

Virtual Memory: Systems

Introduction to Computer Systems
20th Lecture, Nov. 27, 2019

Instructors:

Class 1: Chen Xiangqun, Sun Guangyu

Class 2: Guan Xuetao

Class 3: Lu Junlin

Today

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

Virtual memory reminder/review

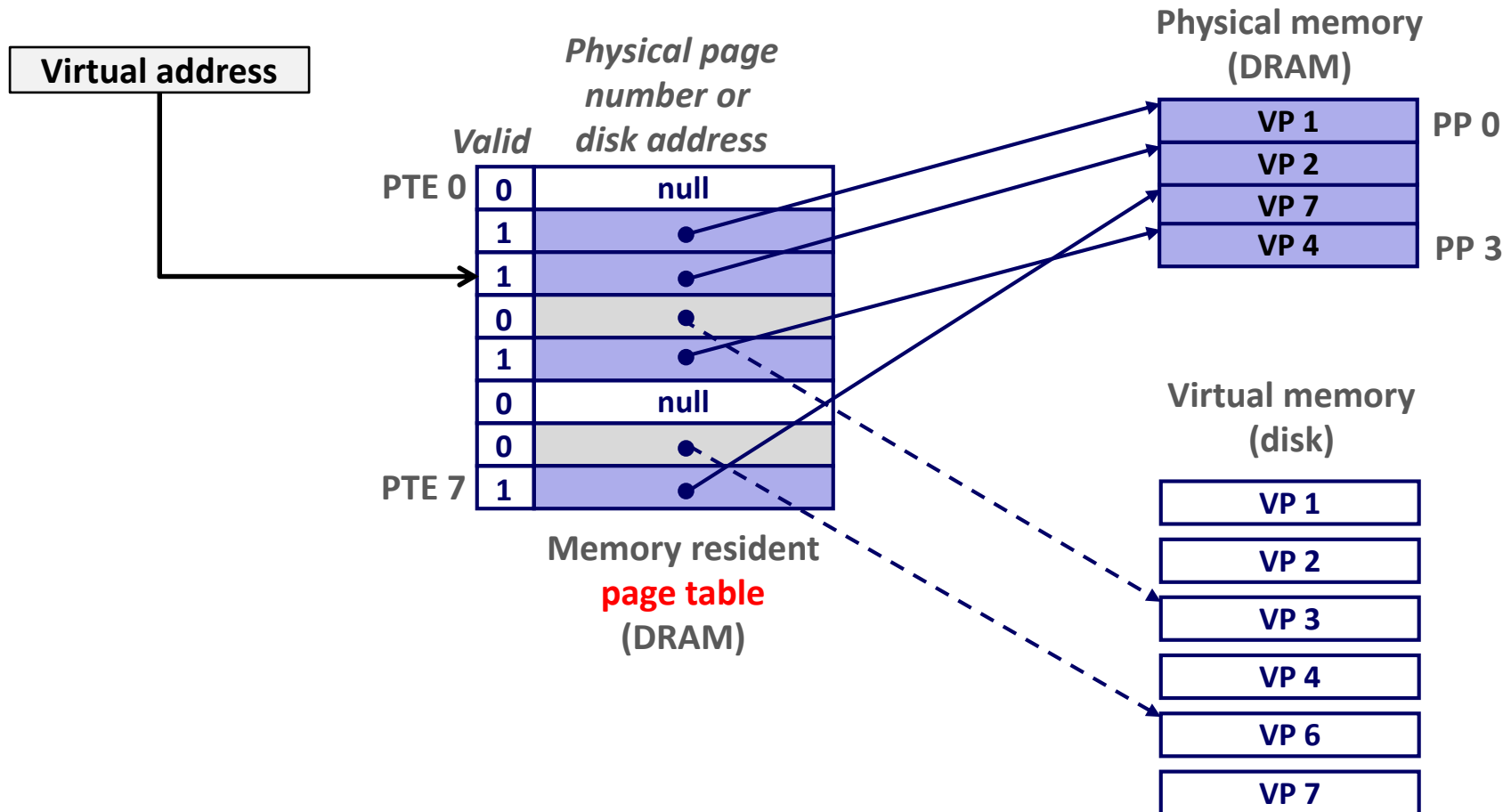
■ Programmer's view of virtual memory

- Each process has its own private linear address space
- Cannot be corrupted by other processes

■ System view of virtual memory

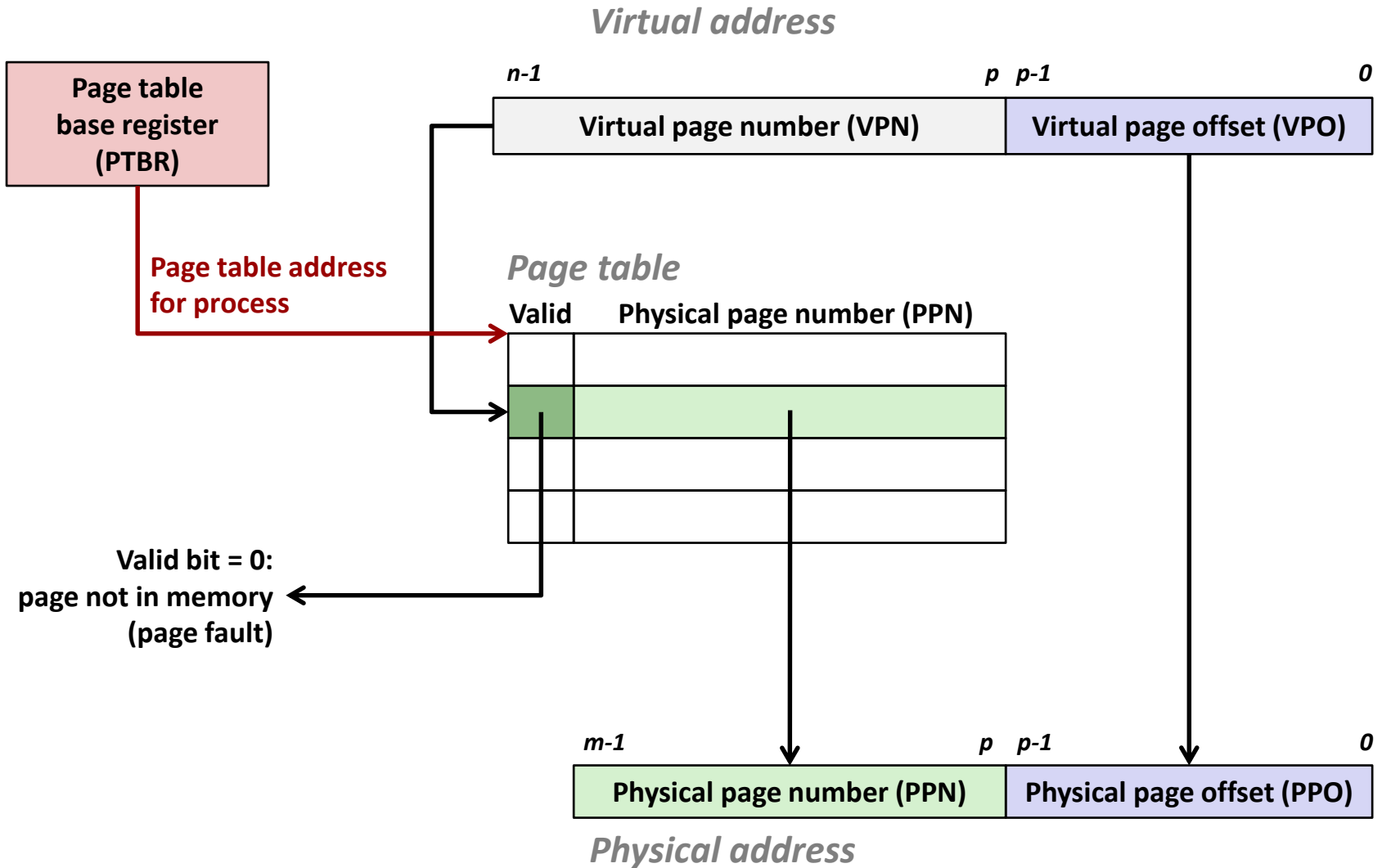
- Uses memory efficiently by caching virtual memory pages
 - Efficient only because of locality
- Simplifies memory management and programming
- Simplifies protection by providing a convenient interpositioning point to check permissions

