Virtual Memory: Details

15-213/14-513/15-513: Introduction to Computer Systems
12th Lecture, October 5, 2023

Announcements

- **Written #5: Midterm Written**
  - Available on Gradescope (not canvas)
  - Covers first half of course, with questions similar to the final exam
  - Due Wed Oct 11 at 11:59 pm
  - No peer grading. Instead, graded by TAs before Fall break
  - Counts double a normal Written (can be 2 of your 2 dropped Writtens)
    - For mid-semester grades, we will temporarily count it as 20%

- **Cache Lab**
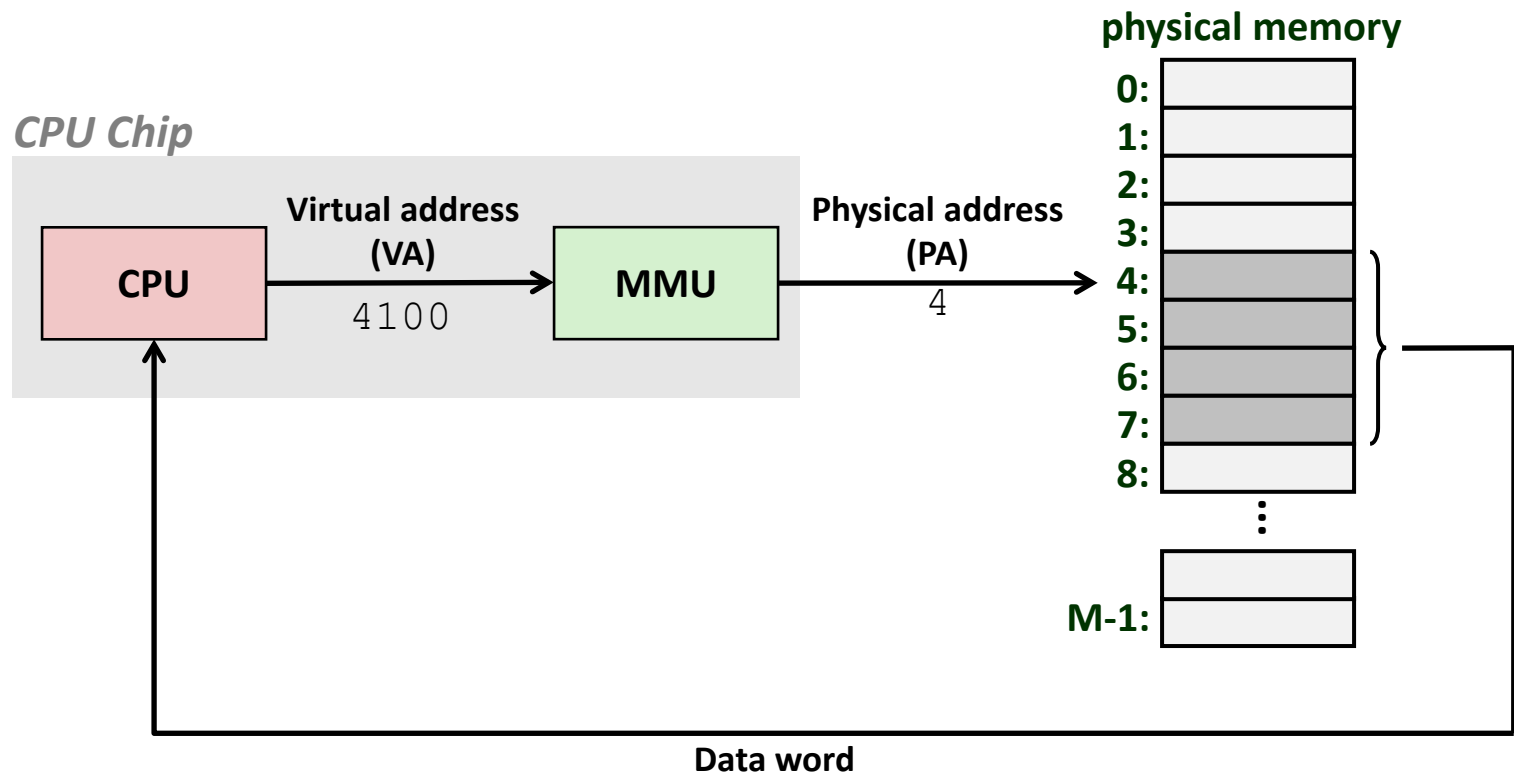 - Due Thurs Oct 12
  - Code review signup due Fri Oct 13

- **Fall Break**
  - Sat Oct 14 to Sun Oct 22

Today

- **Review concepts from last lecture**
- **Simple memory system example**   **CSAPP 9.6.4**
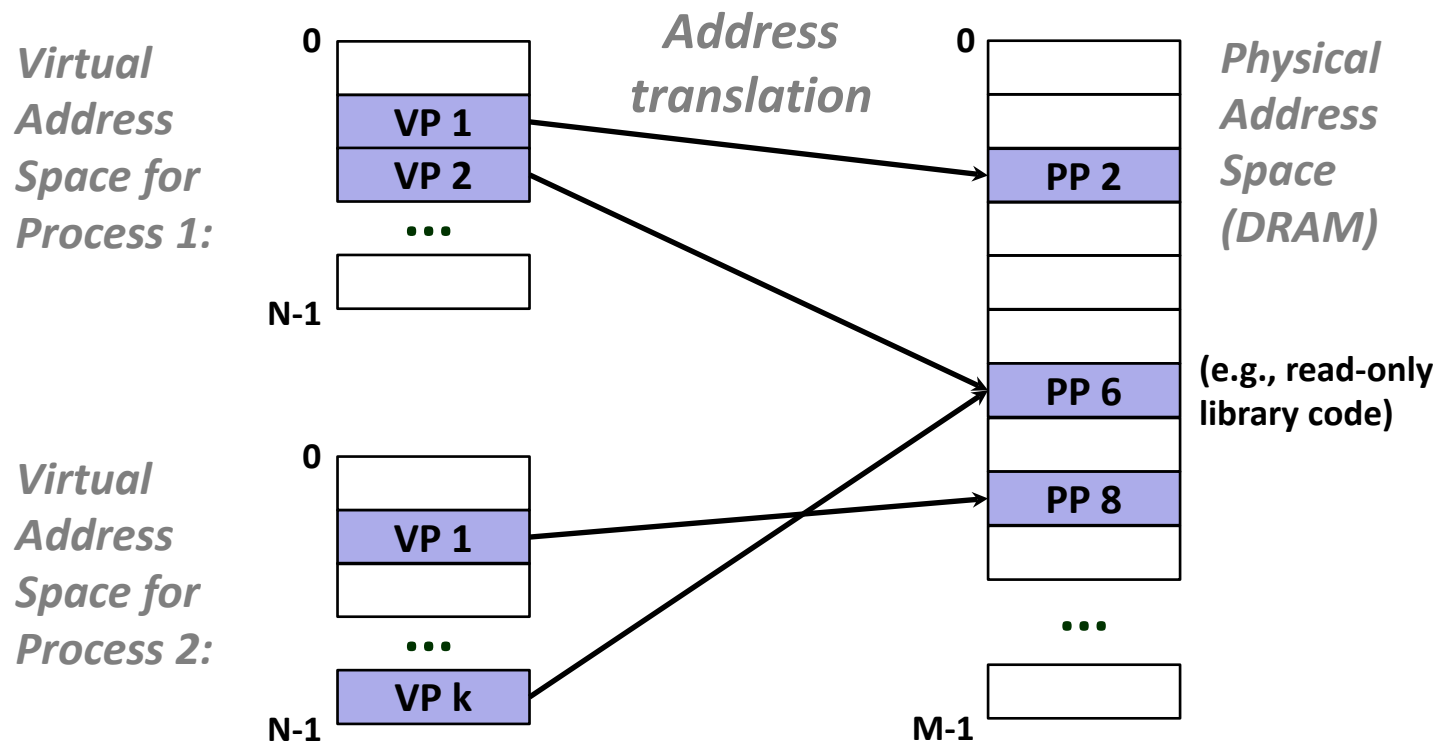- **Case study: Core i7/Linux memory system**   **CSAPP 9.7**
- **Memory mapping**   **CSAPP 9.8**

Review: Virtual Addressing

physical memory

CPU Chip

CPU	Virtual address (VA) `4100`	MMU	Physical address (PA) `4`

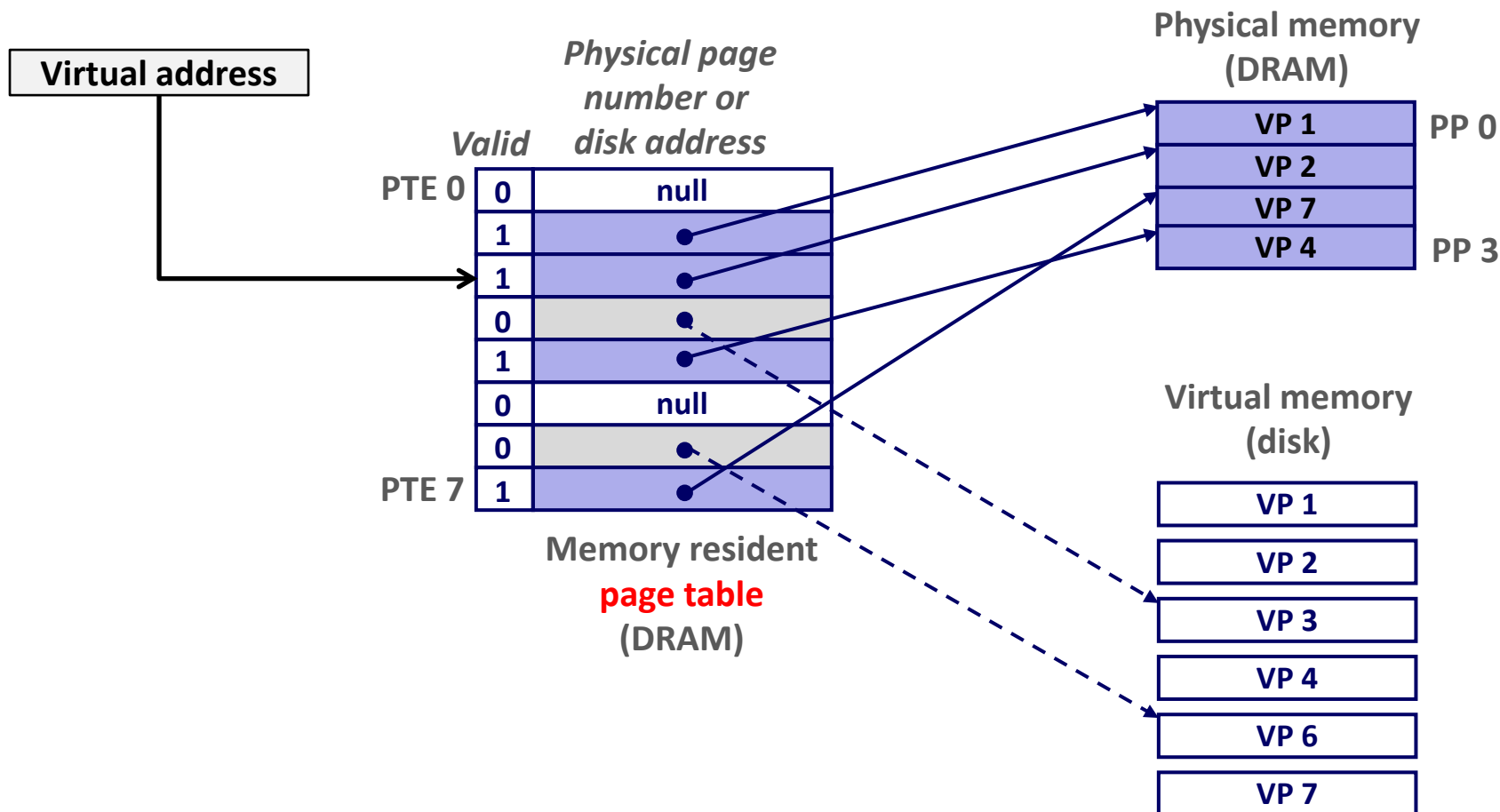|	
0:	
1:	
2:	
3:	
4:	
5:	
6:	
7:	
8:	
	⋮
M-1:	

