Virtual Memory: Systems

Introduction to Computer Systems

Today

- **Virtual memory questions and answers**
- **Simple memory system example**
- **Case study: Core i7/Linux memory system**
- **Memory mapping**
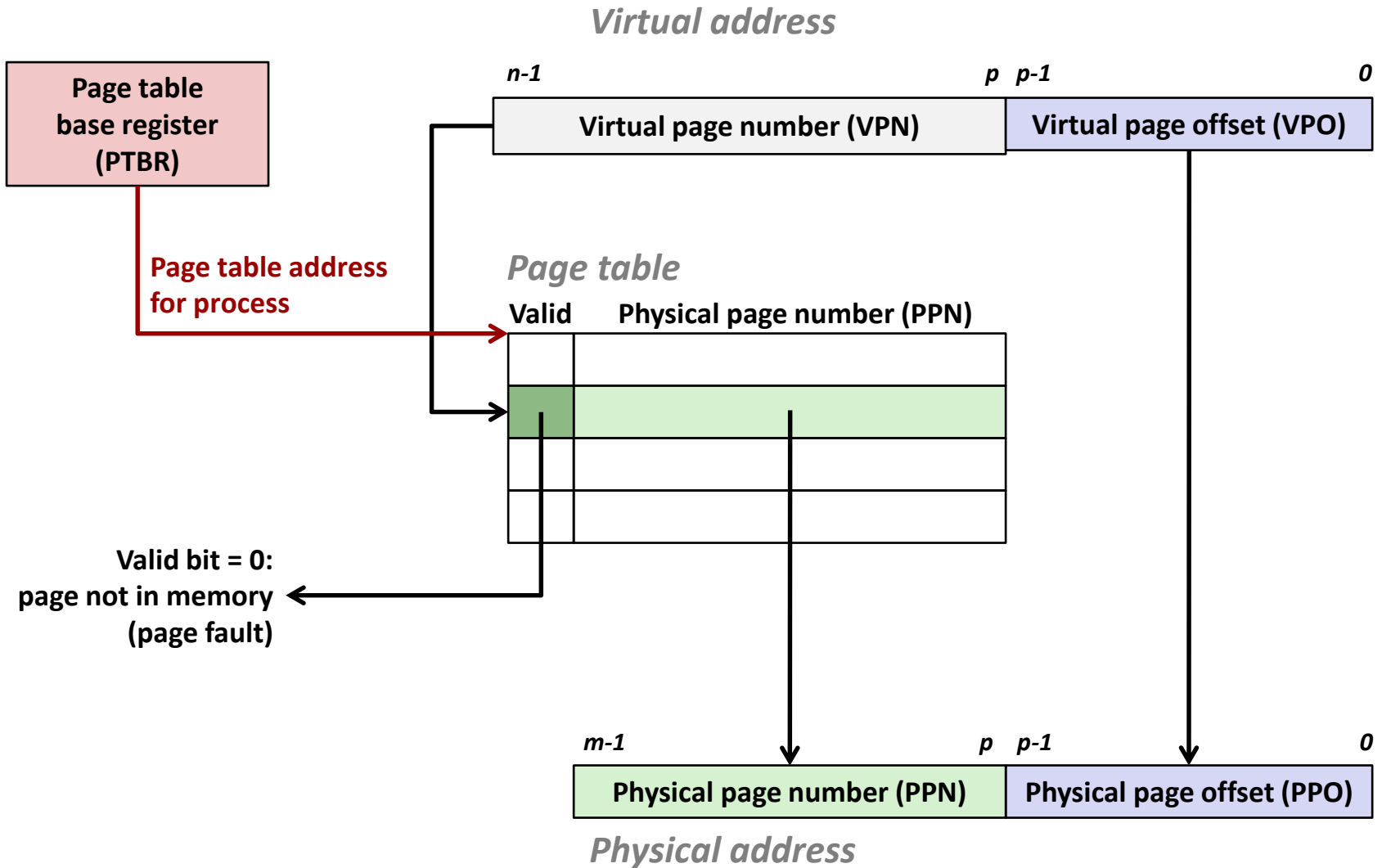
Virtual memory reminder/review

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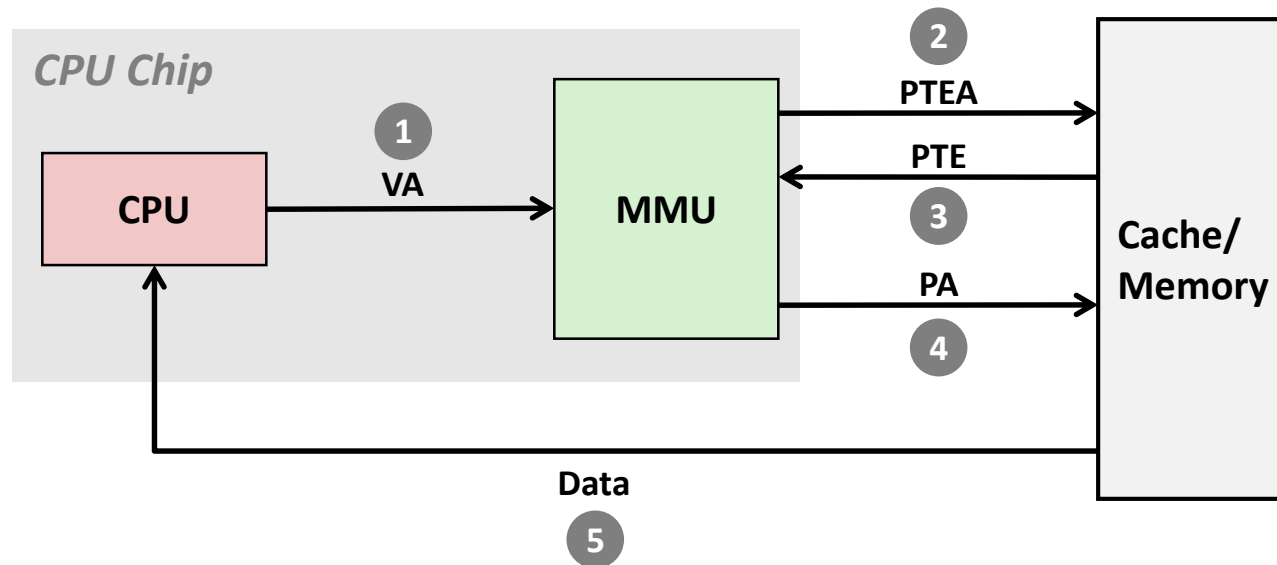 - Cannot be corrupted by other processes

- **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

Recall: Address Translation With a Page Table

Virtual address

Page table base register (PTBR)

Page table address for process

	Virtual page number (VPN)	Virtual page offset (VPO)

$n-1$ ... p $p-1$ ... 0

Page table

Valid **Physical page number (PPN)**

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

Physical page number (PPN)	Physical page offset (PPO)

