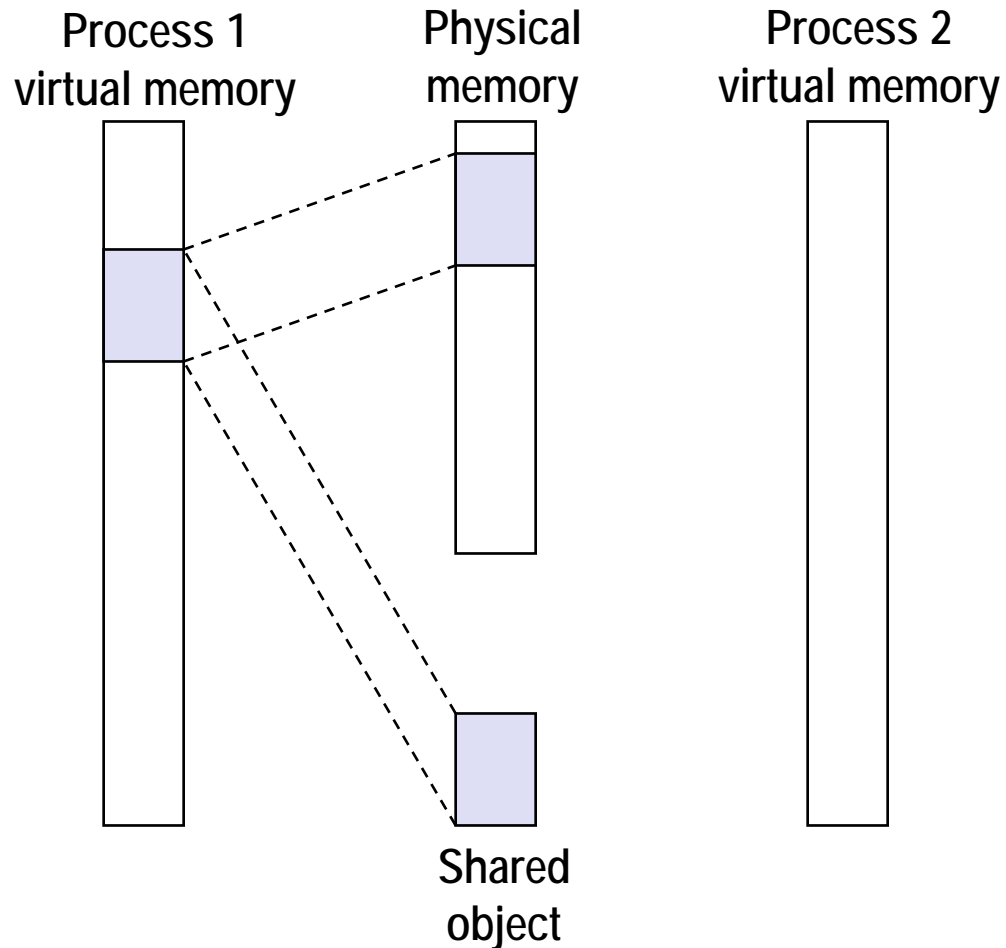
- Simple memory system example
- Case study: Core i7/Linux memory system
- **Memory mapping**