Data word

- ■ **Virtual address space is an abstraction, not real memory**
- ■ **Physical memory refers to the actual computer memory (DRAM)**

Review: Per-process Virtual Address Space

- **Each process has its own *virtual address space***
- **All processes share the same *Physical Memory***

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 k

N-1

Review: Page Table

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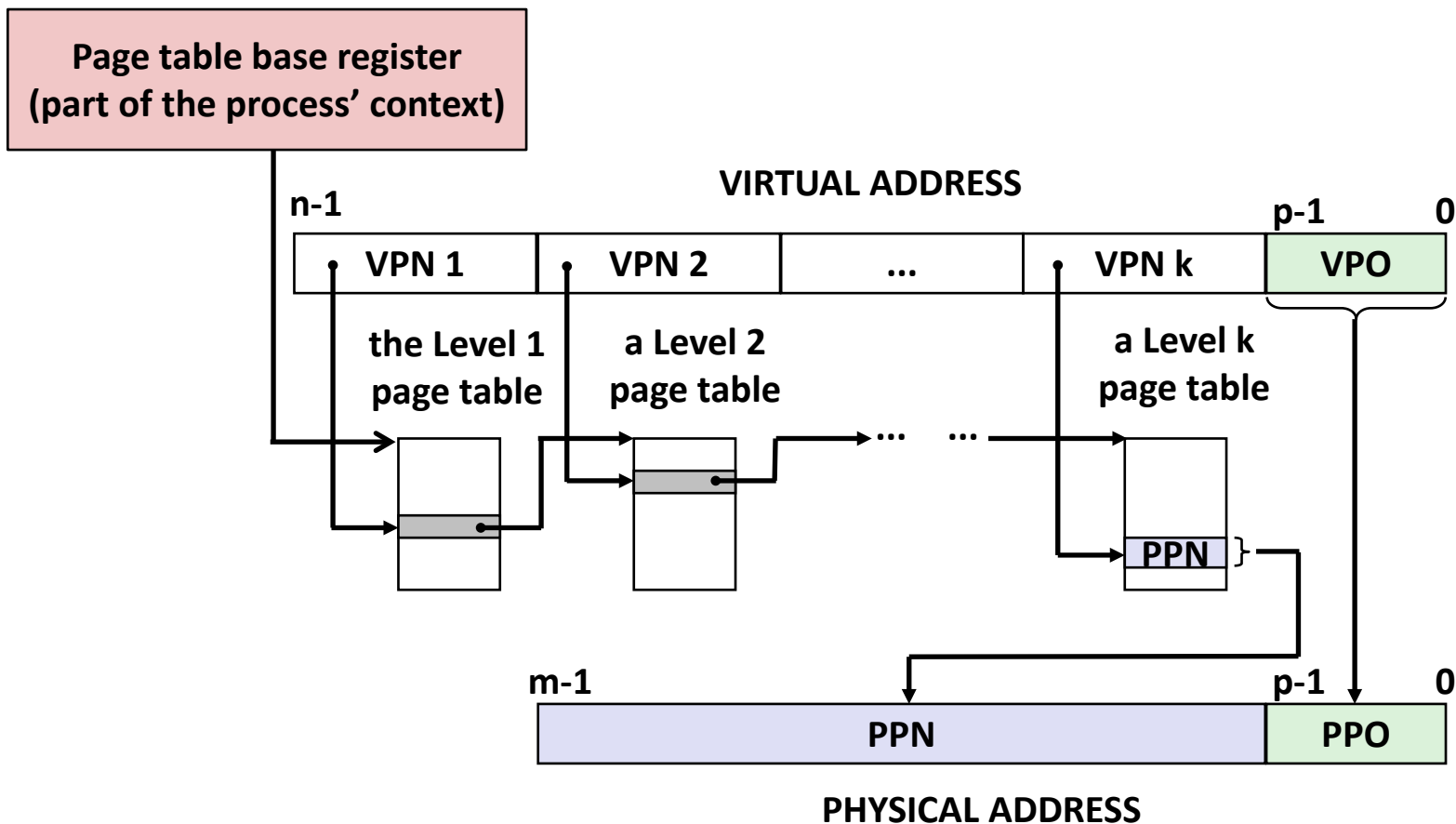
Conceptual Question

The MMU must know the *physical* address of the page table in order to read page table entries from memory. Why does it need a physical address?

If the MMU knew only a virtual address for the page table, then, in order to find the page table in memory, it would first need to look up the physical address of the page table, in the page table itself, …

Review: Translating with a k-level Page Table

- Having multiple levels greatly reduces total 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

Conceptual Questions

Why are one-level page tables impractical?

For typical system sizes, the table would require more physical memory (e.g., 512 GBs) than most computers have.

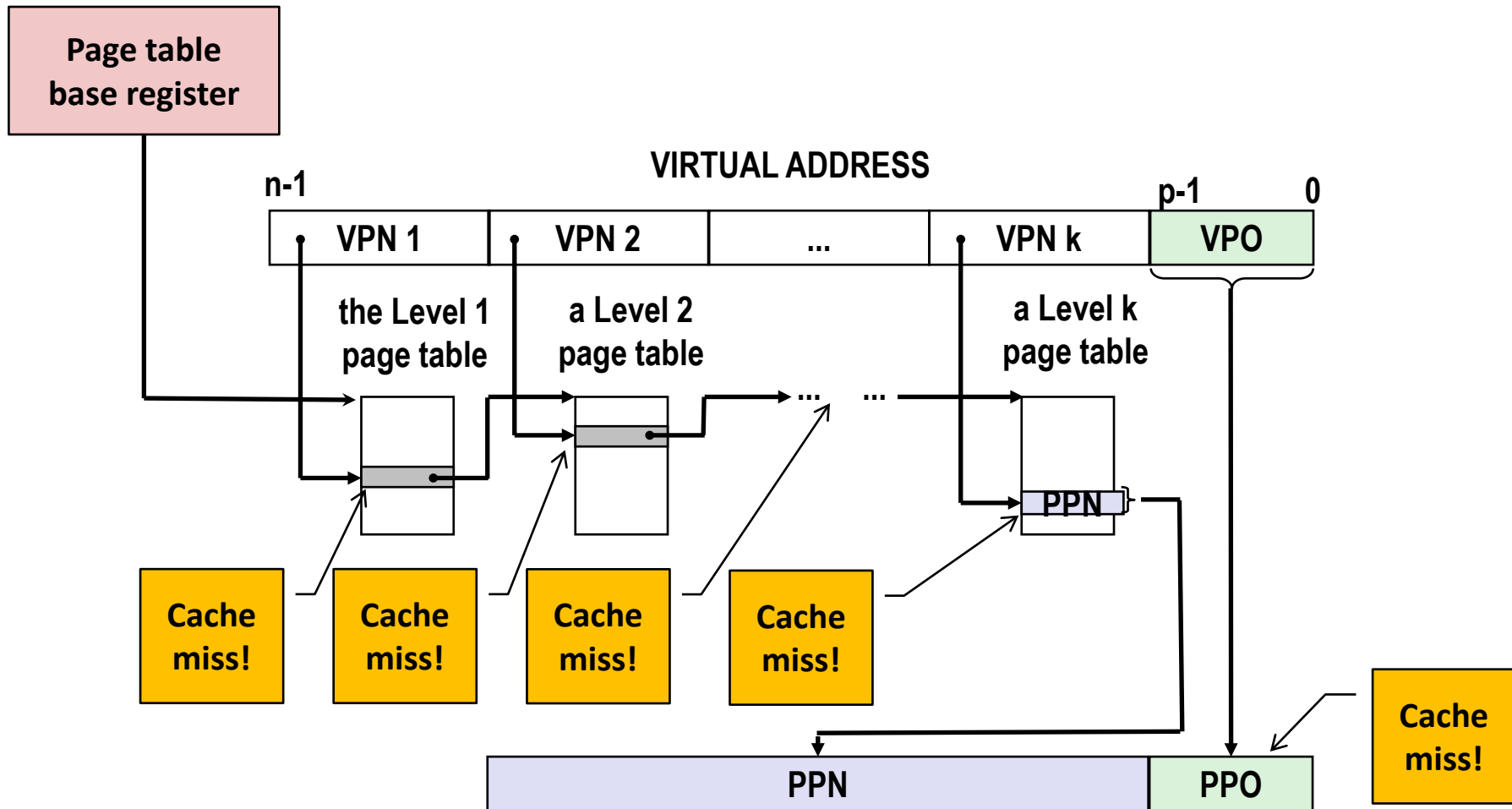**How does a multi-level page table fix this problem?**

Only allocates the part of the page table tree that's needed for the virtual addresses the program uses.

Why is memory access slower with a multi-level page table than with a one-level page table?

A k-level page table requires k memory loads in order to determine the physical address. There is no spatial locality to these loads (see next slide).

The problem (with k-level page tables)

Page table base register

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

Cache miss!

Cache miss!

Cache miss!

Cache miss!

PPN

PPN	PPO

Cache miss!

Review: Translation Lookaside Buffer (TLB)

- **A small cache dedicated to storing mappings from virtual addresses to physical addresses (page table entries)**

- **MMU consults the TLB for each address as its first action. If there is a TLB hit, it does not need to fetch anything from the page table (avoiding k lookups)**
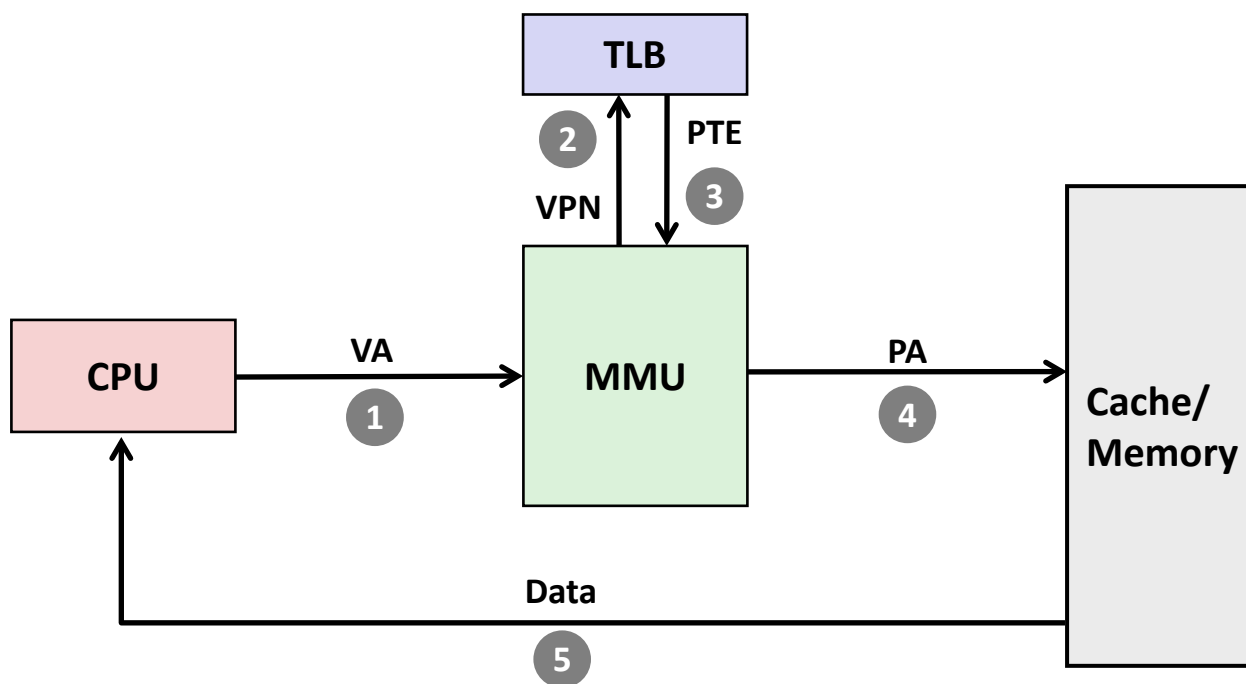
Review: Accessing the TLB