$m-1$ ... p $p-1$ ... 0

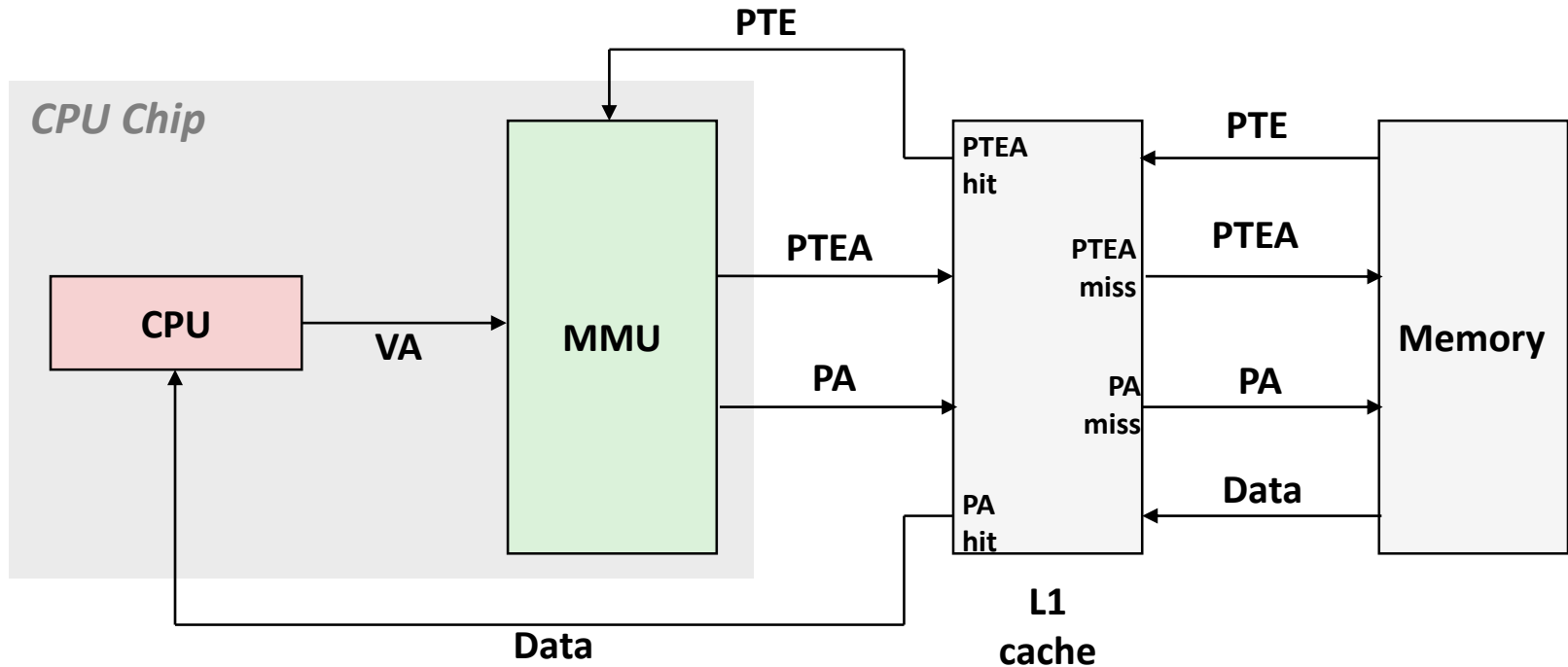*Physical address*

4

Recall: 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

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

CPU Chip

TLB

PTE

2

VPN

3

1

VA

CPU

MMU

PA

4

Cache/
Memory

Data

5
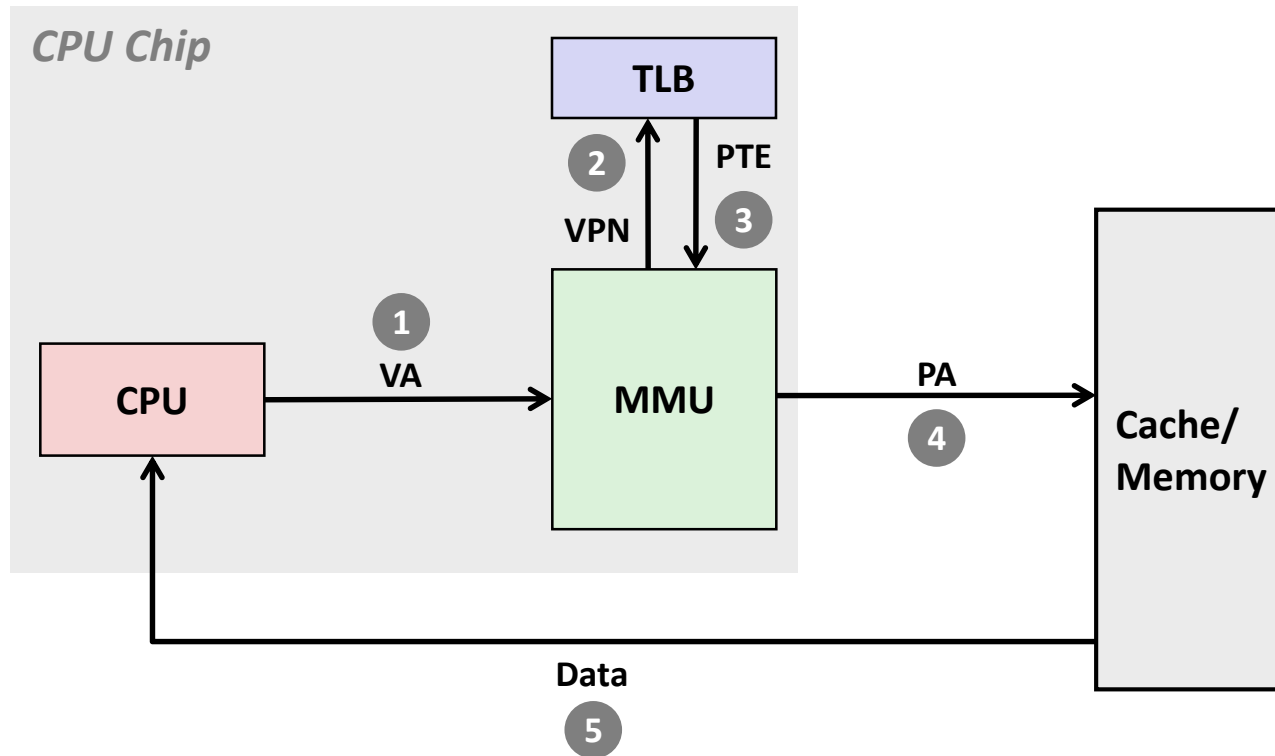
A TLB hit eliminates a memory access

TLB Miss
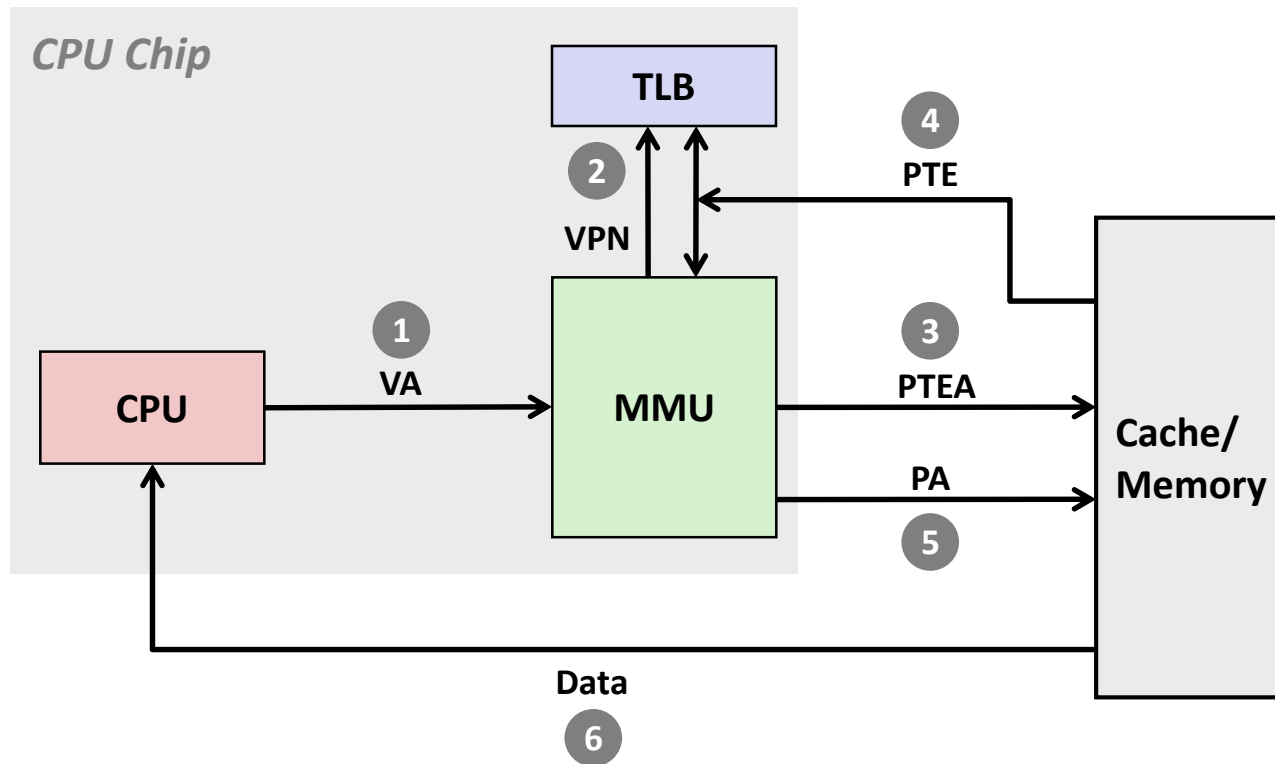
A TLB miss incurs an additional memory access (the PTE)

Fortunately, TLB misses are rare. Why?

Question #3

- **Aren't the TLB contents wrong after a context switch?**

- **Yes, they would be, so something must be done..**
  - Option 1: flush TLB on context switch
  - Option 2: associate a process ID with each TLB entry

Question #4

- **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

Level 2 Tables

- **Suppose:**
  - 4KB (2^{12}) page size, 64-bit address space, 8-byte PTE
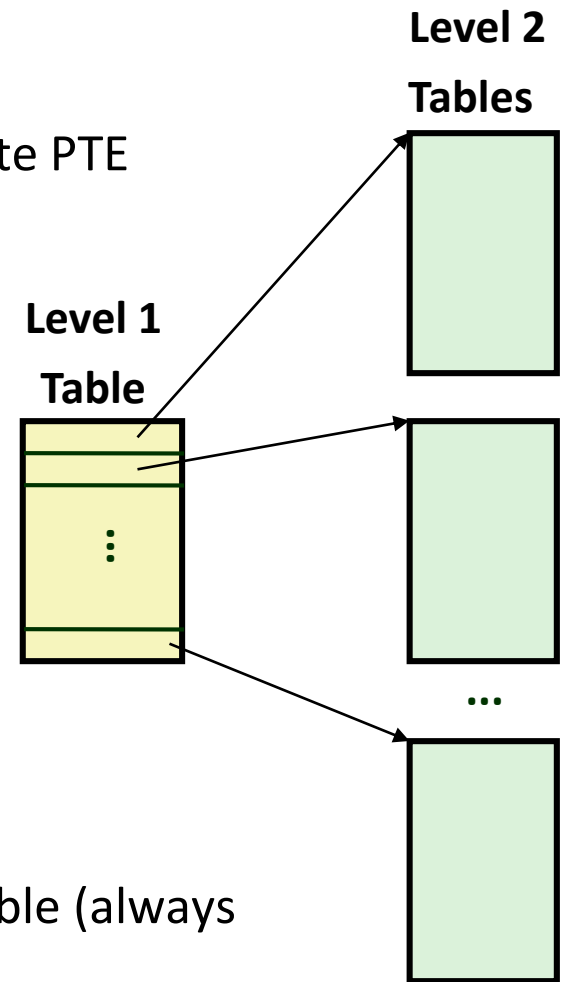
- **Problem:**
  - Would need a 32,000 TB page table!
    - $2^{64} * 2^{-12} * 2^3 = 2^{55}$ bytes

Level 1 Table

- **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

Level 1 page table	Level 2 page tables	Virtual memory

Level 1 page table

PTE 0
PTE 1
PTE 2 (null)
PTE 3 (null)
PTE 4 (null)
PTE 5 (null)
PTE 6 (null)
PTE 7 (null)
PTE 8
(1K - 9) null PTEs

