

Virtual Memory: Systems

15-213: Introduction to Computer Systems
16th Lecture, Oct. 19, 2010

Instructors:

Randy Bryant and Dave O'Hallaron

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

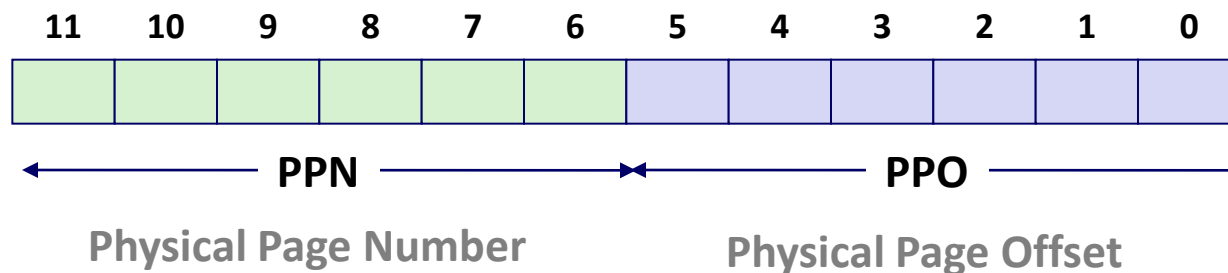
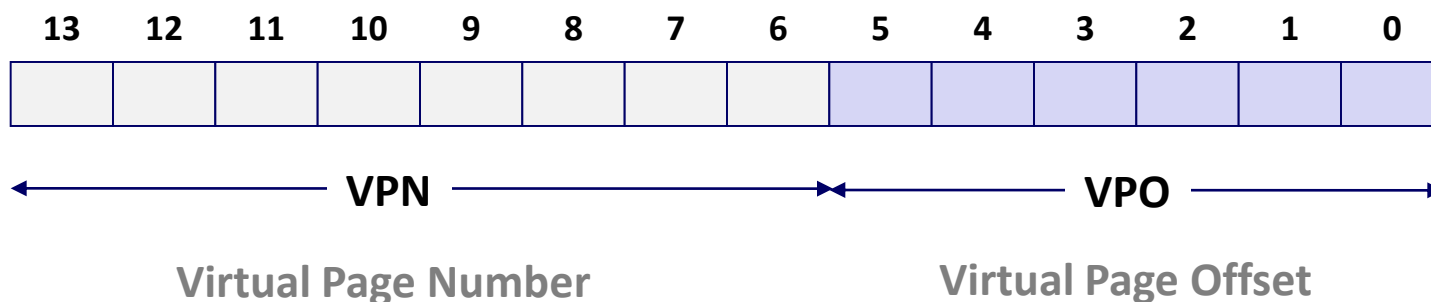
■ Components of the physical address (PA)

- 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 Page Table

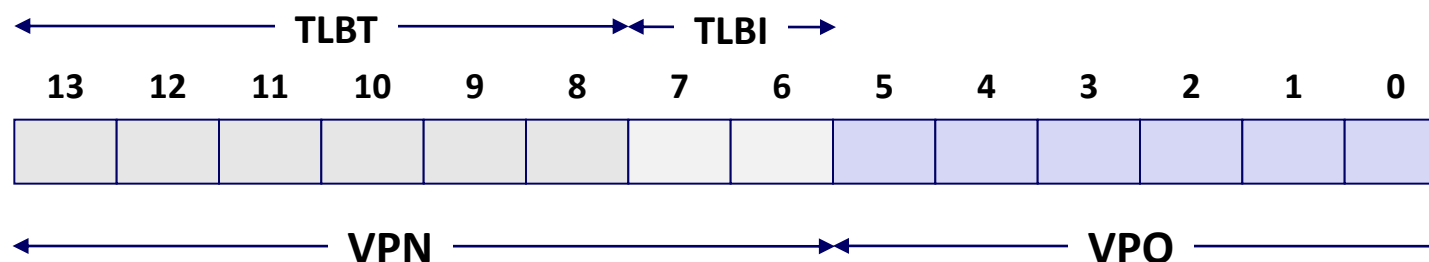
Only show first 16 entries (out of 256)

<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

Simple Memory System TLB

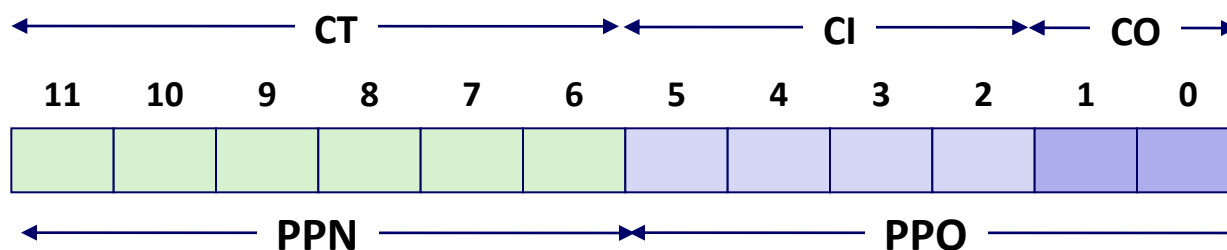
- 16 entries
- 4-way associative



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

Simple Memory System Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped

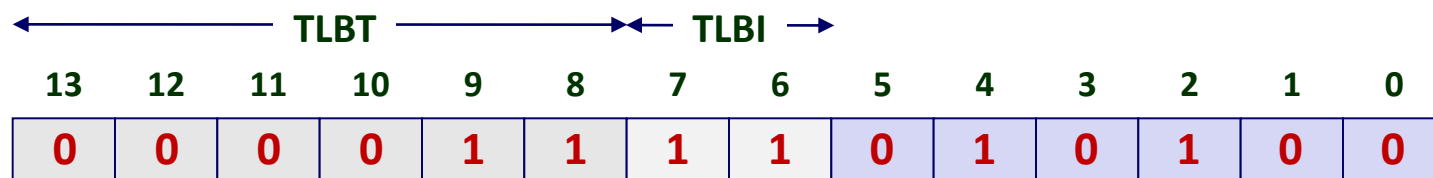


<i>Idx</i>	<i>Tag</i>	<i>Valid</i>	<i>B0</i>	<i>B1</i>	<i>B2</i>	<i>B3</i>
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