# Memory Mapping

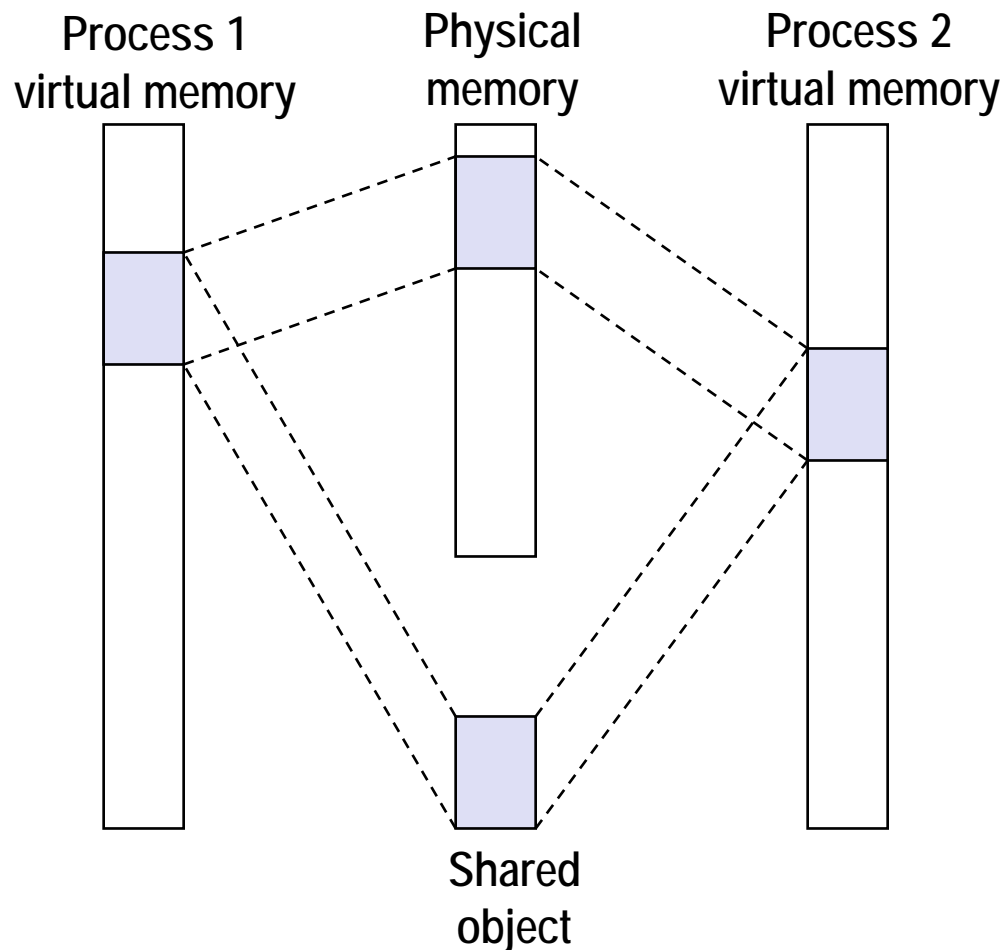
- VM areas initialized by associating them with disk objects.
  - Process is known as *memory mapping*.
- Area can be *backed by* (i.e., get its initial values from) :
  - *Regular file* on disk (e.g., an executable object file)
    - Initial page bytes come from a section of a file
  - *Anonymous file* (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*.