Level 2 page tables

PTE 0
...
PTE 1023

PTE 0
...
PTE 1023

1023 null PTEs
PTE 1023

Virtual memory

0

VP 0
...
VP 1023
VP 1024
...
VP 2047

2K allocated VM pages for code and data

Gap

6K unallocated VM pages

1023 unallocated pages

1023 unallocated pages

VP 9215

1 allocated VM page for the stack

32 bit addresses, 4KB pages, 4-byte PTEs

Translating with a k-level Page Table
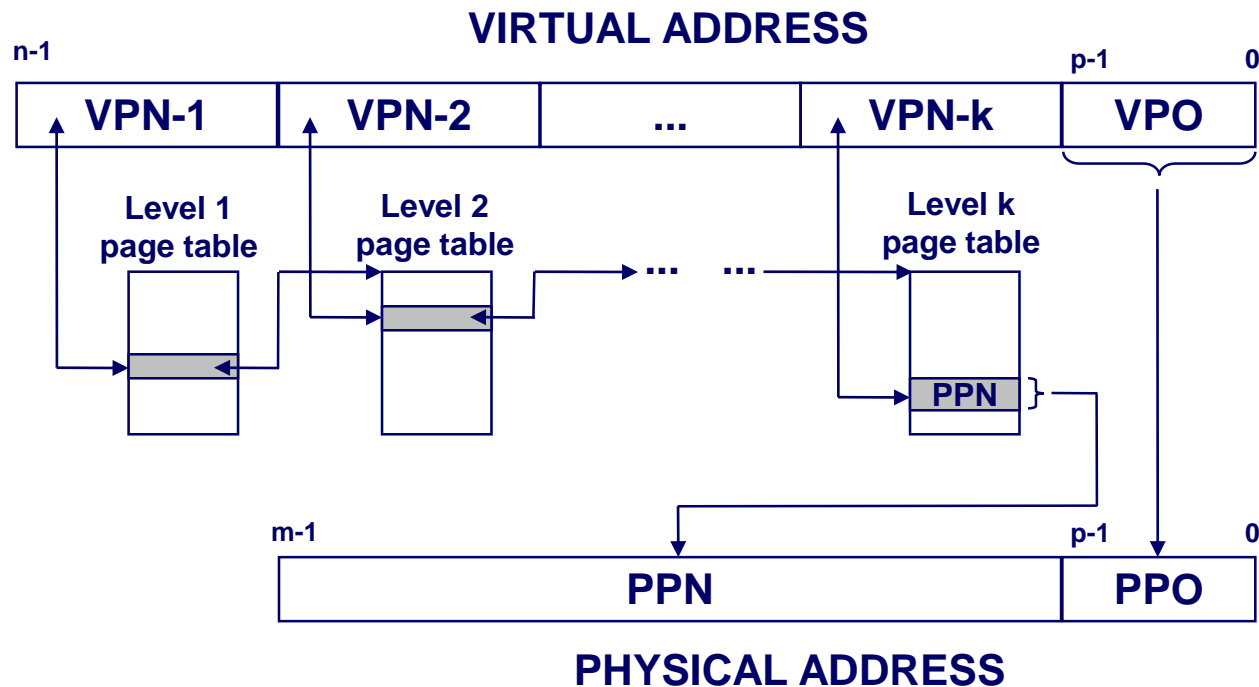
VIRTUAL ADDRESS

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

Level 1 page table   Level 2 page table   ... ...   Level k page table

PPN

m-1	p-1	0
PPN	PPO	

PHYSICAL ADDRESS

16

Today

- **Virtual memory questions and answers**
- **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

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**
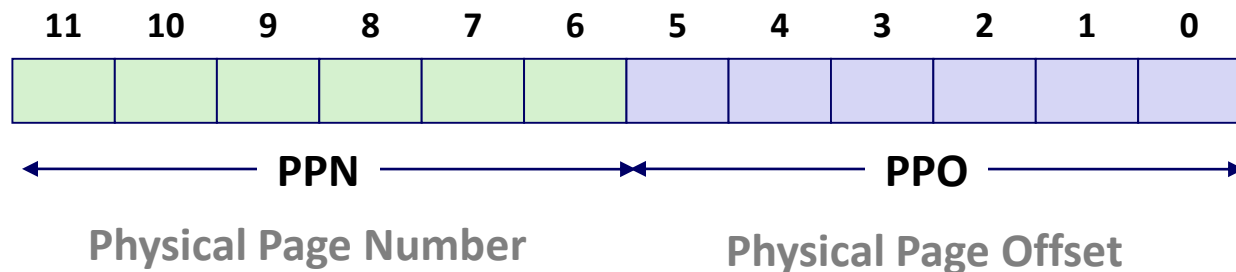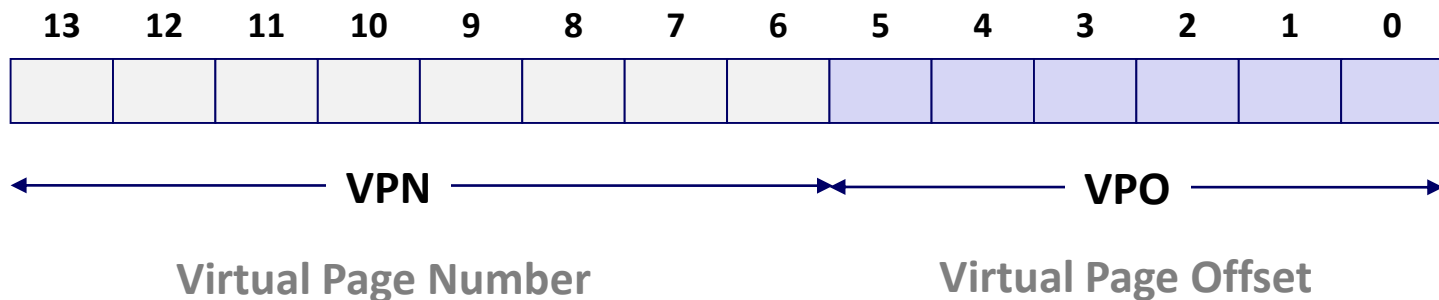
1. Simple Memory System TLB

- **16 entries**
- **4-way associative**
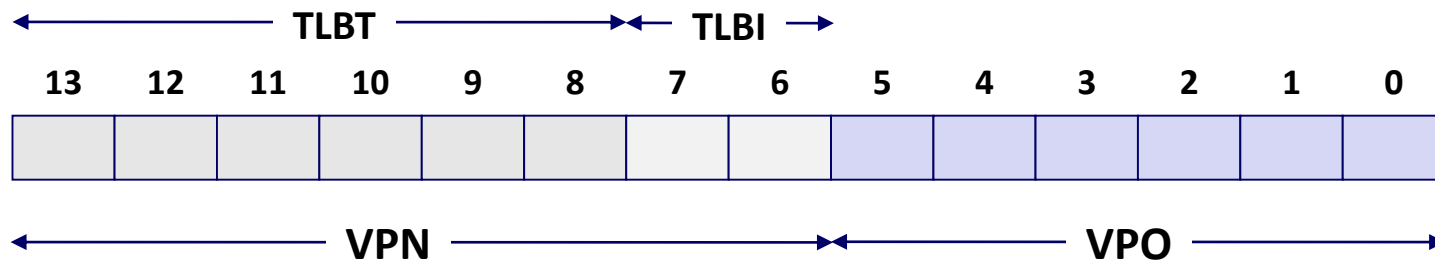
Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

2. Simple Memory System Page Table

Only show 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

3. Simple Memory System Cache

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

- **Physically addressed**

- **Direct mapped**

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

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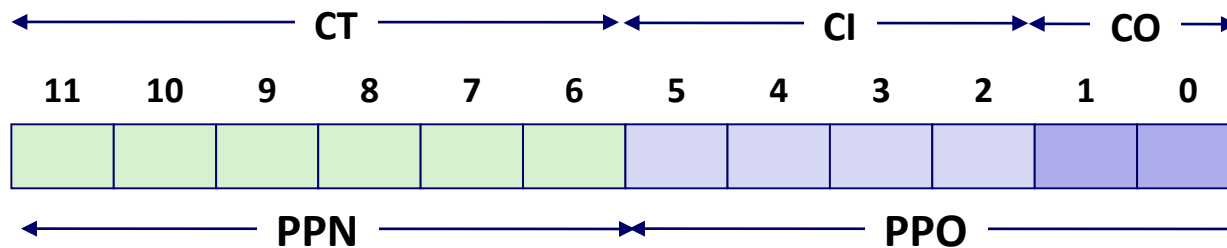
Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

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 #1

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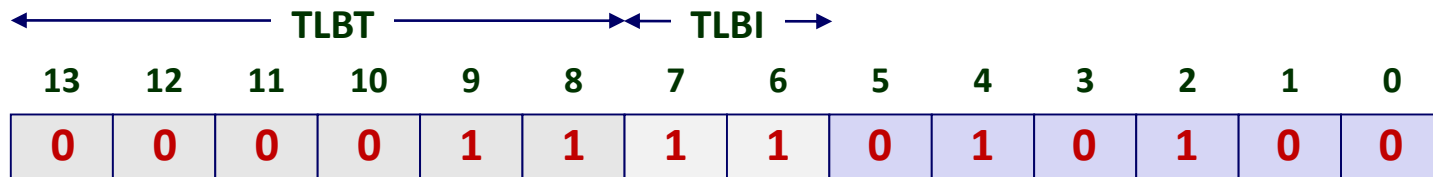
VPN **0x0F**   TLBI **0x3**   TLBT **0x03**   TLB Hit? **Y**   Page Fault? **N**   PPN: **0x0D**

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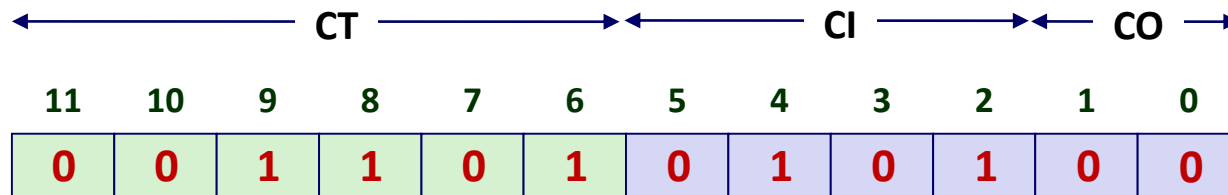
CO **0**   CI **0x5**   CT **0x0D**   Hit? **Y**   Byte: **0x36**

Address Translation Example #2

Virtual Address: `0x0B8F`

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

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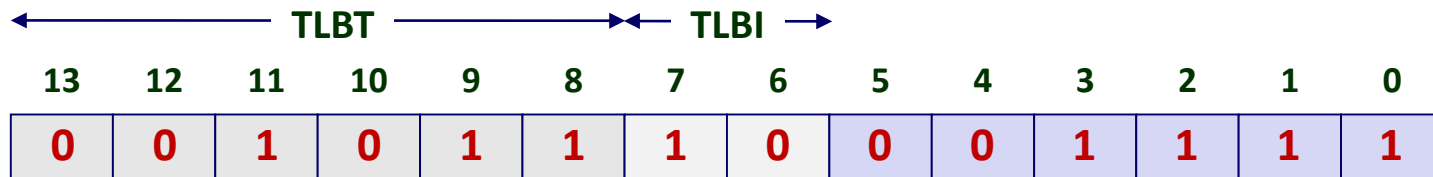
VPN **0x2E**   TLBI **2**   TLBT **0x0B**   TLB Hit? **N**   Page Fault? **Y**   PPN: **TBD**