<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	–	–	–	–

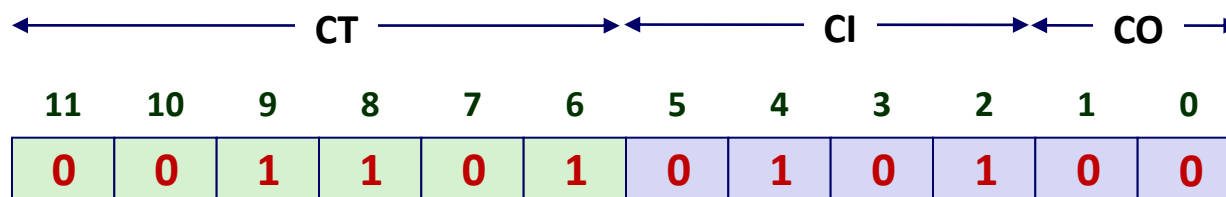
Address Translation Example #1

Virtual Address: 0x03D4



VPN 0x TLBI 0x TLBT 0x TLB Hit? Page Fault? PPN: 0x

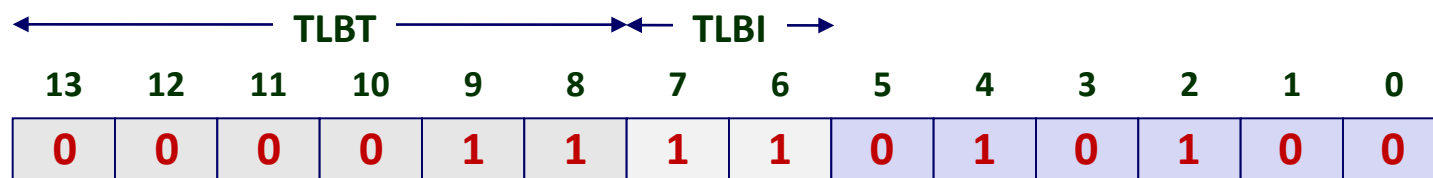
Physical Address



CO CI 0x CT 0x Hit? Byte: 0x

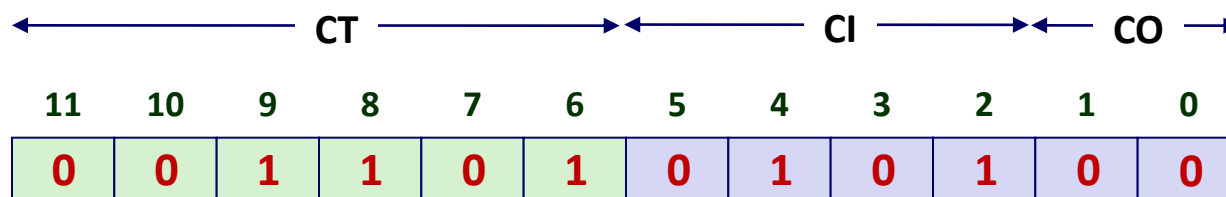
Address Translation Example #1

Virtual Address: 0x03D4



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

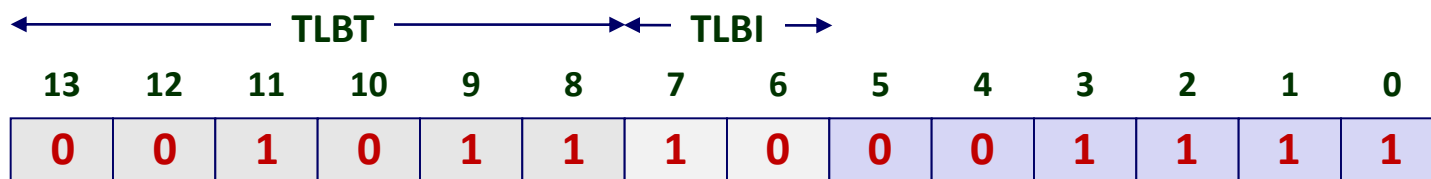
Physical Address



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

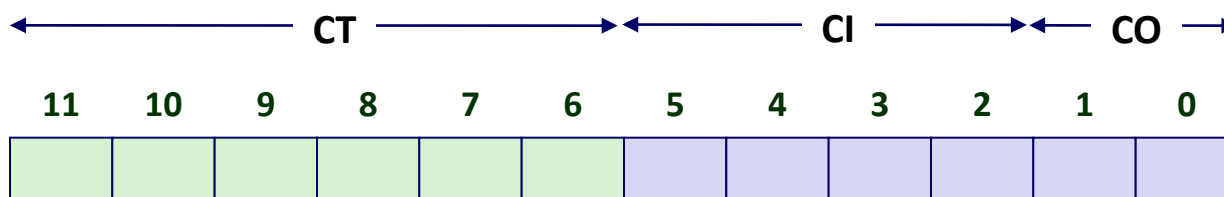
Address Translation Example #2

Virtual Address: 0x0B8F



VPN 0x TLBI 2 TLBT 0x TLB Hit? Page Fault? PPN:

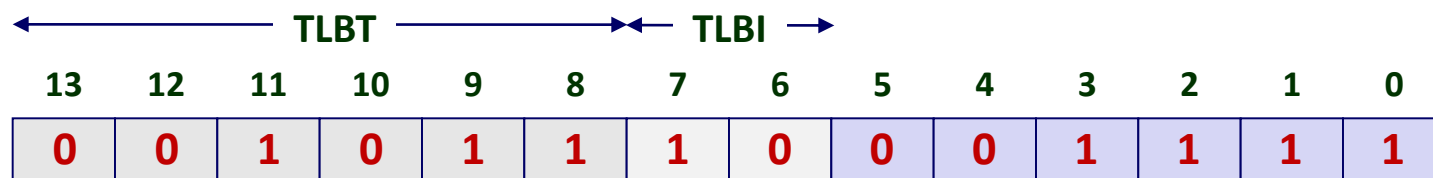
Physical Address



CO CI CT Hit? Byte:

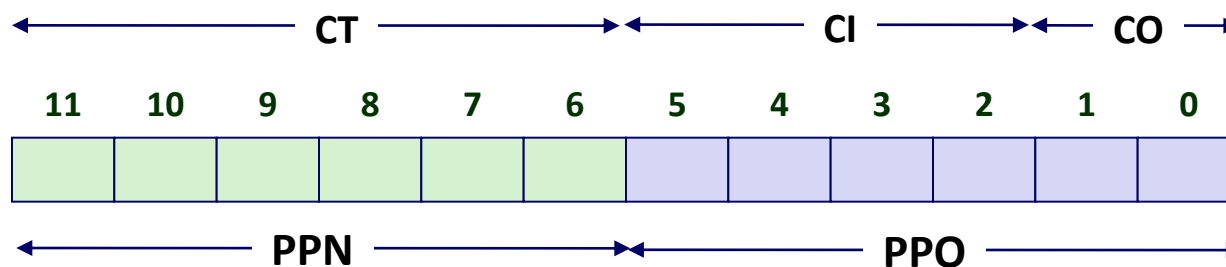
Address Translation Example #2

Virtual Address: 0x0B8F



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

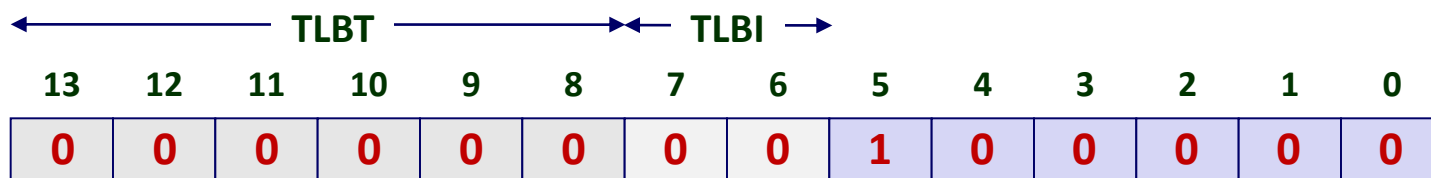
Physical Address



CO ____ CI ____ CT ____ Hit? ____ Byte: ____

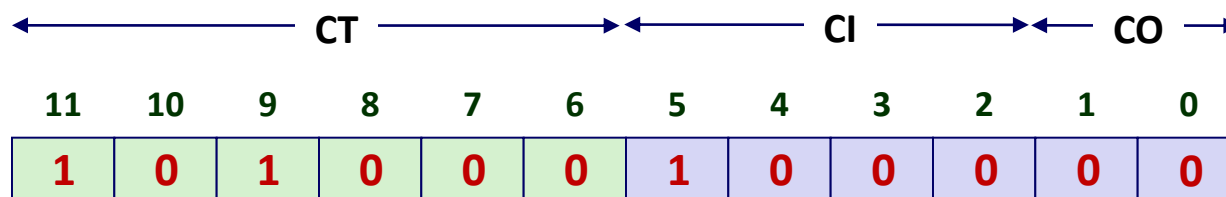
Address Translation Example #3

Virtual Address: 0x0020



VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? N Page Fault? N PPN: 0x28

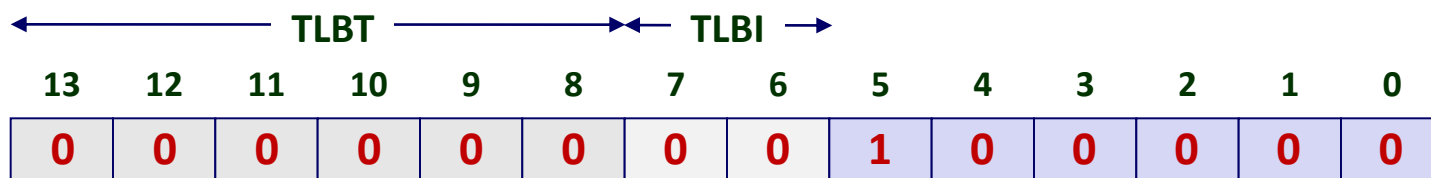
Physical Address



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

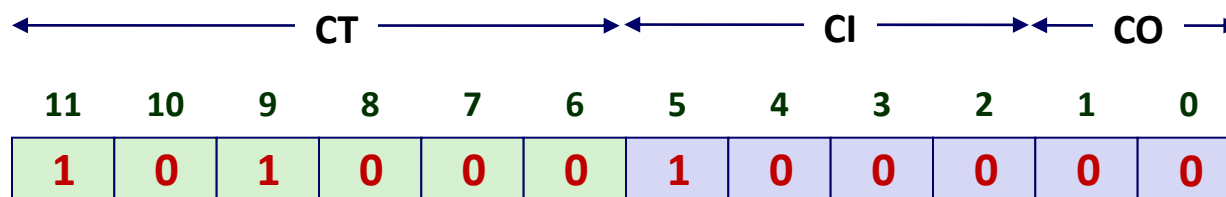
Address Translation Example #3

Virtual Address: 0x0020



VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? N Page Fault? N PPN: 0x28

Physical Address



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

Today

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

Single level page table

- **64bit address space, 4KB page size, 4GB physical memory**
- **One entry per virtual page**
 - 2^{64} addressable bytes / 2^{12} bytes per page = 2^{52} page table entries
- **Page table entry size**
 - One page table entry contains:
 - Physical page number + Access control bits
 - 4GB physical memory = 2^{32} bytes
 - 2^{32} bytes memory / 2^{12} bytes per page = 2^{20} physical pages
 - 20 bits needed for physical page number
 - Page table entry = ~ 4 bytes
- **Page table size**
 - 2^{52} page table entries * 4 bytes = 2^{54} bytes (16 petabytes) per process