■ **MMU uses the VPN portion of the virtual address to access the TLB:**

$T = 2^t$ **sets**

VPN

TLBT matches tag of line within set

n-1	p+t	p+t-1	p	p-1	0
TLB tag (TLBT)		TLB index (TLBI)		VPO	

Set 0 — | v | tag | PTE | | v | tag | PTE |

TLBI selects the set

Set 1 — | v | tag | PTE | | v | tag | PTE |

⋮

Set T-1 — | v | tag | PTE | | v | tag | PTE |
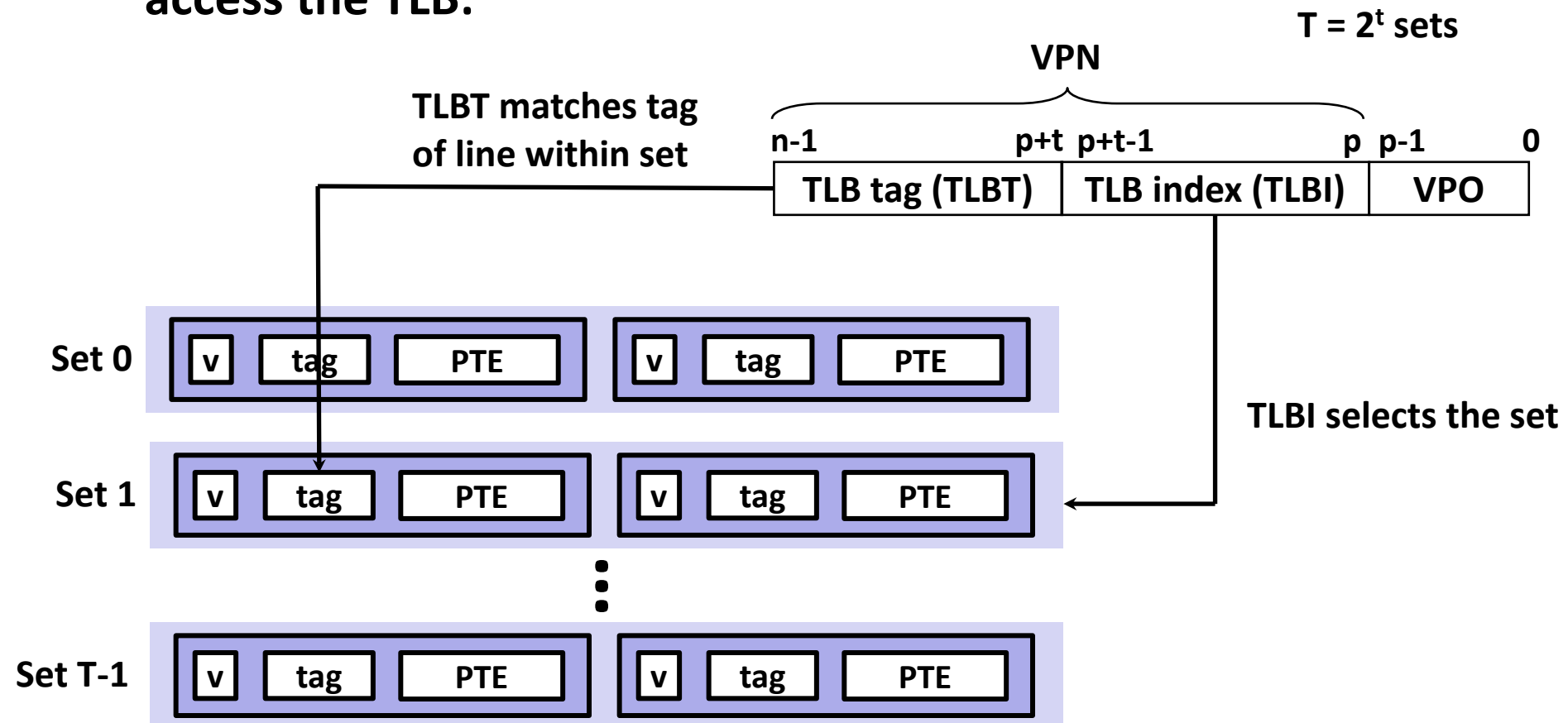
Conceptual Question

How does virtual memory interact with the CPU cache(s)?

The cache's function is to speed up access to whatever data is most frequently used. The MMU sits "in between" the CPU and the cache; the cache works only with physical addresses. This means data from multiple processes may coexist in the cache (or compete for cache space).

TLBT — TLBI

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

VPN — VPO

Virtual Page Number — **Virtual Page Offset**

CT — CI — CO

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

PPN — PPO

Physical Page Number — **Physical Page Offset**
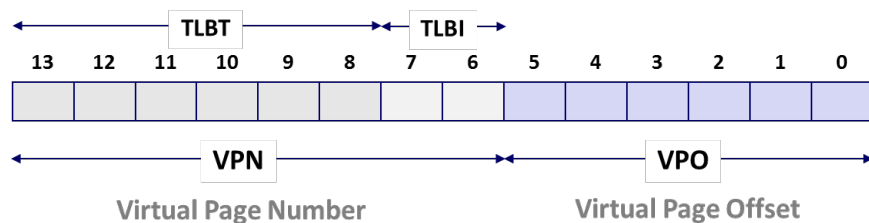
1. MMU uses VA to find PTE & get PA   **2. PA is used to look in cache for data**
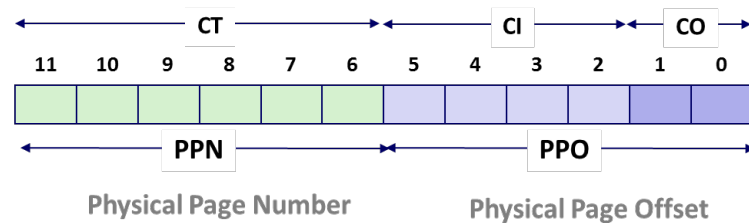
Today

- **Review concepts from last lecture**
- **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

Why is the VPO 6 bits?   **Why is the PPO 6 bits?**

Why is the VPN 8 bits?   **Why is the PPN 6 bits?**

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**
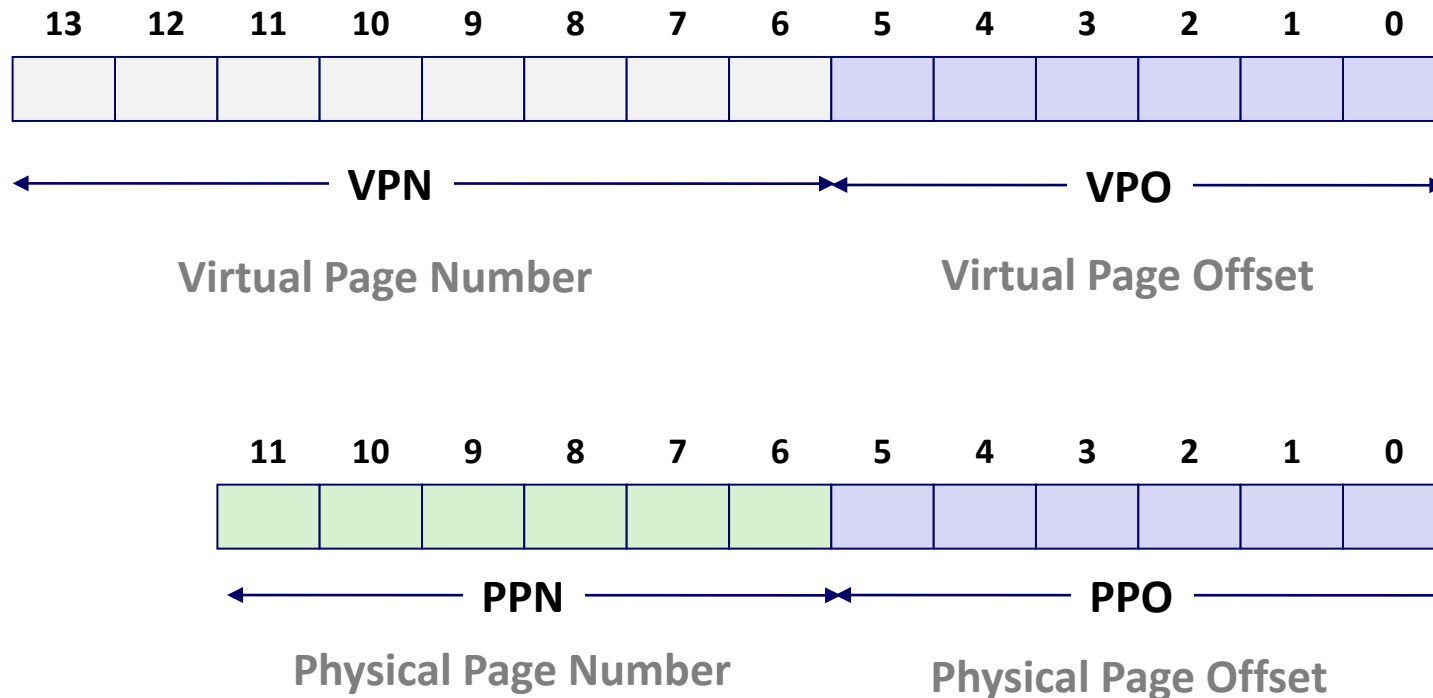
Simple Memory System TLB

- **16 entries**

- **4-way associative**
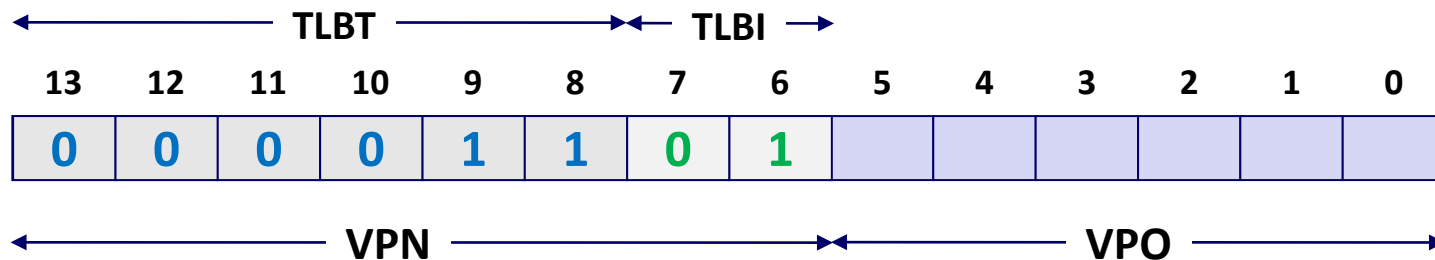
$$VPN = 0b1101 = 0x0D$$

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

	TLBT					TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1						
		VPN					VPO						

