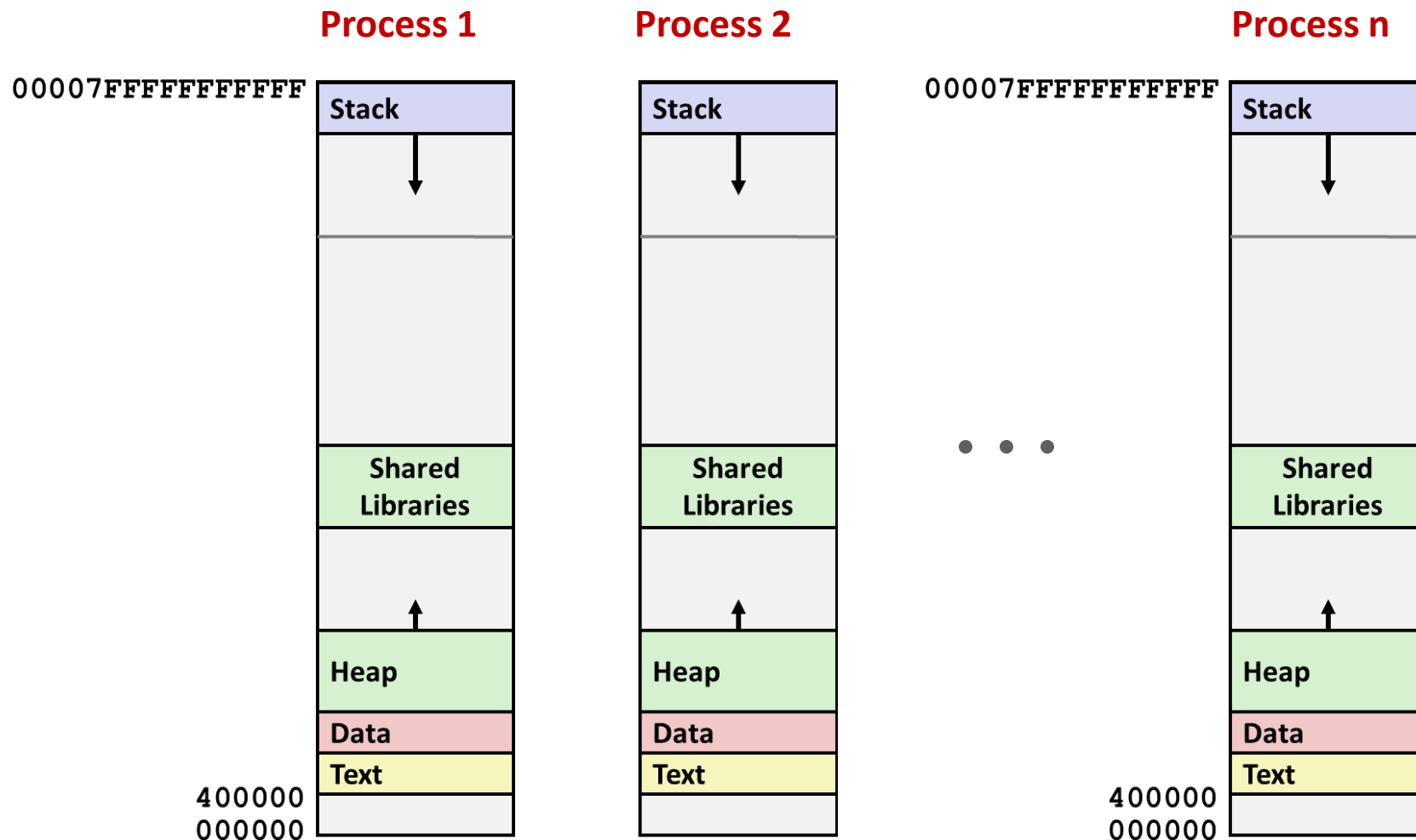# Virtual Memory: Systems

15-213: Introduction to Computer Systems
18th Lecture, March 27, 2018

Instructor:
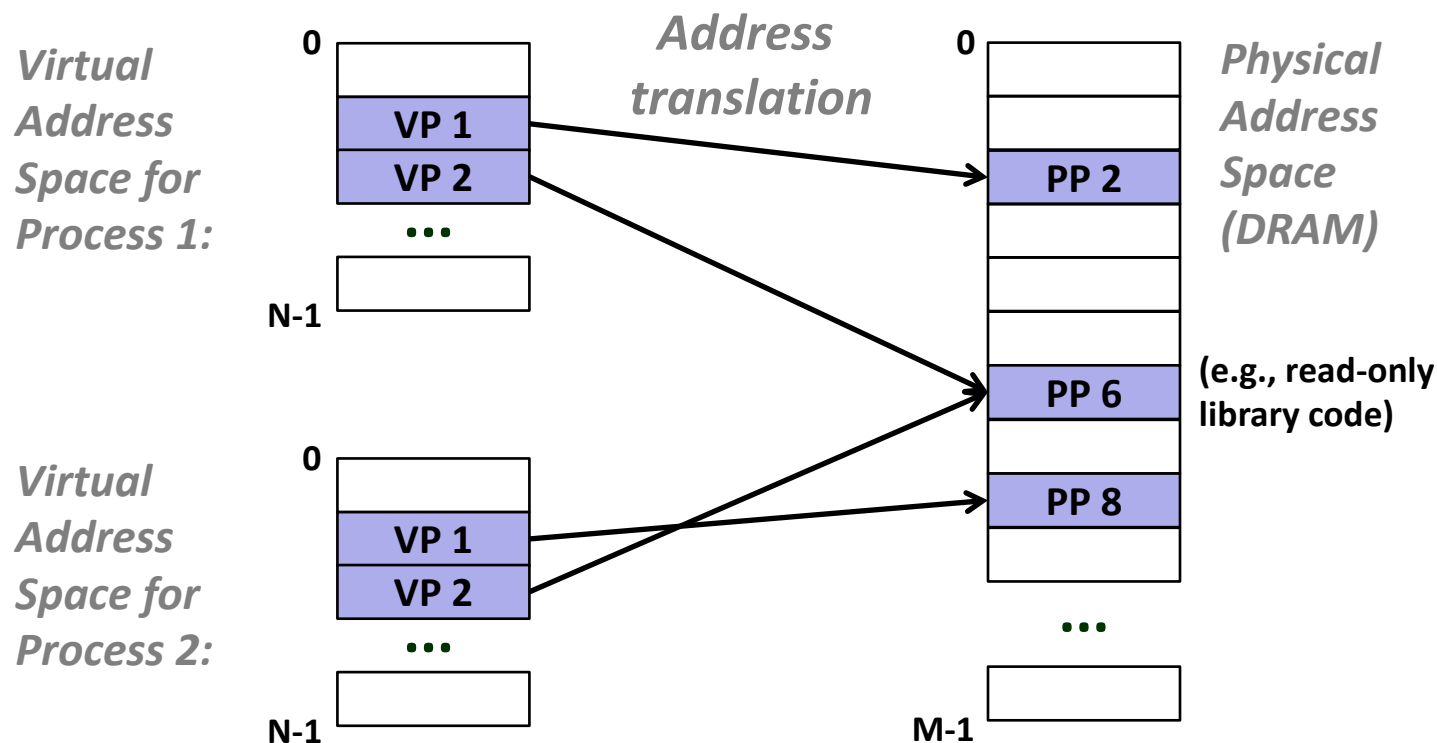
Franz Franchetti, Seth Copen Goldstein, and Brian Railing

Recap: Hmmm, How Does This Work?!

Process 1

Process 2

Process n

VM as a Tool for Memory Management

- **Simplifying memory allocation**
- **Sharing code and data among processes**

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)