Multilevel page table

- **Let's make sure that any page table requires only a single page (4KB)**
 - Page table entry = ~ 4 bytes
 - $4\text{KB page} / 4\text{bytes per PTE} = 1024$ entries
 - 10 bits of address space needed
 - We have 52 bit address for page table
 - We need $\text{ceiling}(52/10) = 6$ levels needed !
- **For each page table access, we need 6 memory accesses**

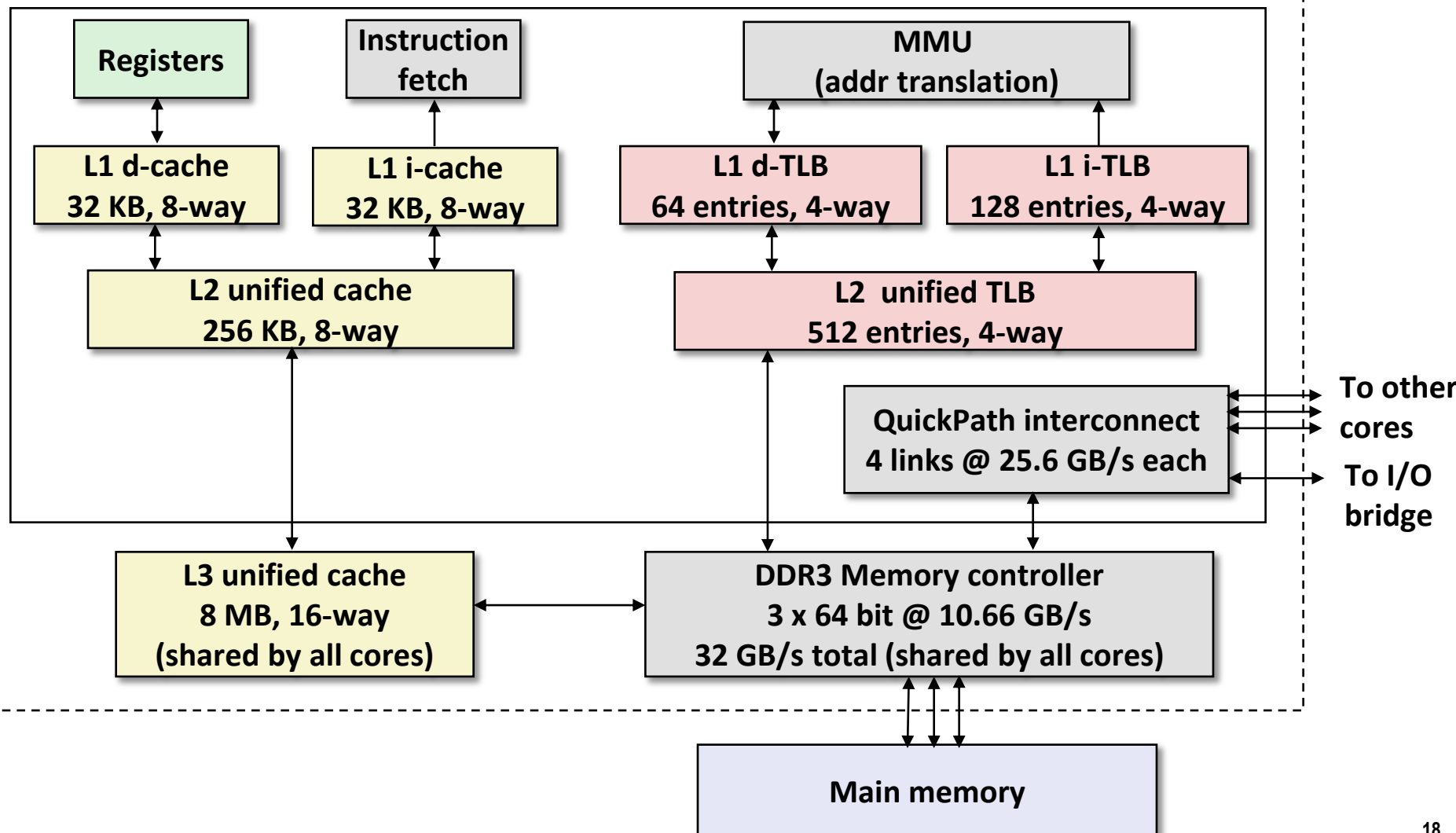
Inverted Page Table

- **We only have 4GB physical memory**
 - 2^{32} bytes memory / 2^{12} bytes per page = 2^{20} physical pages
- **If we only store a single page table entry per physical page**
- **(table is indexed by PPN)**
 - Each page table entry has process ID, VPN, and access info
 - 16 bit process ID, 52bit VPN, 12 bit access info = 80 bits = 10 bytes
 - 2^{20} PTE * 10 bytes = 10MB
- **For each (VPN, pid) pair, we need to find where it is in the page table by comparing it with all entries**
 - Linear time, up to 2^{20} memory accesses.....
- **Hashing to the rescue -> Hashed Inverted Page Tables**

Intel Core i7 Memory System

Processor package

Core x4



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)

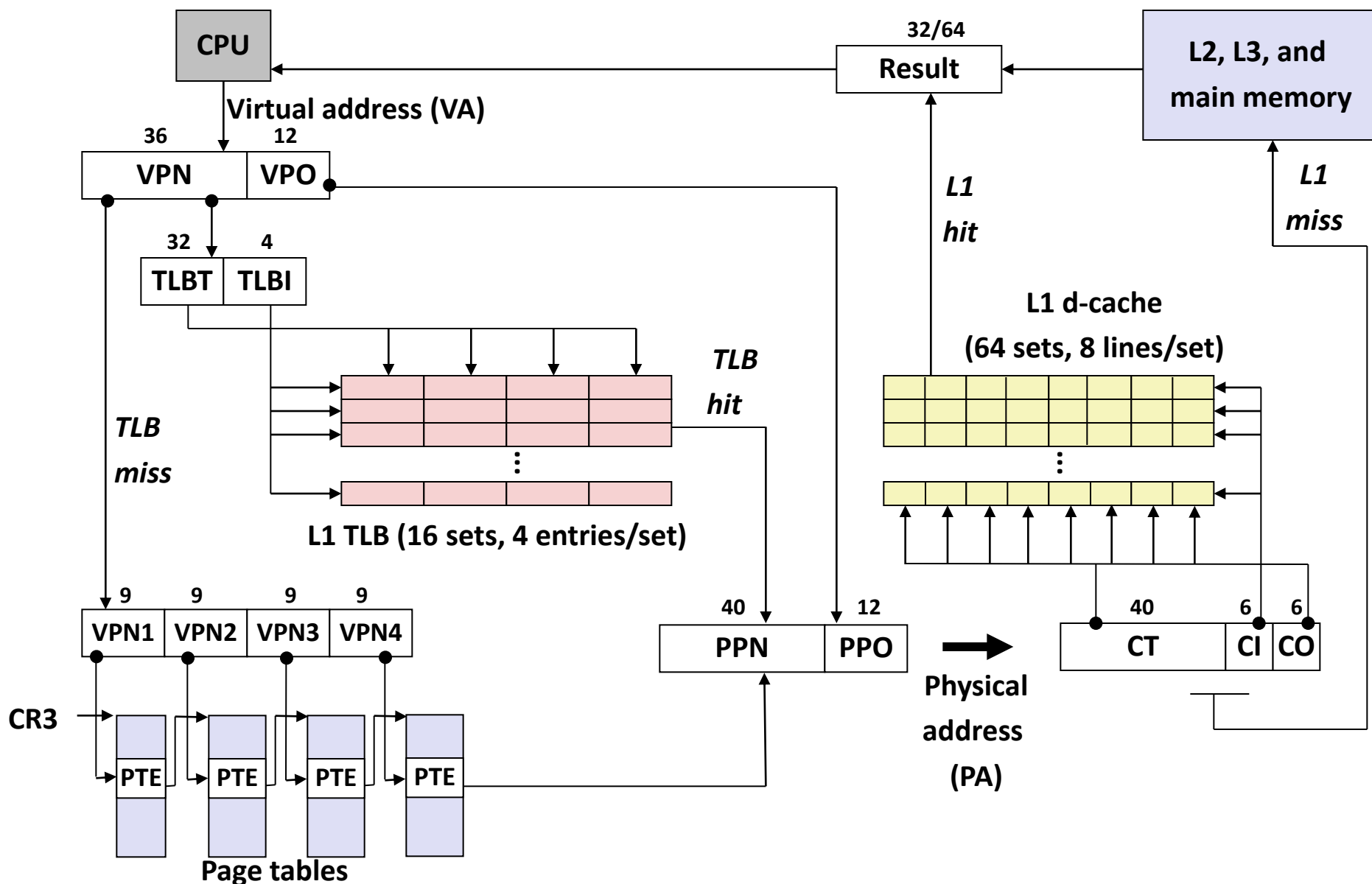
■ Components of the virtual address (VA)