Recall: Virtual Memory & Physical Memory

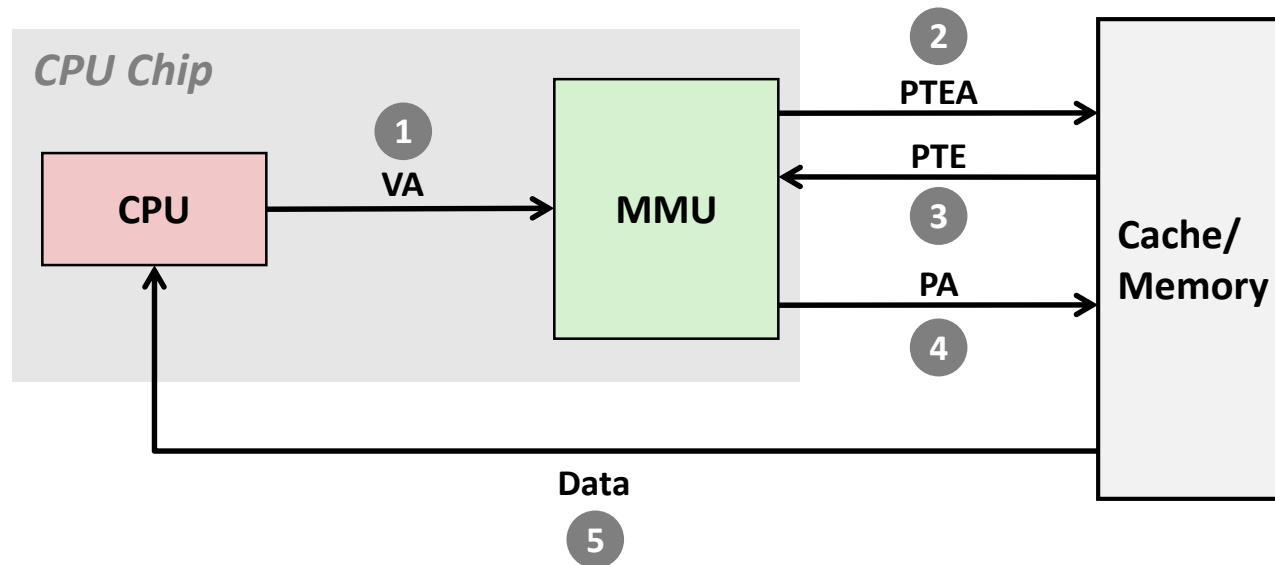


- A **page table** contains page table entries (PTEs) that map virtual pages to physical pages.

Recall: Address Translation With a Page Table



Recall: Address Translation: Page Hit

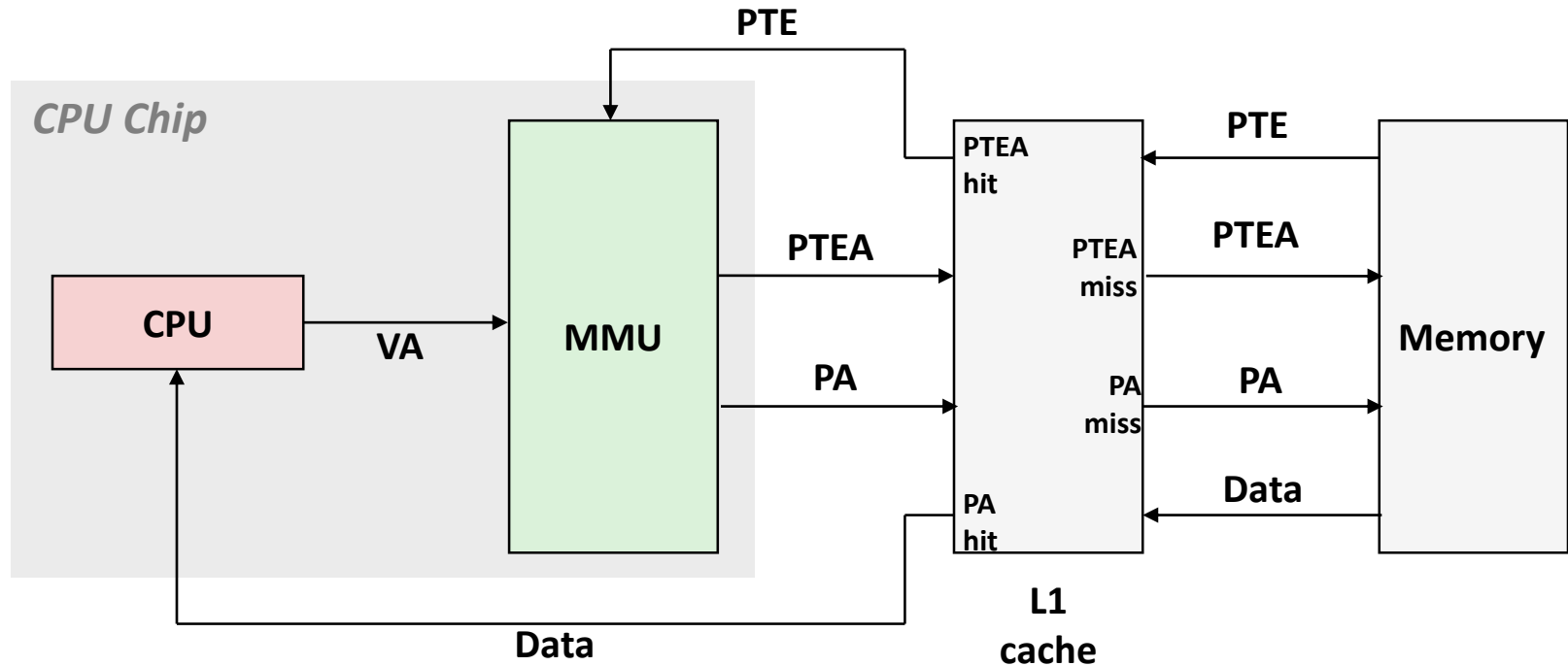


- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

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

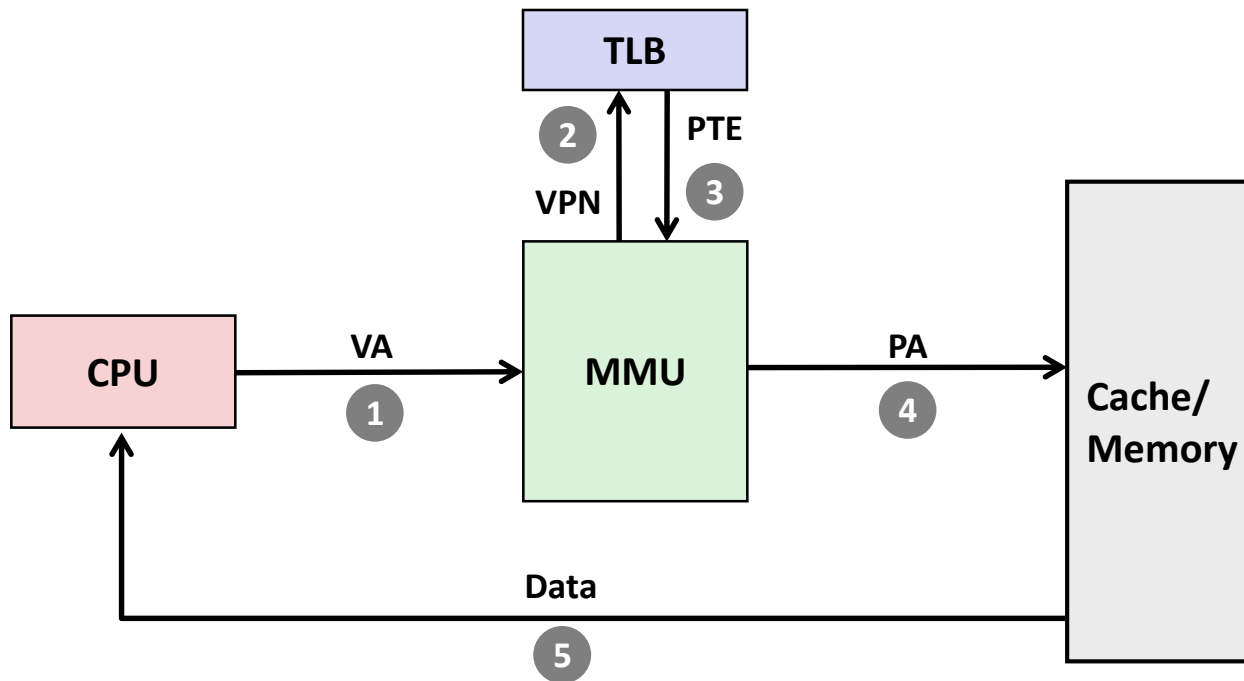
- Isn't it slow to have to go to memory twice every time?
- Yes, it would be... so, real MMUs don't