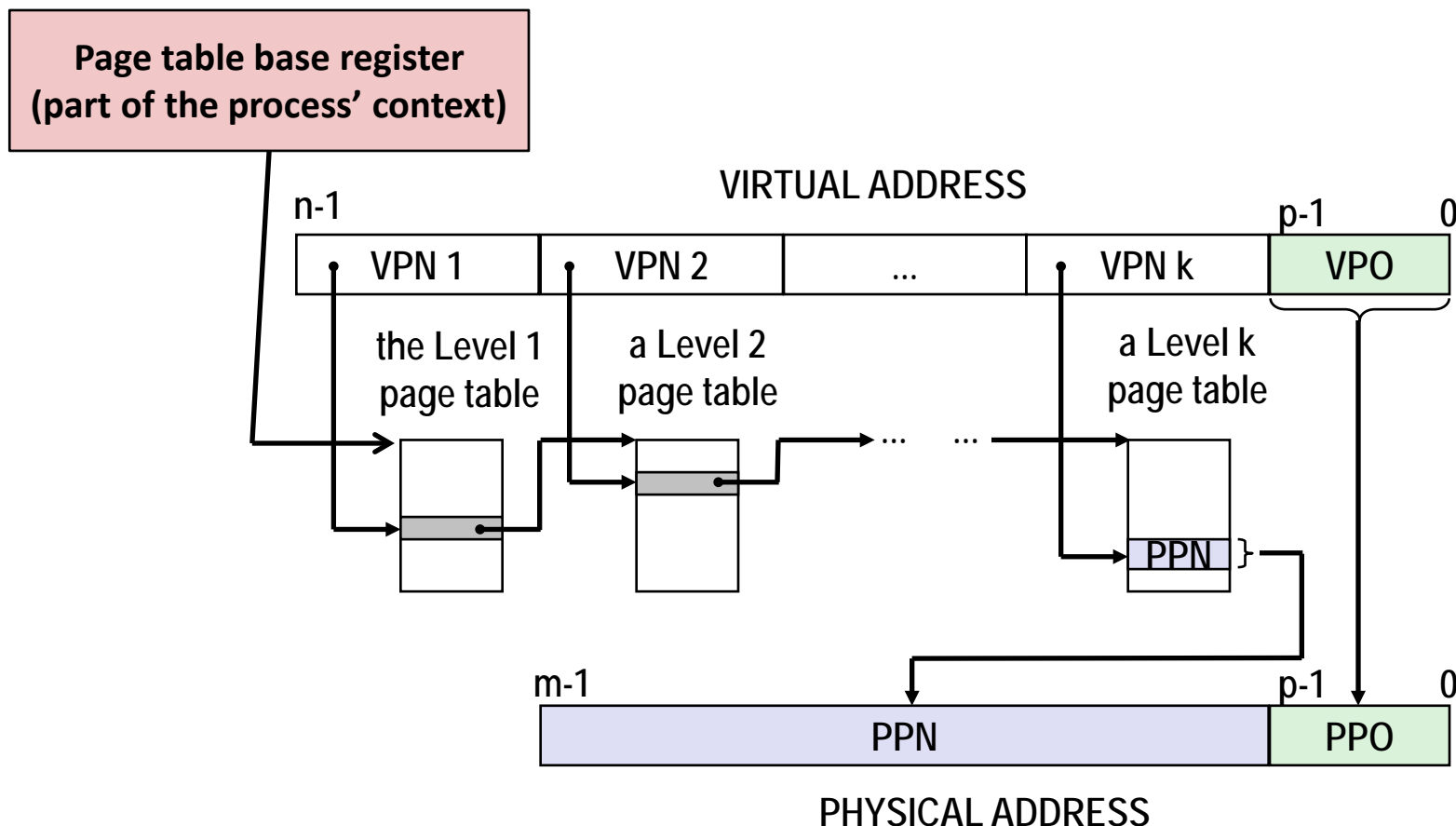*Virtual Address Space for Process 2:*

0
VP 1
VP 2
...
N-1

Review: Virtual Memory & Physical Memory
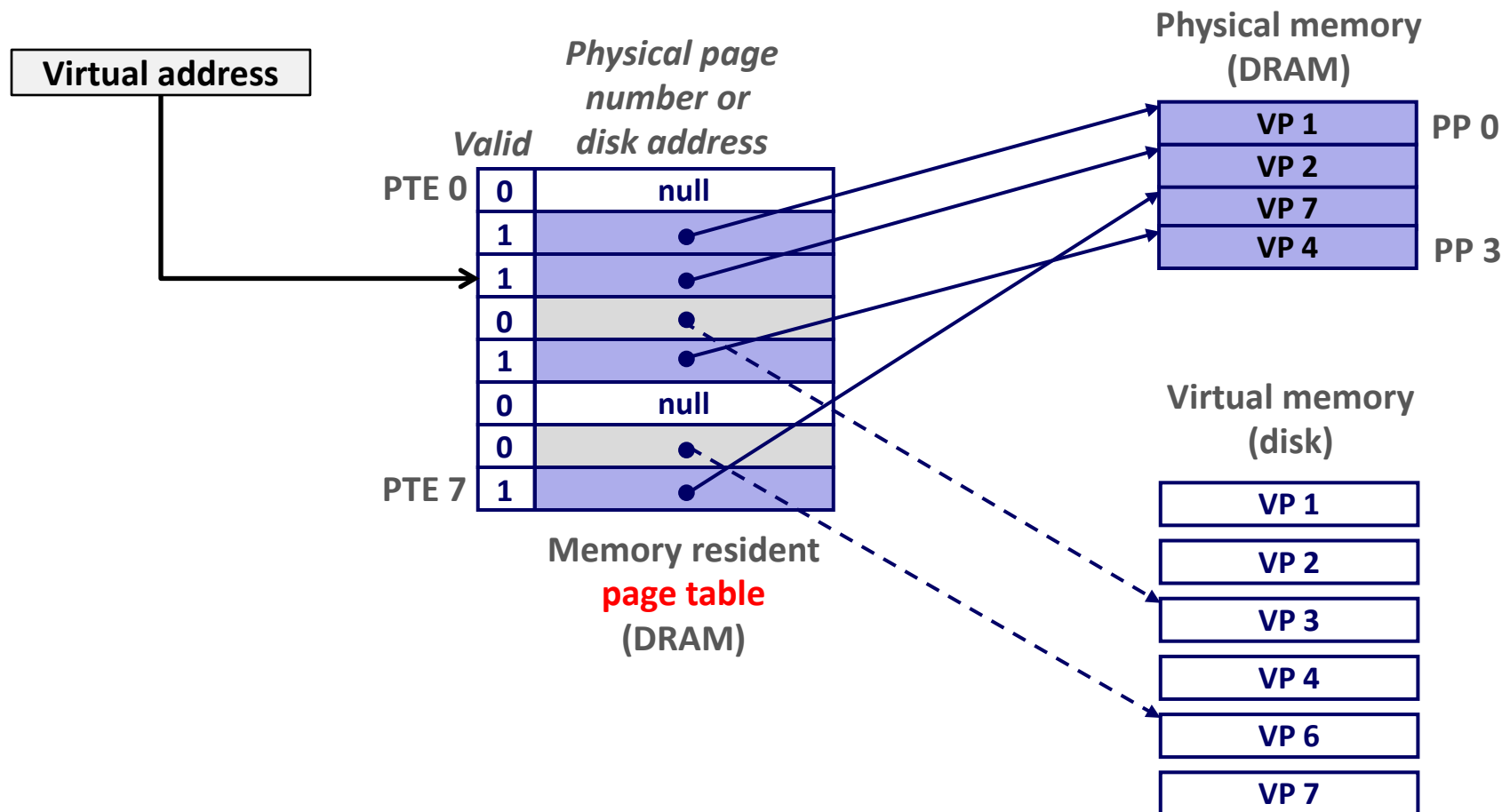
■ A *page table* contains page table entries (PTEs) that map virtual pages to physical pages.

Translating with a k-level Page Table

- **Having multiple levels greatly reduces page table size**

Page table base register
(part of the process' context)

VIRTUAL ADDRESS

n-1			p-1	0
VPN 1	VPN 2	...	VPN k	VPO

the Level 1 page table   a Level 2 page table   a Level k page table

... ...

PPN

m-1	p-1	0
PPN	PPO	

PHYSICAL ADDRESS

Translation Lookaside Buffer (TLB)

■ **A small cache of page table entries with fast access by MMU**
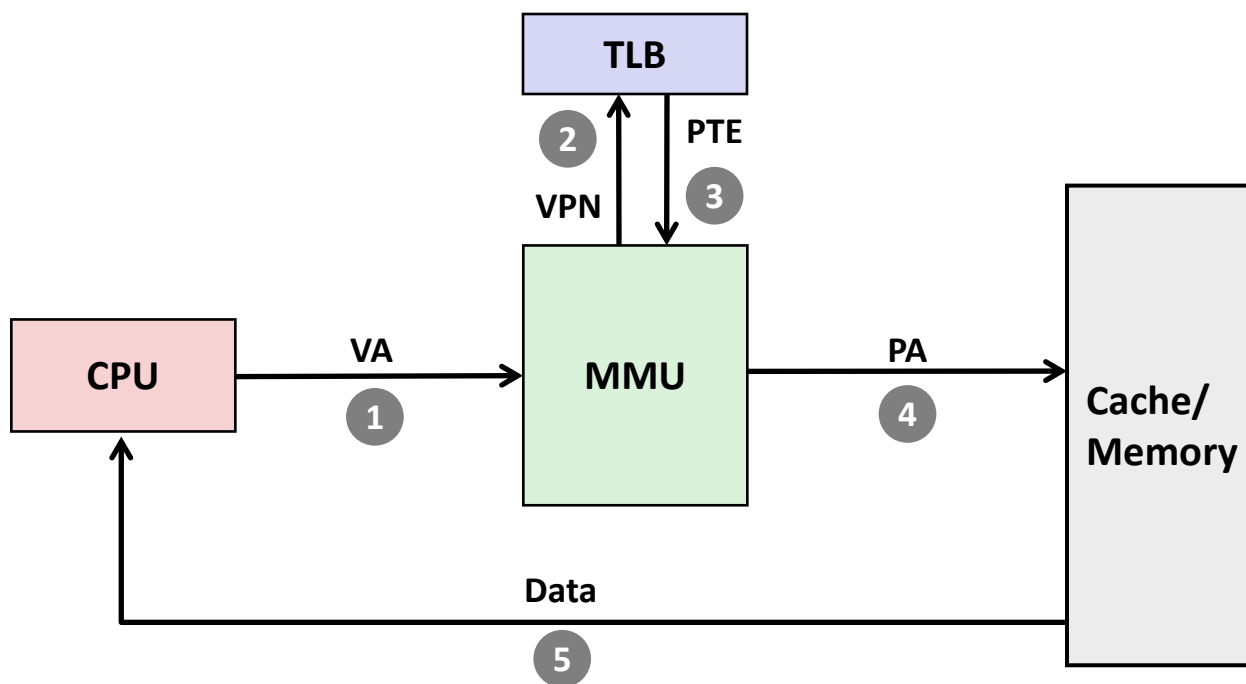
Typically, a TLB hit eliminates the k memory accesses required to do a page table lookup.

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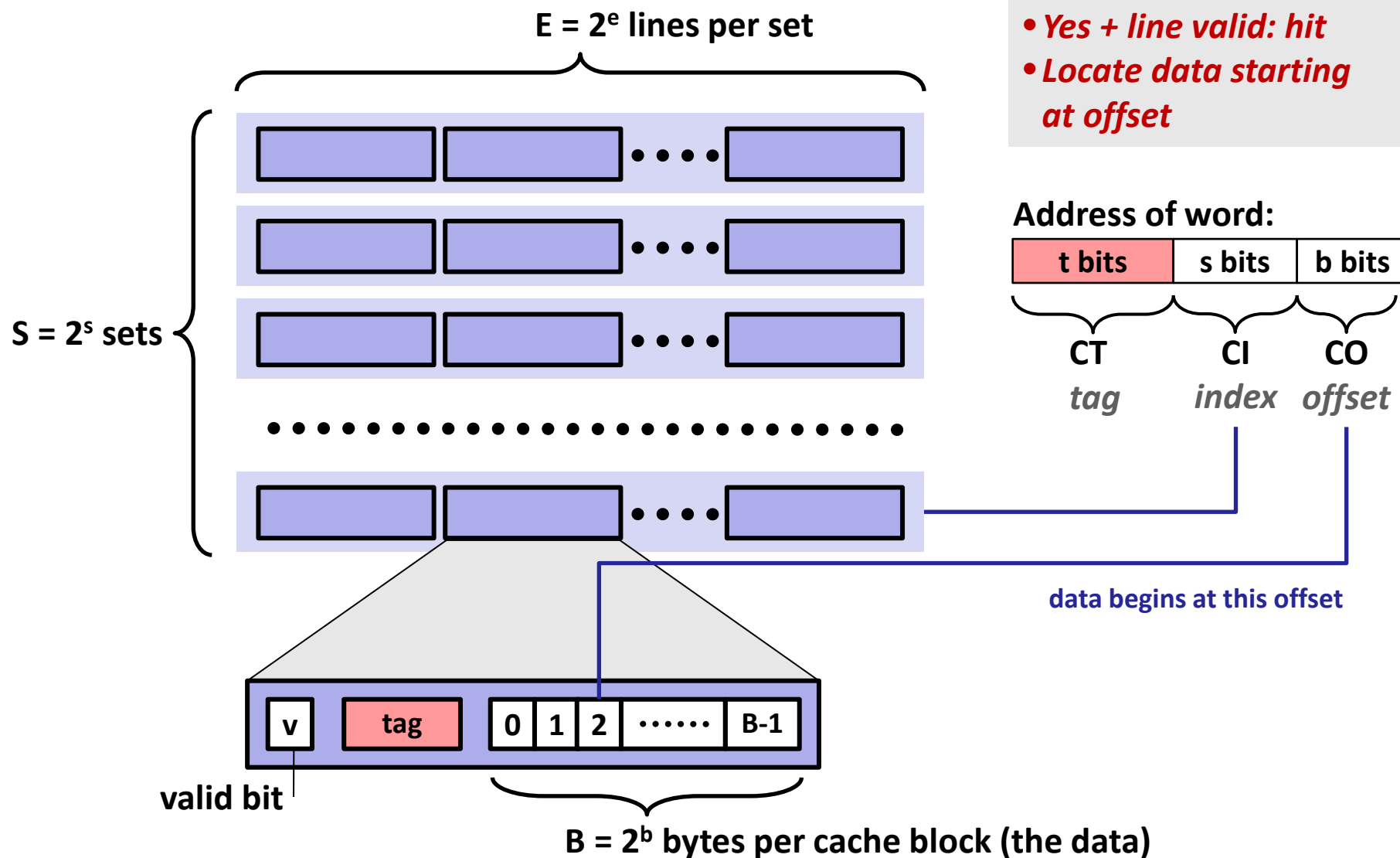
- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*