Physical Address

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

PPN — PPO
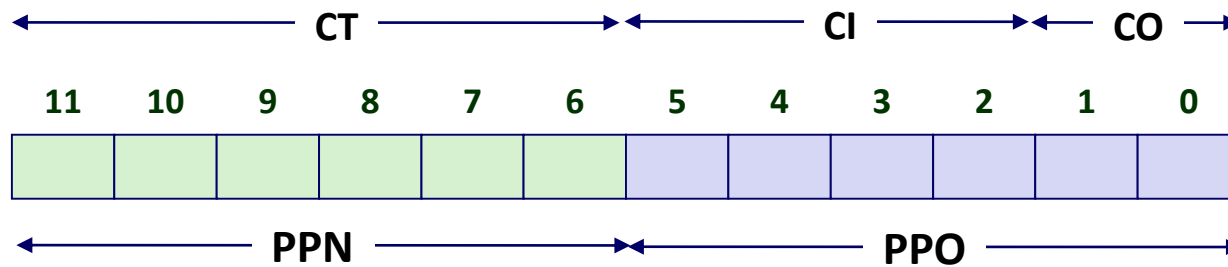
CO ___   CI ___   CT ____   Hit? __   Byte: ____

Address Translation Example #3

Virtual Address: `0x0020`

TLBT							TLBI						
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

VPN ←——————————————→ VPO
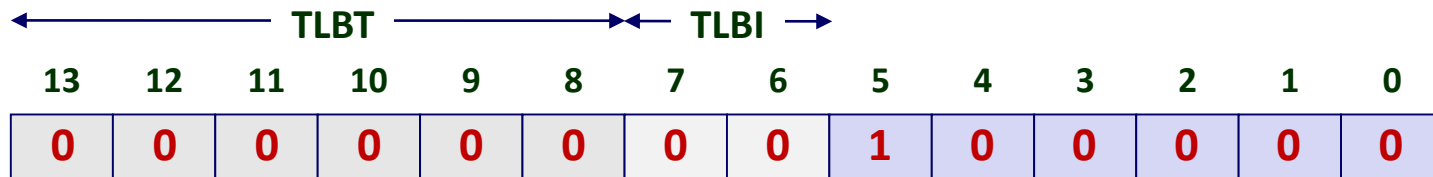
VPN **0x00**   TLBI **0**   TLBT **0x00**   TLB Hit? **N**   Page Fault? **N**   PPN: **0x28**

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
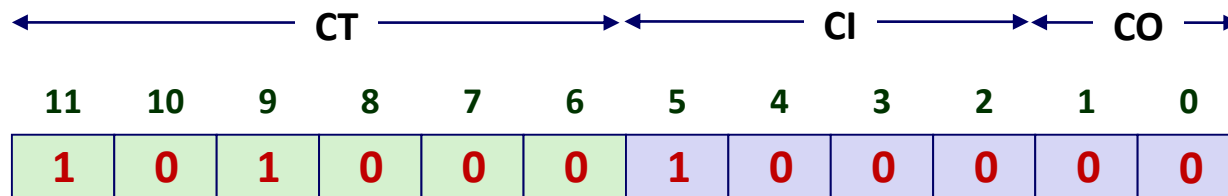
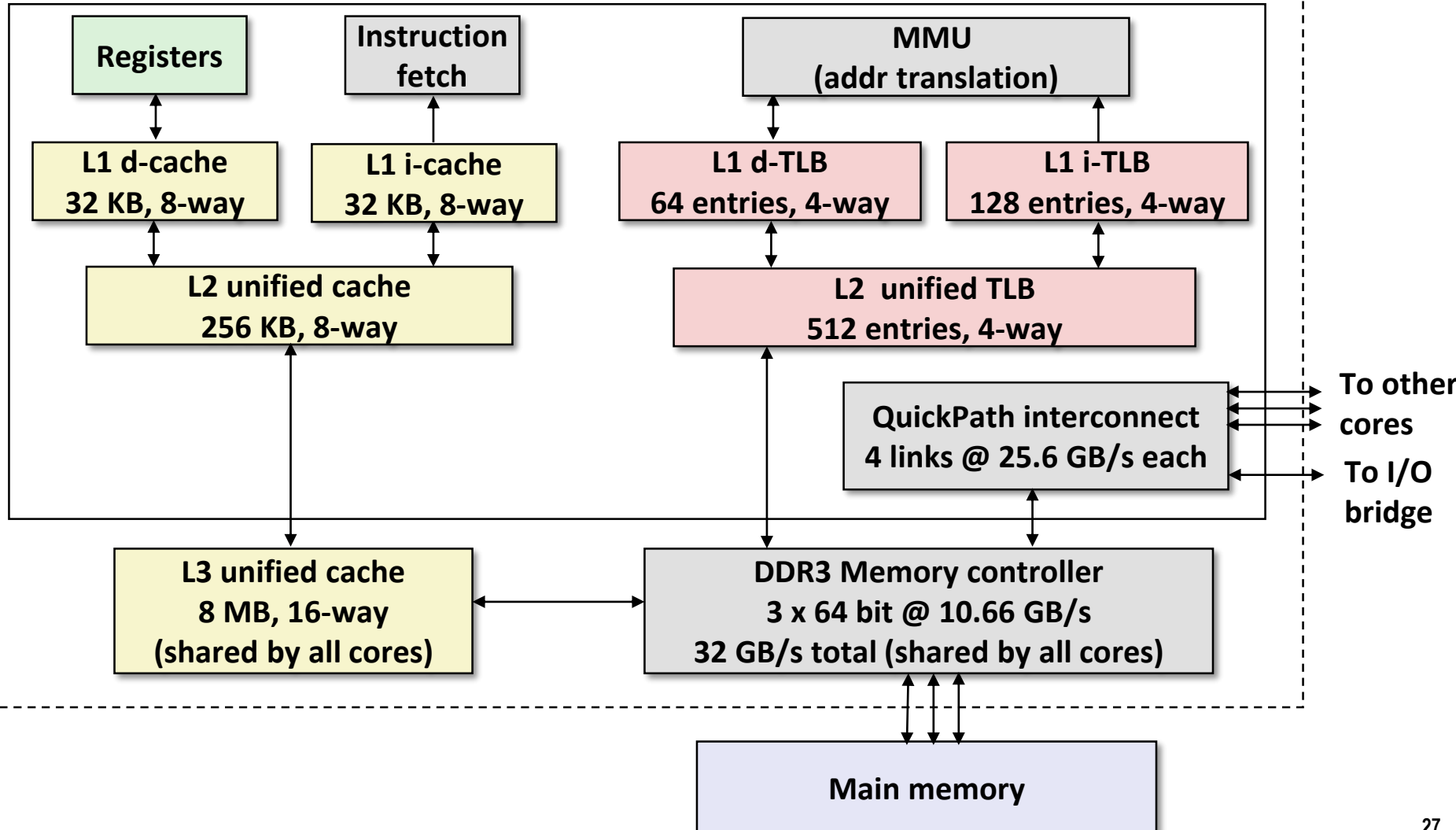CO **0**   CI **0x8**   CT **0x28**   Hit? **N**   Byte: **Mem**

Today

- **Virtual memory questions and answers**
- **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

27

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
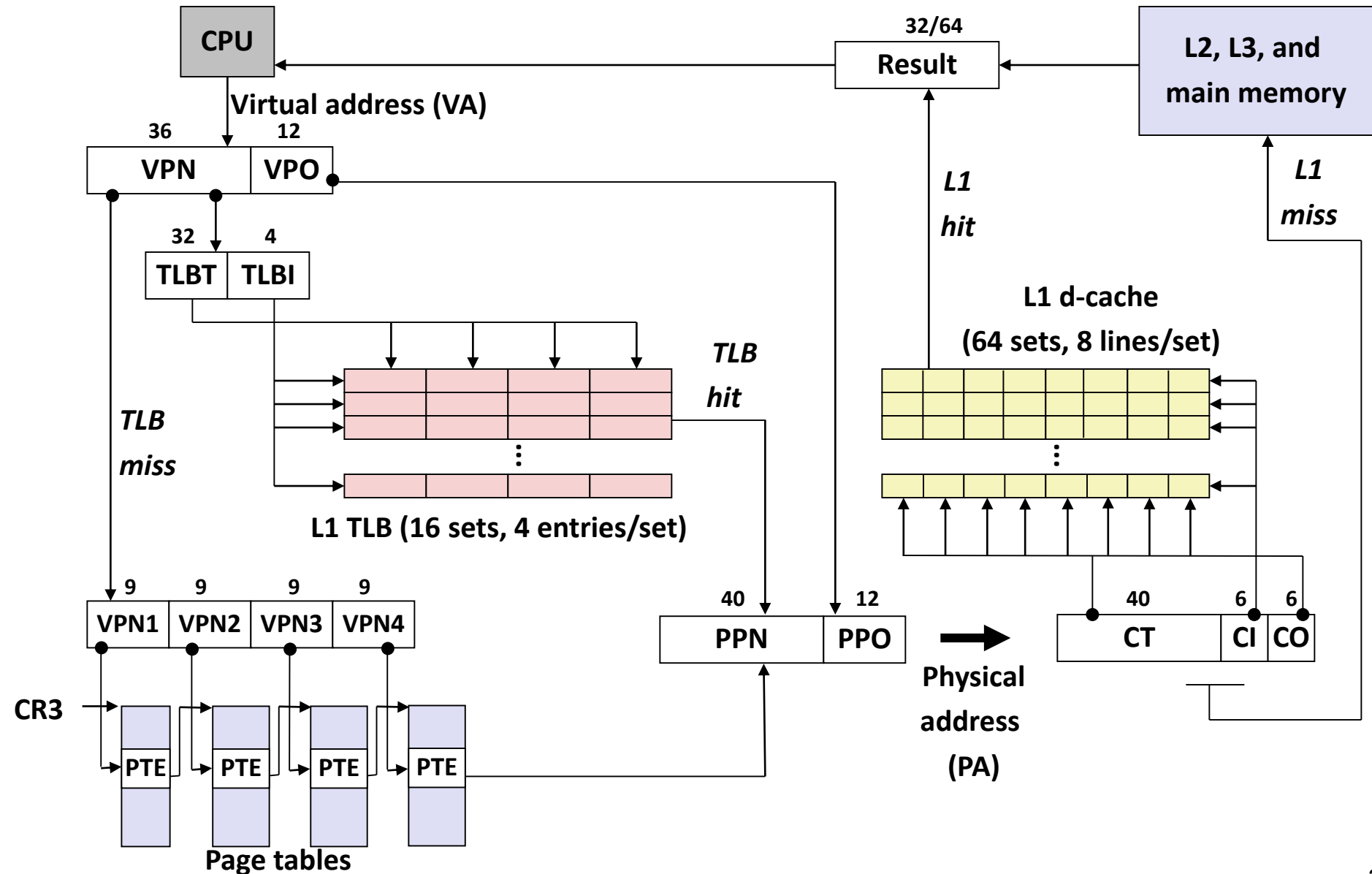 - **P = 2^p** : Page size (bytes)
- **Components of the virtual address (VA)**
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
- **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number
  - **CO**: Byte offset within cache line
  - **CI:** Cache index
  - **CT**: Cache tag