➡

				PPN				PPO				
11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1						
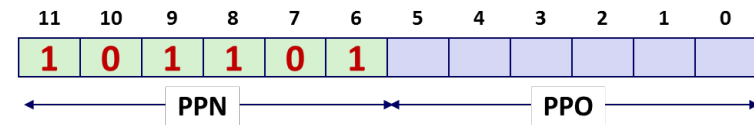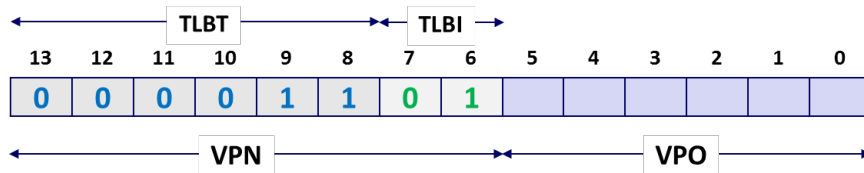
Simple Memory System Cache

- **16 lines, 4-byte cache line 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	–	–	–	–

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

19

Address Translation Example

Virtual Address: `0x03D4`

	TLBT								TLBI		
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0
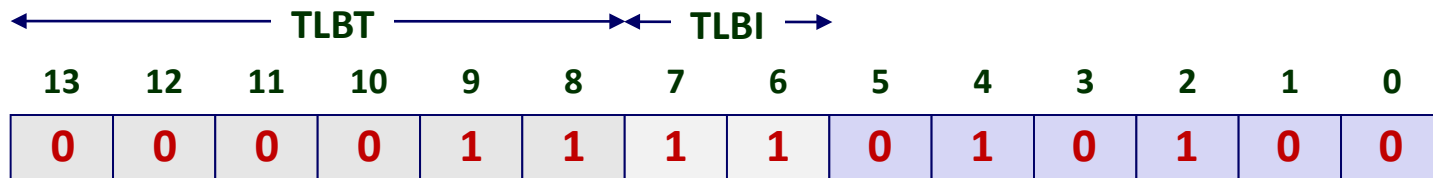
VPN **0x0F**   TLBI **0x3** TLBT **0x03**   TLB Hit? **Y**   Page Fault? **N**   PPN: **0x0D**

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

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

PPN   PPO
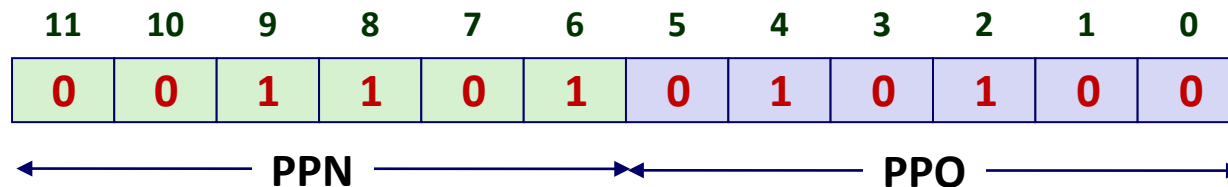
Address Translation Example

Physical Address

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

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

PPN ⟷ PPO
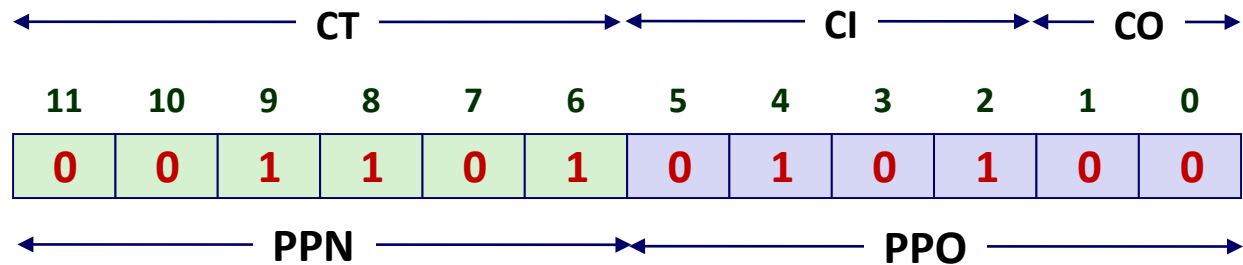
CO **0**   CI **0x5**   CT **0x0D**   Hit? **Y**   Byte: **0x36**

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	–	–	–	–

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

Address Translation Example: TLB/Cache Miss

Virtual Address: 0x0020

$$\xleftarrow{\hspace{2em}} \text{TLBT} \xrightarrow{\hspace{2em}} \xleftarrow{} \text{TLBI} \xrightarrow{}$$

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

$$\xleftarrow{\hspace{3em}} \text{VPN} \xrightarrow{\hspace{3em}} \xleftarrow{\hspace{2em}} \text{VPO} \xrightarrow{\hspace{2em}}$$
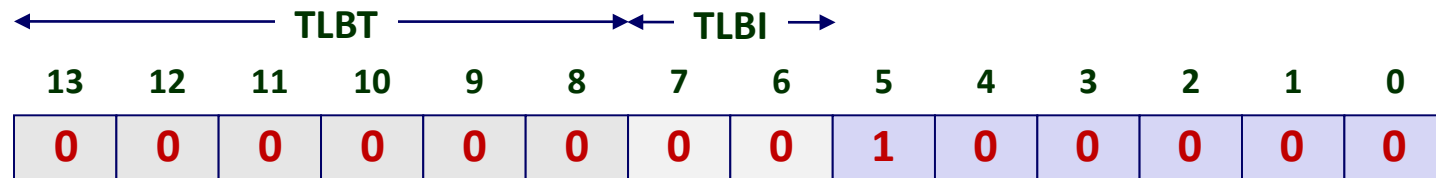
VPN **0x00**   TLBI **0**   TLBT **0x00**   TLB Hit? **N**   Page Fault? **N**   PPN: **0x28**

Physical Address

$$\xleftarrow{\hspace{2em}} \text{CT} \xrightarrow{\hspace{2em}} \xleftarrow{\hspace{2em}} \text{CI} \xrightarrow{\hspace{2em}} \xleftarrow{} \text{CO} \xrightarrow{}$$

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

$$\xleftarrow{\hspace{2em}} \text{PPN} \xrightarrow{\hspace{2em}} \xleftarrow{\hspace{2em}} \text{PPO} \xrightarrow{\hspace{2em}}$$
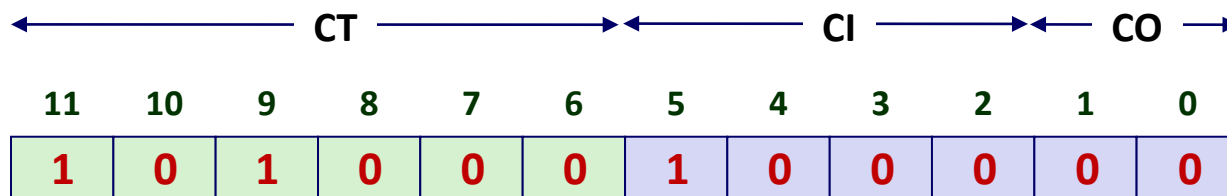
CO **0**   CI **0x8**   CT **0x28**   Hit? __   Byte: ____

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

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__
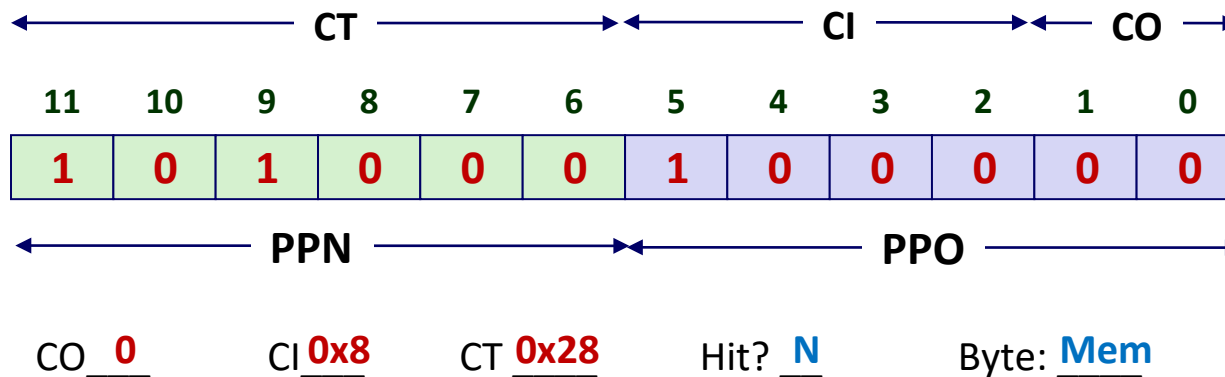
Quiz Time!

Canvas Quiz: Day 12 – Virtual Memory: Details
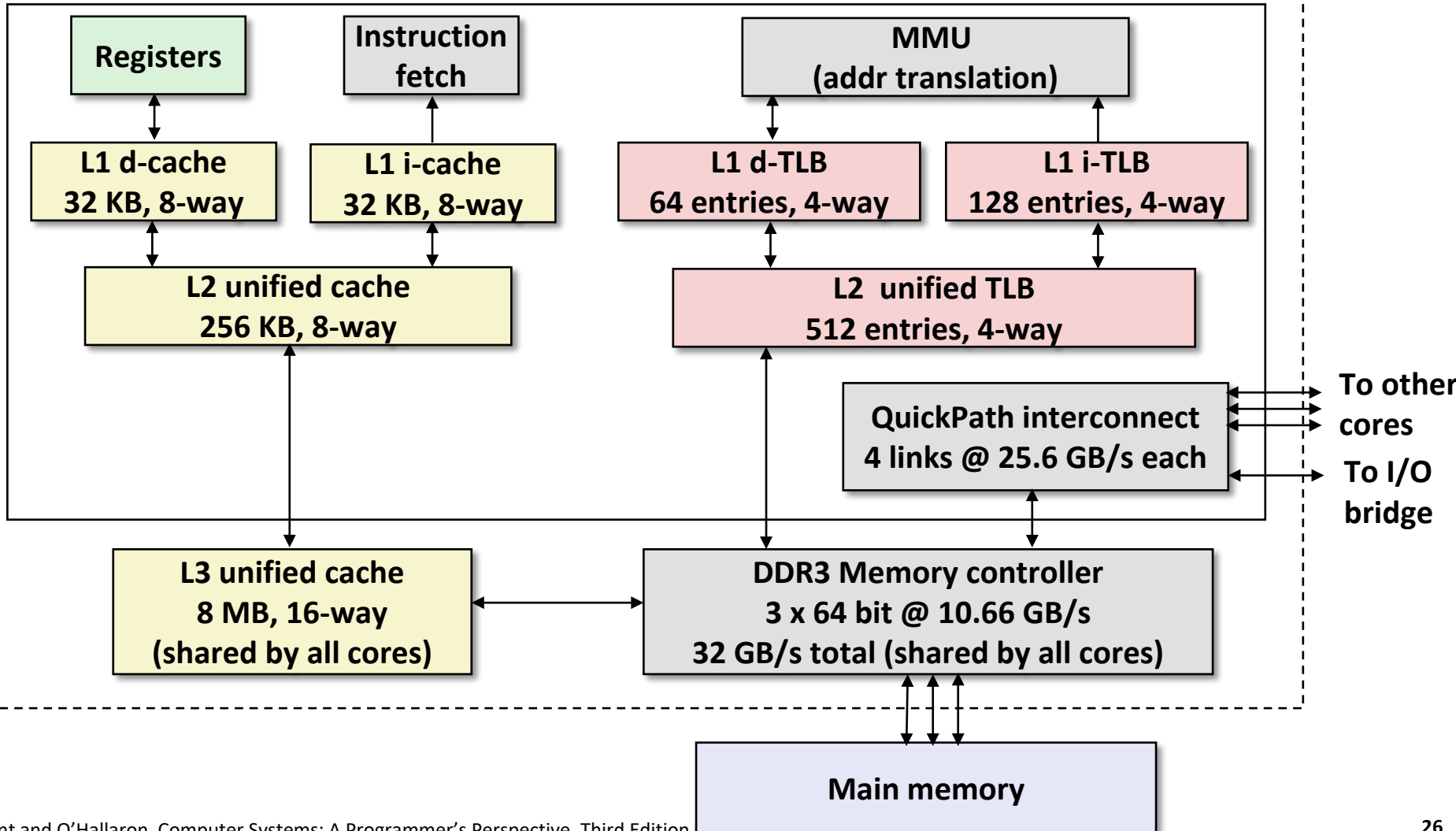
Today