E = 2^e lines per set

S = 2^s sets

Address of word:

t bits	s bits	b bits

CT
tag

CI
index

CO
offset

data begins at this offset

v	tag	0	1	2	······	B-1

valid bit

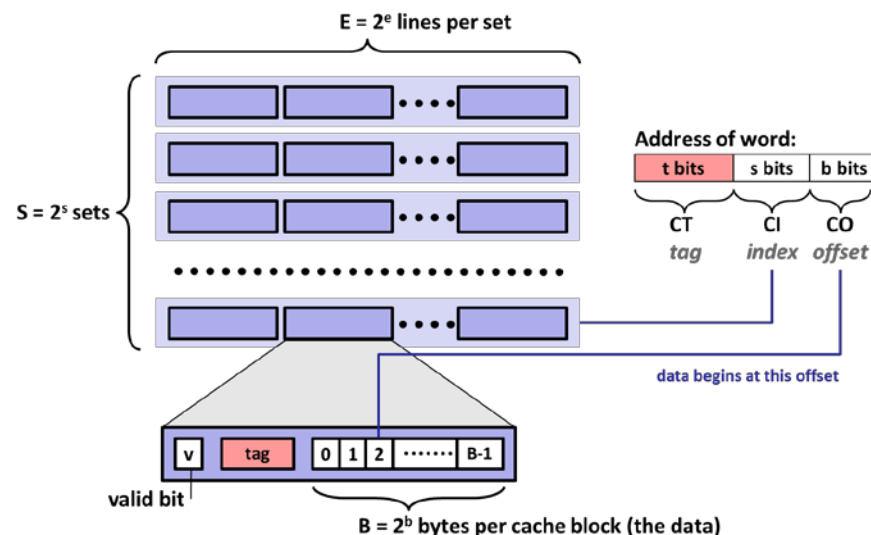**B = 2^b bytes per cache block (the data)**

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
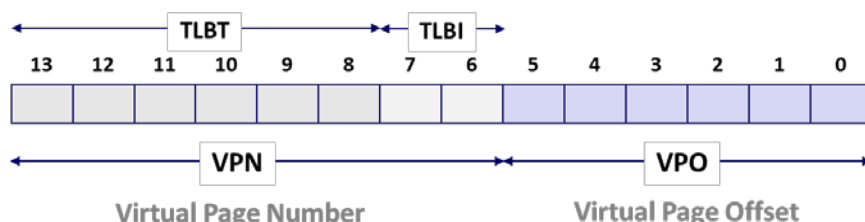
(bits per field for our simple example)

Today

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

Simple Memory System Example

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

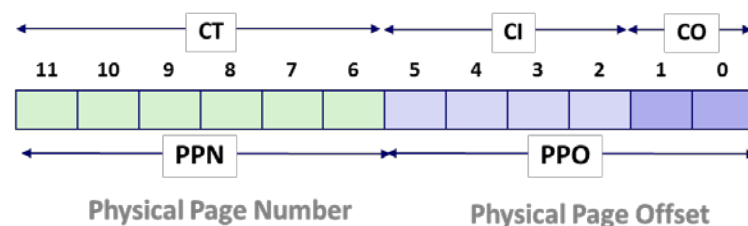| 13	12	11	10	9	8	7	6	5	4	3	2	1	0

◄————————————————— VPN ———————————————►◄——————————— VPO ———————————►

Virtual Page Number   **Virtual Page Offset**

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

◄——————————— PPN ———————————►◄——————————— PPO ———————————►

Physical Page Number   **Physical Page Offset**
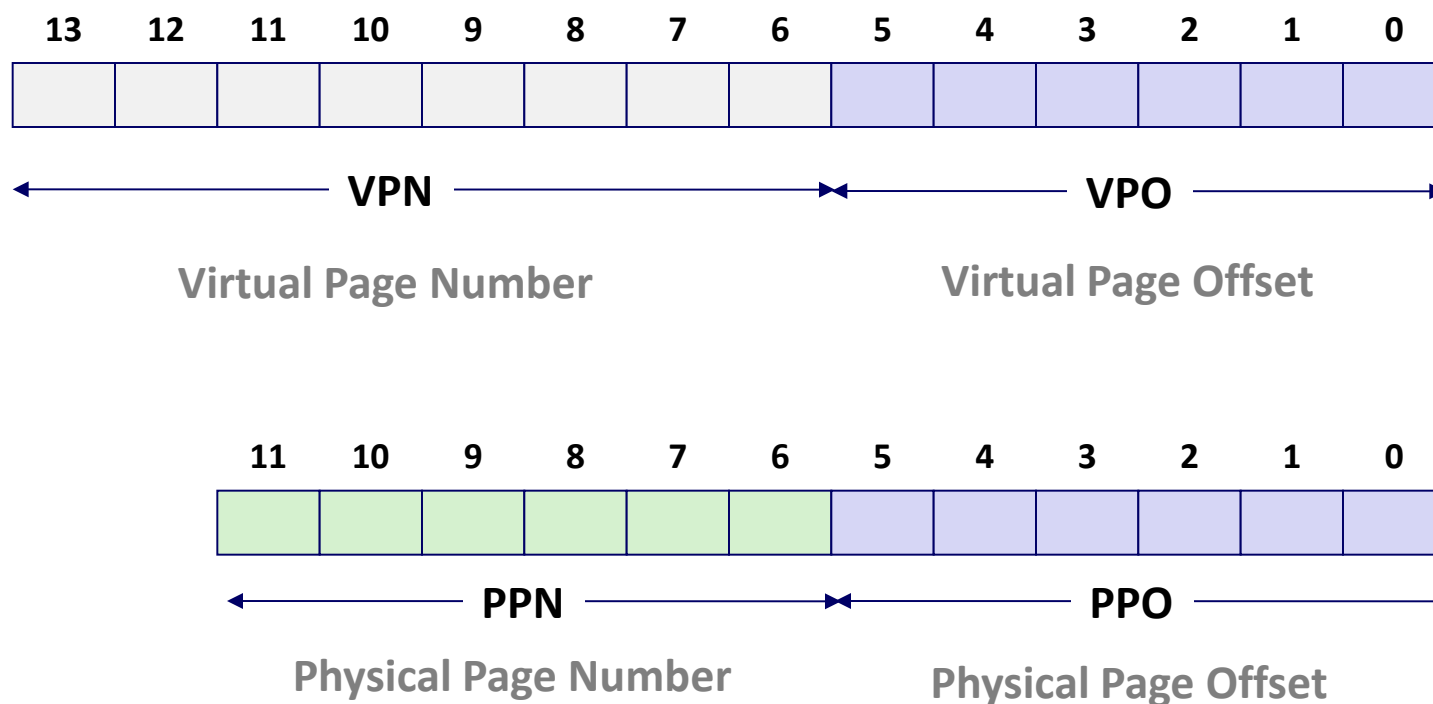
Simple Memory System TLB

- **16 entries**
- **4-way associative**

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

TLBT ← → TLBI

VPN ← → VPO
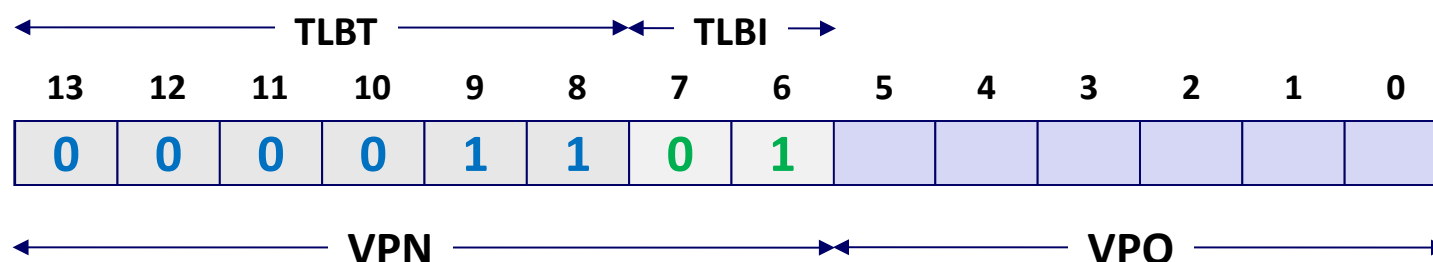
VPN = 0b1101 = 0x0D

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)

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

0x0D → 0x2D

Simple Memory System Cache

- ## 16 lines, 4-byte block size

- ## Physically addressed

- ## Direct mapped

V[0b00001101101001] = V[0x369]

P[0b101101101001] = P[0xB69] = **?**

	CT						CI				CO	

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

PPN — PPO

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	–	–	–	–

Simple Memory System Cache

- **16 lines, 4-byte block size**

- **Physically addressed**

- **Direct mapped**

V[0b00001101101001] = V[0x369]
P[0b101101101001] = P[0xB69] = **0x15**

| | | CT | | | | | | CI | | | CO | |

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

PPN ← → PPO

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

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

Set	CT			CT			CI			CO		
	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

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

	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**

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

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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
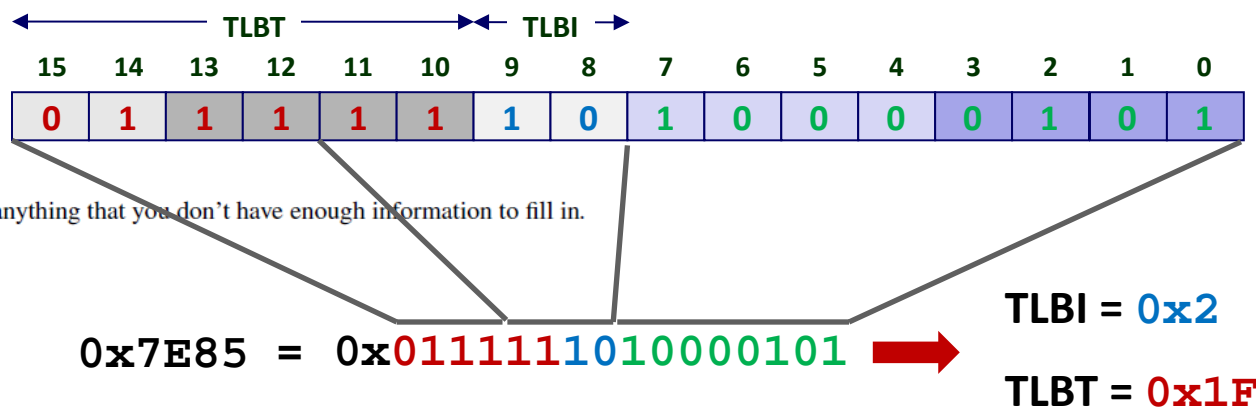4. 2^{16} bytes of virtual memory
5. one (or more) boats