- 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

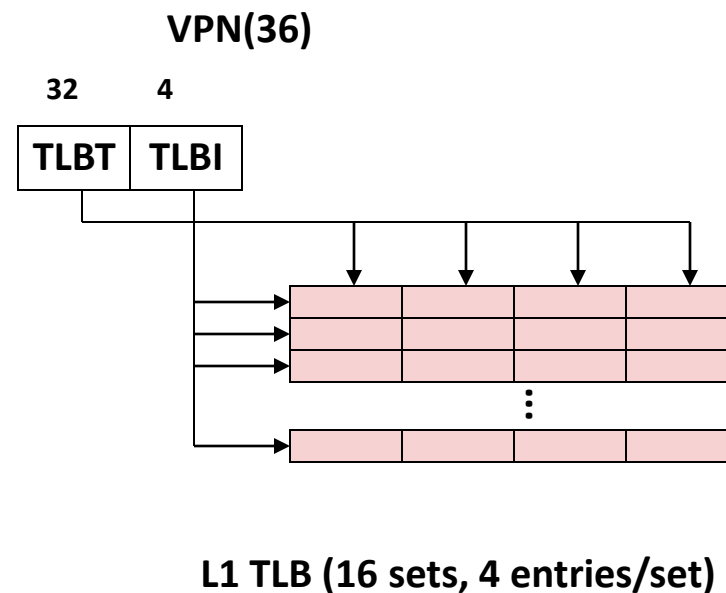
End-to-end Core i7 Address Translation



Core i7 L1 TLB

■ 16 sets, 4 entries/set

- 16 sets -> TLBI : 4 bits
- 36 bit VPN = 32 bit TLBT + 4 bit TLBI
- Each TLBT entry has $4 / 2^{32}$ chance of hit !



Core i7 L2 unified TLB

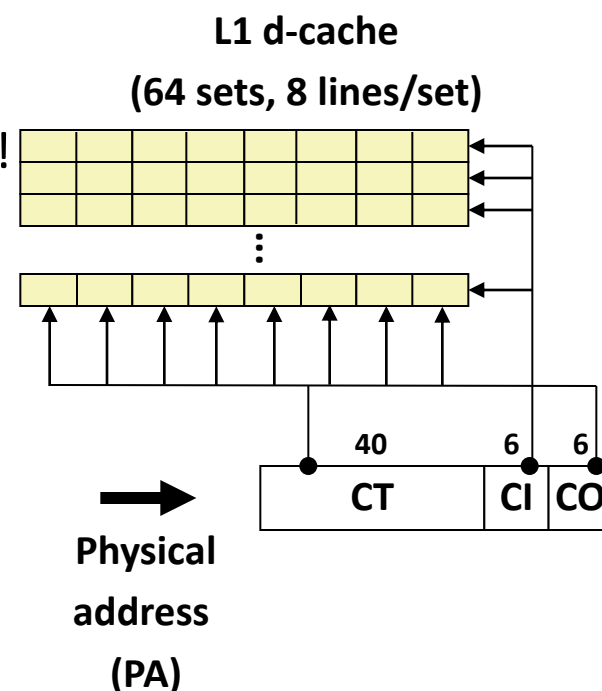
L2 unified TLB
512 entries, 4-way

- 512 entries, 4 way -> 128 sets
- 128 sets -> TLBI : 7 bits
- 36 bit VPN = 29 bit TLBT + 7 bit TLBI
- Each TLBT entry has $4 / 2^{29}$ chance of hit !

Core i7 L1 d-cache

■ 64 sets, 8 entries/set

- Each cache line has 64 bytes \rightarrow CO : 6 bits
- 64 sets \rightarrow CI : 6 bits
- 52 bit PA = 40 bit CT + 6 bit CO + 6 bit CO
- Each Cache entry has $8 / 2^{40}$ chance of hit !



Core i7 L2/L3 unified cache

L2 unified cache
256 KB, 8-way

- $256 \text{ KB} = 8 \times 32\text{KB} = 8 \times 512 \times 64 \text{ byte}$
- 512 sets \rightarrow CI : 9 bits
- 52 bit PA = 37 bit CT + 9 bit CI + 6 bit CO
- Each Cache entry has $8 / 2^{37}$ chance of hit !

L3 unified cache
8 MB, 16-way
(shared by all cores)

- $8 \text{ MB} = 16 \times 512 \text{ KB} = 16 \times 8192 \times 64 \text{ byte}$
- 8192 sets \rightarrow CI : 13 bits
- 52 bit PA = 34 bit CT + 13 bit CI + 6 bit CO
- Each Cache entry has $16 / 2^{34}$ chance of hit !

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

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.

CD: Caching disabled or enabled 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).

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

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

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

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

CD: Cache disabled (1) or enabled (0)

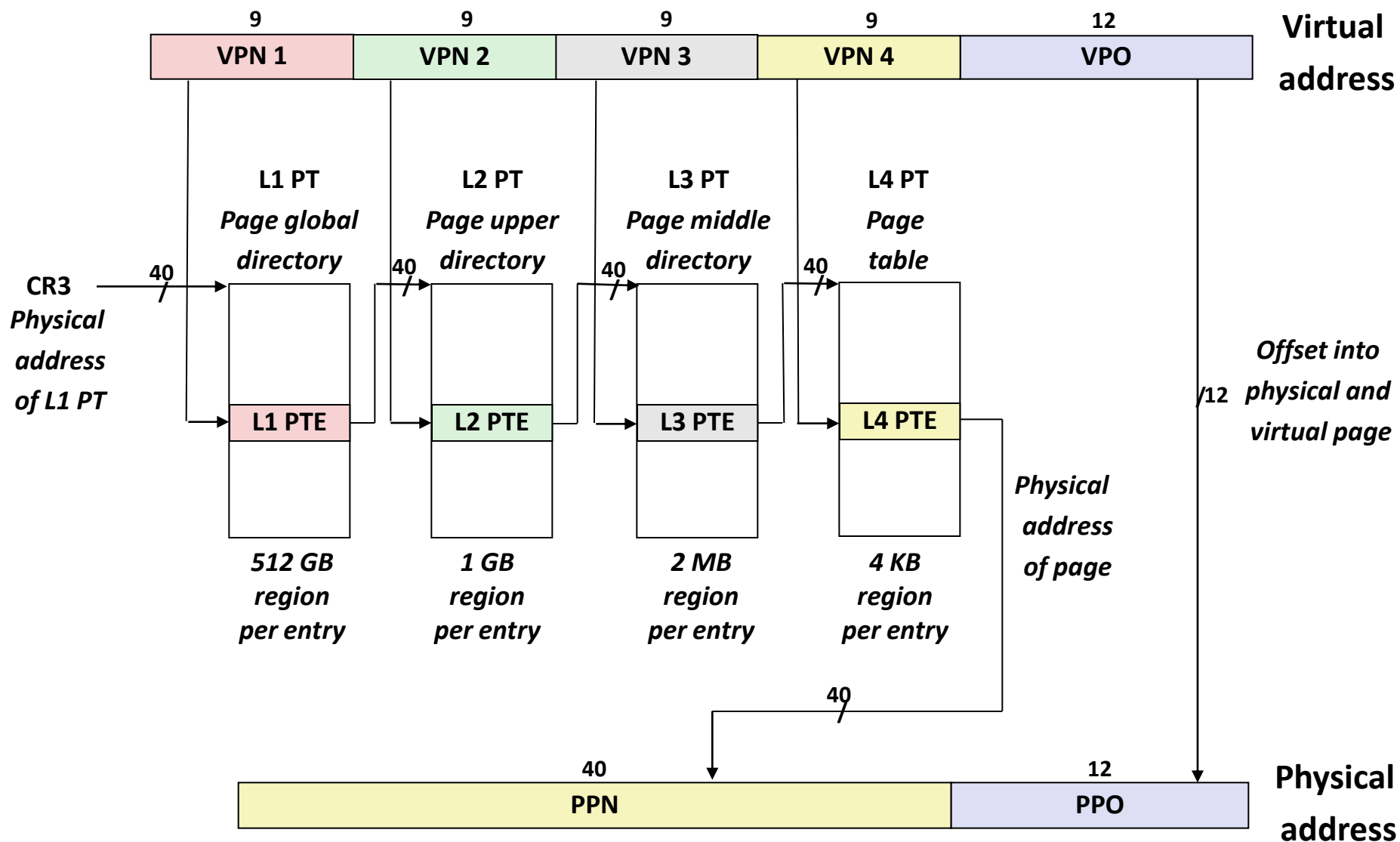
A: Reference bit (set by MMU on reads and writes, cleared by software)