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

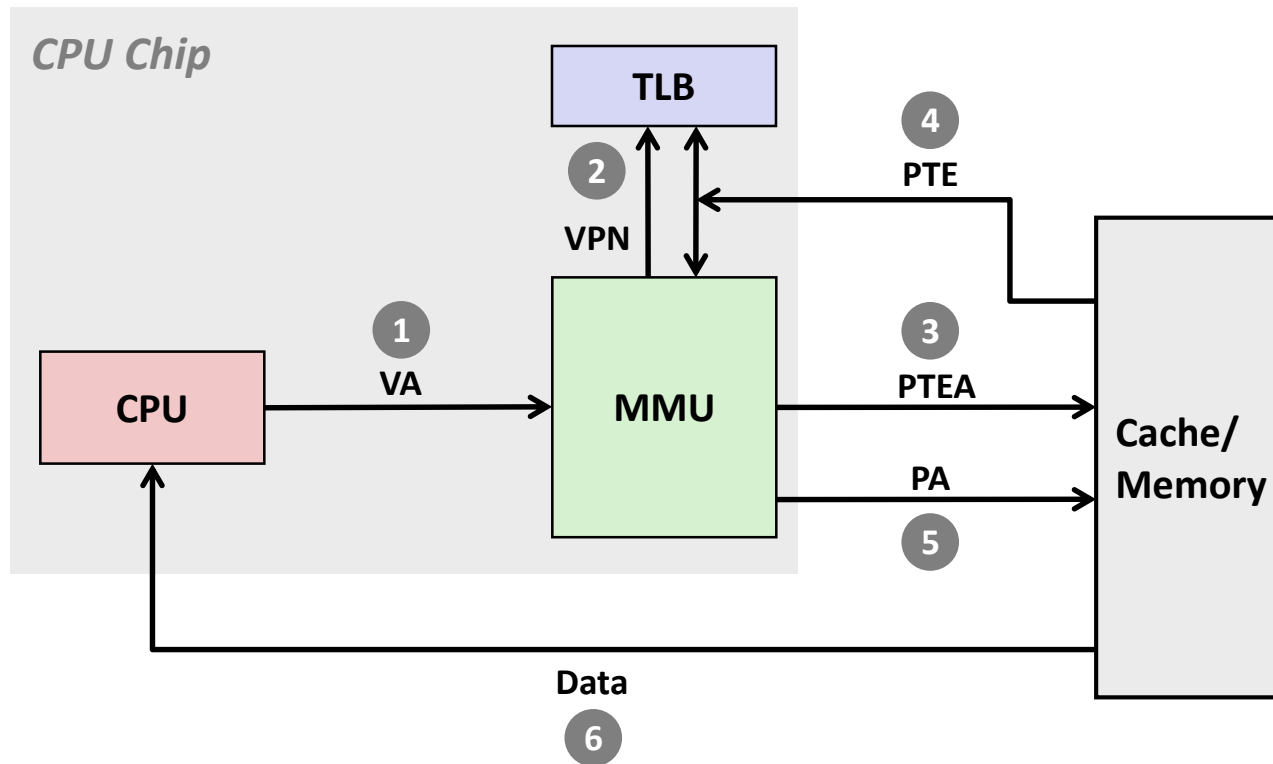
Translation Lookaside Buffer (TLB)

- 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.

TLB Miss



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

Fortunately, TLB misses are rare. Why?

Question #3

- 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

■ Suppose:

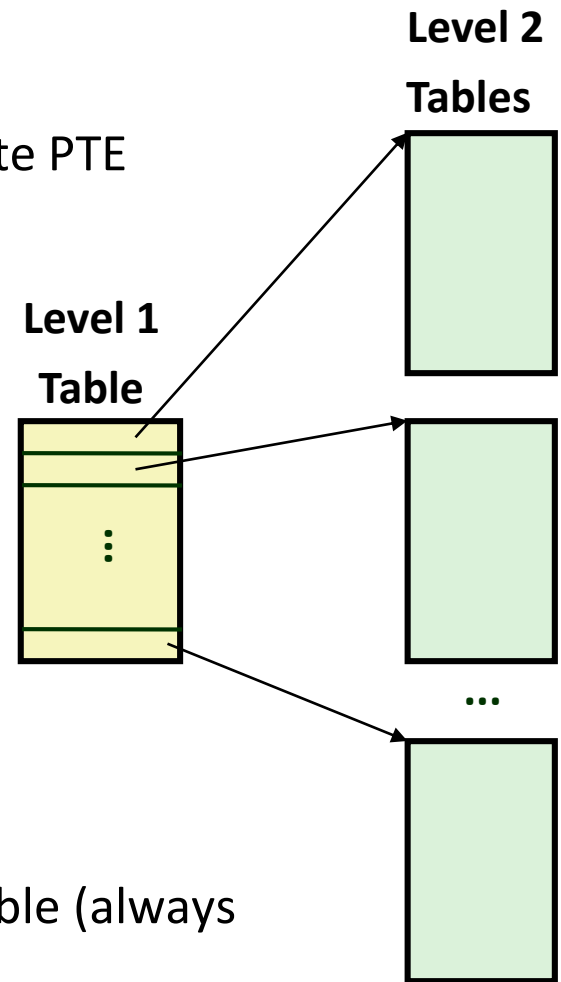
- 4KB (2^{12}) page size, 64-bit address space, 8-byte PTE

■ Problem:

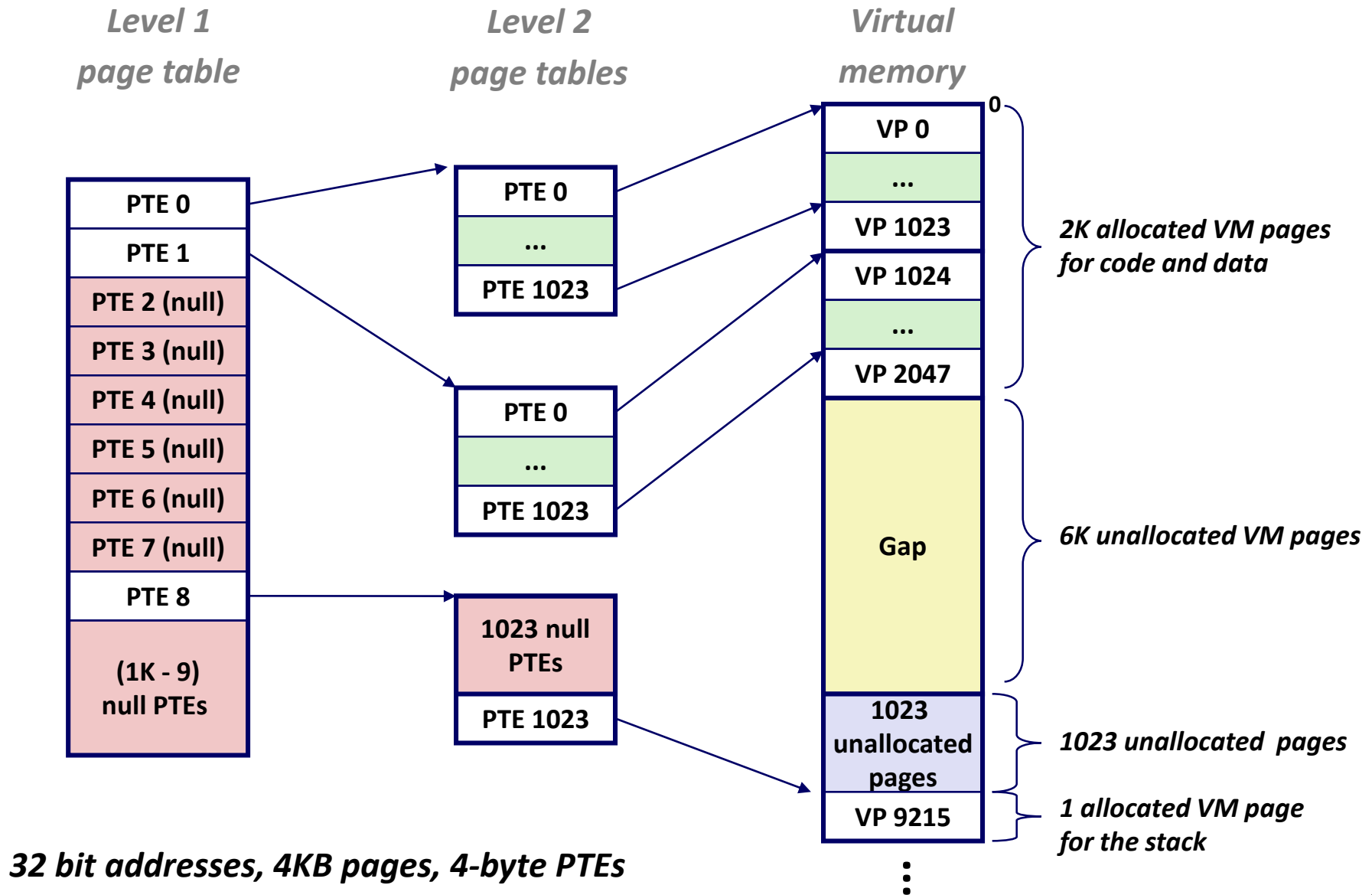
- Would need a 32,000 TB page table!
 - $2^{64} * 2^{-12} * 2^3 = 2^{55}$ bytes

■ 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)