TLB

Index	Tag	Frame Number	Valid
0	0x13	0x30	1
	0x34	0x58	0
1	0x1F	0x80	0
	0x2A	0x72	1
2	0x1F	0x95	1
	0x20	0xAA	0
3	0x3F	0x20	1
	0x3E	0xFF	0

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

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

TLBT ← → TLBI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	0	0	0	0	1	0	1

0x7E85 = 0x0111111010000101 →

TLBI = 0x2

TLBT = 0x1F

0x7E85 → 0x9585

Break Time!

flummox: "To confuse (a lot)"

Check out:

Quiz: day 18: VM
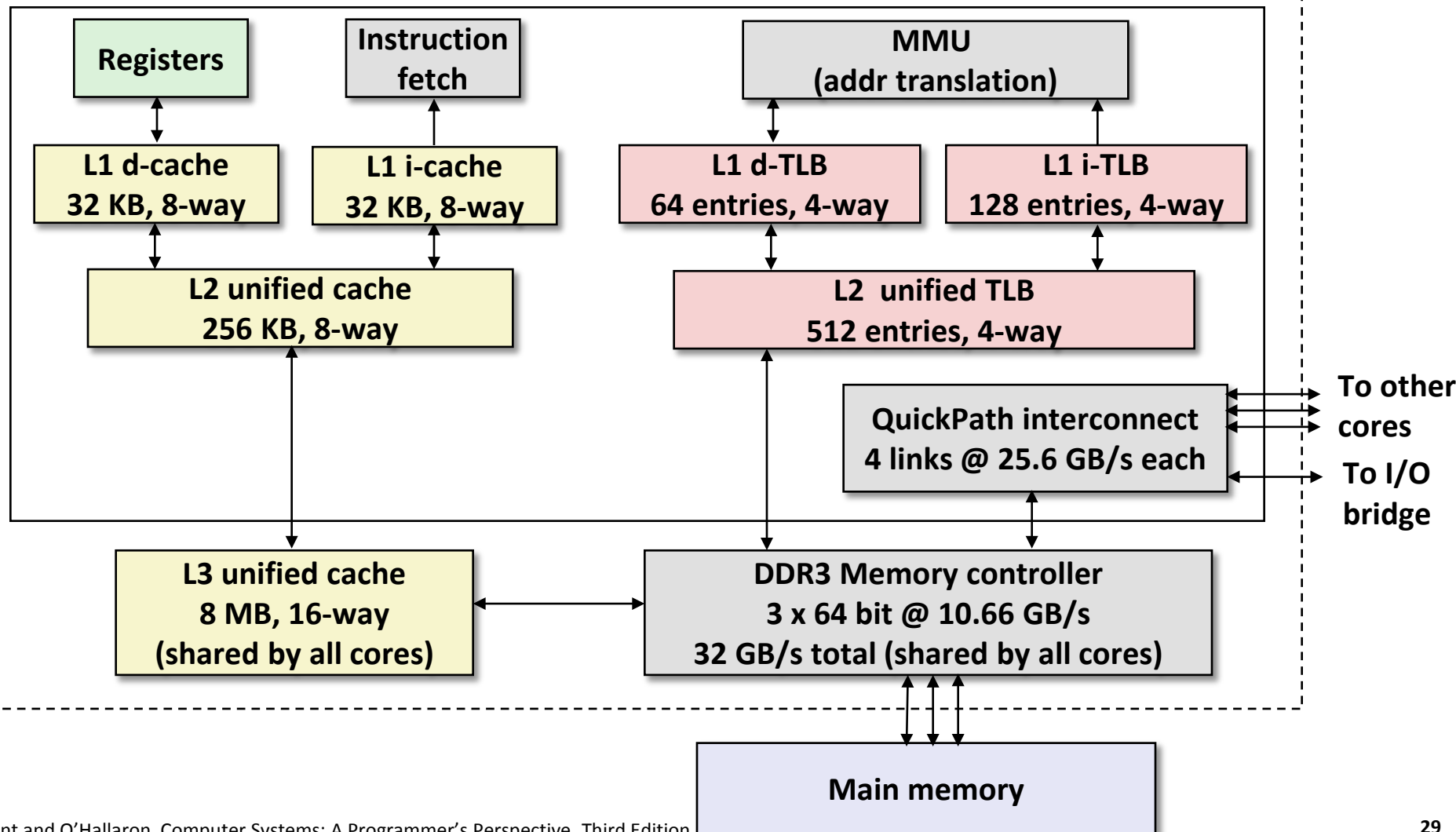
https://canvas.cmu.edu/courses/3822

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

End-to-end Core i7 Address Translation
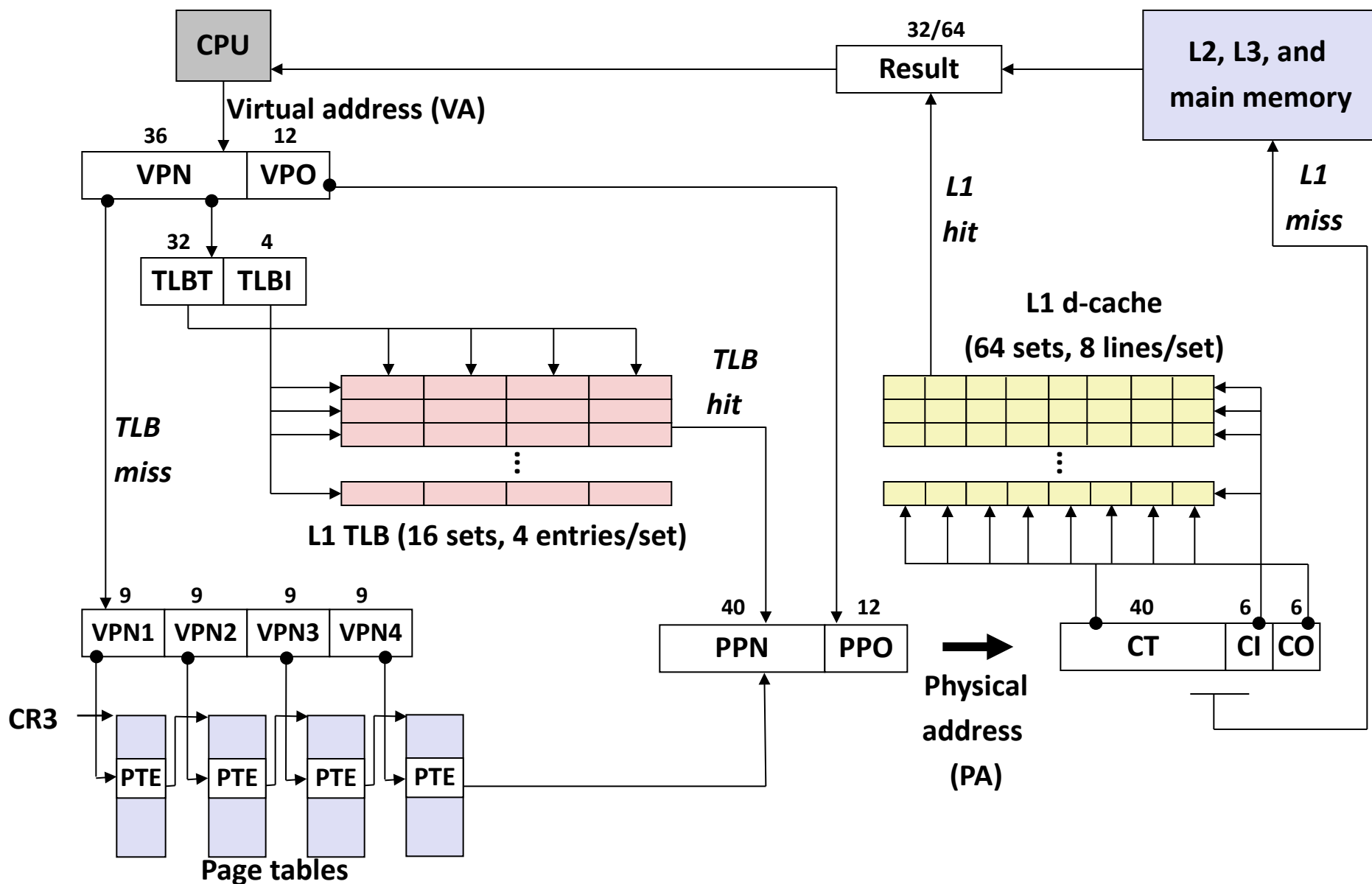
Core i7 Level 1-3 Page Table Entries

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

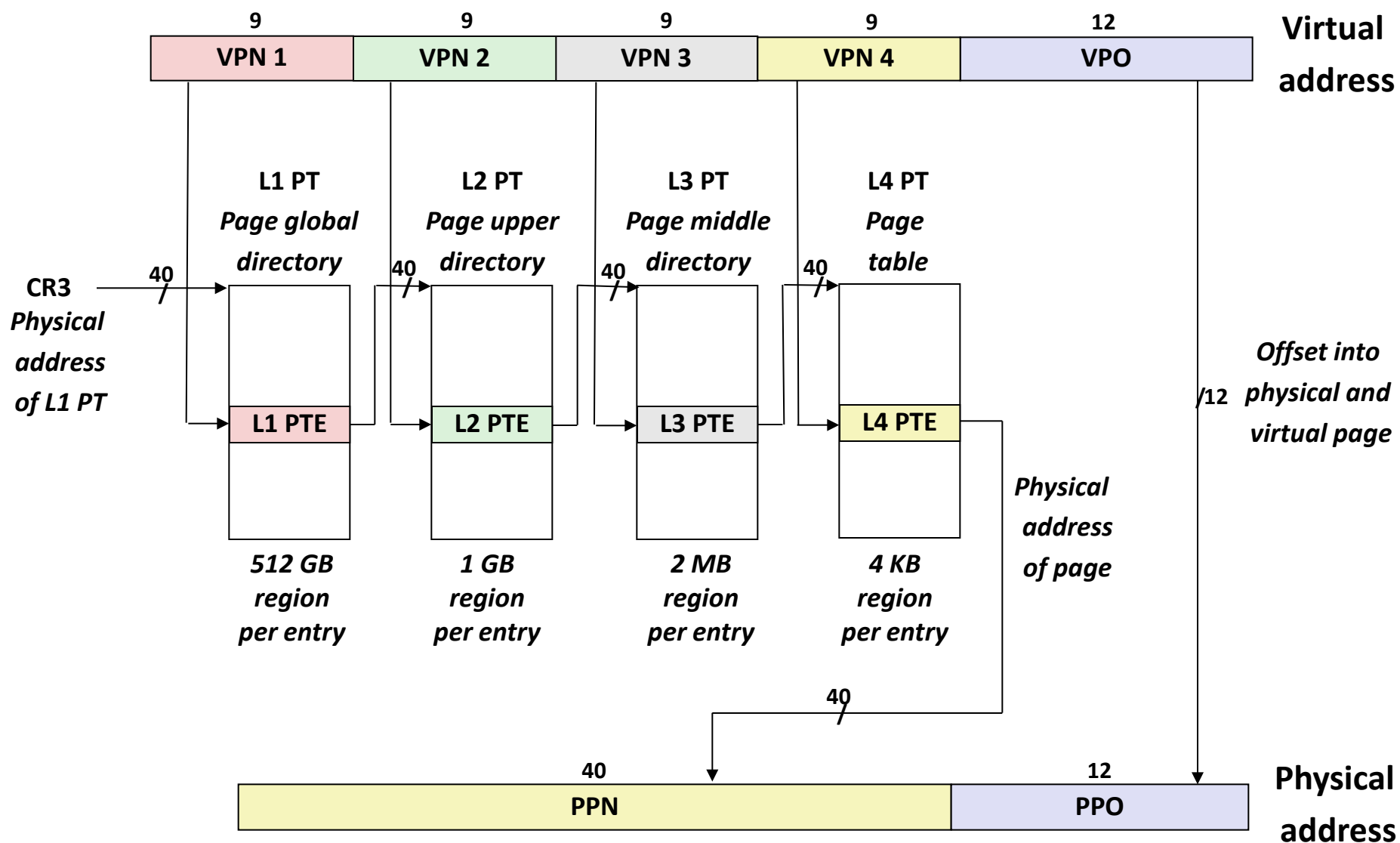**WT:** Write-through or write-back cache policy for this page