End-to-end Core i7 Address Translation

CPU

32/64

Result

L2, L3, and
main memory

Virtual address (VA)

36
VPN

12
VPO

32
TLBT

4
TLBI

TLB
hit

L1
hit

L1
miss

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

TLB
miss

L1 TLB (16 sets, 4 entries/set)

9
VPN1

9
VPN2

9
VPN3

9
VPN4

40
PPN

12
PPO

40
CT

6
CI

6
CO

Physical
address
(PA)

CR3

PTE   PTE   PTE   PTE

Page tables

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 3 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

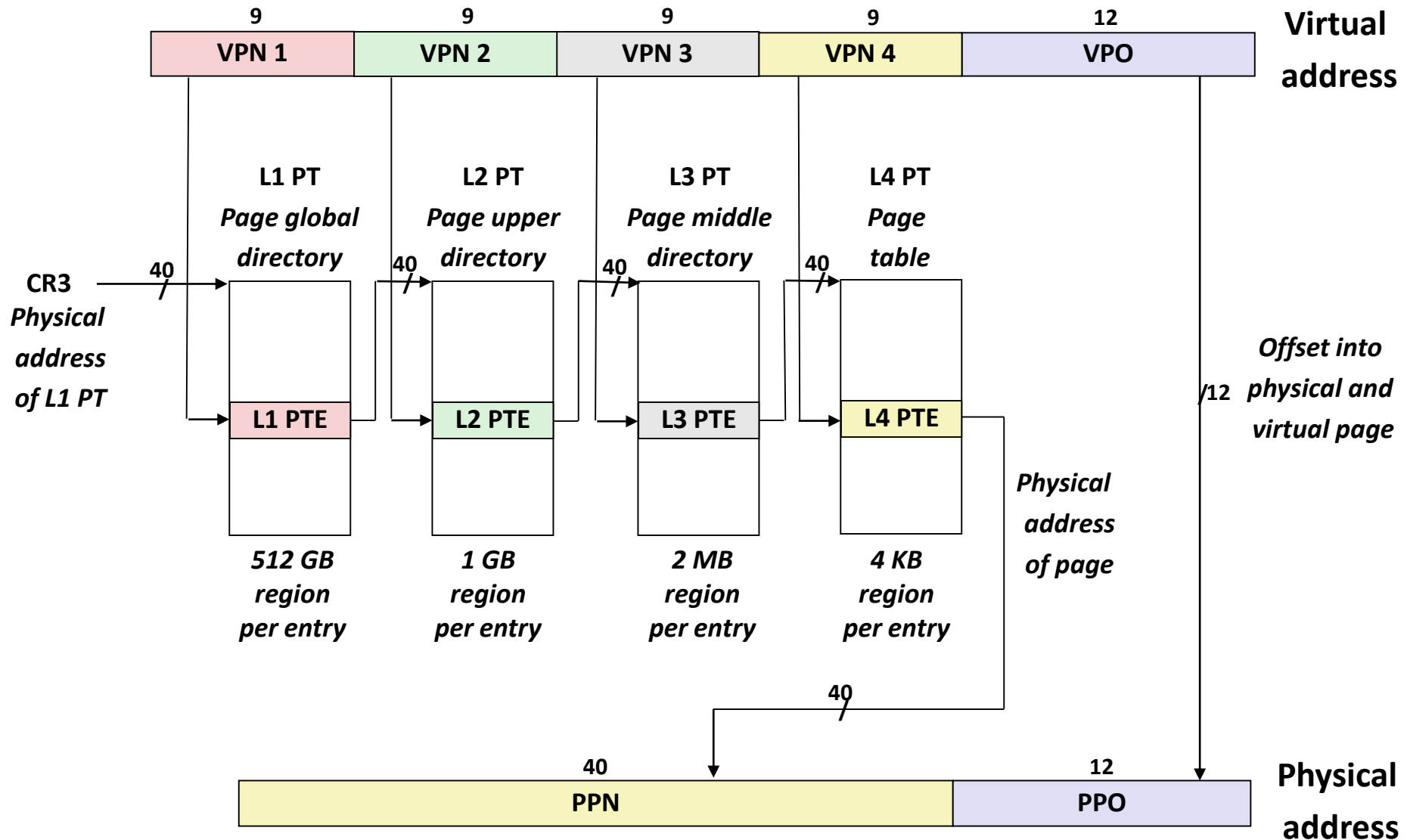**WT:** Write-through or write-back cache policy for this page

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

D: Dirty bit (set by MMU on writes, cleared by software)
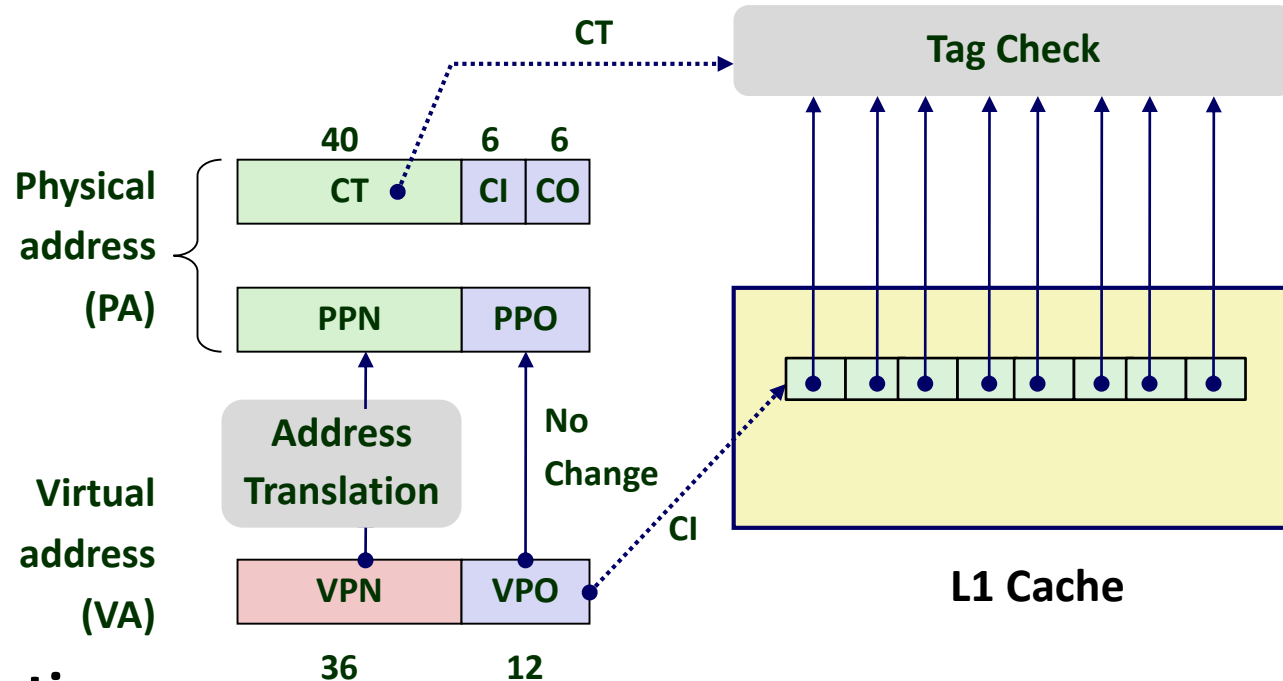
Page physical base address: 40 most significant bits of physical page address (forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.
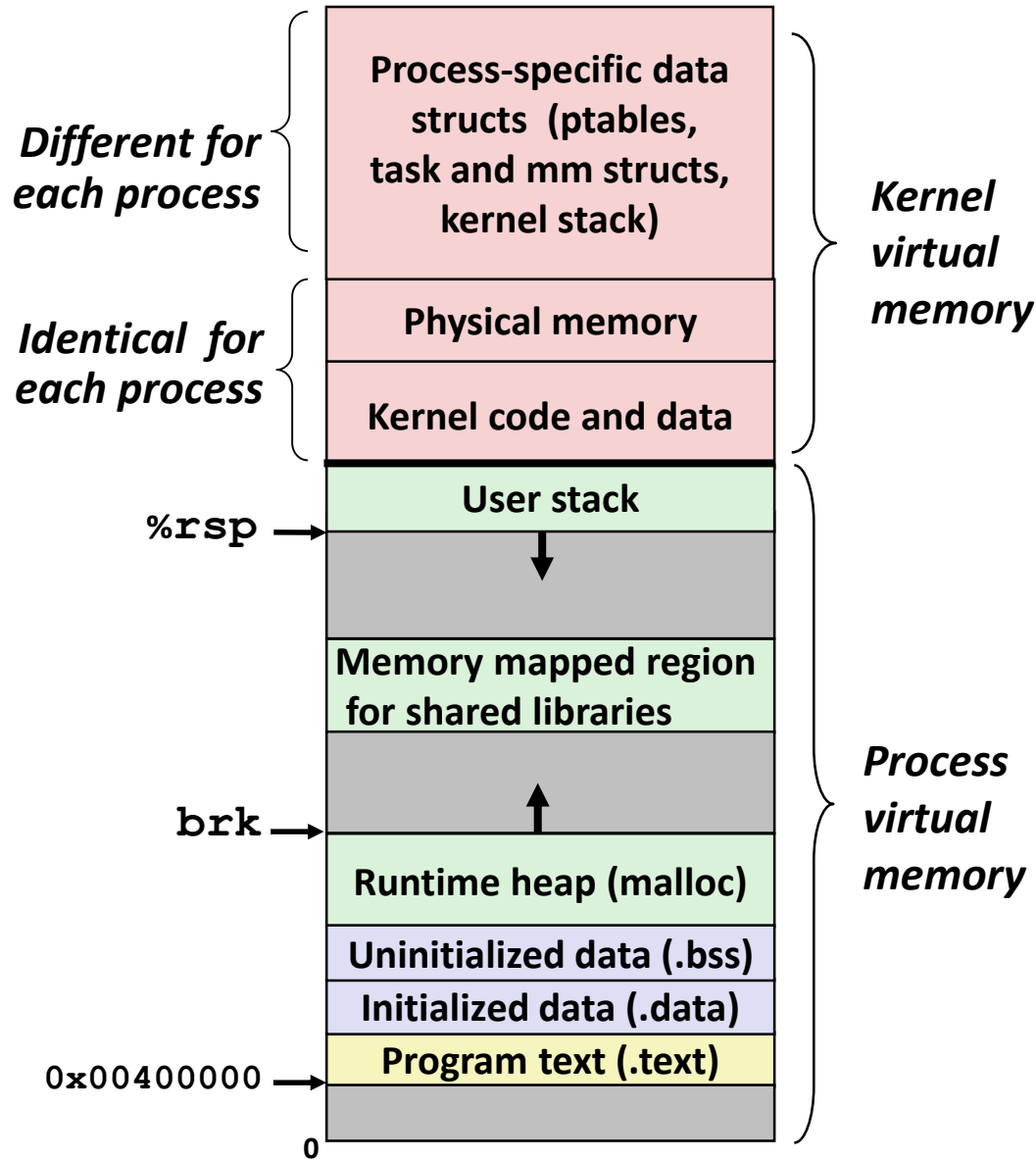
Core i7 Page Table Translation