# Sharing Revisited: Shared Objects



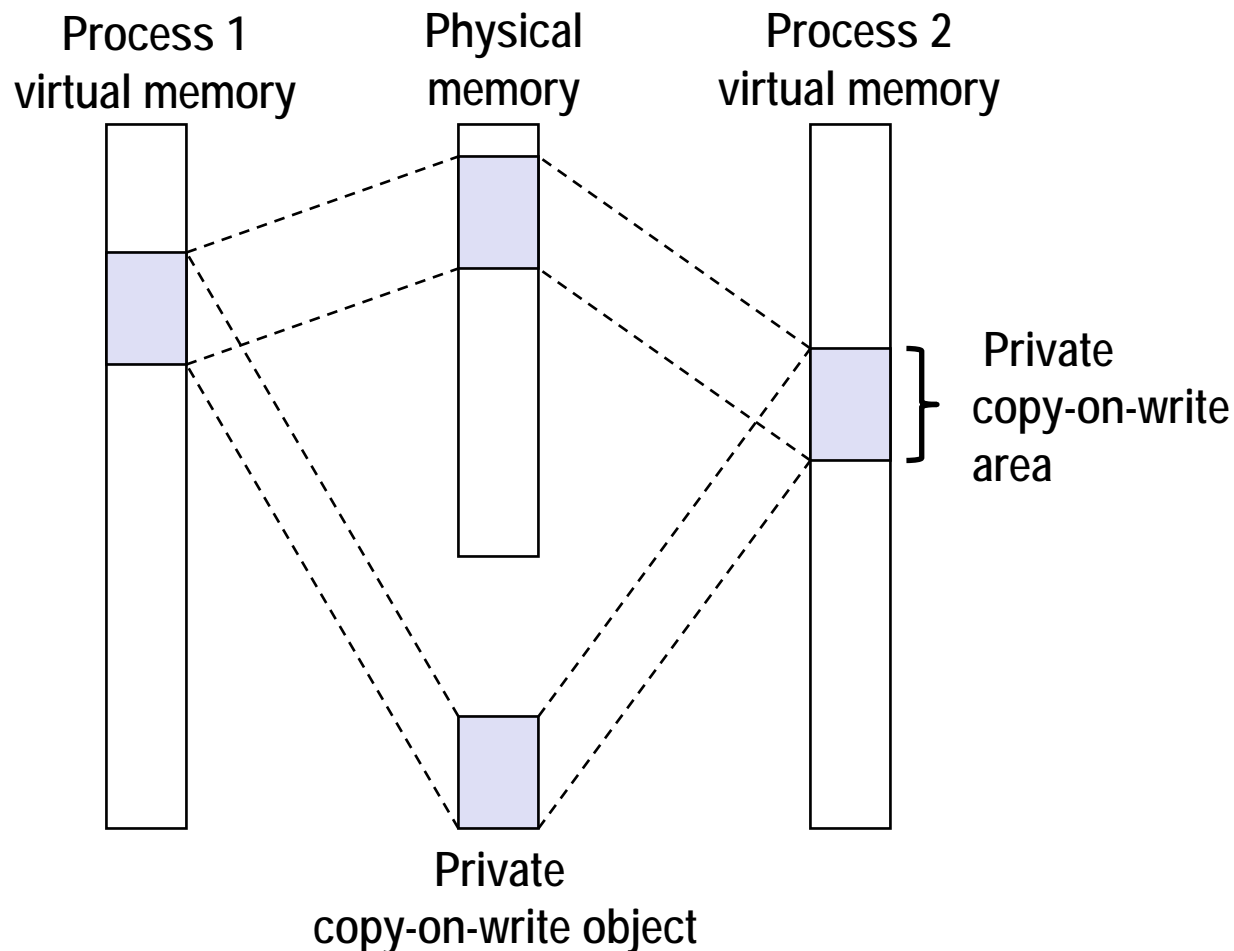
- **Process 1 maps the shared object.**

# Sharing Revisited: Shared Objects



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

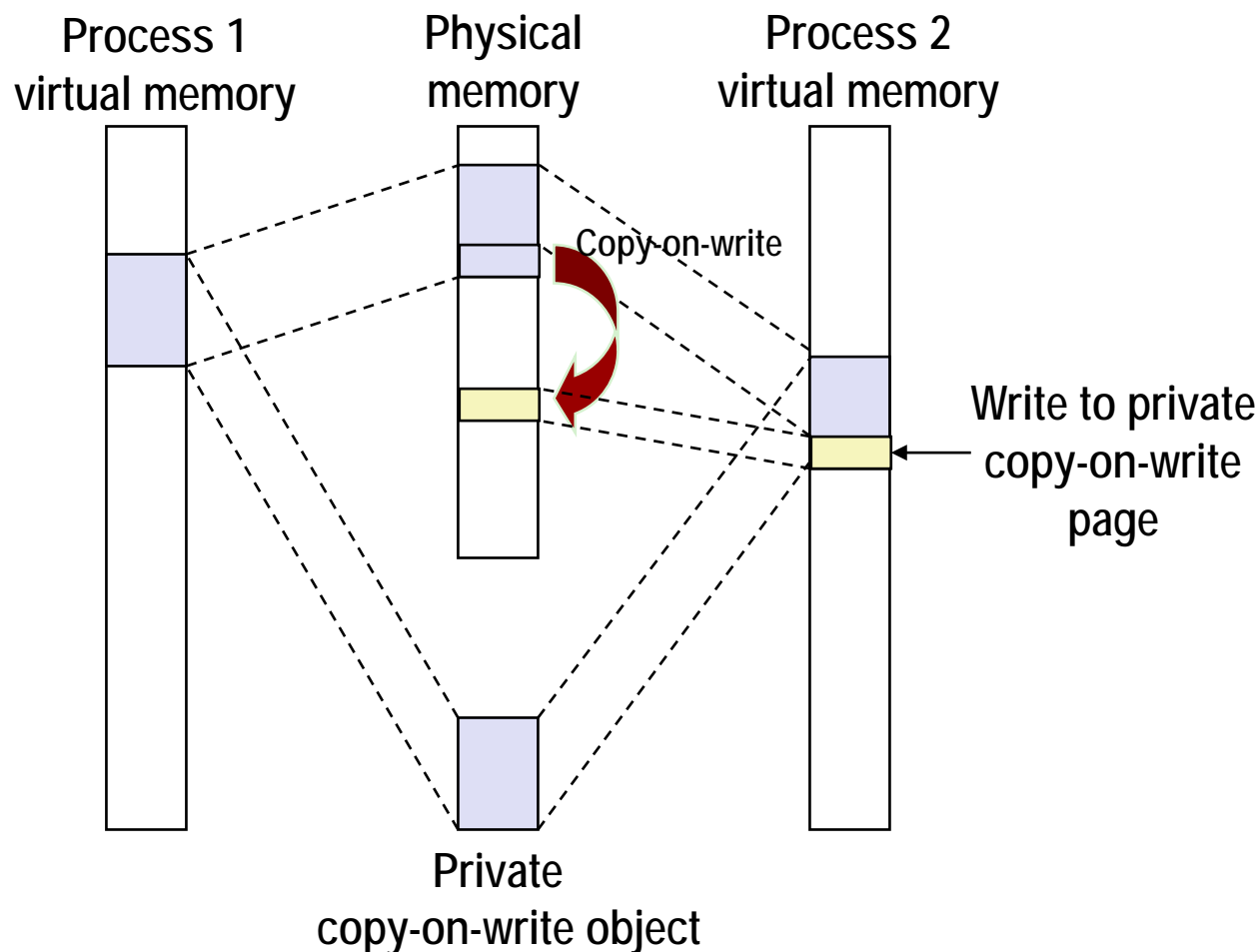
# Sharing Revisited: Private Copy-on-write (COW) Objects