- **Review concepts from last lecture**
- **Simple memory system example**
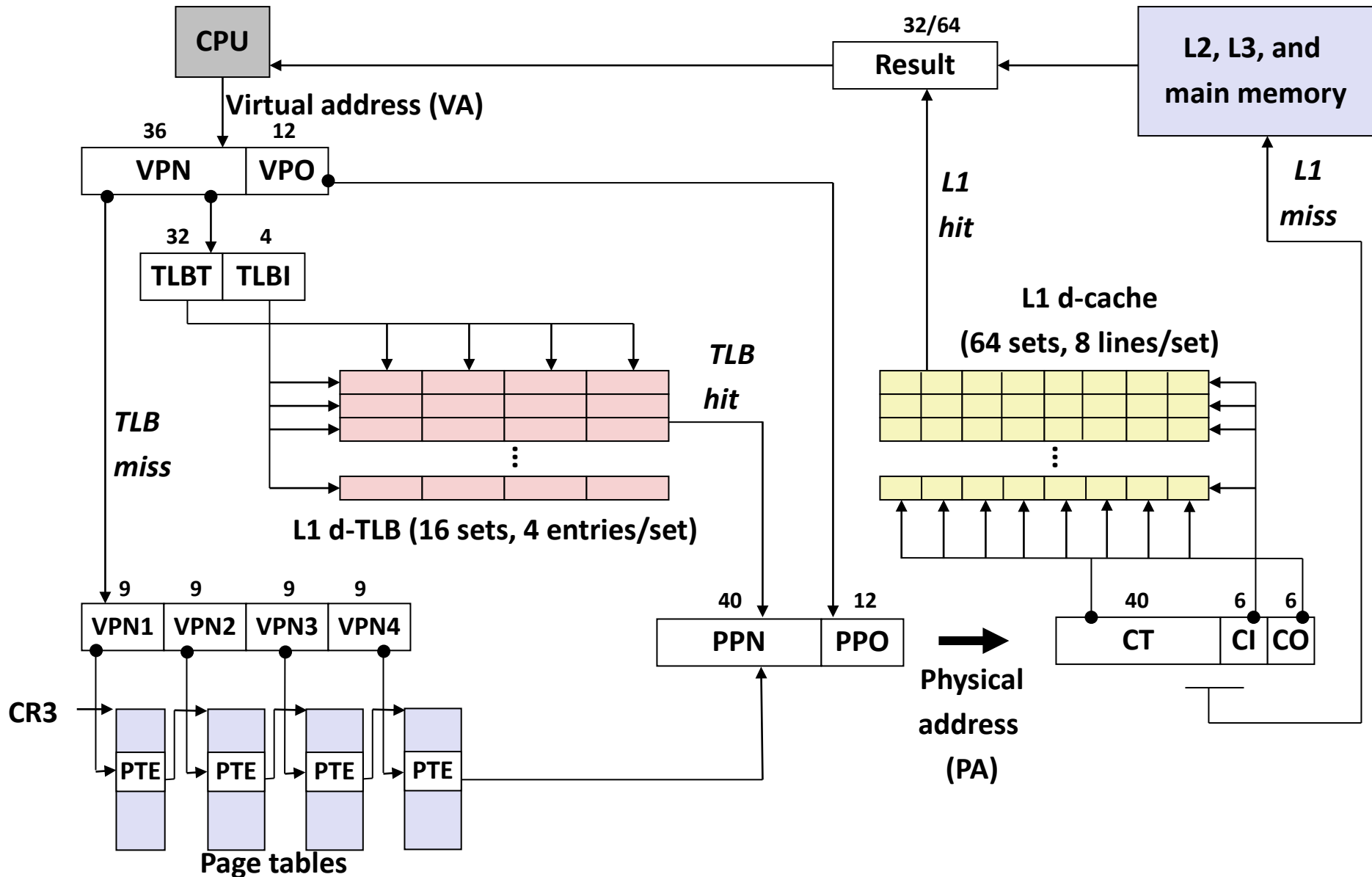- **Case study: Core i7/Linux memory system**
- **Memory mapping**

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

CPU

Virtual address (VA)

36 — **VPN**   12 — **VPO**

32 — **TLBT**   4 — **TLBI**

TLB miss

L1 d-TLB (16 sets, 4 entries/set)

TLB hit

9 — **VPN1**   9 — **VPN2**   9 — **VPN3**   9 — **VPN4**

CR3

PTE   **PTE**   **PTE**   **PTE**

Page tables

40 — **PPN**   12 — **PPO**

Physical address (PA)

Result   32/64

L1 hit

L1 d-cache (64 sets, 8 lines/set)

40 — **CT**   6 — **CI**   6 — **CO**

L2, L3, and main memory

L1 miss

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

27

Core i7 Level 1-3 Page Table Entries

63	62 52	51 Page table physical base address 12	11 Unused 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.

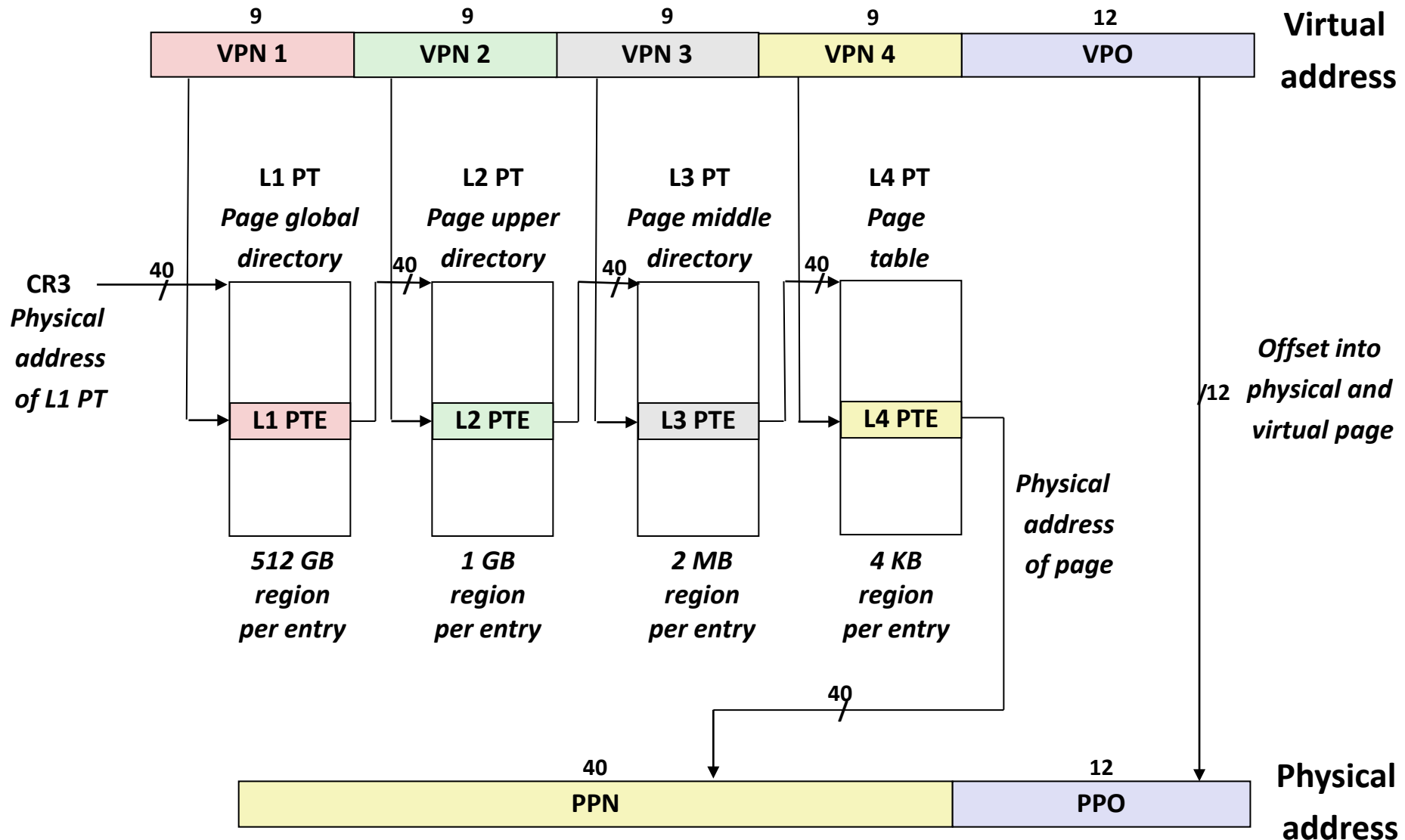**A:** Reference bit (set by MMU on reads and writes, cleared by software).

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)

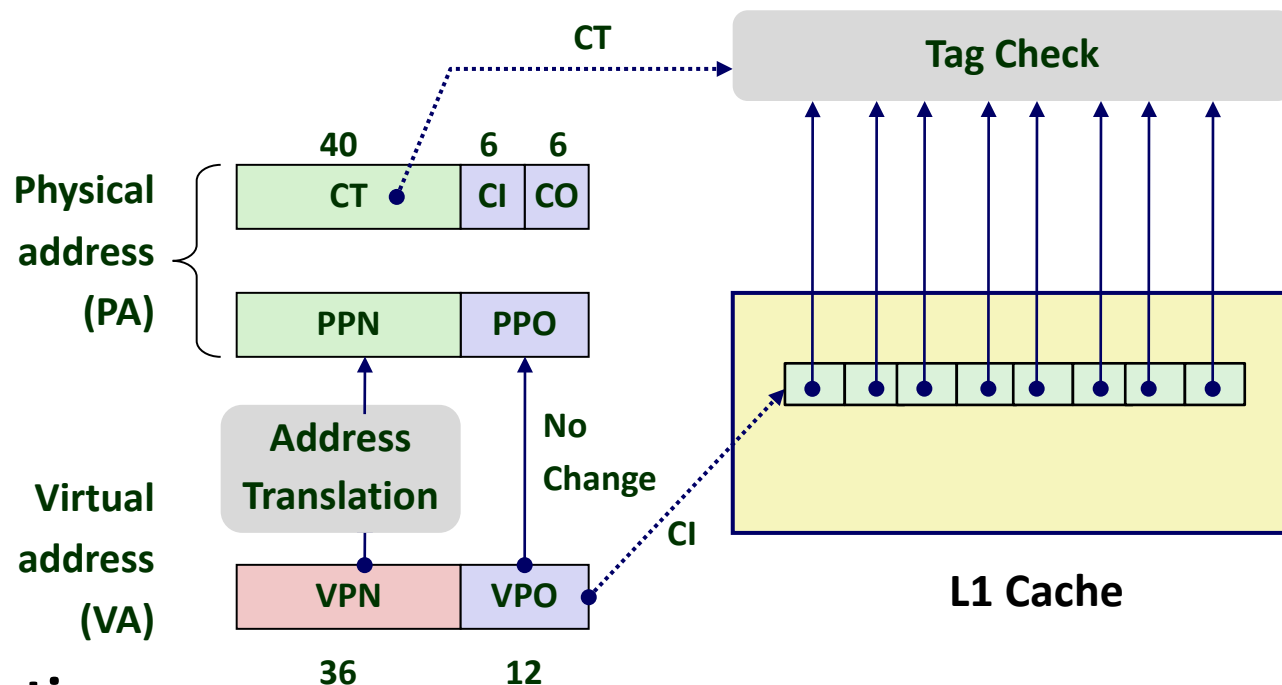**G:** Global page (don't evict from TLB on task switch)