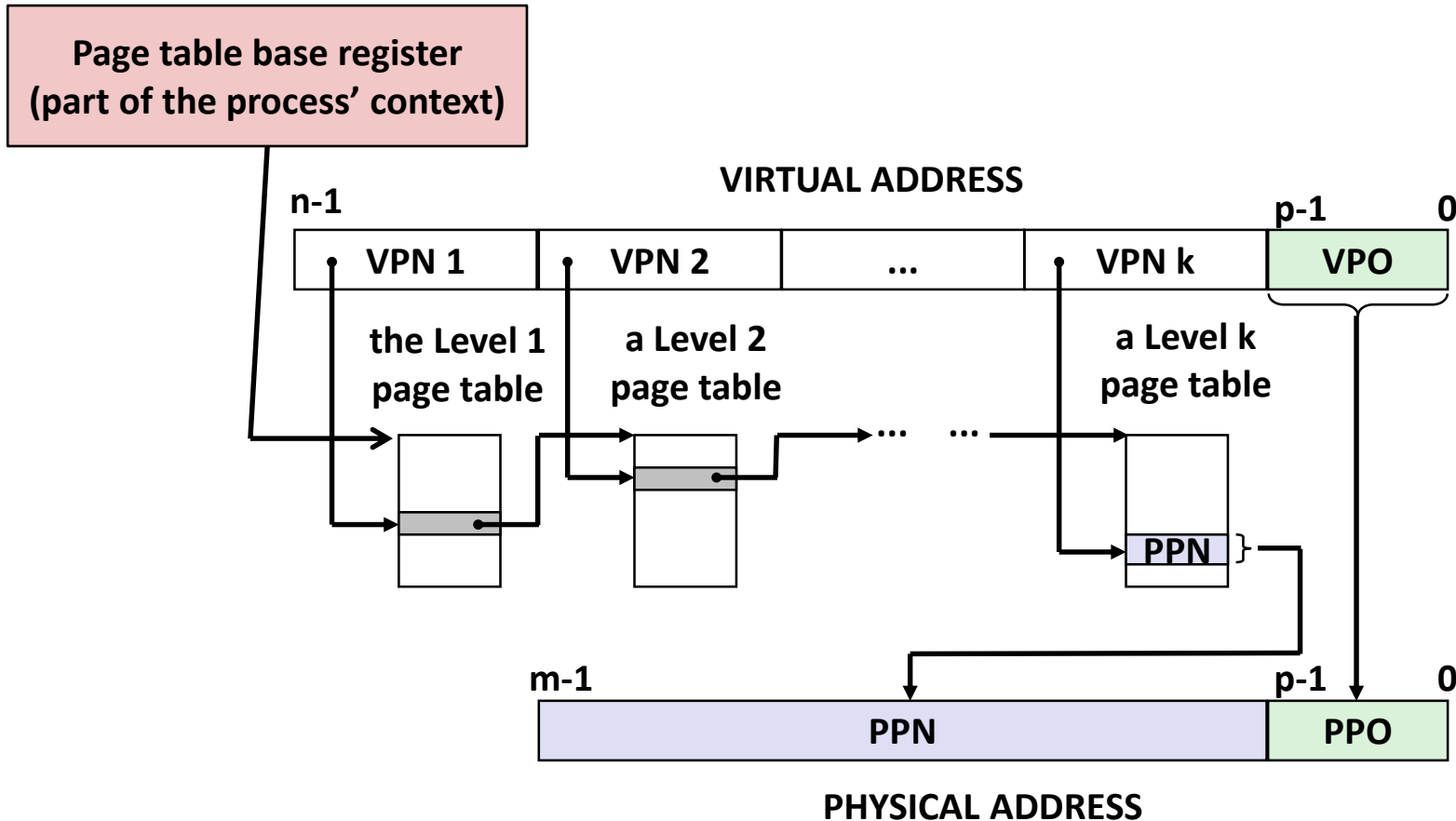


A Two-Level Page Table Hierarchy



Translating with a k-level Page Table

- Having multiple levels greatly reduces page table size



Question #4

- 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

Today

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

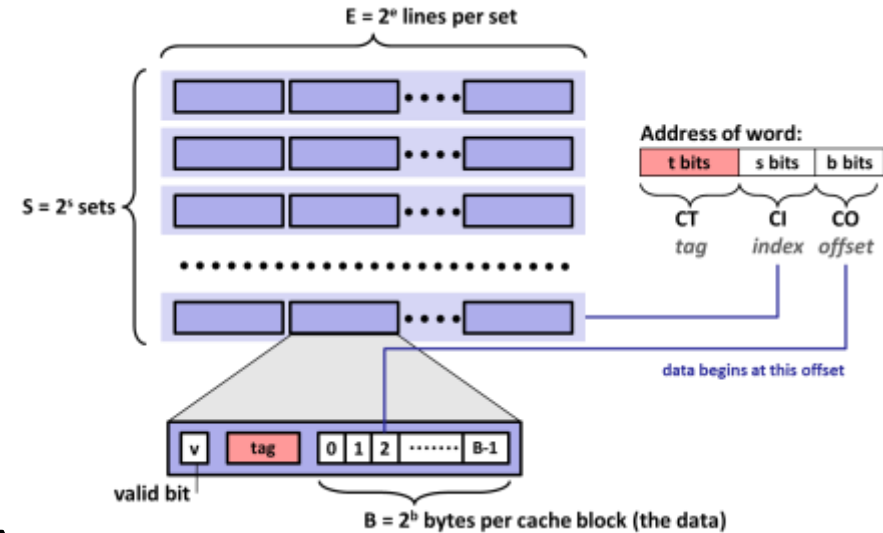
Review of Symbols

■ 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)

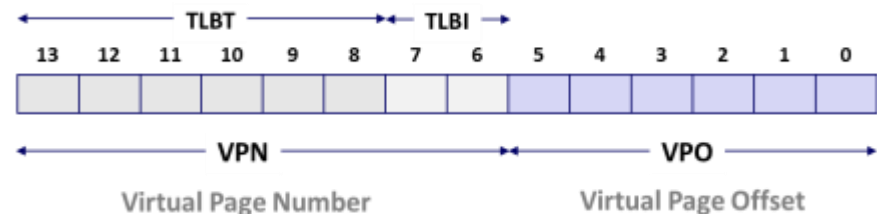
■ Components of the *virtual address* (VA)

- TLBI: TLB index
- TLBT: TLB tag
- VPO: Virtual page offset
- VPN: Virtual page number

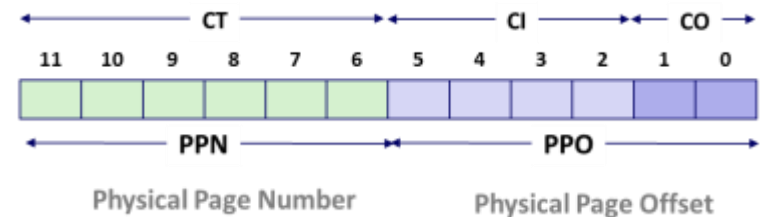


■ Components of the *physical address* (PA)

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



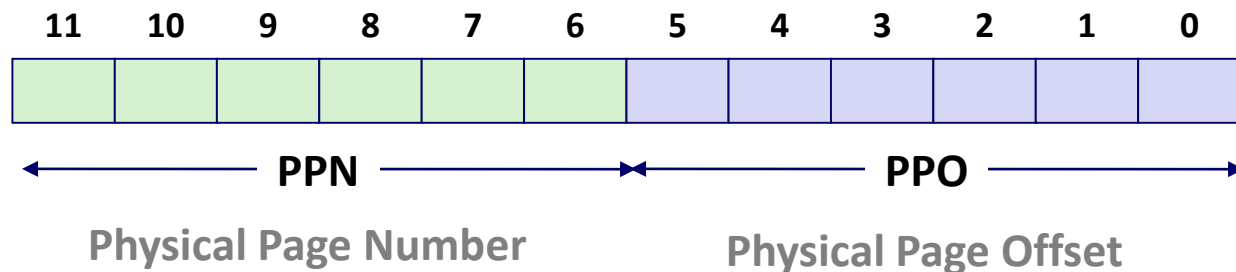
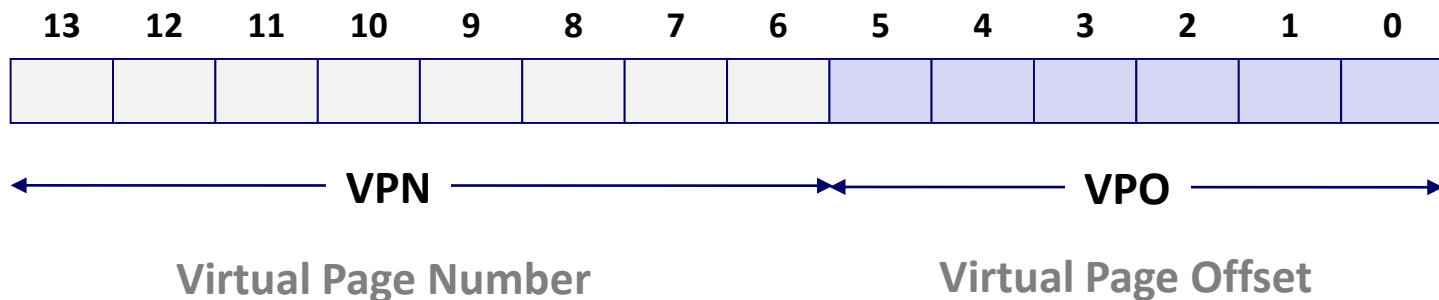
(bits per field for our simple example)