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

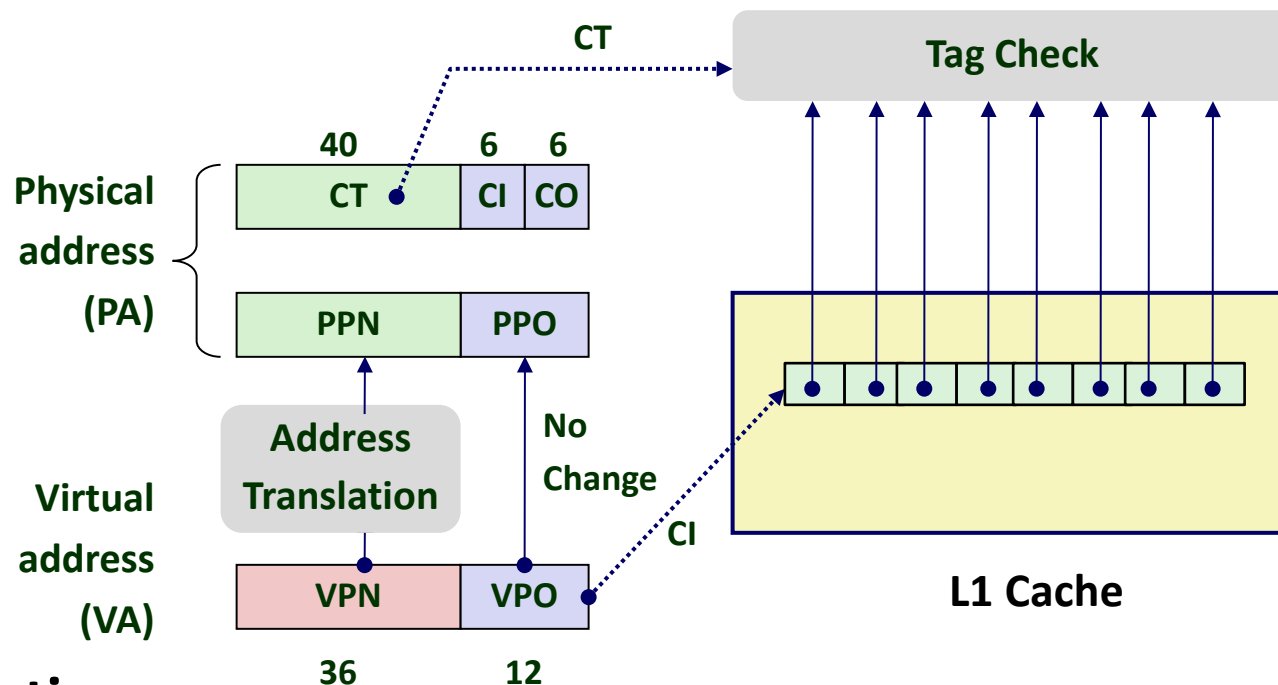**D:** Dirty bit (set by MMU on writes, cleared by software)

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.
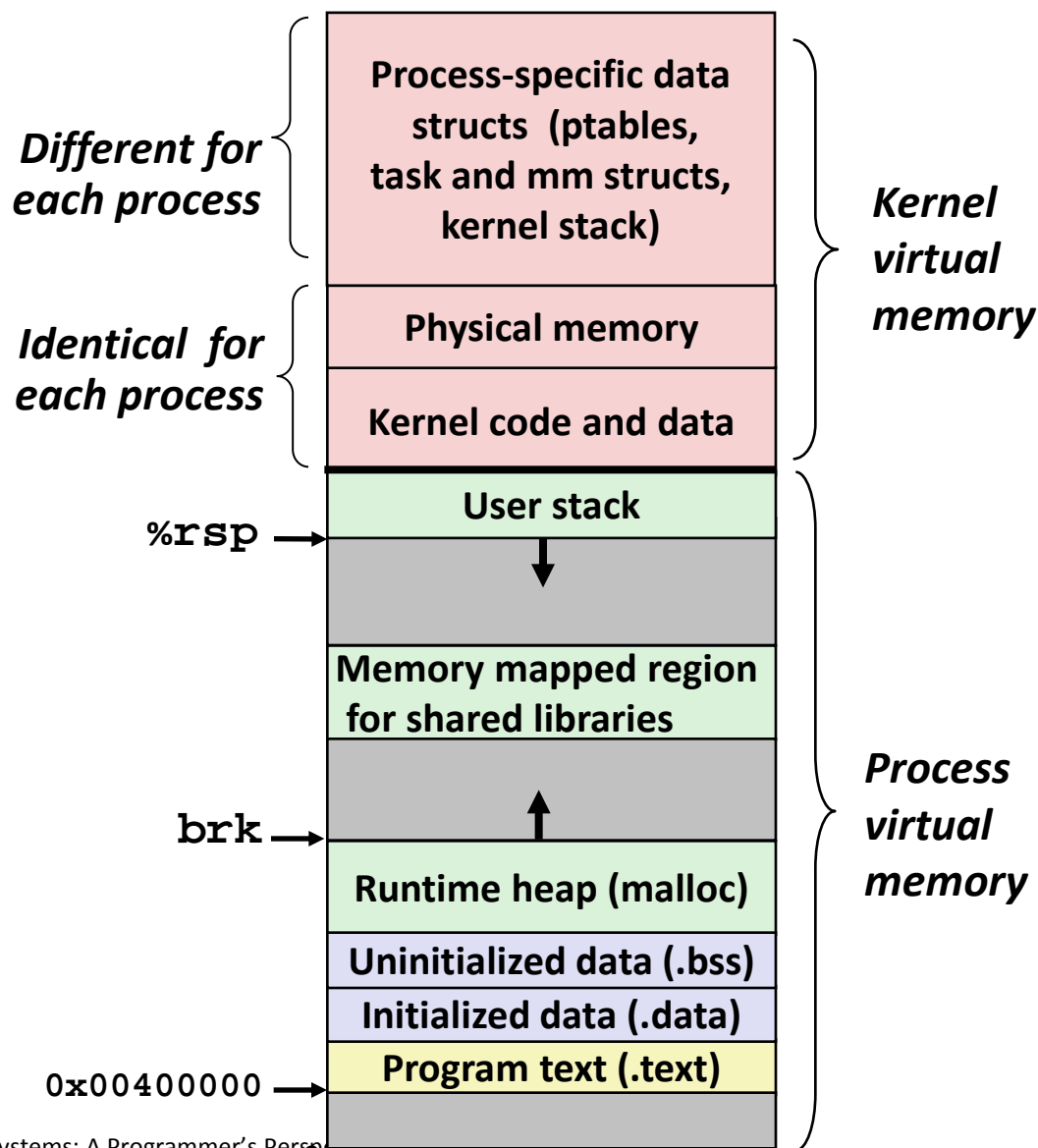
Core i7 Page Table Translation