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

9	9	9	9	12
VPN 1	VPN 2	VPN 3	VPN 4	VPO

Virtual address

L1 PT
Page global directory

L2 PT
Page upper directory

L3 PT
Page middle directory

L4 PT
Page table

CR3
Physical address of L1 PT

40

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

CT → **Tag Check**

Physical address (PA): **40** **CT** | **6** **CI** | **6** **CO**

PPN | **PPO**

Address Translation

No Change

Virtual address (VA): **VPN** | **VPO**

36 **12**

CI

L1 Cache

■ 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

Today

- **Review concepts from last lecture**
- **Simple memory system example**
- **Case study: Core i7/Linux memory system**
- **Memory mapping**

Memory-Mapped Files

- **Paging = every page of a program's physical memory is *backed* by some page of disk***

- **Normally, those pages belong to *swap space***

- **But what if some pages were backed by … files?**

* This is how it used to work 20 years ago.
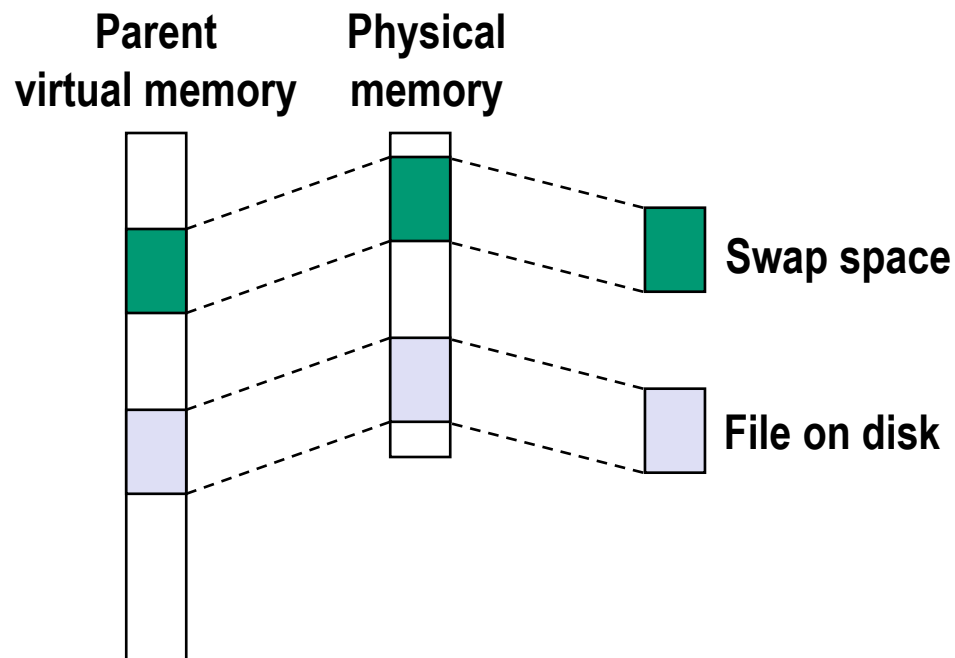Nowadays, not always true.

Memory-Mapped Files

Process
virtual memory

Physical
memory

Swap space

File on disk
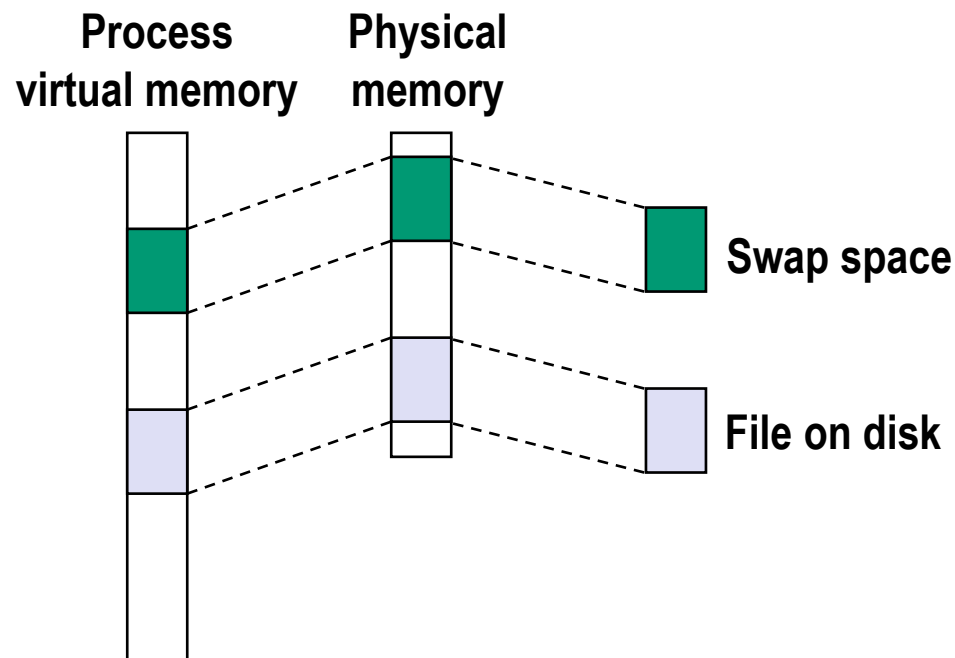
Memory-Mapped Files
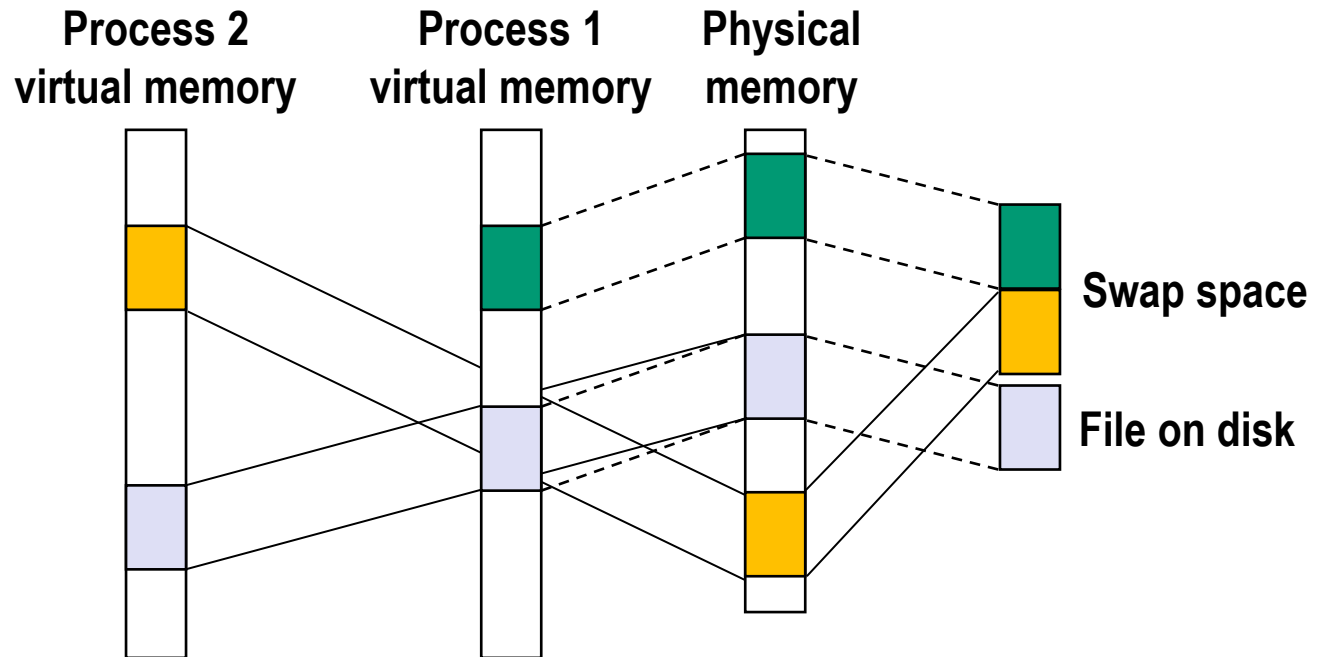
Copy-on-write sharing

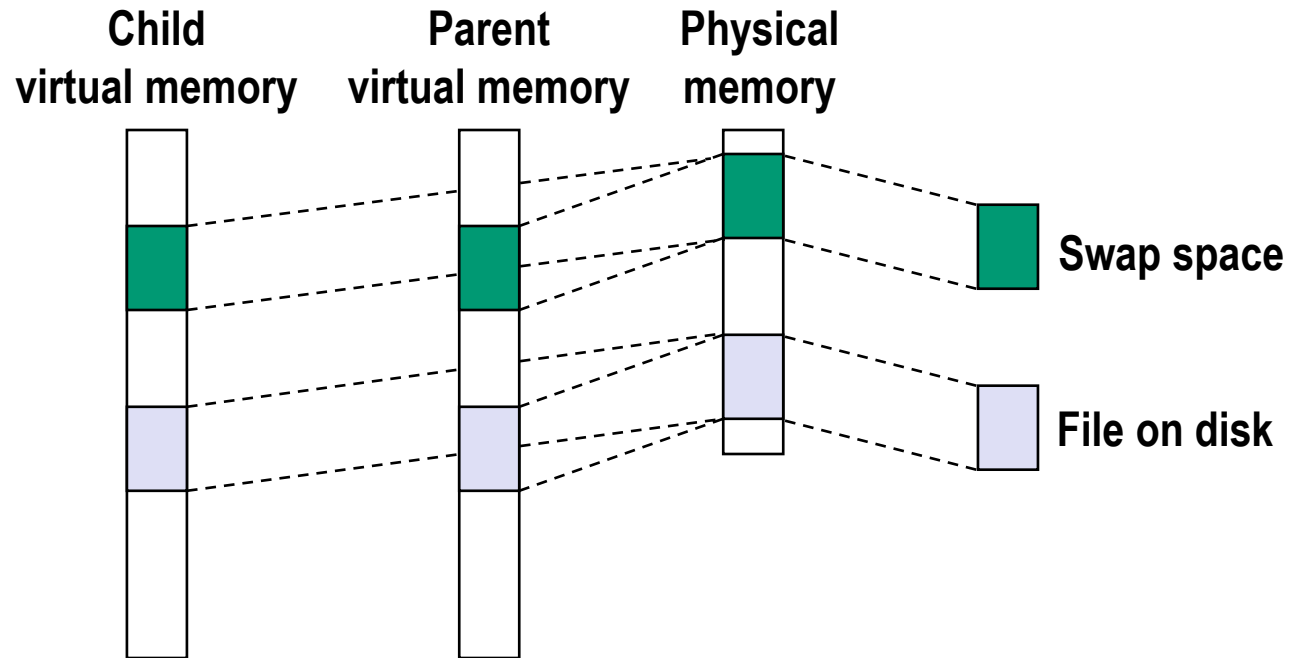- **`fork` creates a new process by copying the entire address space of the parent process**
  - That sounds slow
  - It *is* slow