Simple Memory System Example

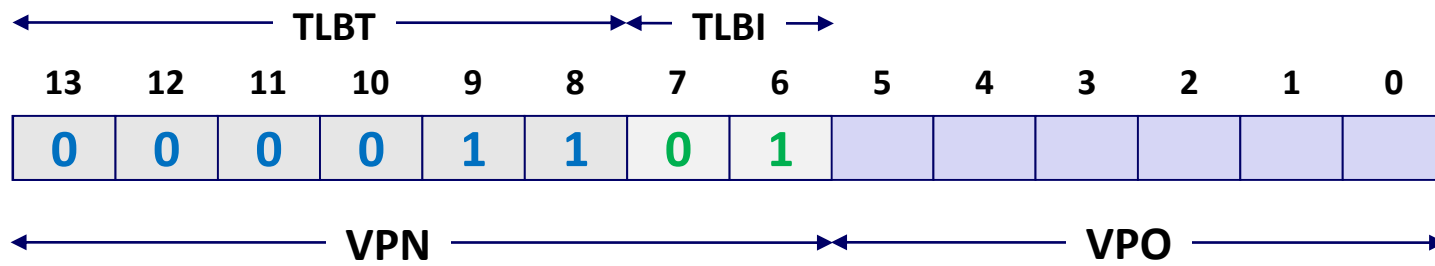
■ Addressing

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



Simple Memory System TLB

- 16 entries
- 4-way associative



$$\text{VPN} = 0b1101 = 0x0D$$

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

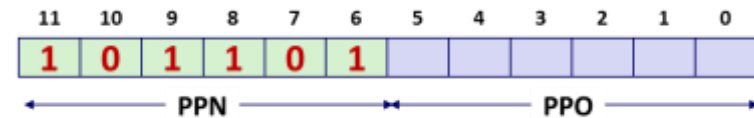
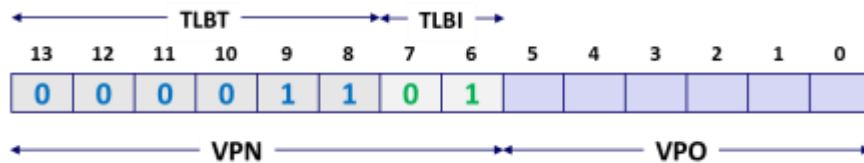
Simple Memory System Page Table

Only showing the first 16 entries (out of 256)

<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
00	28	1
01	—	0
02	33	1
03	02	1
04	—	0
05	16	1
06	—	0
07	—	0

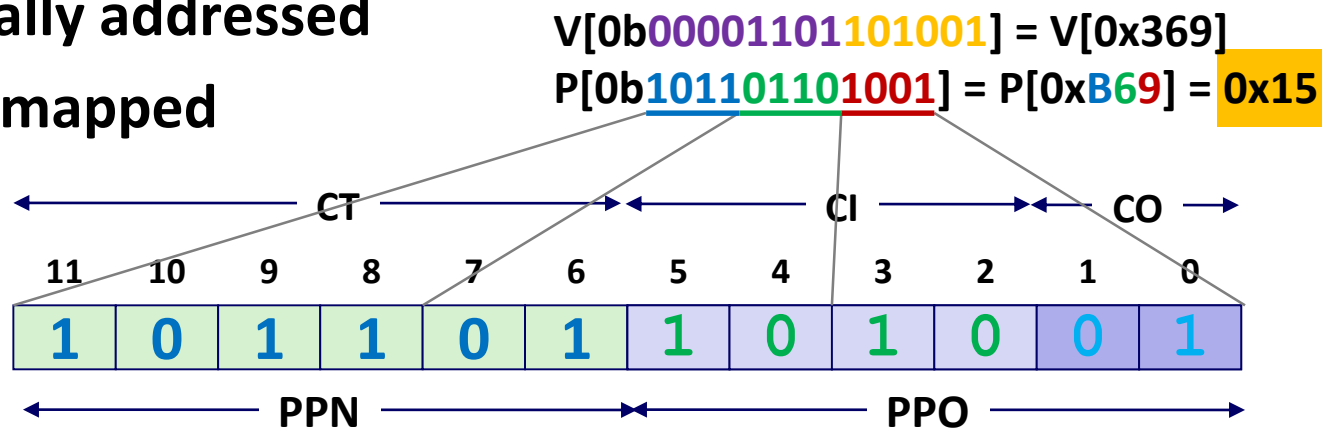
<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
08	13	1
09	17	1
0A	09	1
0B	—	0
0C	—	0
0D	2D	1
0E	11	1
0F	0D	1

0x0D → 0x2D



Simple Memory System Cache

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

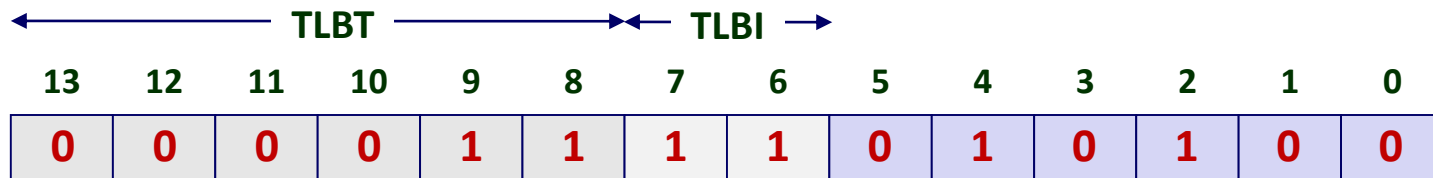


Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

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: TLB/Cache Hit

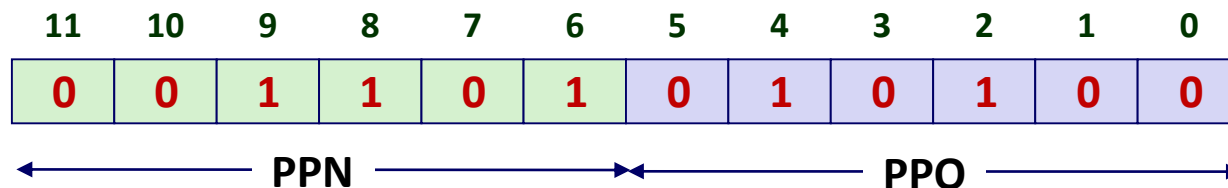
Virtual Address: 0x03D4



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

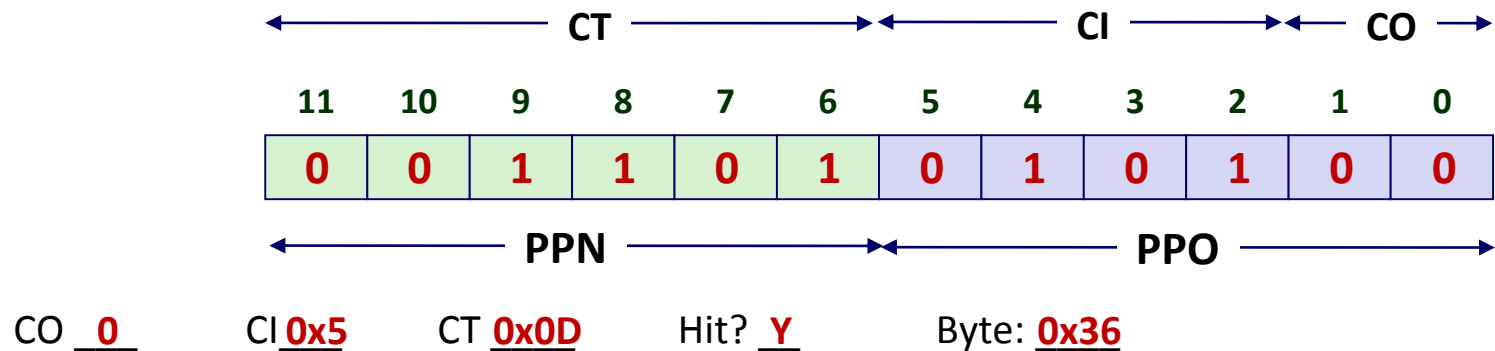
TLB	Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
	0	03	–	0	09	0D	1	00	–	0	07	02	1
	1	03	2D	1	02	–	0	04	–	0	0A	–	0
	2	02	–	0	08	–	0	06	–	0	03	–	0
	3	07	–	0	03	0D	1	0A	34	1	02	–	0