- 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

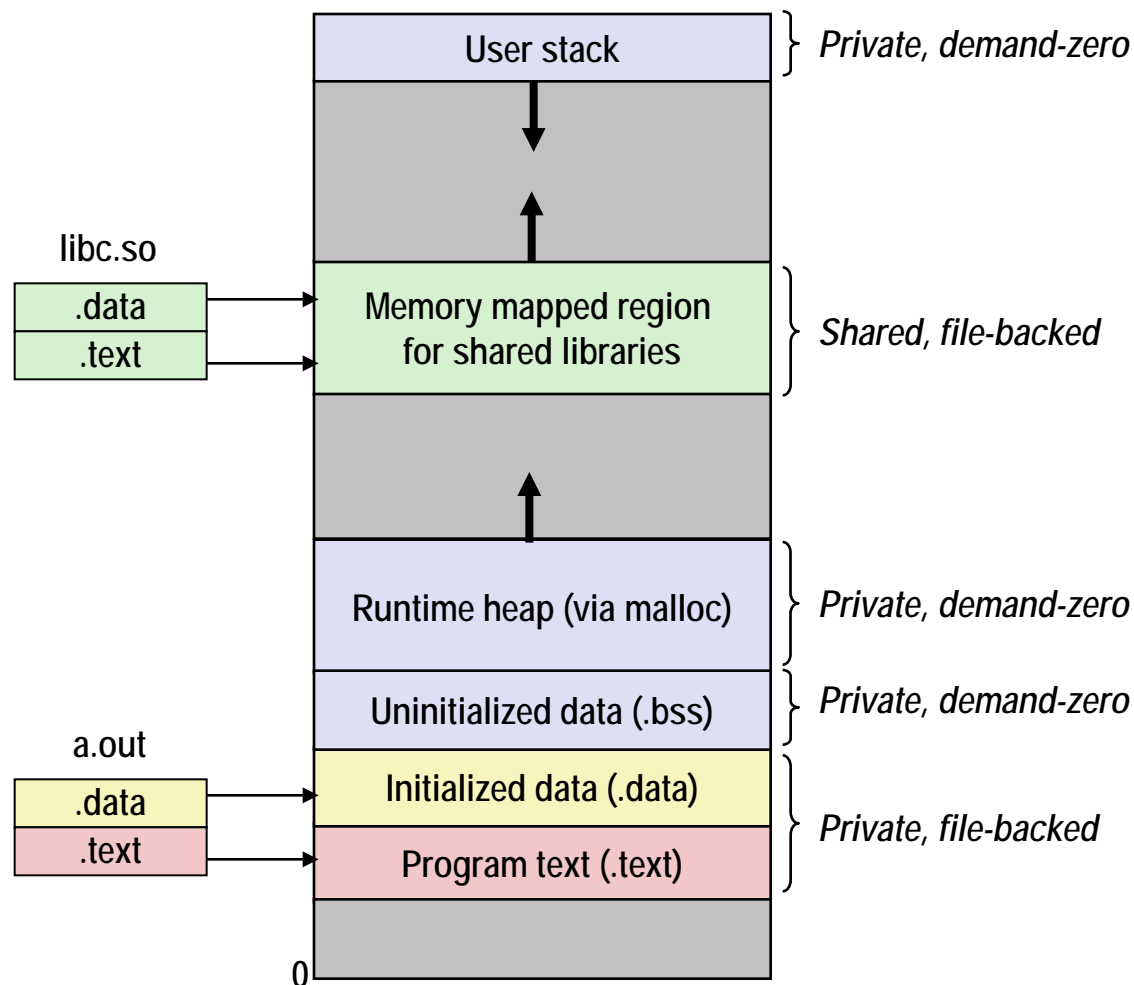


- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

# The `fork` Function Revisited

- VM and memory mapping explain how `fork` provides private address space for each process.
- **To create virtual address for new new process**
  - Create exact copies of current `mm_struct`, `vm_area_struct`, and page tables.
  - Flag each page in both processes as read-only
  - Flag each `vm_area_struct` in both processes as private COW
- **On return, each process has exact copy of virtual memory**
- **Subsequent writes create new pages using COW mechanism.**