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

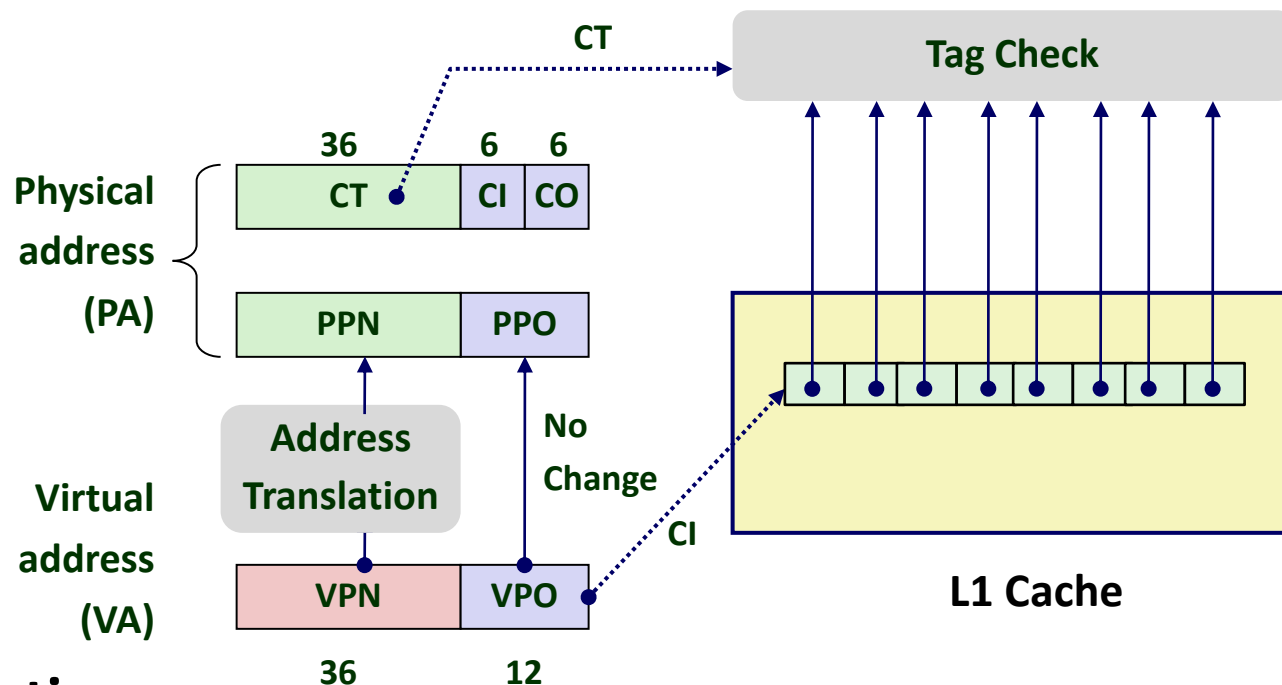
G: Global page (don't evict from TLB on task switch)