Physical Address



Address Translation Example: TLB/Cache Hit

Physical Address



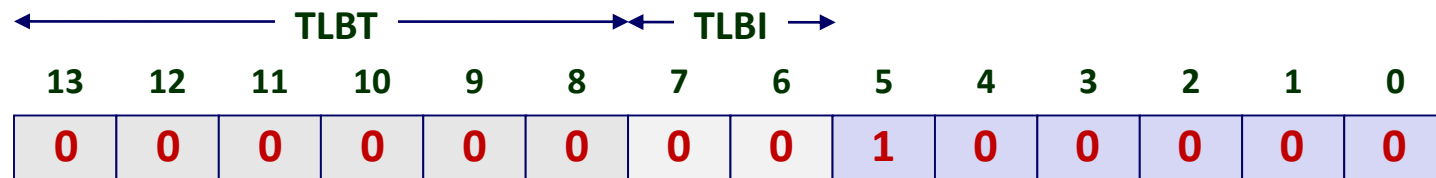
Cache

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	–	–	–	–

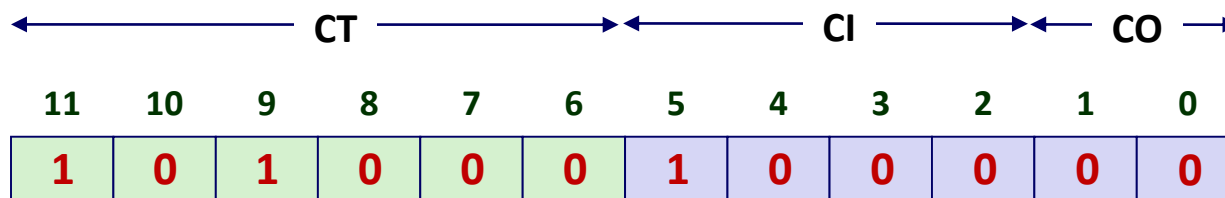
Address Translation Example: TLB/Cache Miss

Virtual Address: 0x0020



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

Physical Address



PPN PPO

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

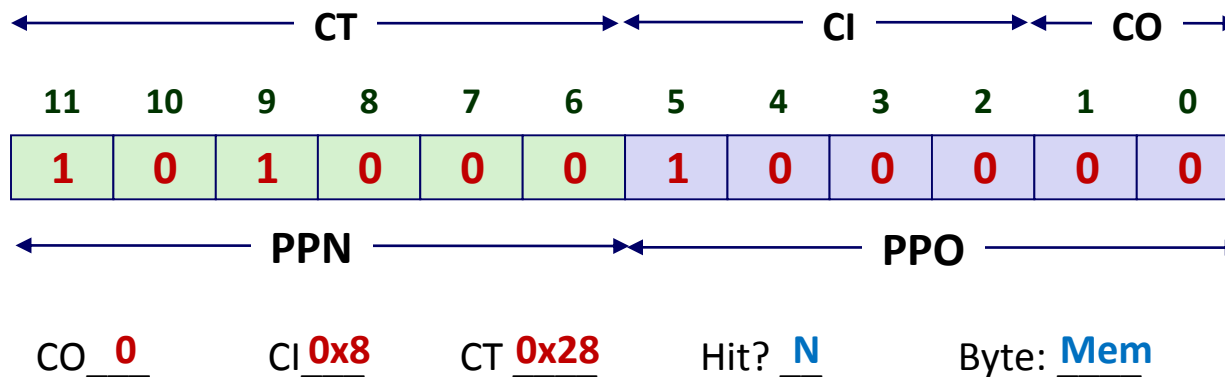
Address Translation Example: TLB/Cache Miss

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

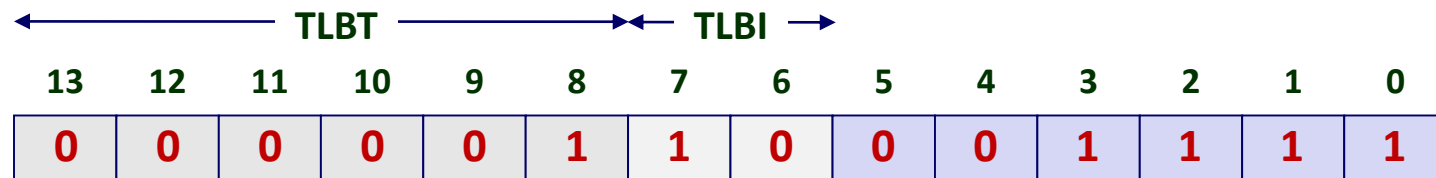
Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Physical Address



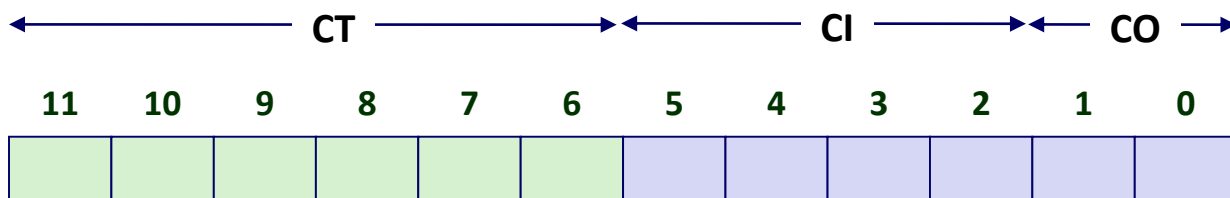
Address Translation Example: Page Fault

Virtual Address: 0x018F



VPN 0x06 TLBI 0x2 TLBT 0x01 TLB Hit? N Page Fault? Y PPN: TBD

Physical Address



PPN PPO

CO ____ CI ____ CT ____ 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

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

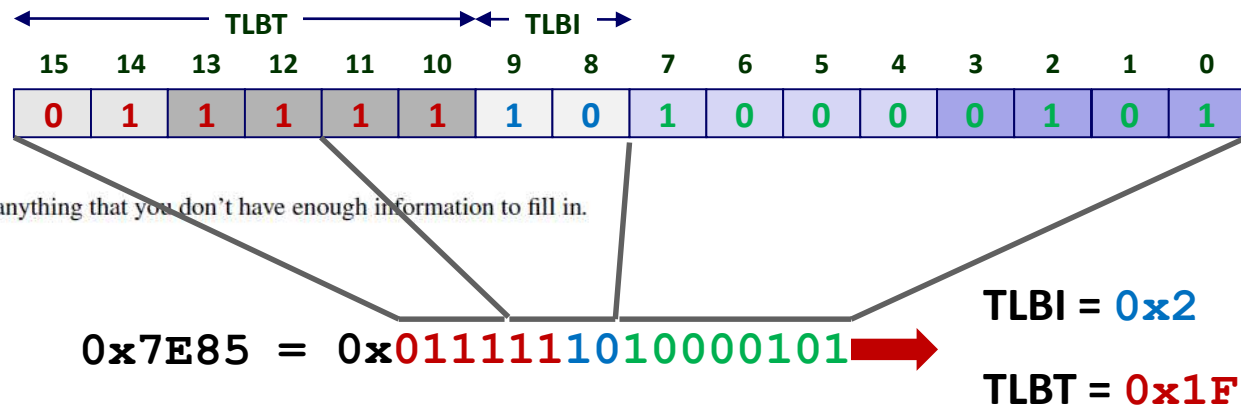
TLB			
Index	Tag	PPN	Valid
0	0x13	0x30	1
	0x34	0x58	0
1	0x1F	0x80	0
	0x2A	0x72	1
2	0x1F	0x95	1
	0x20	0xAA	0
3	0x3F	0x20	1
	0x3E	0xFF	0