# 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.

# User-Level Memory Mapping

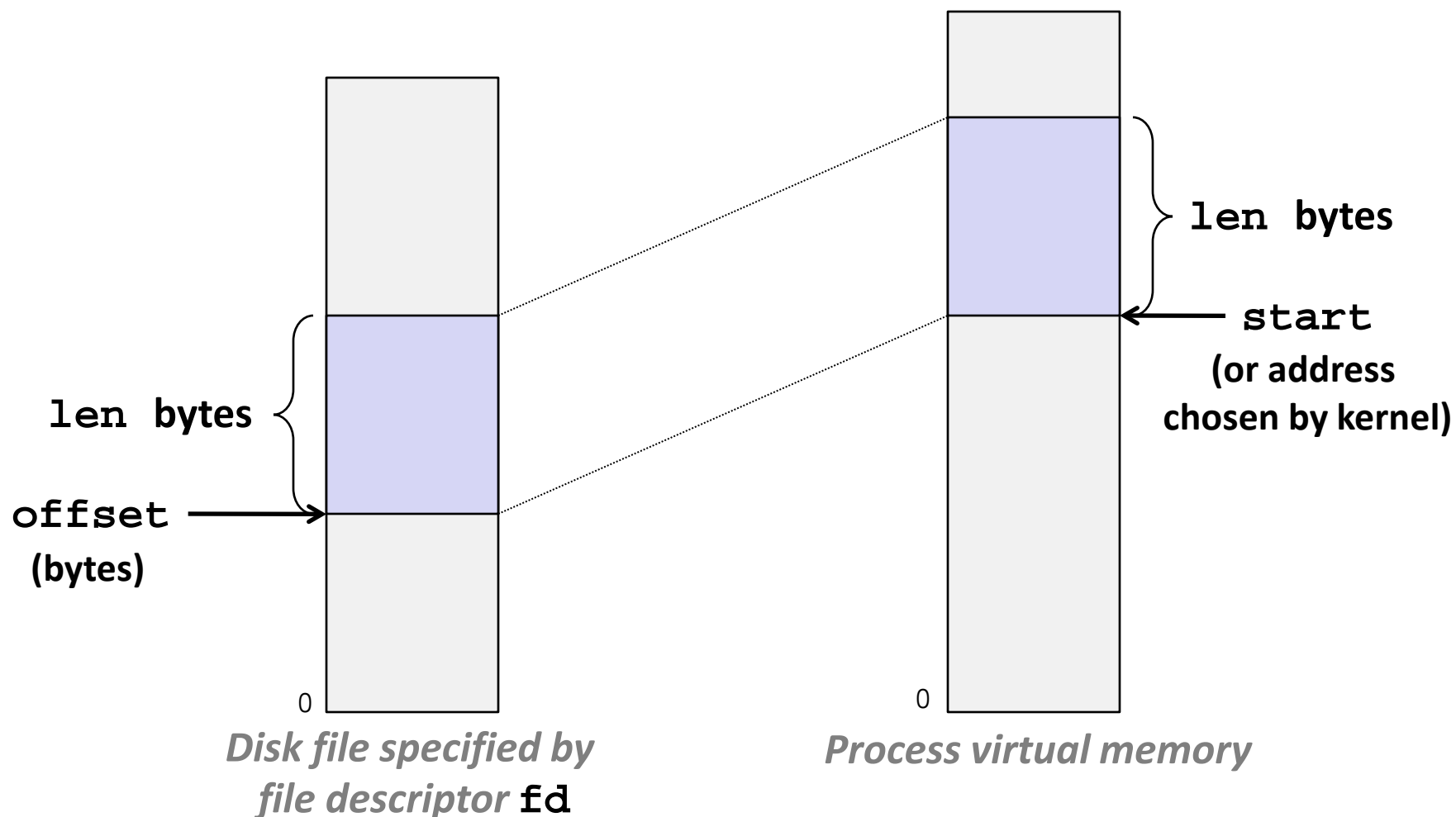
```
void *mmap(void *start, int len,  
           int prot, int flags, int fd, int offset)
```

- **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)
```



# Example: Using `mmap` to Copy Files

- Copying a file to `stdout` without transferring data to user space .

```
#include "csapp.h"

void mmapcopy(int fd, int size)
{

    /* Ptr to memory mapped area */
    char *bufp;

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

mmapcopy.c

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

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

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

mmapcopy.c

# Summary: Virtual Memory & Caches

## ■ 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