	9	9	9	9	12	
	VPN 1	VPN 2	VPN 3	VPN 4	VPO	**Virtual address**

CR3
Physical address of L1 PT

40

L1 PT
Page global directory

L2 PT
Page upper directory

L3 PT
Page middle directory

L4 PT
Page table

40   40   40

L1 PTE   L2 PTE   L3 PTE   L4 PTE

512 GB region per entry   *1 GB region per entry*   *2 MB region per entry*   *4 KB region per entry*

Physical address of page

Offset into physical and virtual page

/12

40

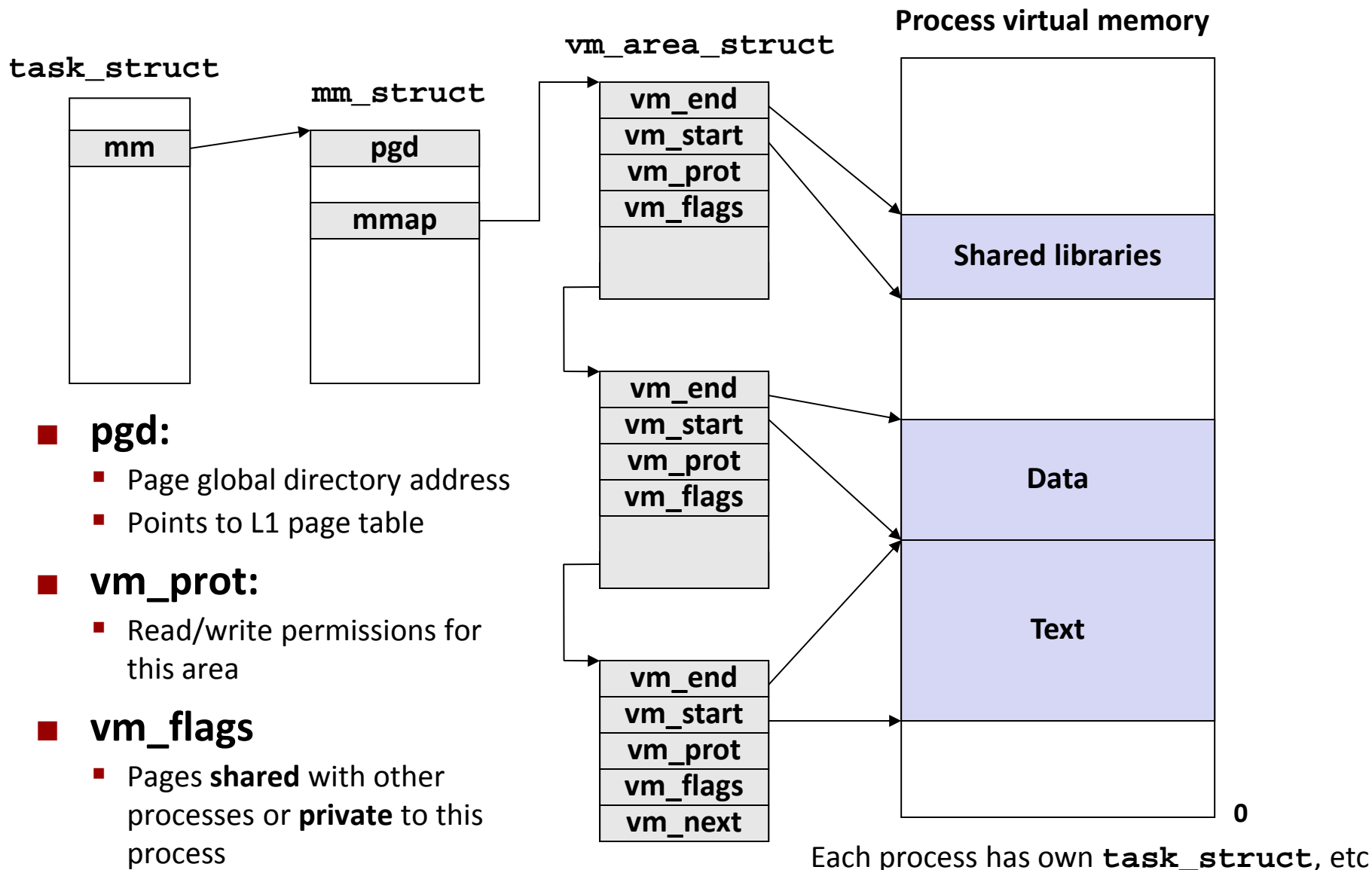| 40	12	
PPN	PPO	**Physical address**

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"*
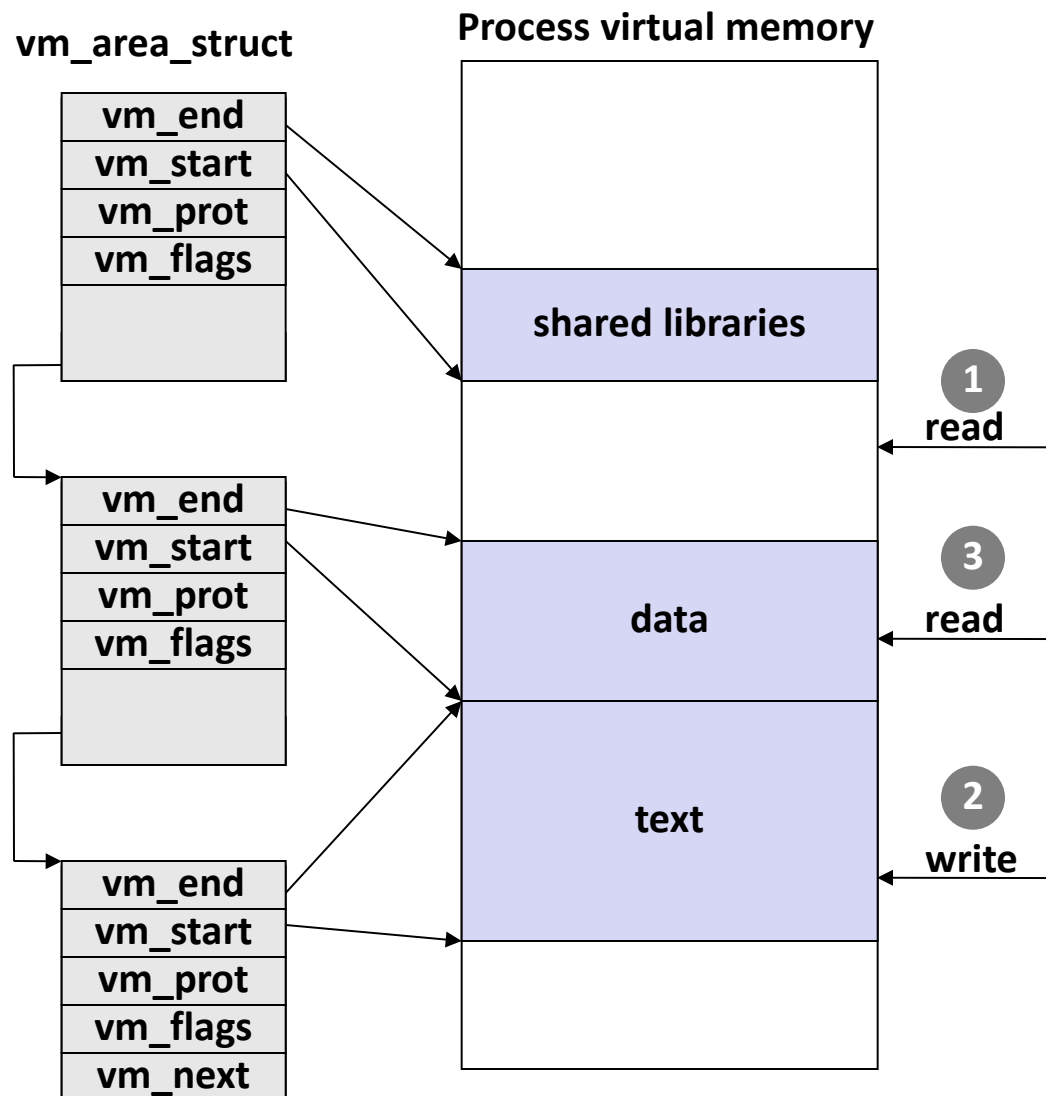- Cache carefully sized to make this possible

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)

`0x00400000`

Program text (.text)

Process virtual memory

0

Linux Organizes VM as Collection of "Areas"

task_struct

mm_struct

vm_area_struct

Process virtual memory

task_struct
mm

mm_struct
pgd
mmap

vm_area_struct
vm_end
vm_start
vm_prot
vm_flags

vm_end
vm_start
vm_prot
vm_flags

vm_end
vm_start
vm_prot
vm_flags
vm_next

Process virtual memory
Shared libraries
Data
Text

0

- **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

Each process has own **task_struct**, etc

Linux Page Fault Handling

vm_area_struct

Process virtual memory

| vm_end |
| vm_start |
| vm_prot |
| vm_flags |

shared libraries

1 read

Segmentation fault:
accessing a non-existing page

| vm_end |
| vm_start |
| vm_prot |
| vm_flags |

data

3 read

Normal page fault

text

2 write

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

| vm_end |
| vm_start |
| vm_prot |
| vm_flags |
| vm_next |

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**

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

Sharing Revisited: Shared Objects

Process 1
virtual memory

Physical
memory
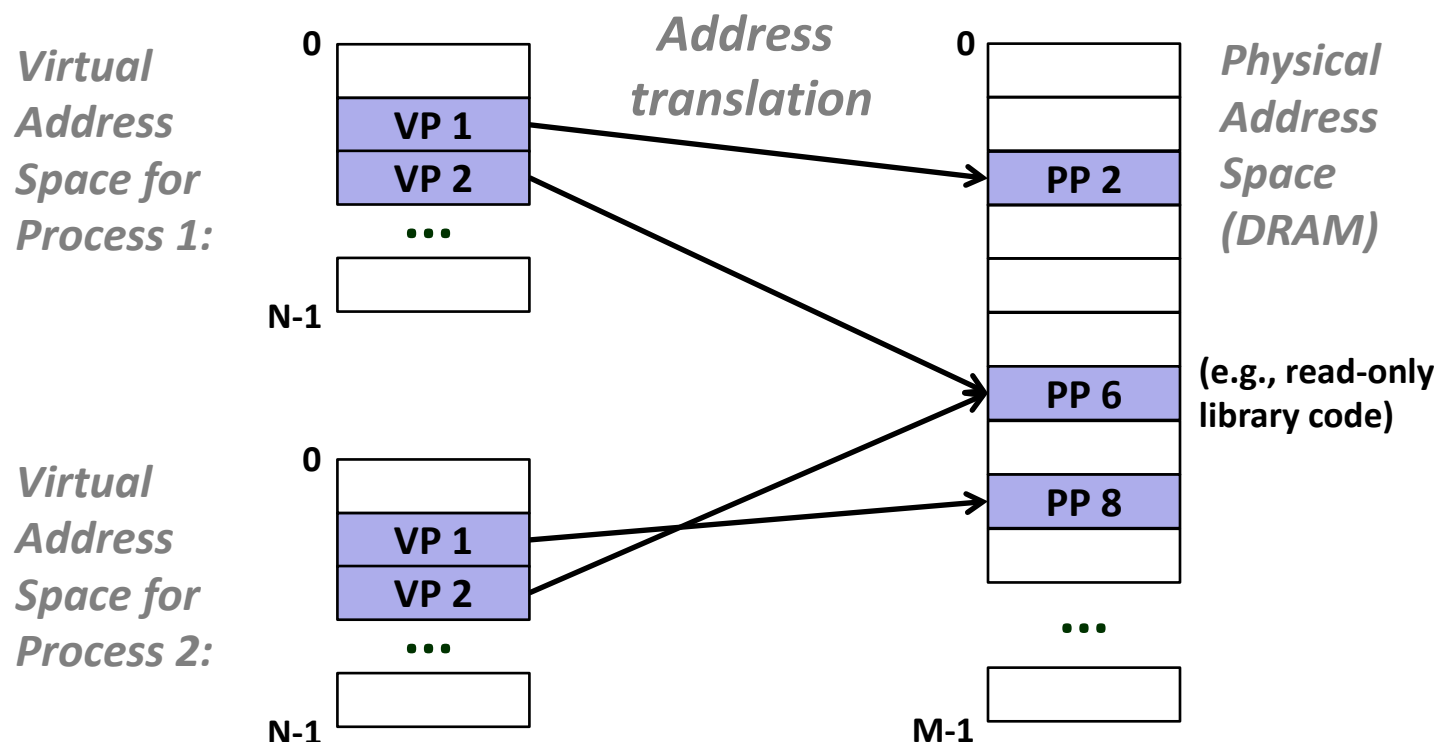
Process 2
virtual memory
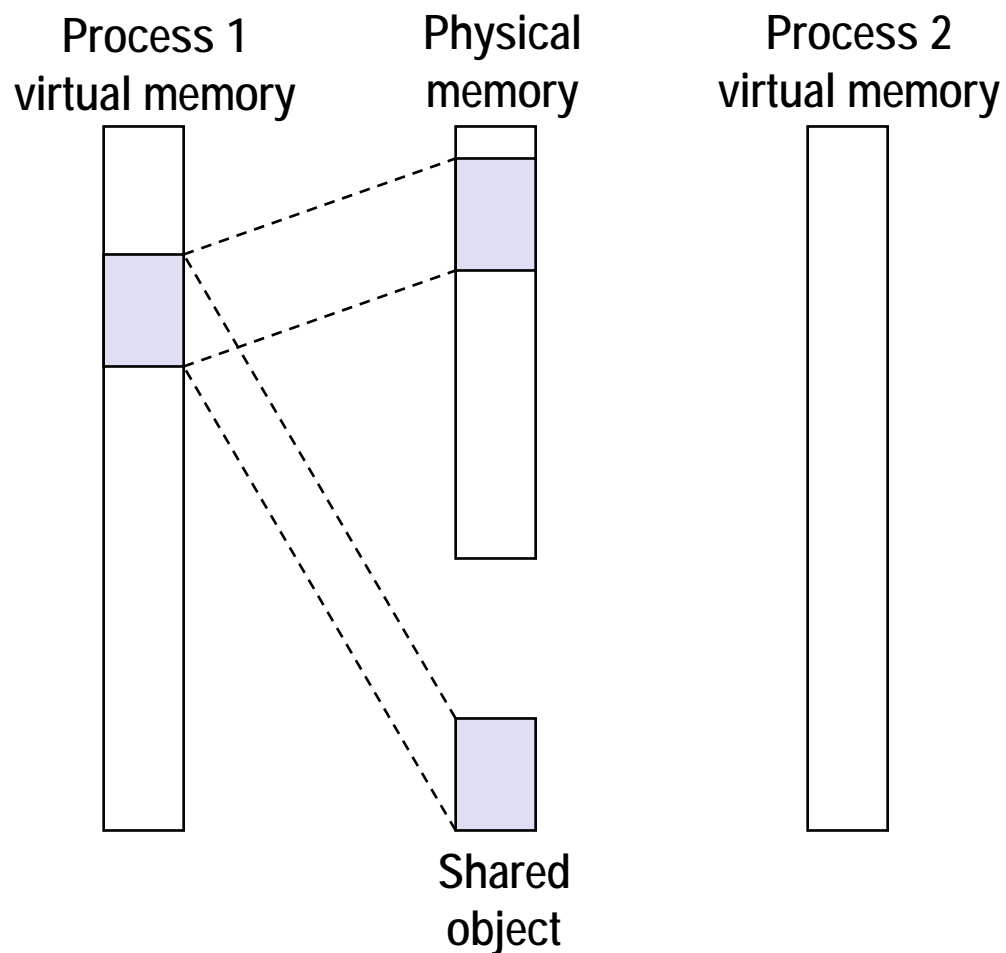
Shared
object