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

Core i7 Page Table Translation



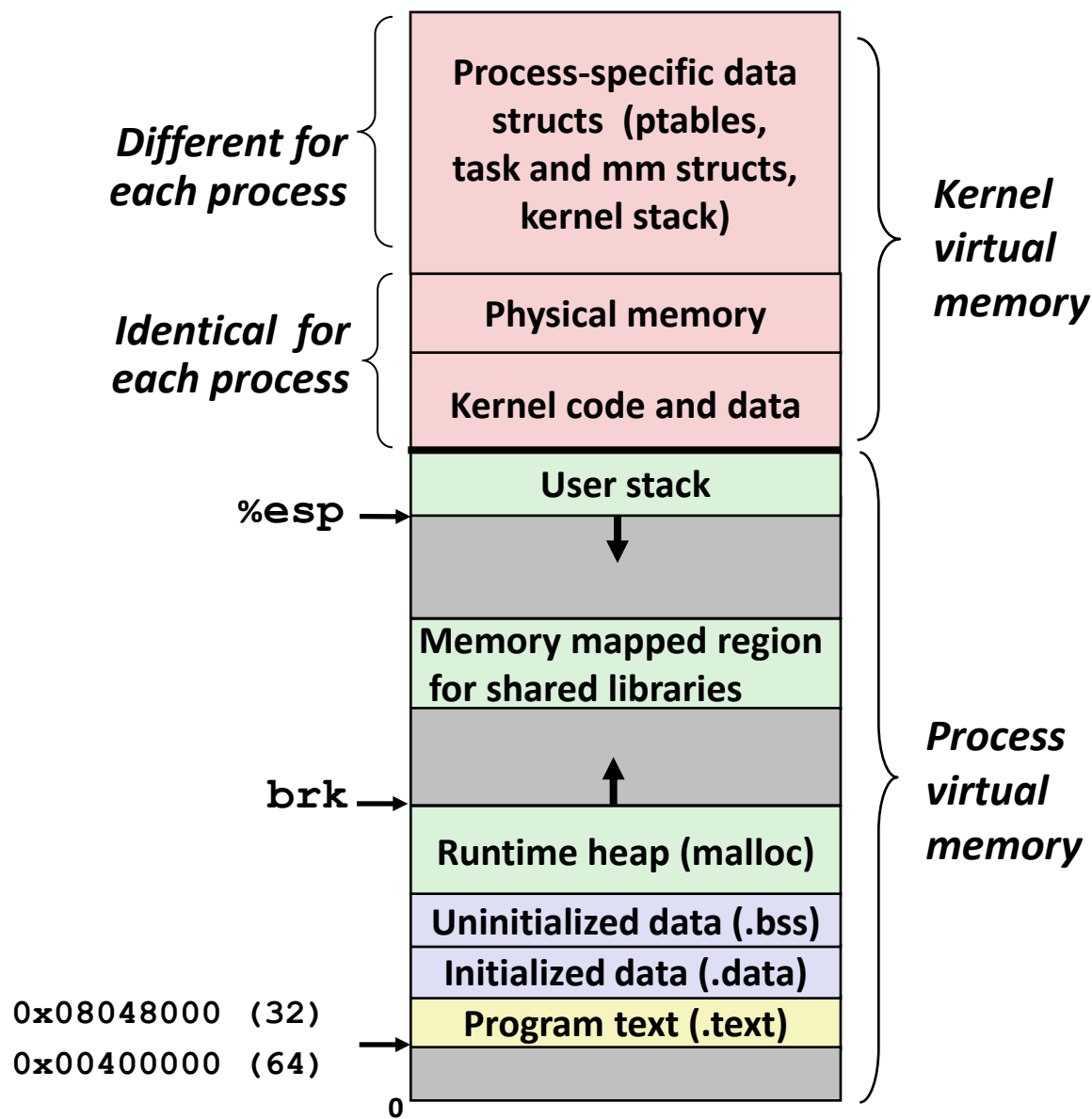
Cute Trick for Speeding Up L1 Access



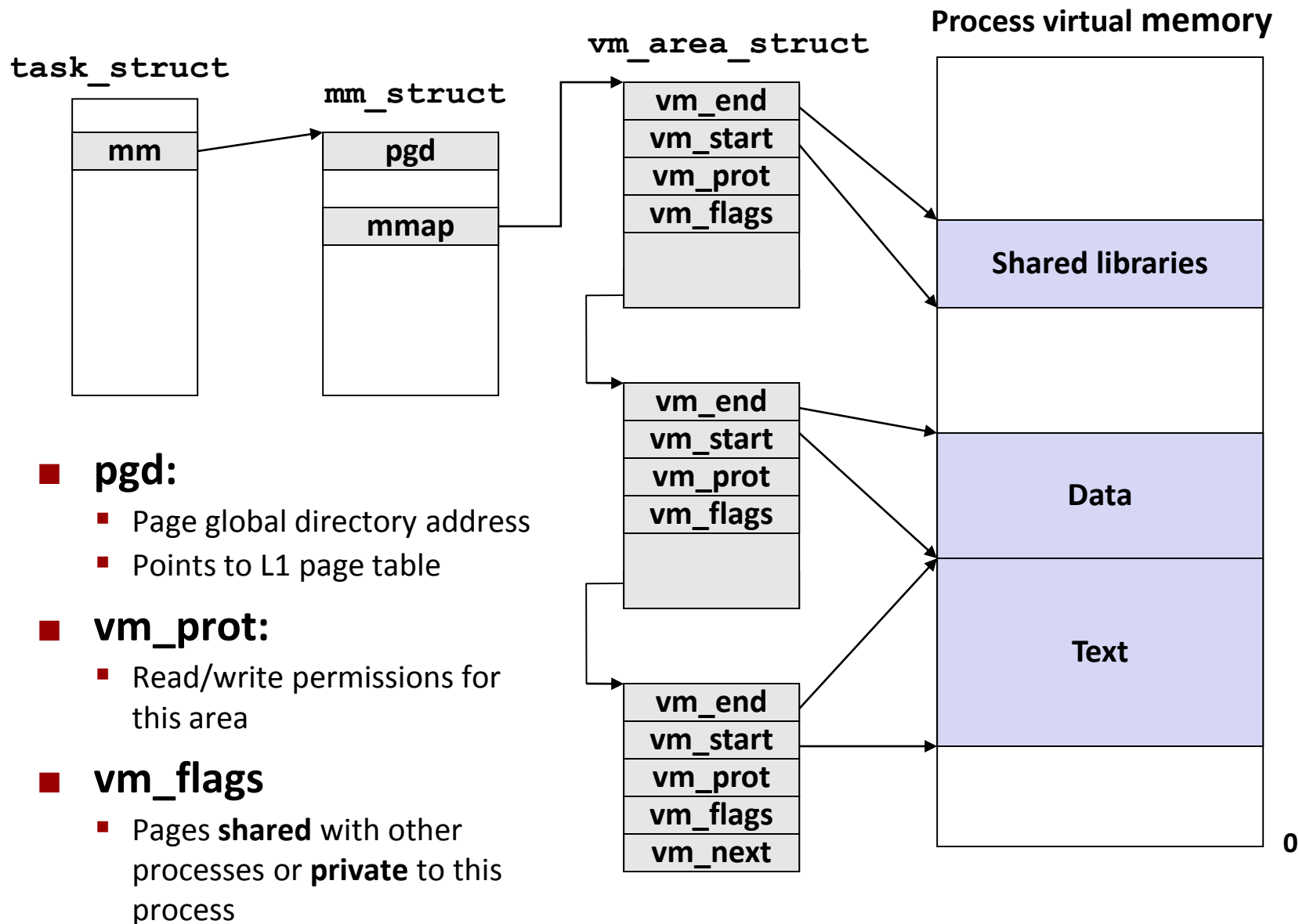
■ Observation

- 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

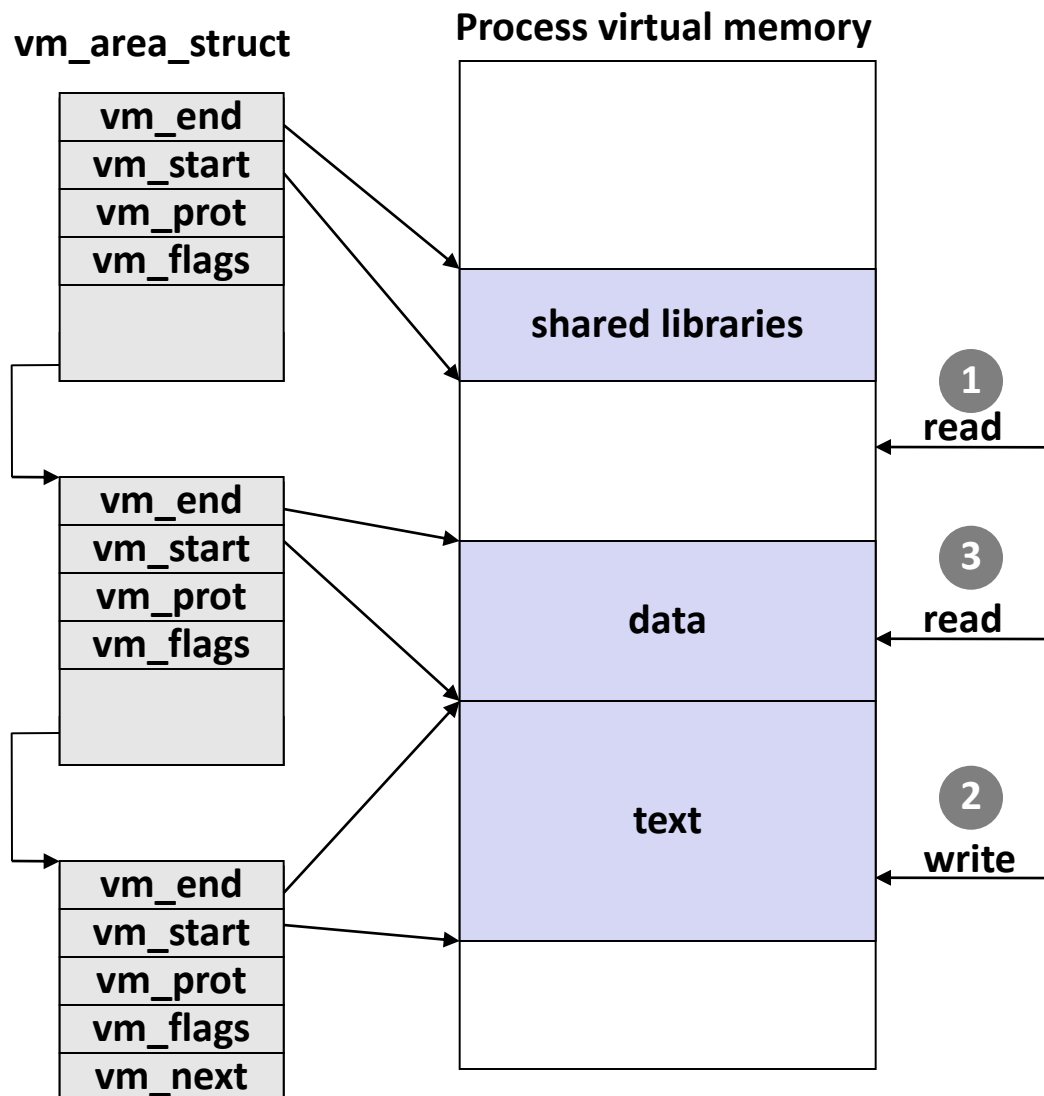
Virtual Memory of a Linux Process



Linux Organizes VM as Collection of “Areas”



Linux Page Fault Handling



Segmentation fault:
accessing a non-existing page

Normal page fault

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

Today

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

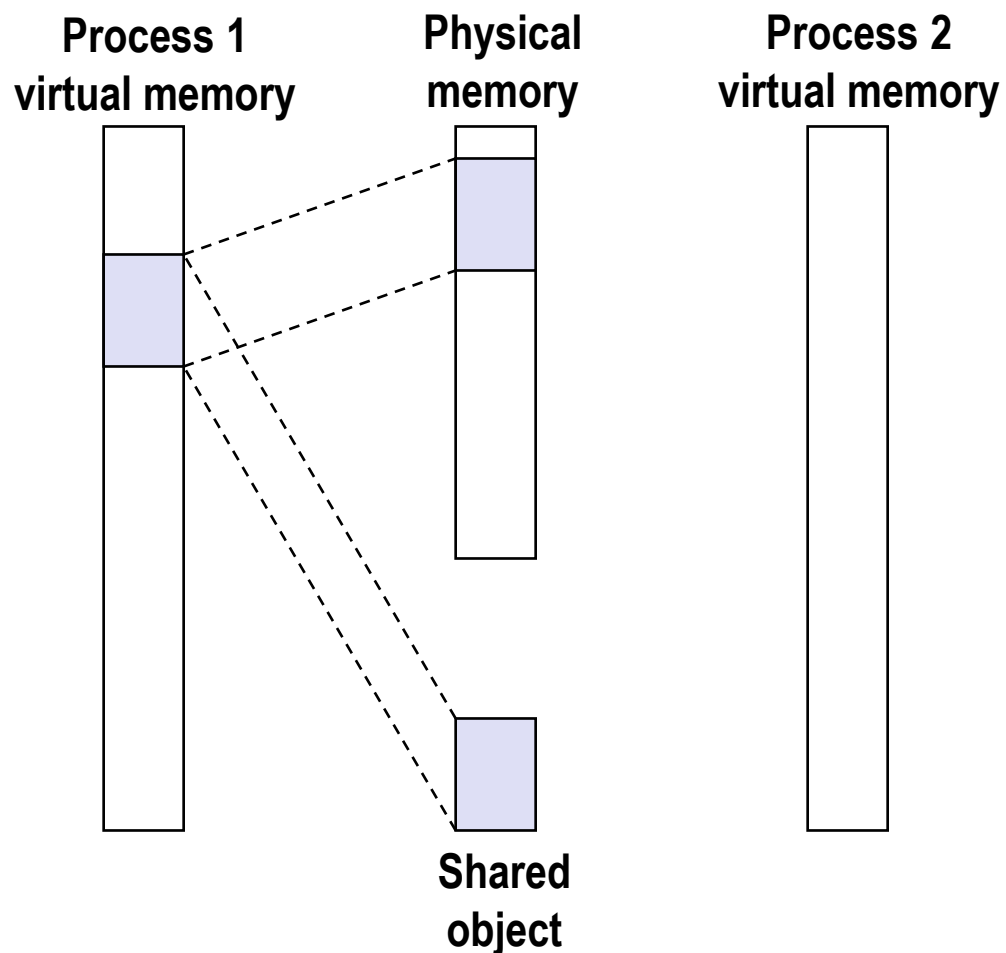
Memory Mapping

- VM areas initialized by associating them with disk objects.
 - 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*.