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



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



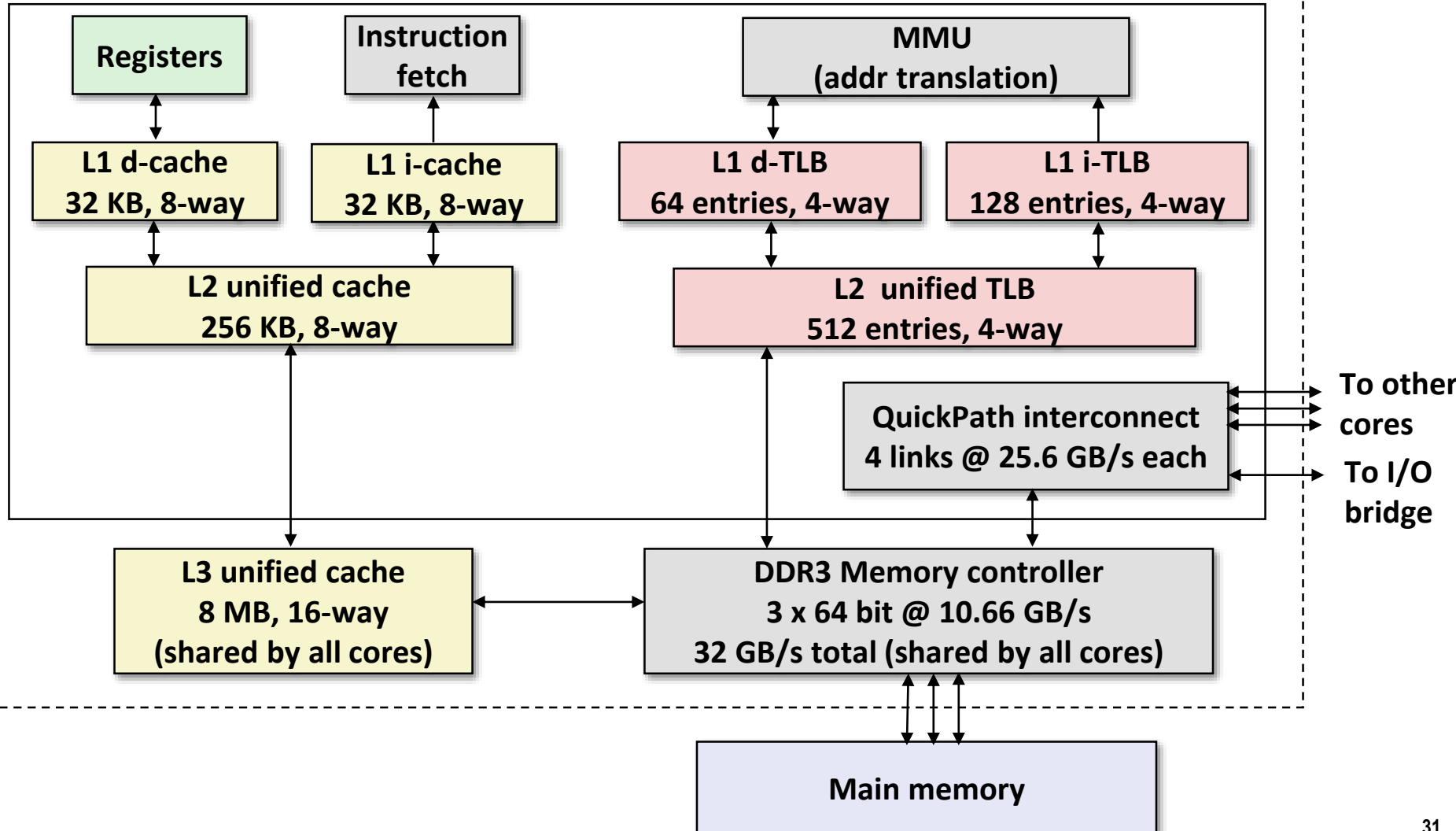
Today

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

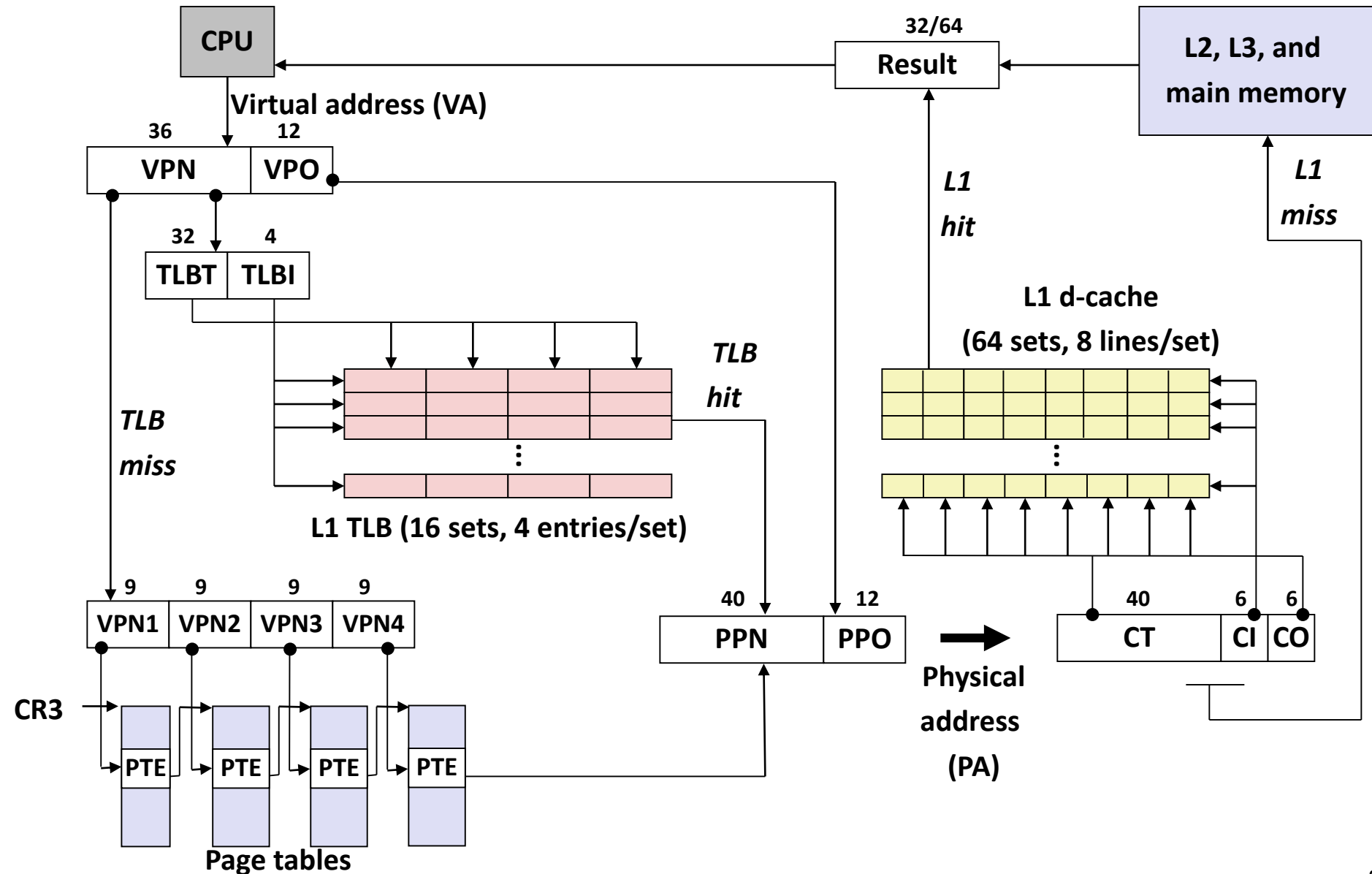
Intel Core i7 Memory System

Processor package

Core x4



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 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