Cute Trick for Speeding Up L1 Access

CT

Tag Check

| 40 | 6 | 6 |

Physical address (PA)

CT | CI | CO

PPN | PPO

Address Translation

No Change

CI

L1 Cache

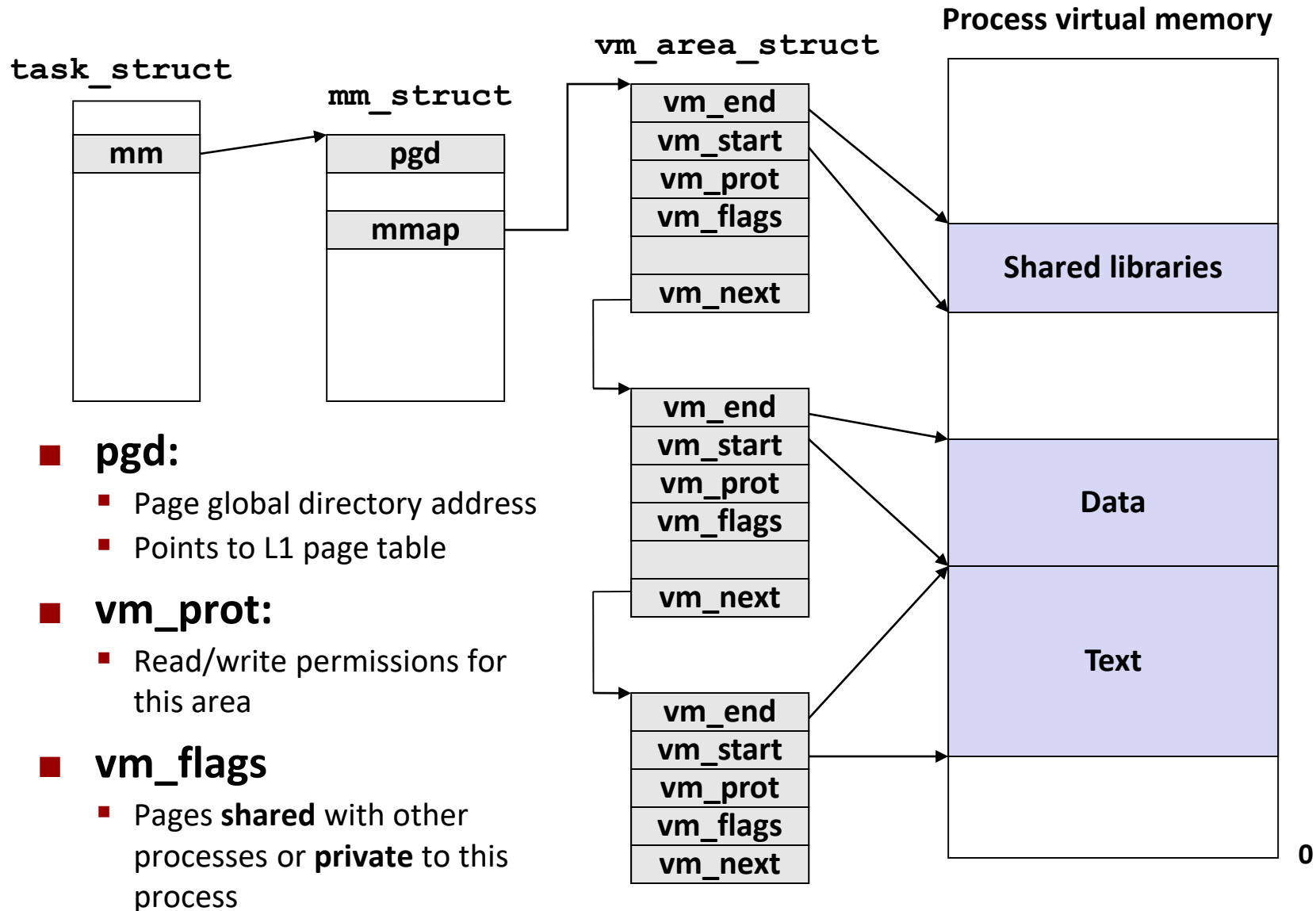**Virtual address (VA)**

VPN | VPO

36 | 12

- **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

33

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

User stack

`%rsp` →

Memory mapped region for shared libraries

Process virtual memory

`brk` →

Runtime heap (malloc)

Uninitialized data (.bss)

Initialized data (.data)

`0x00400000` → **Program text (.text)**

0

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_next |

Shared libraries

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

Data

Text

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

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

35

Linux Page Fault Handling

vm_area_struct

Process virtual memory

vm_end
vm_start
vm_prot
vm_flags
vm_next

vm_end
vm_start
vm_prot
vm_flags
vm_next

vm_end
vm_start
vm_prot
vm_flags
vm_next

shared libraries

data

text

1 read
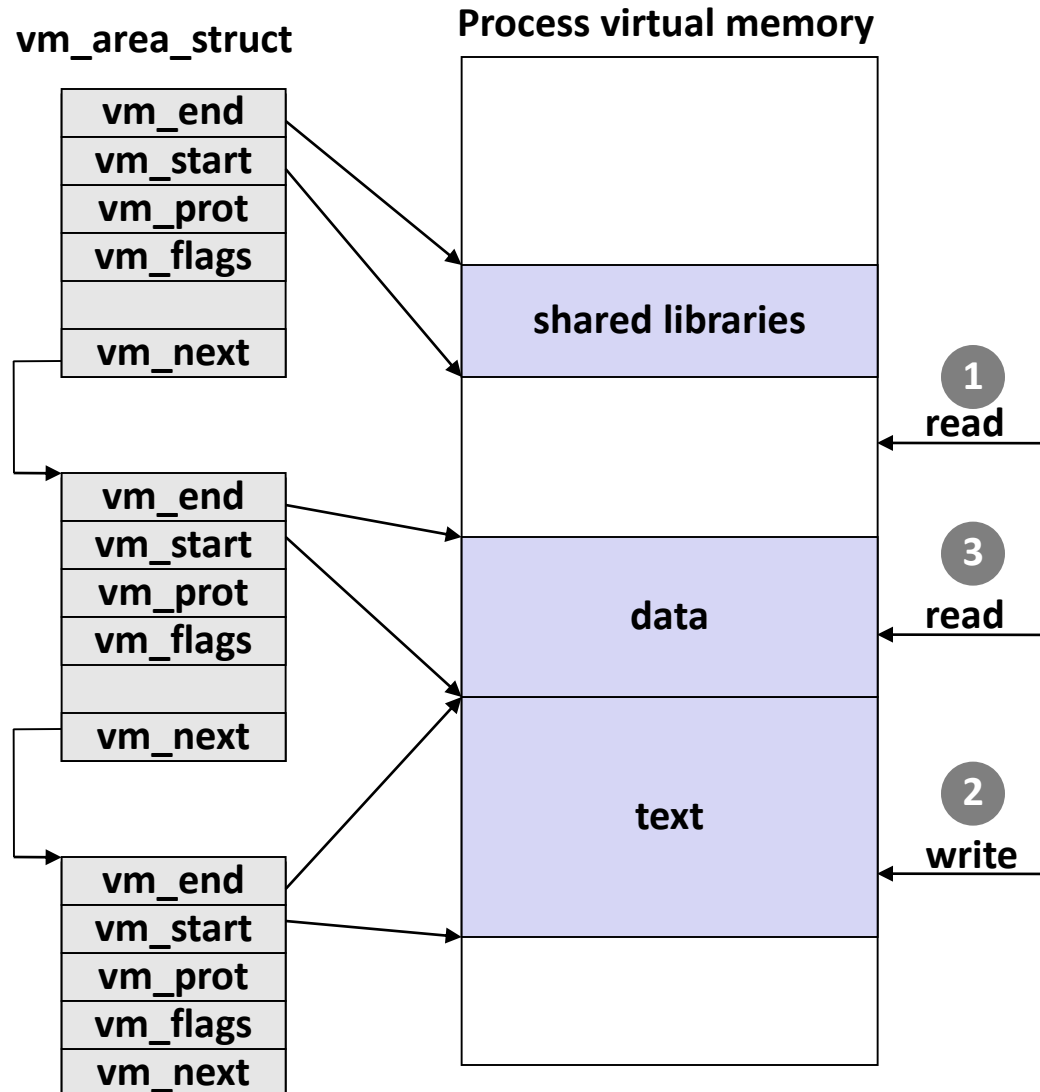
Segmentation fault:
accessing a non-existing page

3 read

Normal page fault

2 write

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

36

Today

- **Virtual memory questions and answers**
- **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*.**

Revisiting fork()