Parent virtual memory **Physical memory**

Swap space

File on disk

- **Clever trick:**
  - Just duplicate the page tables
  - Mark everything read only (PTE permission bits for all pages set to read-only)
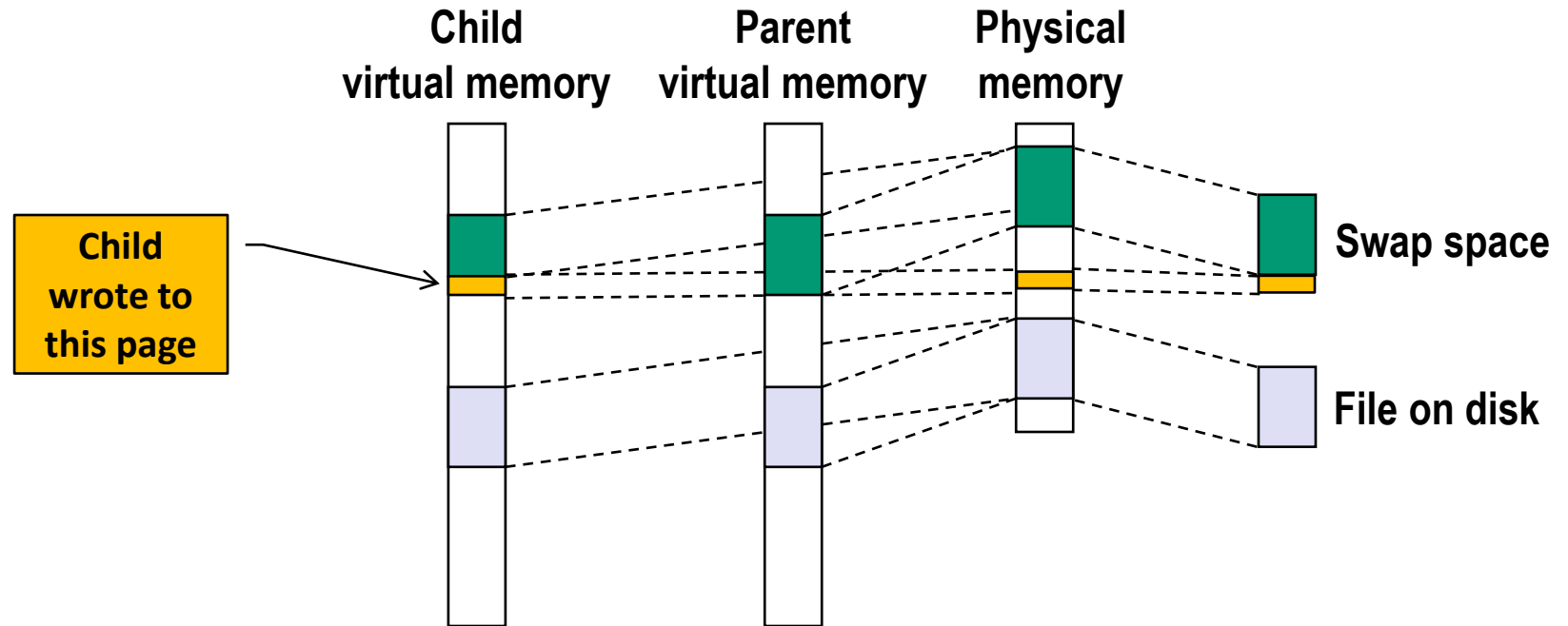 - Copy only on write faults

Copy-on-write sharing

Child virtual memory   Parent virtual memory   Physical memory

Swap space

File on disk

- **Clever trick:**
  - Just duplicate the page tables
  - Mark everything read only
  - Copy only on write faults

Copy-on-write sharing

Child virtual memory   **Parent virtual memory**   **Physical memory**

Child wrote to this page

Swap space

File on disk

- **Clever trick:**
  - Just duplicate the page tables
  - Mark everything read only
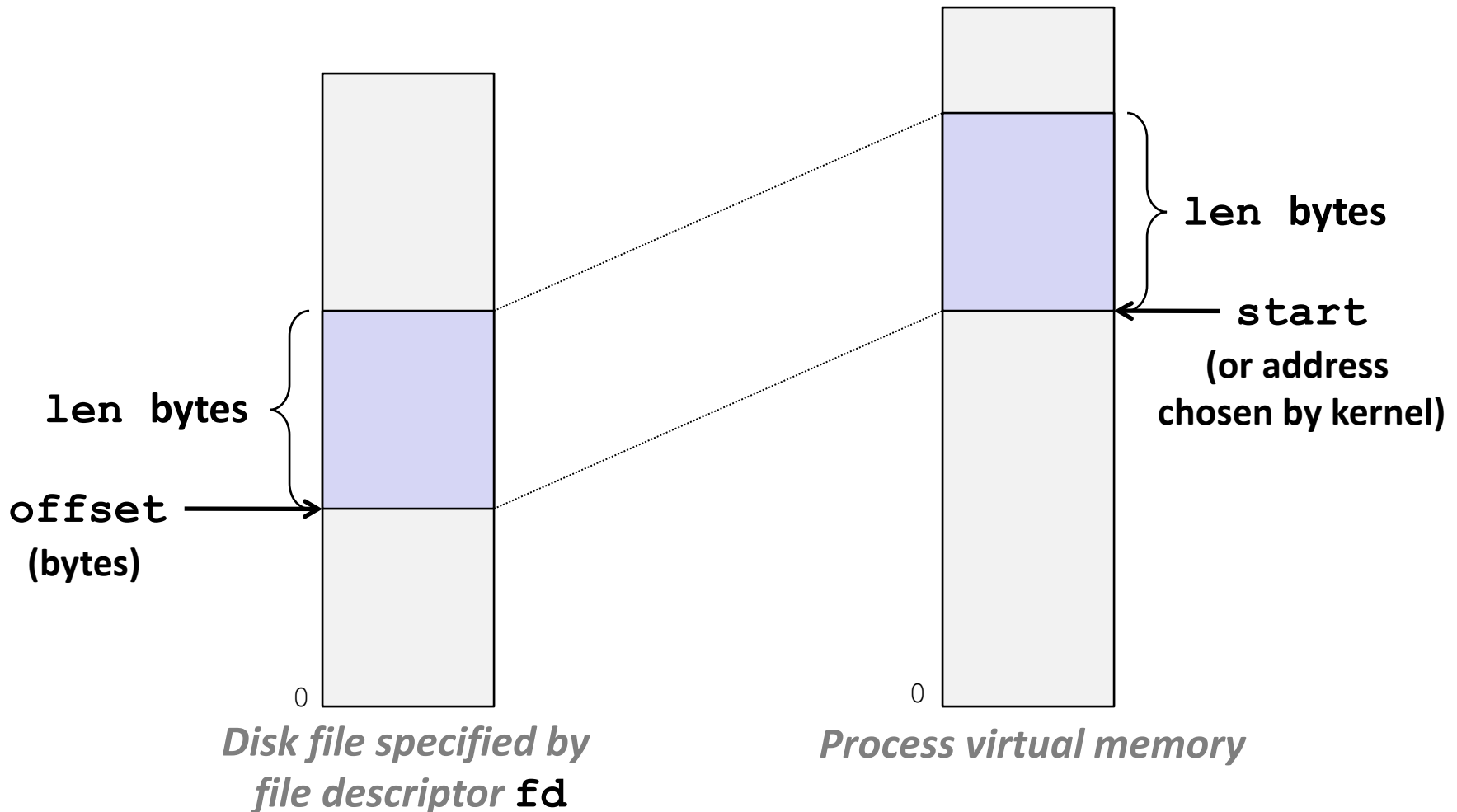 - Copy only on write faults

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

start
(or address
chosen by kernel)

len bytes

offset
(bytes)

*Disk file specified by
file descriptor* `fd`

Process virtual memory

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
  - When unmap region, file will be updated via write-back
  - Can implement load from file / update / write back to file

- **Enable Attack Lab**
  - Allow students to execute code on the stack (which is forbidden on shark machines)
  - See backup slides for details

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

Summary

- **Programmer's view of virtual memory**
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes

- **System view of virtual memory**
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions

- **Implemented via combination of hardware & software**
  - MMU, TLB, exception handling mechanisms part of hardware
  - Page fault handlers, TLB management performed in software

Review Question

For a simple system with a one-level page table, what sub-steps does the MMU take when it fetches a PTE from a page table?

The MMU has to split the virtual address into VPN and VPO.

The VPN can then be used to index directly into the page table.

If the valid bit is set on the PTE, the entry contains a PPN and the physical address is PPN followed by PPO (=VPO).

Otherwise, a page fault is triggered.

Address Translation With a Page Table

Virtual address

n-1		*p* *p-1*	*0*
Virtual page number (VPN)		**Virtual page offset (VPO)**	

**Page table
base register (PTBR)
(CR3 in x86)**

**Physical page table
address for the current
process**

Page table

Valid   Physical page number (PPN)

**Valid bit = 0:
Page not in memory
(page fault)**

Valid bit = 1

m-1	*p* *p-1*	*0*
Physical page number (PPN)	**Physical page offset (PPO)**	

Physical address
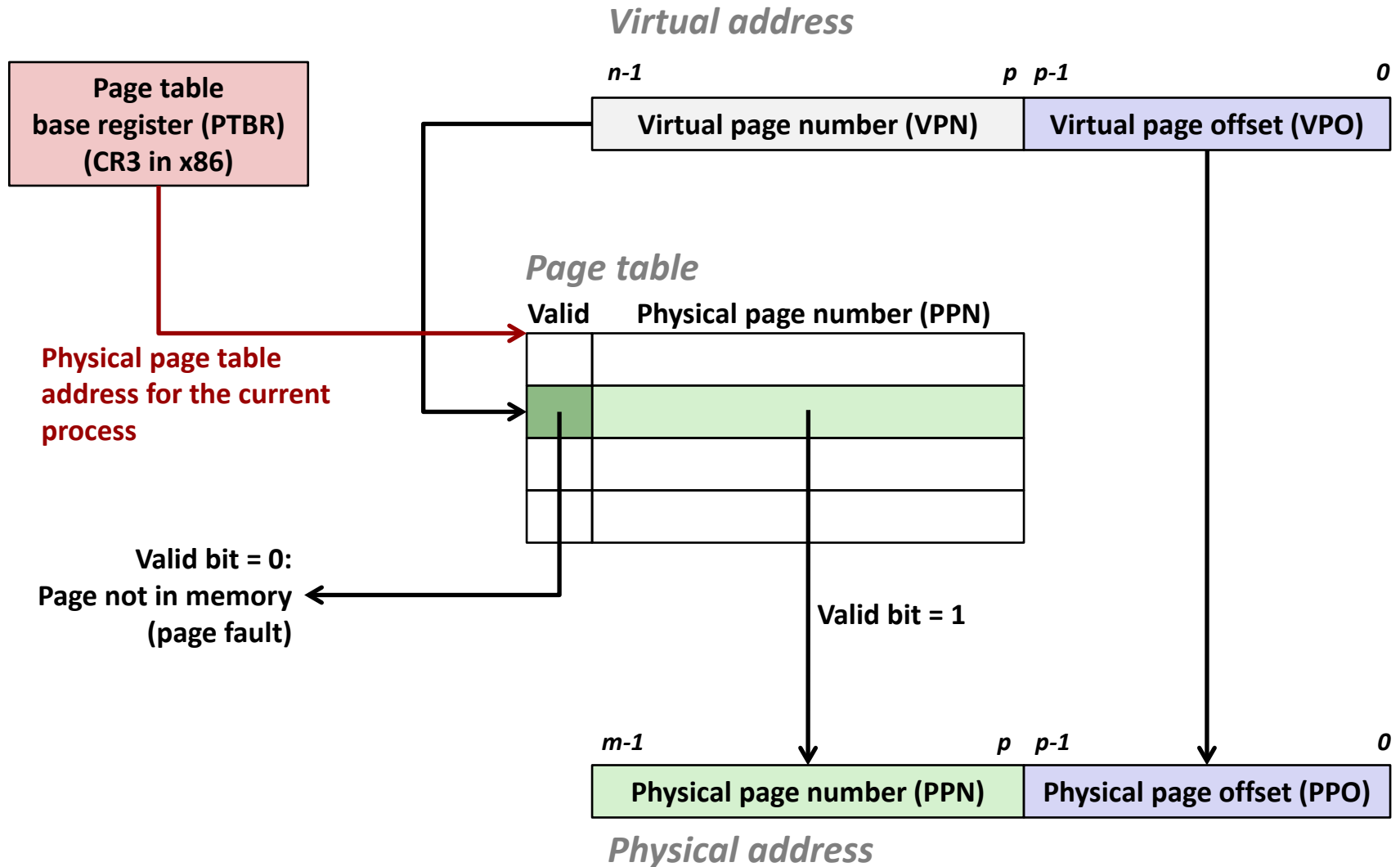
Example: Using `mmap` to Support Attack Lab