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

Sharing Revisited: Shared Objects

Process 1
virtual memory

Physical
memory

Process 2
virtual memory
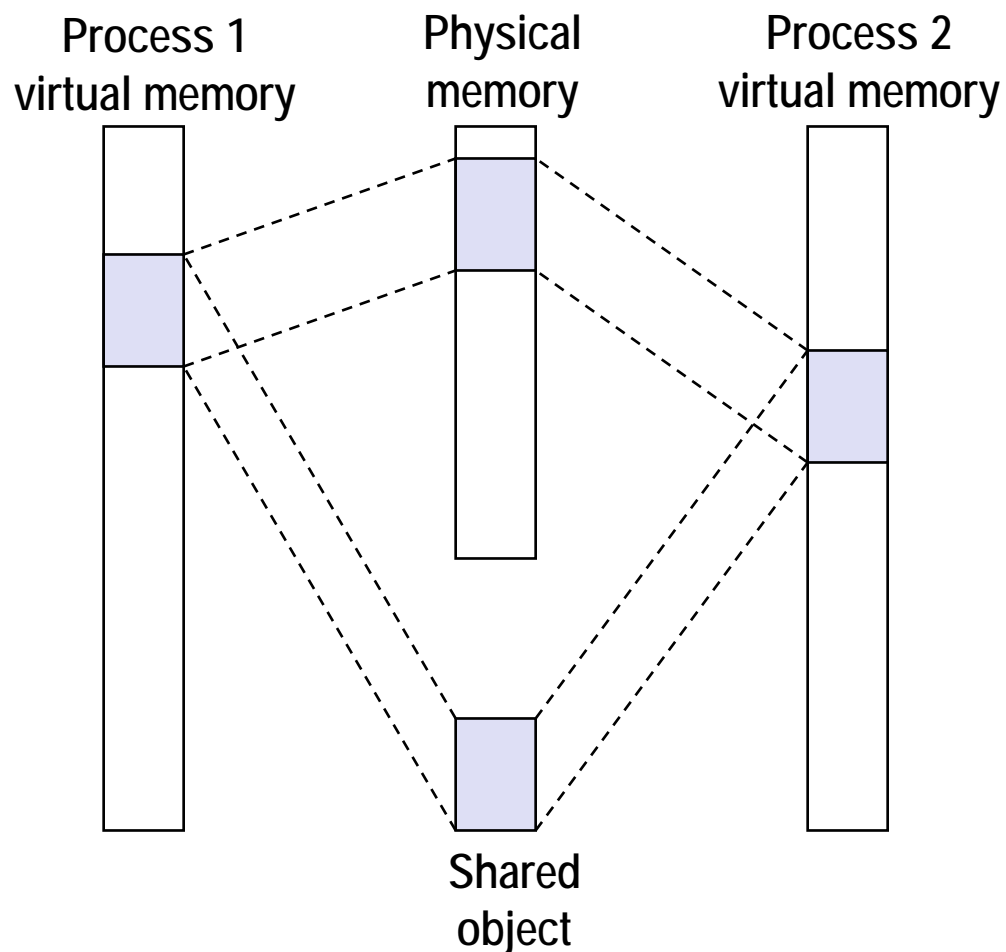
Shared
object

- **Process 2 maps the same shared object.**

- **Notice how the virtual addresses can be different.**

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

Process 1
virtual memory

Physical
memory

Process 2
virtual memory

Private
copy-on-write
area

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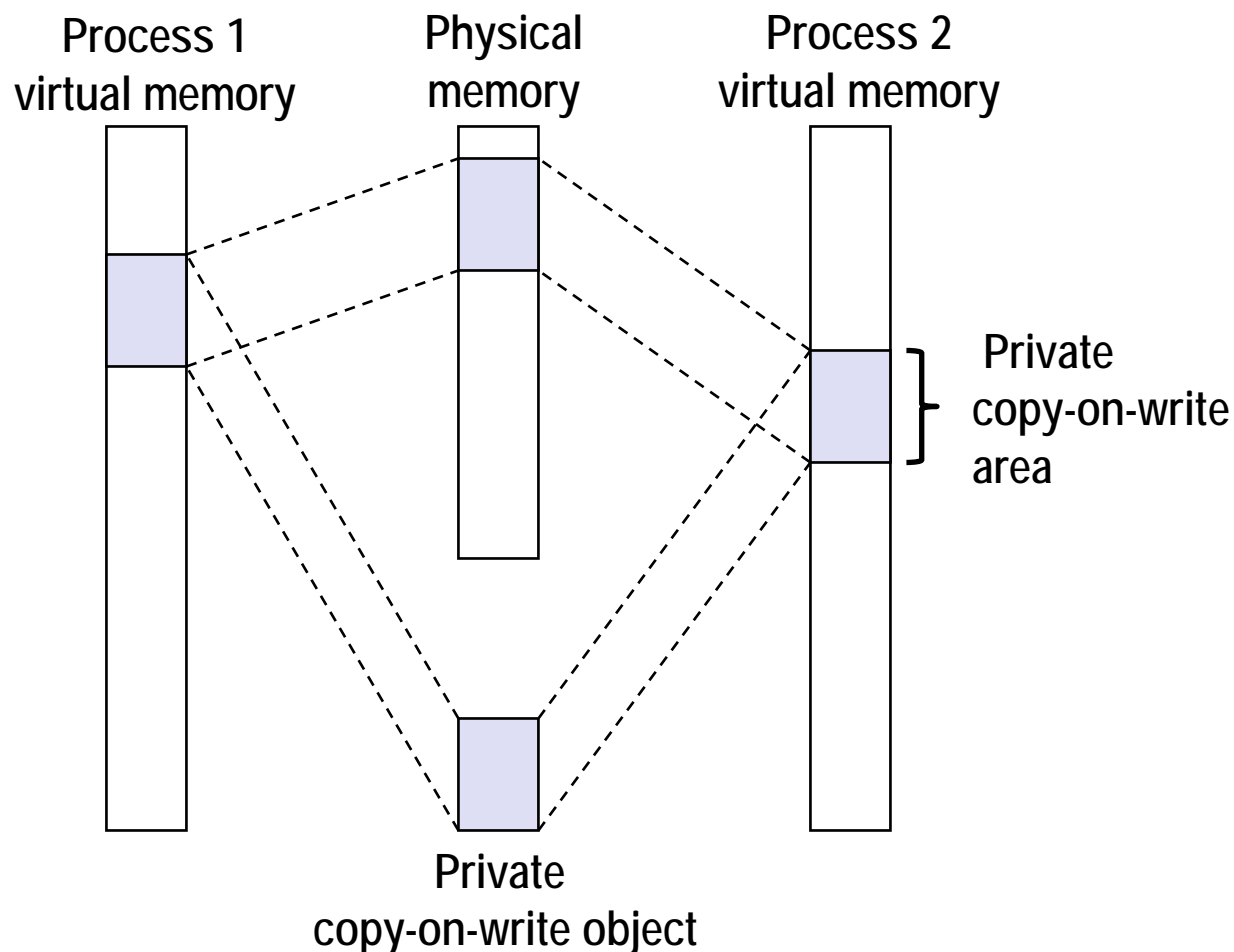
- **Two processes mapping a *private copy-on-write (COW)* object**
- **Area flagged as private copy-on-write**
- **PTEs in private areas are flagged as read-only**