- **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**
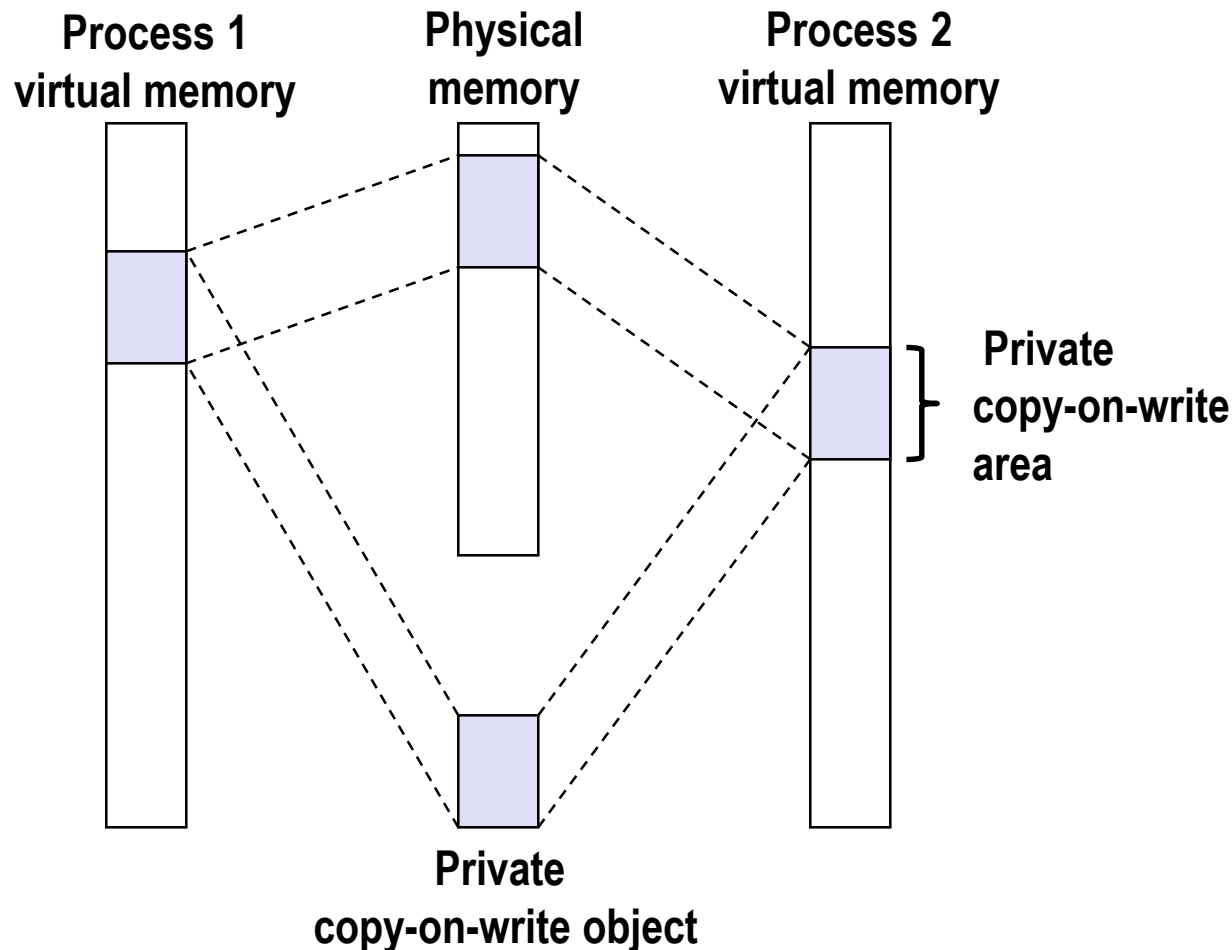
Sharing Revisited: Shared Objects