- **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

Kernel virtual memory
User stack (created at runtime)
Memory-mapped region for shared libraries
Run-time heap (created by `malloc`)
Read/write segment (`.data, .bss`)
Read-only segment (`.init,.text, .rodata`)
Unused

Memory invisible to user code

`%rsp` (stack pointer)

`0x40000000`

`0`

Using `mmap` to Support Attack Lab

Memory
invisible to
user code

Kernel virtual memory
User stack (created at runtime)
Memory-mapped region for shared libraries
Region created by mmap
Run-time heap (created by `malloc`)
Read/write segment (`.data`, `.bss`)
Read-only segment (`.init`, `.text`, `.rodata`)
Unused

`%rsp`
**(stack
pointer)**

0x55586000

0x55586000

0x40000000

0

Using `mmap` to Support Attack Lab

Memory invisible to user code

Kernel virtual memory
User stack (created at runtime)
Memory-mapped region for shared libraries
Region created by mmap
Run-time heap (created by `malloc`)
Read/write segment (.data, .bss)
Read-only segment (.init, .text, .rodata)
Unused

0x55586000

0x40000000

0

`%rsp` (stack pointer)

Frame for launch
Frame for test
Frame for getbuf

0x55586000

Using `mmap` to Support Attack Lab

Kernel virtual memory

User stack
(created at runtime)

Memory-mapped region for
shared libraries

Run-time heap
(created by `malloc`)

Read/write segment
(`.data, .bss`)

Read-only segment
(`.init,.text, .rodata`)

0x40000000

Unused

0

Memory
invisible to
user code

`%rsp`
(stack
pointer)

Restore original %rsp
Use `munmap` to remove mapped region

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);
```
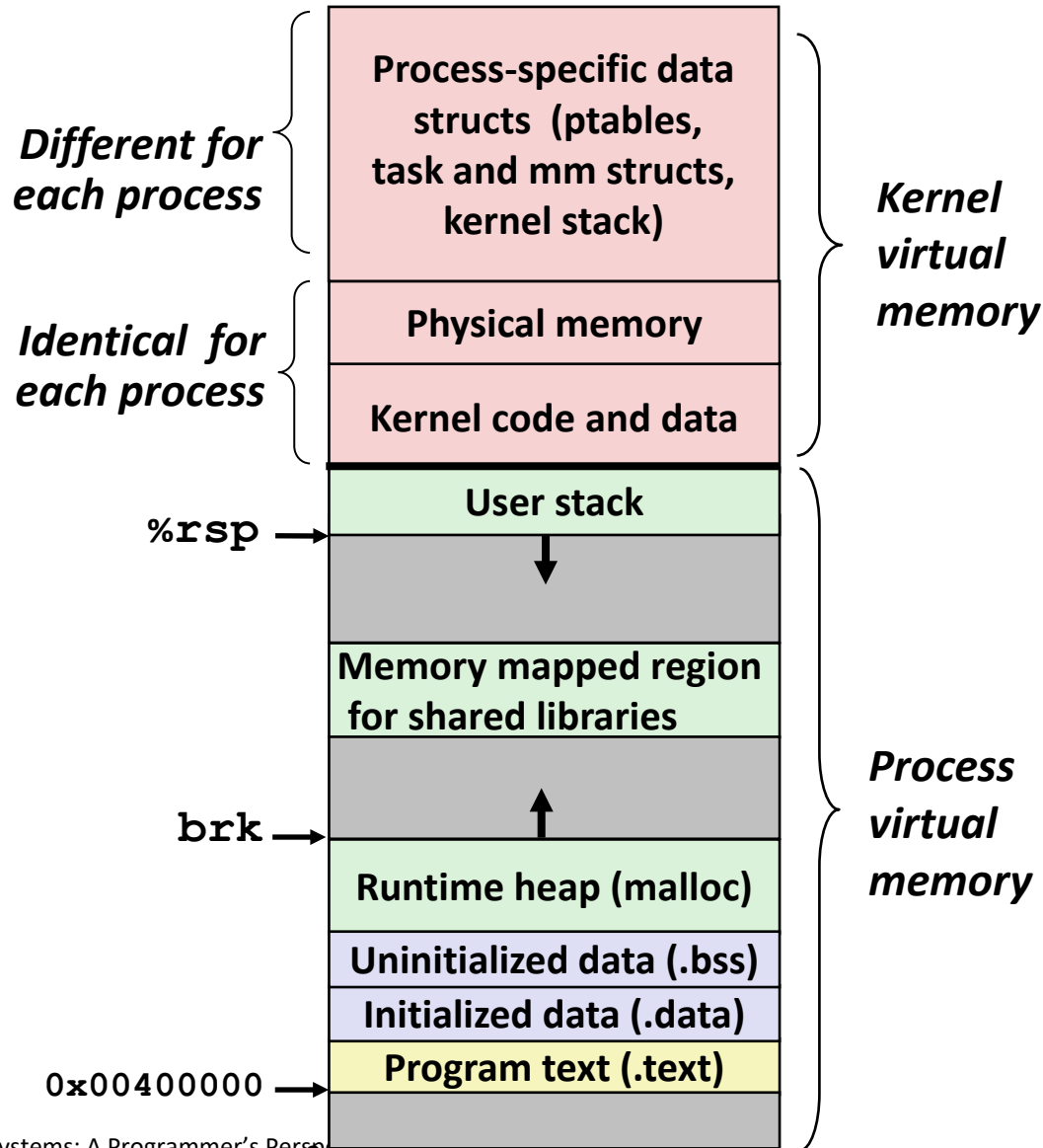
Restore stack and remove region

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

munmap(new_stack, STACK_SIZE);
```

Virtual Address Space of a Linux Process

Different for each process

Process-specific data structs (ptables, task and mm structs, kernel stack)

Kernel virtual memory

Identical for each process

Physical memory

Kernel code and data

`%rsp` →

User stack

Memory mapped region for shared libraries

Process virtual memory

`brk` →

Runtime heap (malloc)

Uninitialized data (.bss)

Initialized data (.data)

`0x00400000` →

Program text (.text)

Linux Organizes VM as Collection of "Areas"

Process virtual memory

vm_area_struct

task_struct

mm_struct

| mm |

| pgd |
| mmap |

| 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 |

Shared libraries

Data

Text

0
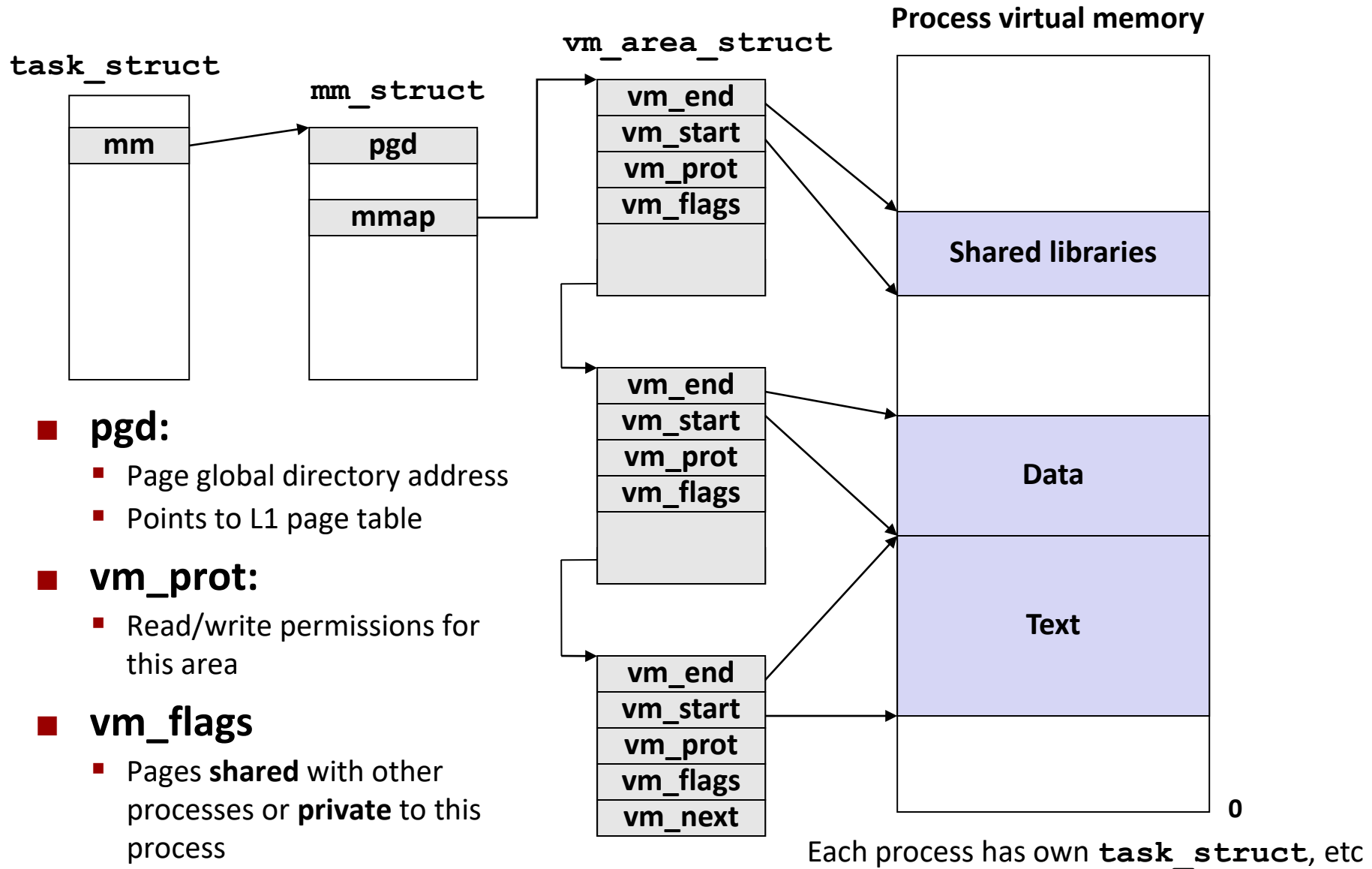
- **pgd:**
  - Page global directory address
  - Points to L1 page table

- **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

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

52

Linux Page Fault Handling

vm_area_struct

Process virtual memory

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)