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

Process 1 virtual memory

Physical memory

Process 2 virtual memory

Copy-on-write

Write to private copy-on-write page

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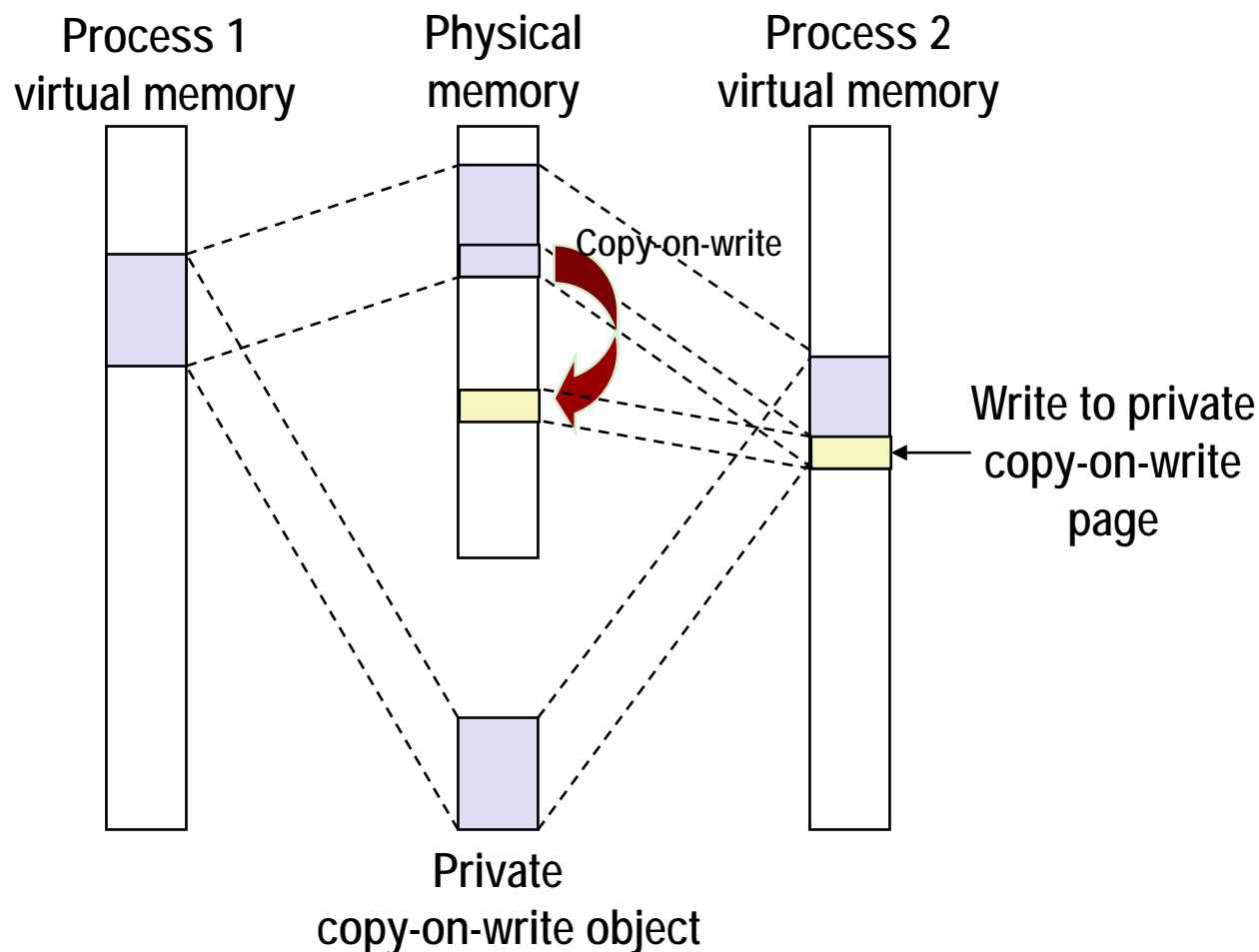
- **Instruction writing to private page triggers protection fault.**
- **Handler creates new R/W page.**
- **Instruction restarts upon handler return.**
- **Copying deferred as long as possible!**

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

44

The `fork` Function Revisited

- **VM and memory mapping explain how `fork` provides private address space for each process.**

- **To create virtual address for 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
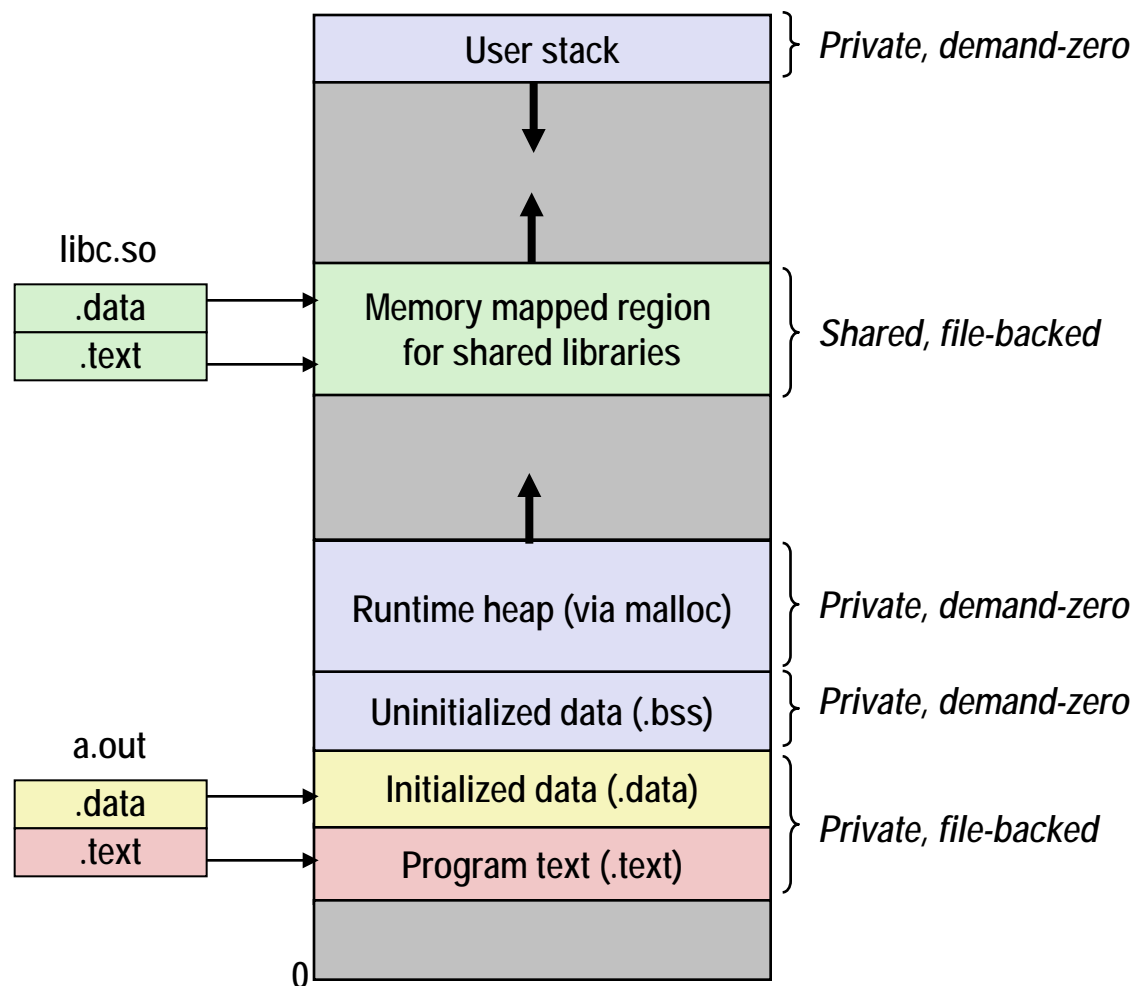
- **To load and run a new program `a.out` in the current process using `execve`:**

- **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.

Finding More Shareable Pages

- **Easy places to identify shareable pages**
  - Child created via `fork`
  - Processes loading the same binary file
    - E.g., bash or python interpreters, web browsers, …
  - Processes loading the same library file

- **What about others?**
  - 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
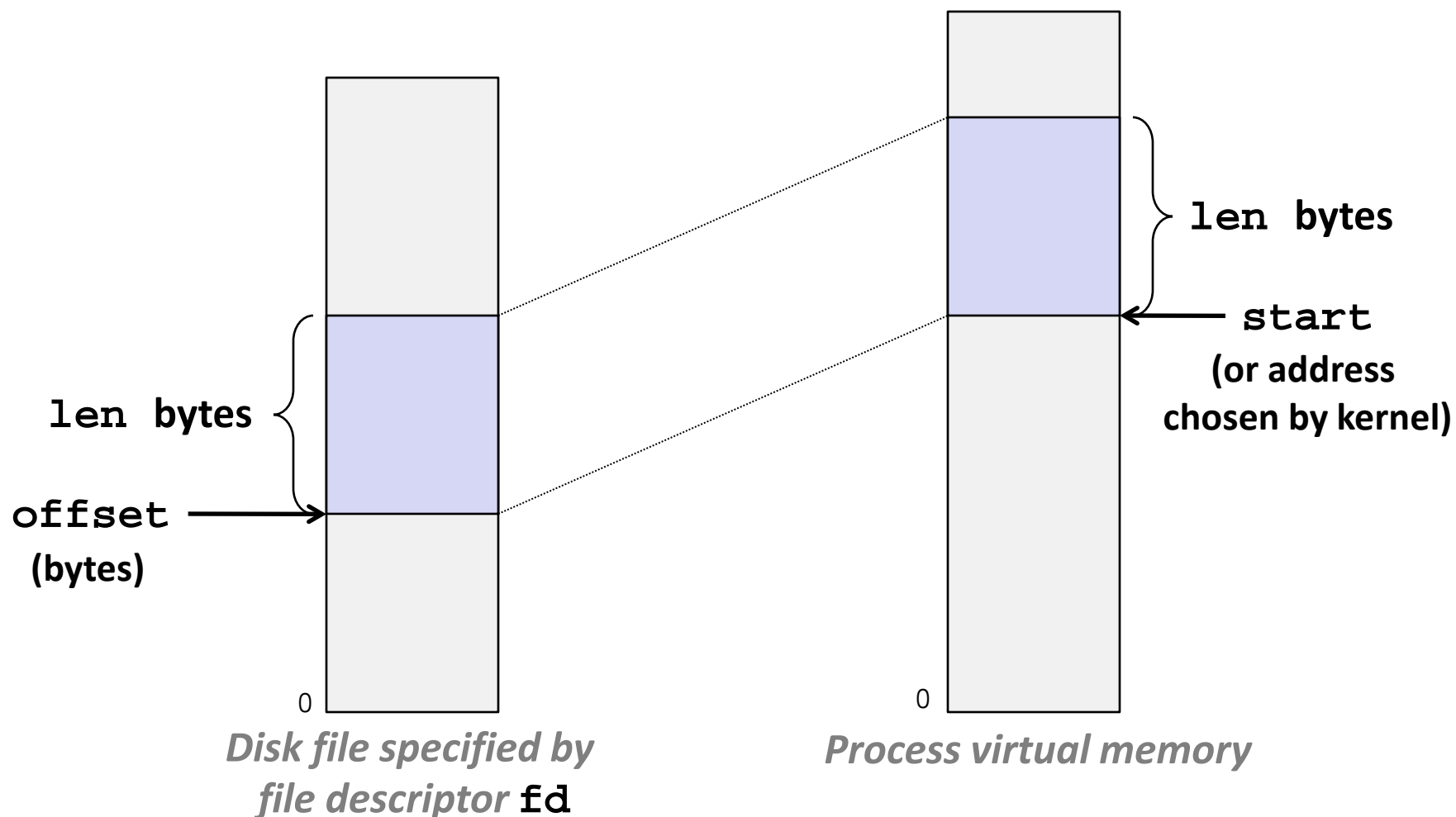
User-Level Memory Mapping

```
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)
```

len bytes

len bytes

offset
(bytes)

start
(or address chosen by kernel)

Disk file specified by
file descriptor `fd`

Process virtual memory

Example: Using `mmap` to Copy Files

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

```c
#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

```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

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Some 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

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
  - Forking processes
  - Providing memory protection

Today: Virtual Memory Systems

- Simple memory system example

- Case study: Core i7/Linux memory system

- Memory mapping

Next Lecture

- Dynamic Memory Allocation