**Process 1
virtual memory**

**Physical
memory**

**Process 2
virtual memory**
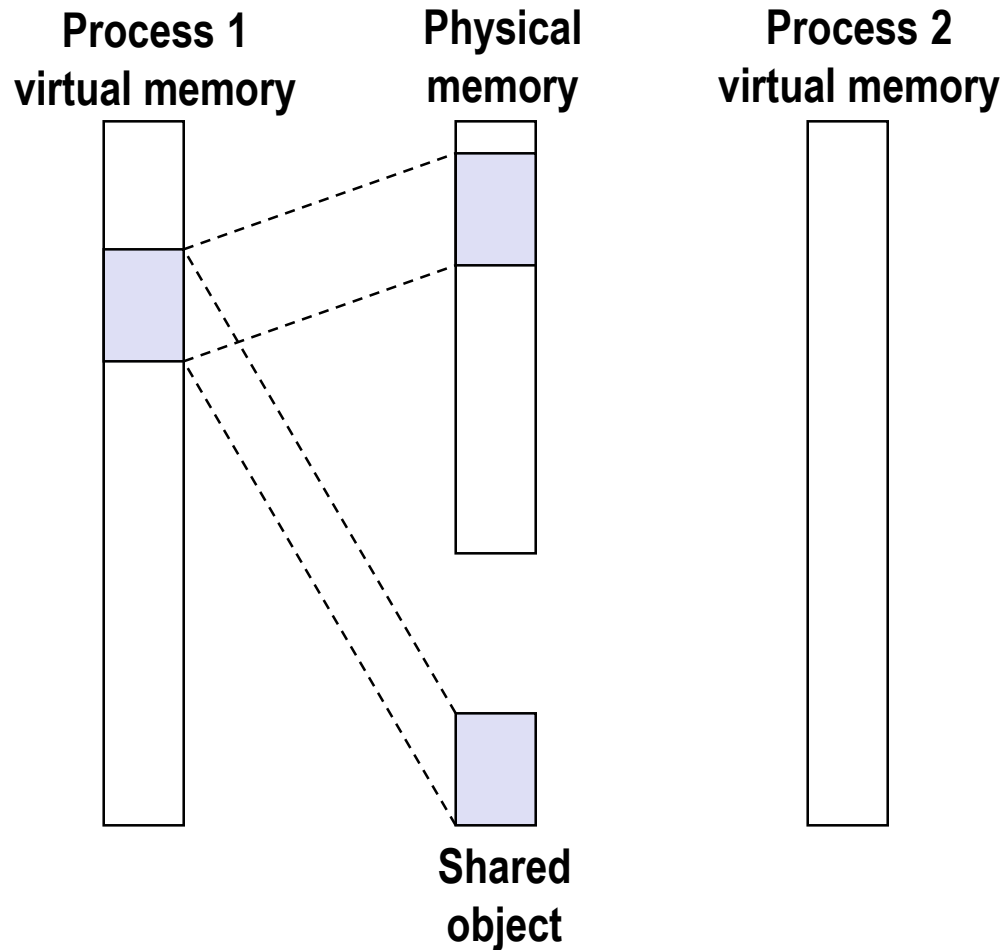
- **Process 1 maps
  the shared
  object.**

**Shared
object**

Sharing Revisited: Shared Objects

**Process 1
virtual memory**

**Physical
memory**

**Process 2
virtual memory**
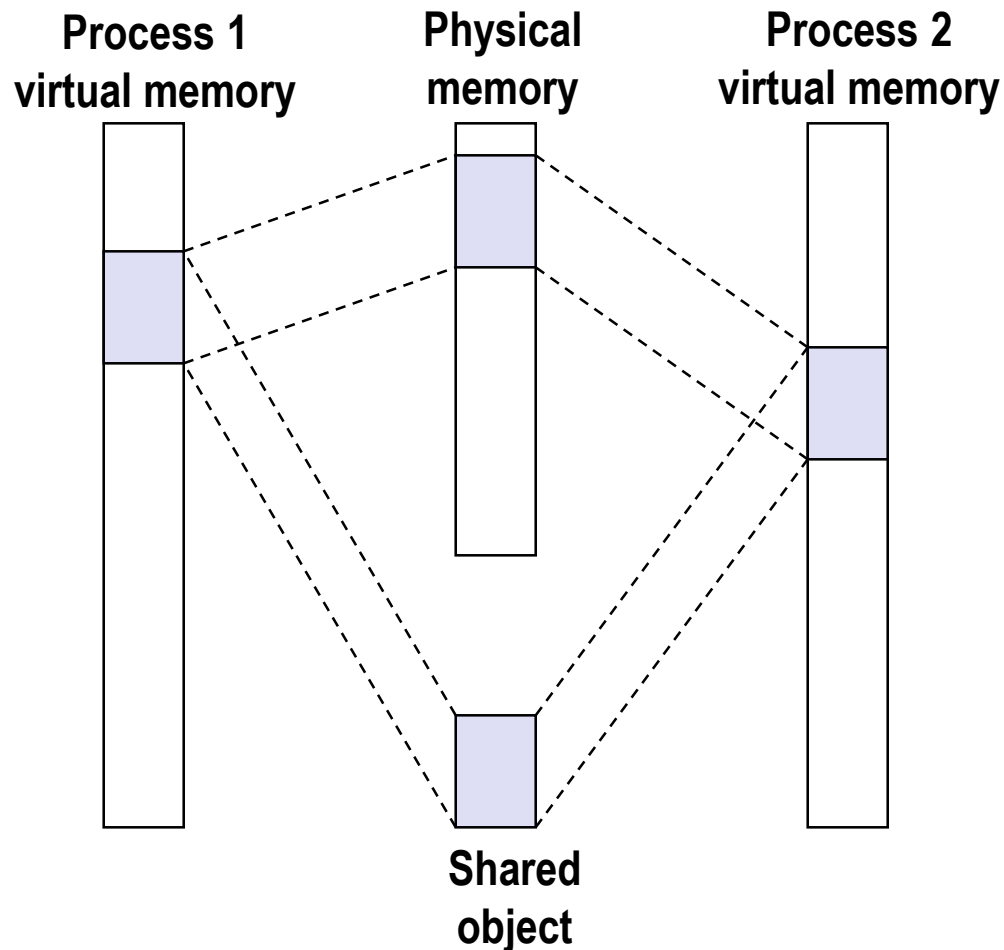
- **Process 2 maps
  the shared
  object.**

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

**Shared
object**

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

Process 1 virtual memory

Physical memory

Process 2 virtual memory

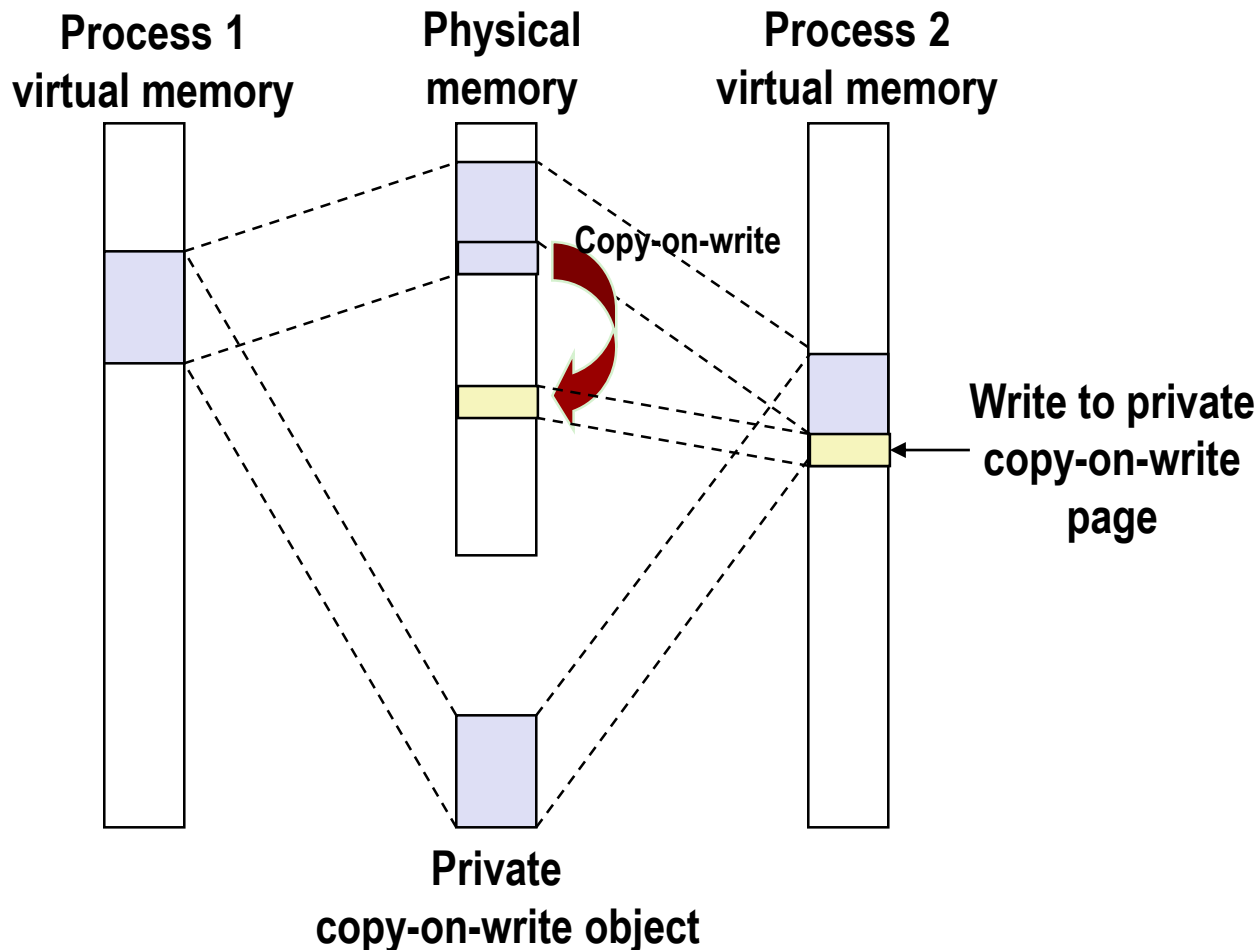Private copy-on-write area

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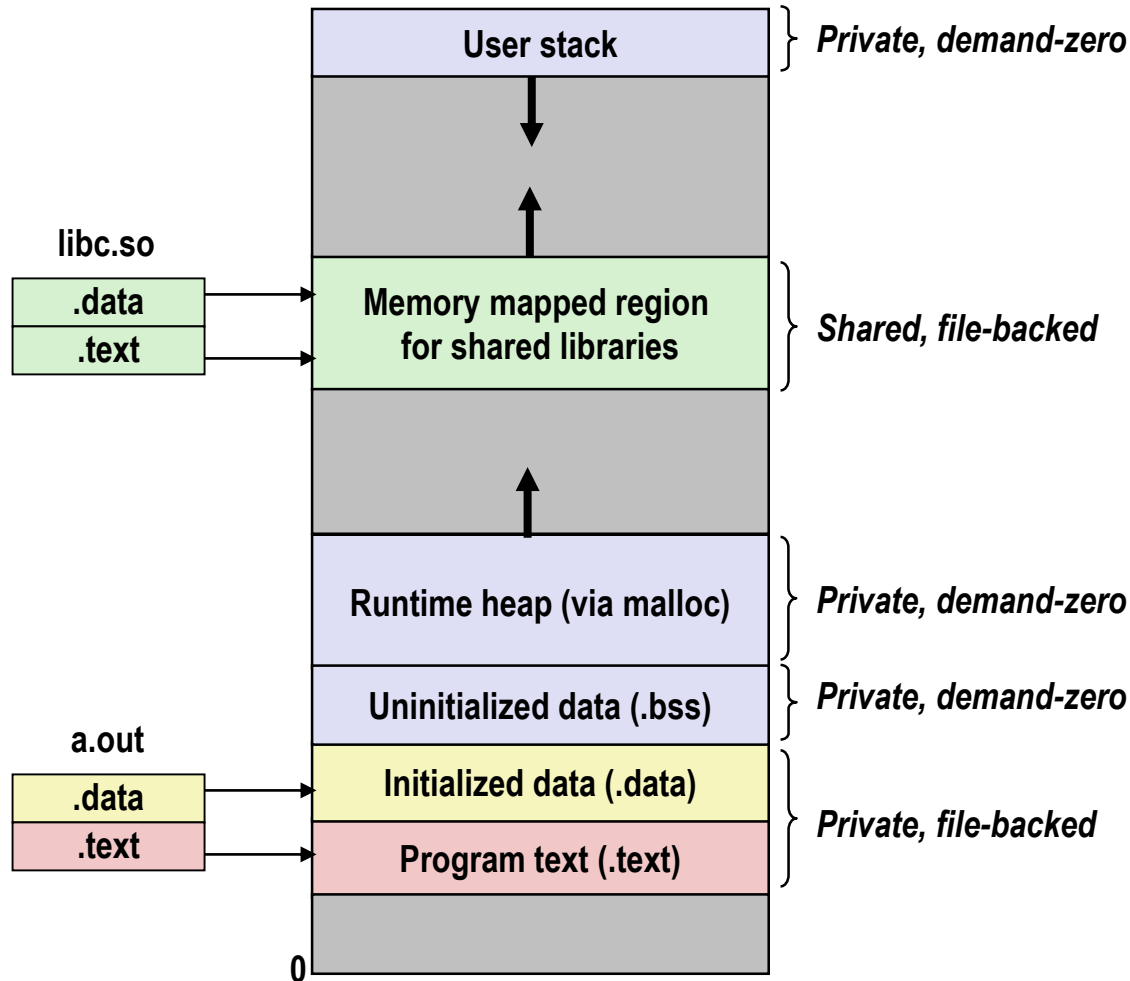
Sharing Revisited: Private Copy-on-write (COW) Objects

**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!**

The `fork` Function Revisited

- **VM and memory mapping explain how `fork` provides private address space for each process.**
  - Perfect approach for common case of fork() followed by exec()

- **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

libc.so

| .data |
| .text |

a.out

| .data |
| .text |

```
                                  ┐
         User stack               ├ Private, demand-zero
                                  ┘
      (blank)

  Memory mapped region            ┐
  for shared libraries            ├ Shared, file-backed
                                  ┘
      (blank)

  Runtime heap (via malloc)       ┐ Private, demand-zero
                                  ┘
  Uninitialized data (.bss)       ┐ Private, demand-zero
                                  ┘
  Initialized data (.data)        ┐
                                  ├ Private, file-backed
  Program text (.text)            ┘
0
```

- **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.
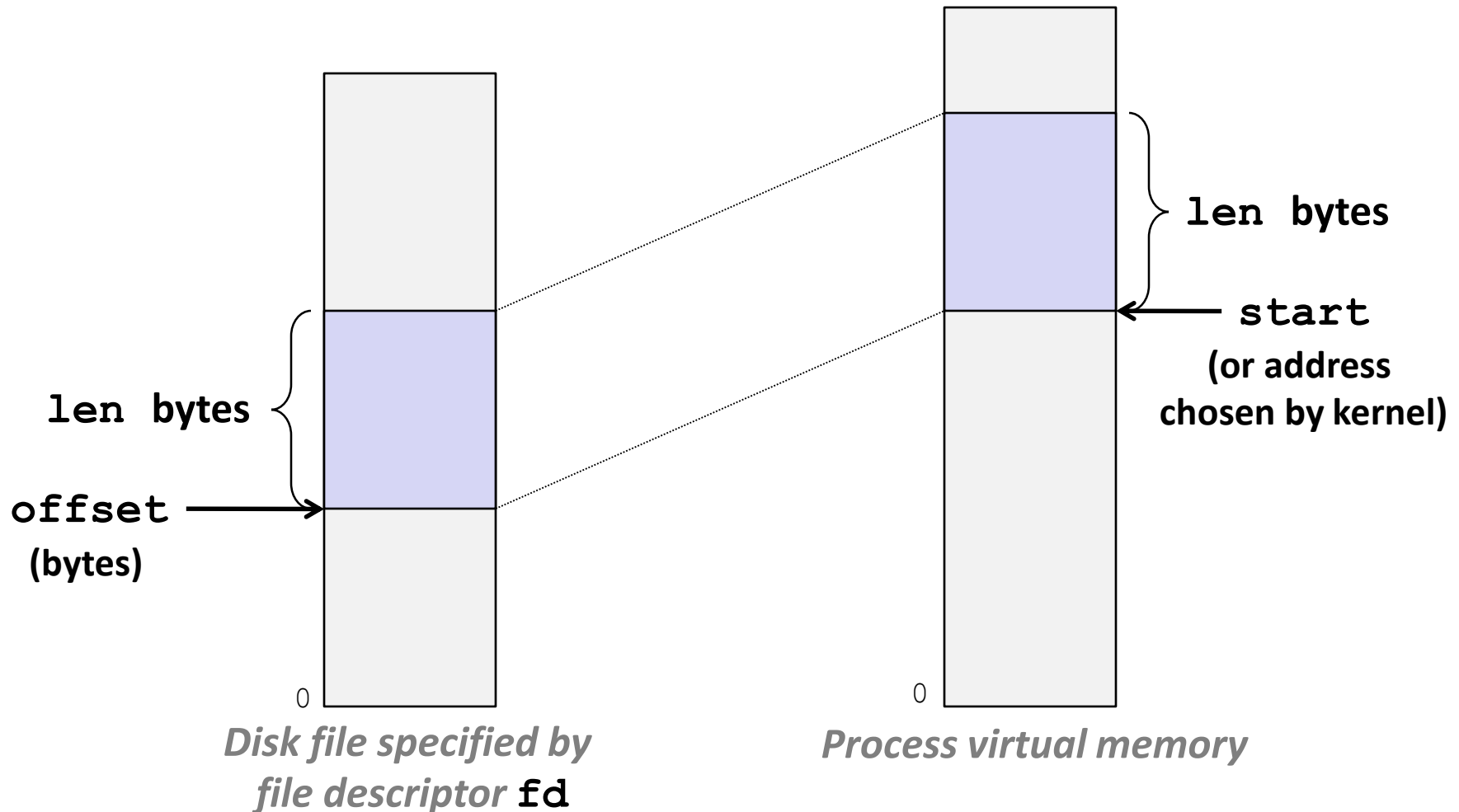
User-Level Memory Mapping

```
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)
```


len bytes

offset
(bytes)

len 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