Demand paging

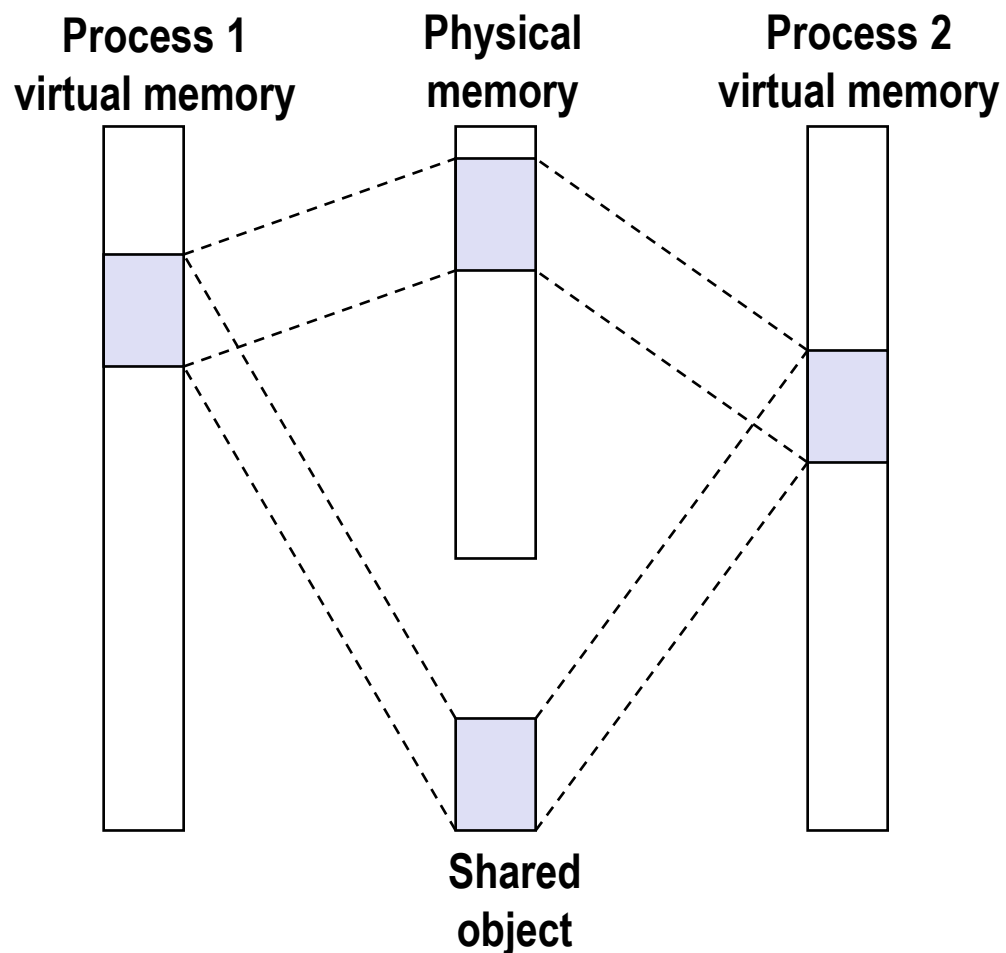
- ***Key point:*** no virtual pages are copied into physical memory until they are referenced!
 - Known as ***demand paging***
- **Crucial for time and space efficiency**

Sharing Revisited: Shared Objects



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

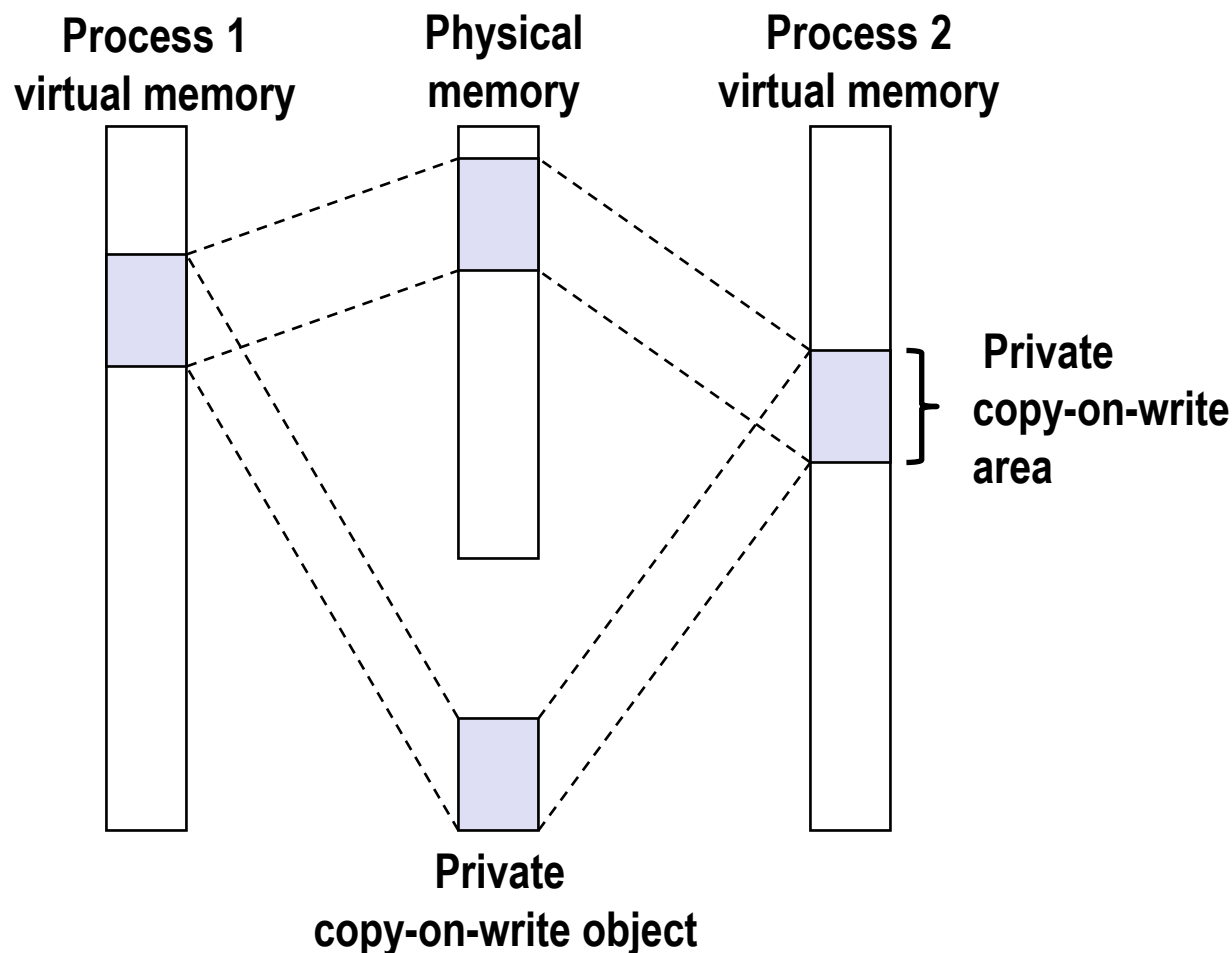
Sharing Revisited: Shared Objects



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

Sharing Revisited:

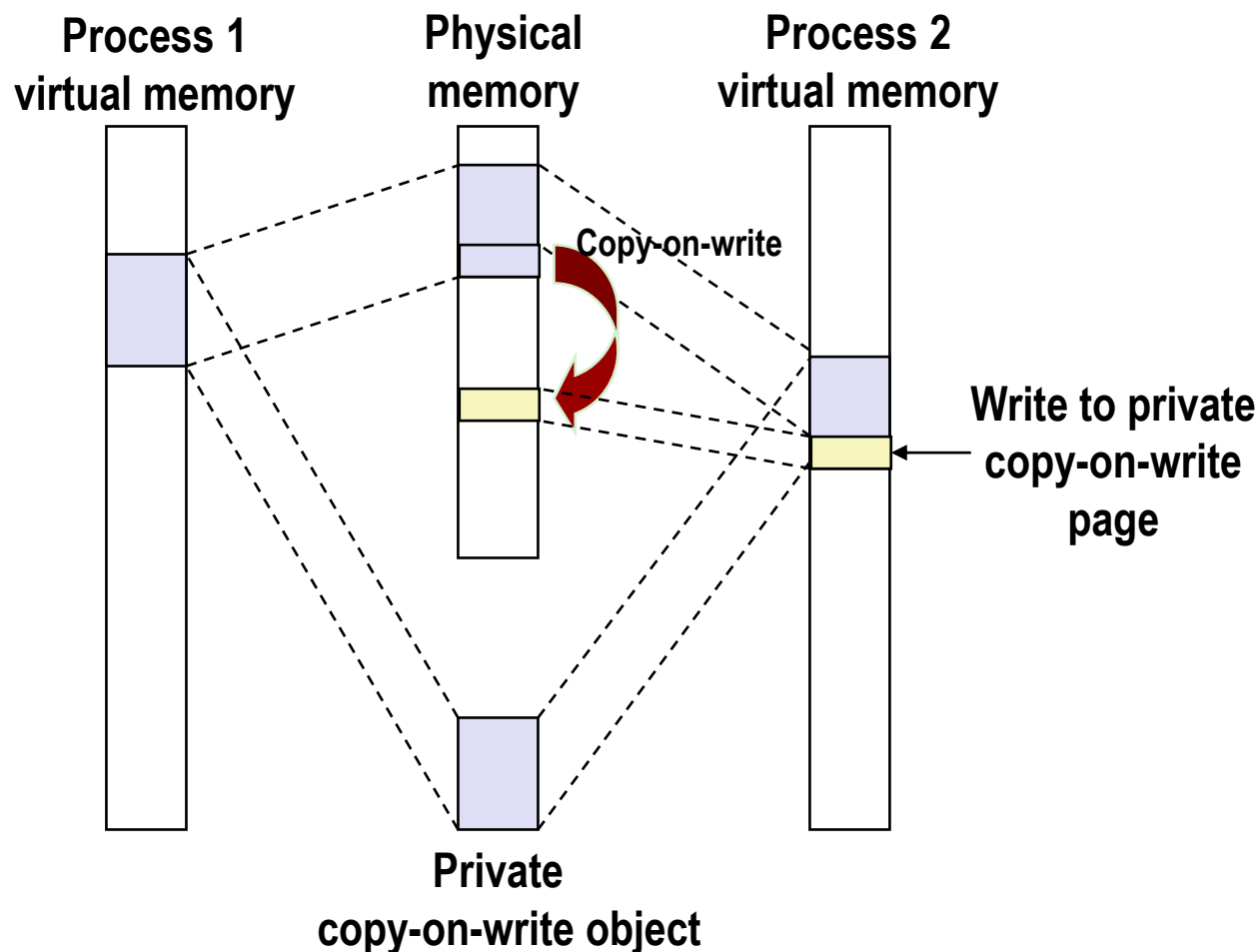
Private Copy-on-write (COW) Objects



- 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

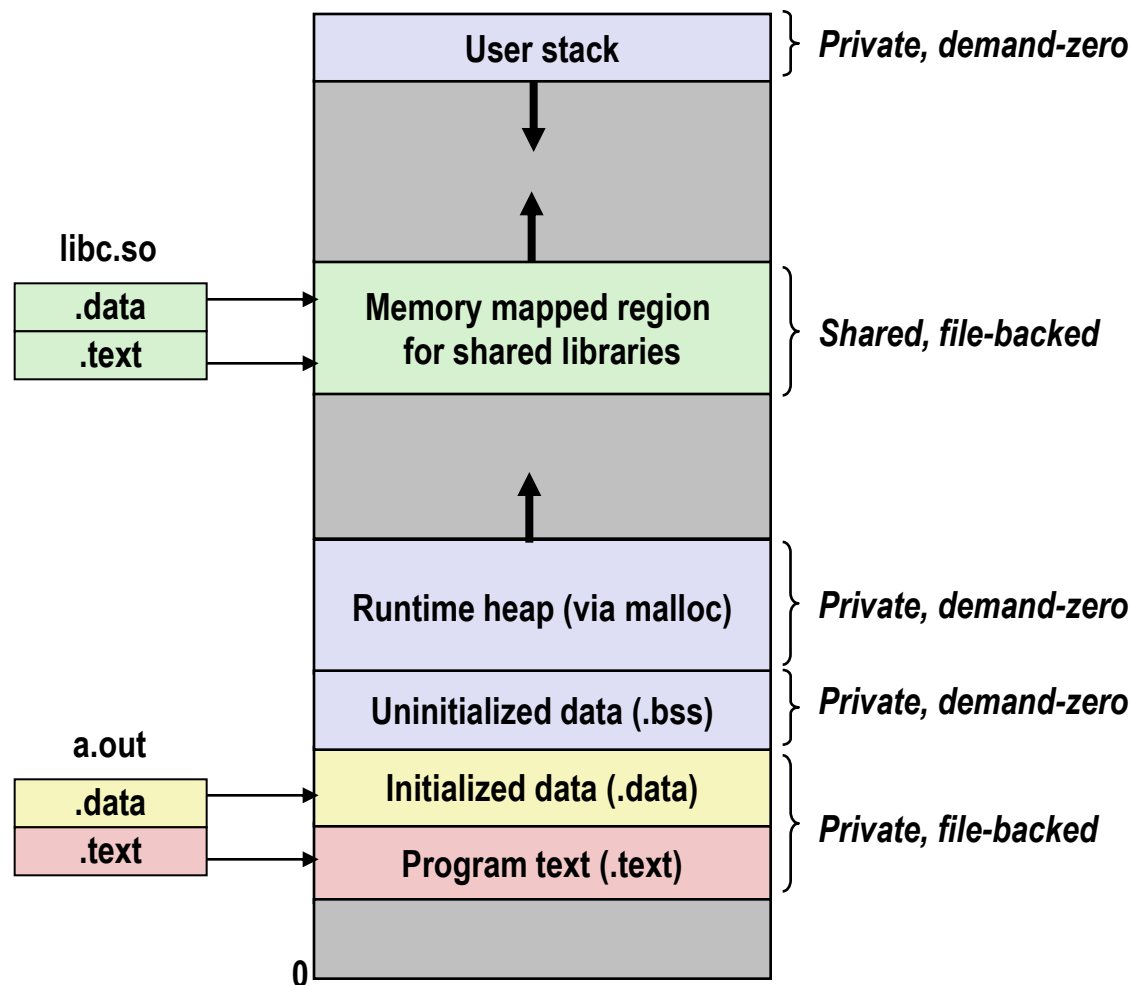


- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

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.

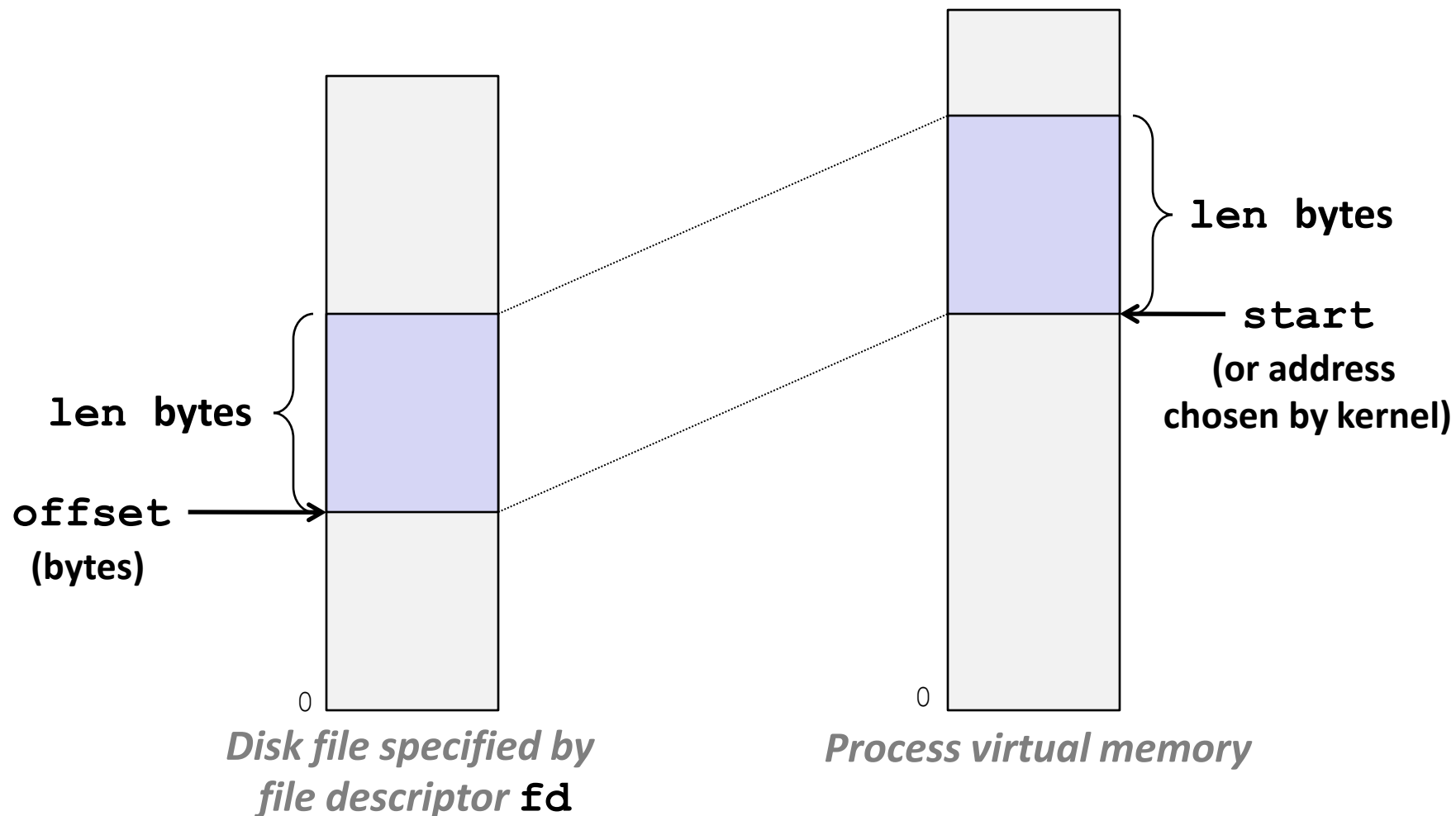
User-Level Memory Mapping

```
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)
```



Using mmap to Copy Files

■ Copying without transferring data to user space .

```
#include "csapp.h"

/*
 * mmapcopy - uses mmap to copy
 *            file fd to stdout
 */
void mmapcopy(int fd, int size)
{
    /* Ptr to mem-mapped VM area */
    char *bufp;

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

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

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

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