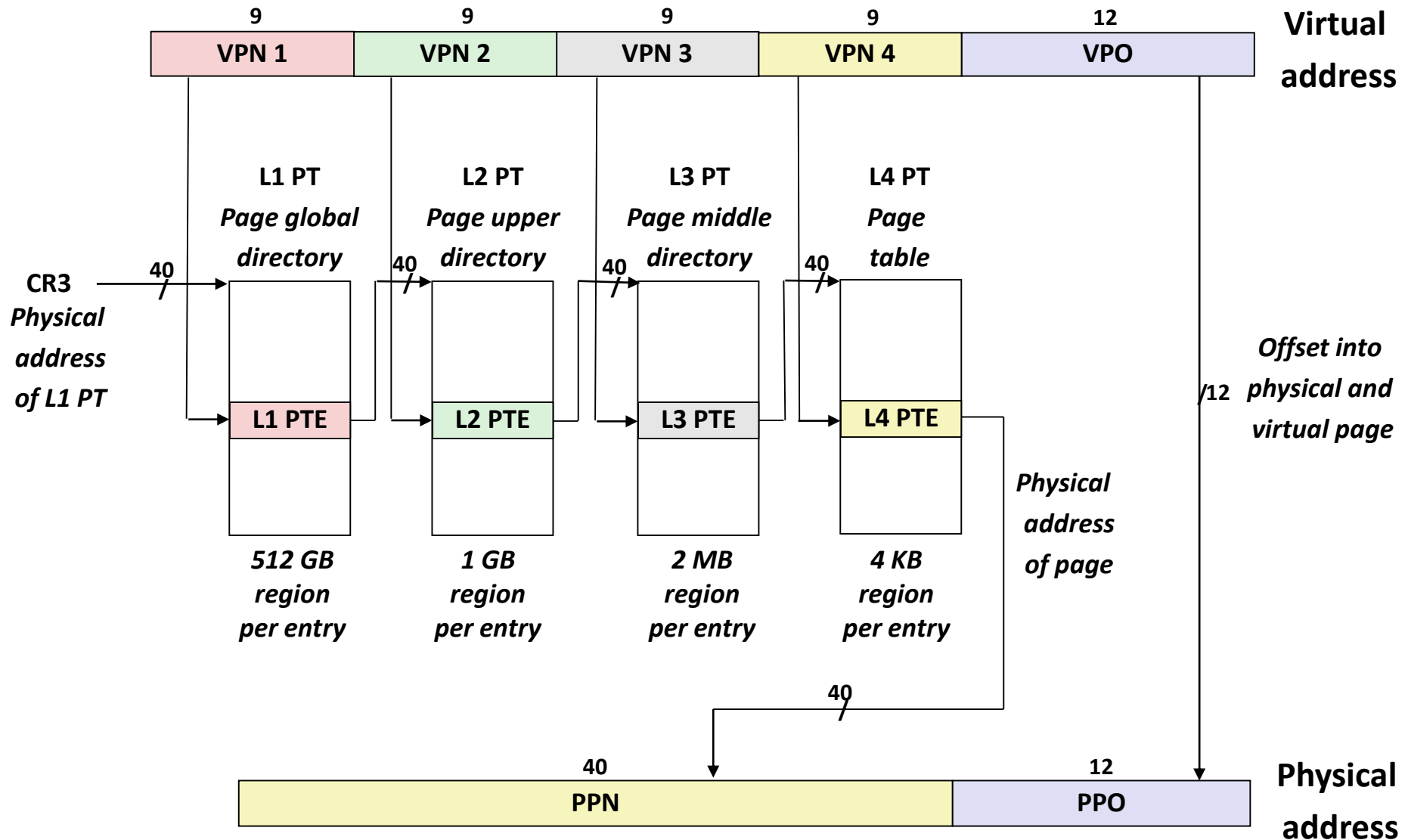
A: Reference bit (set by MMU on reads and writes, cleared by software)

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

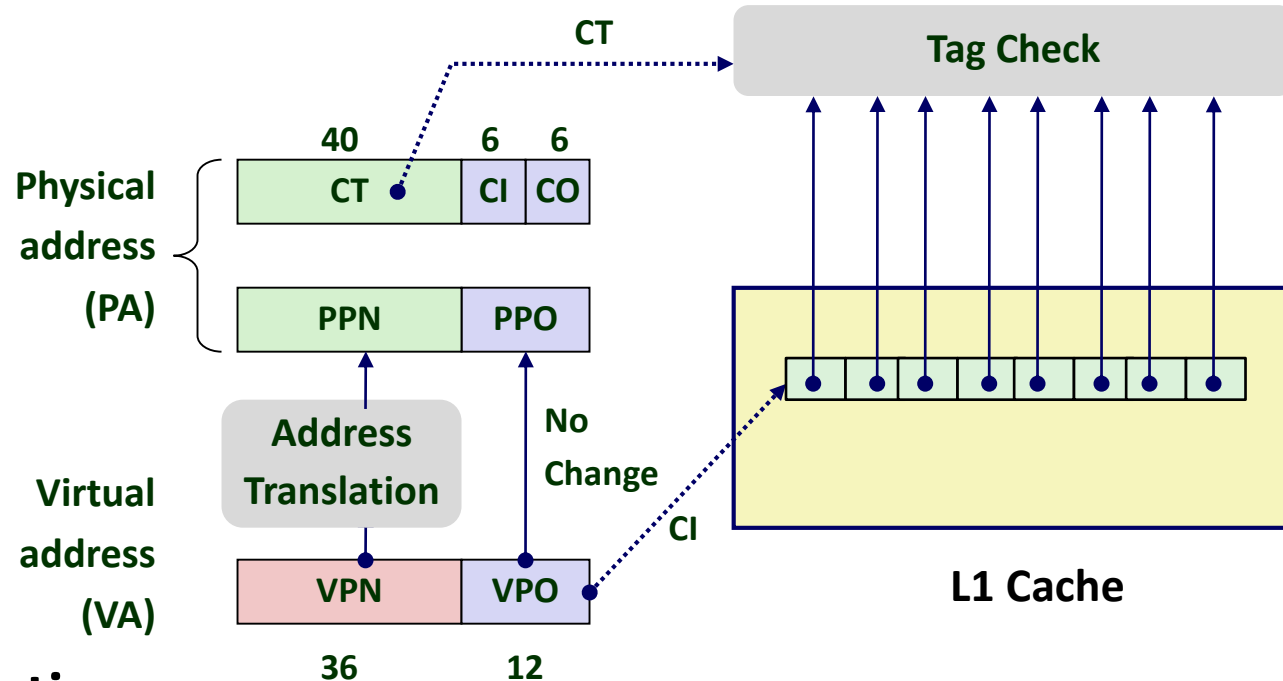
Page physical base address: 40 most significant bits of physical page address
(forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.

Core i7 Page Table Translation



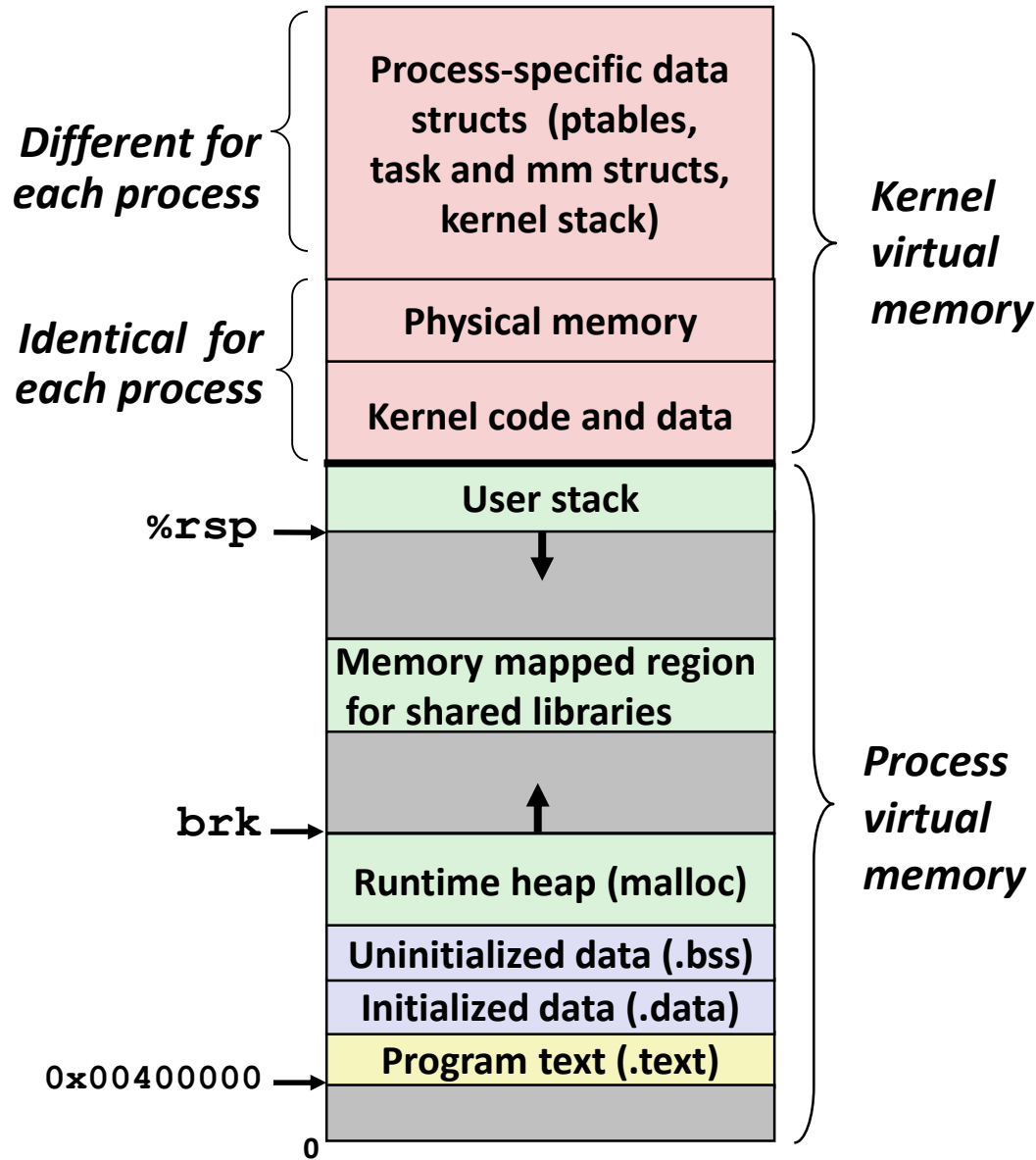
Cute Trick for Speeding Up L1 Access



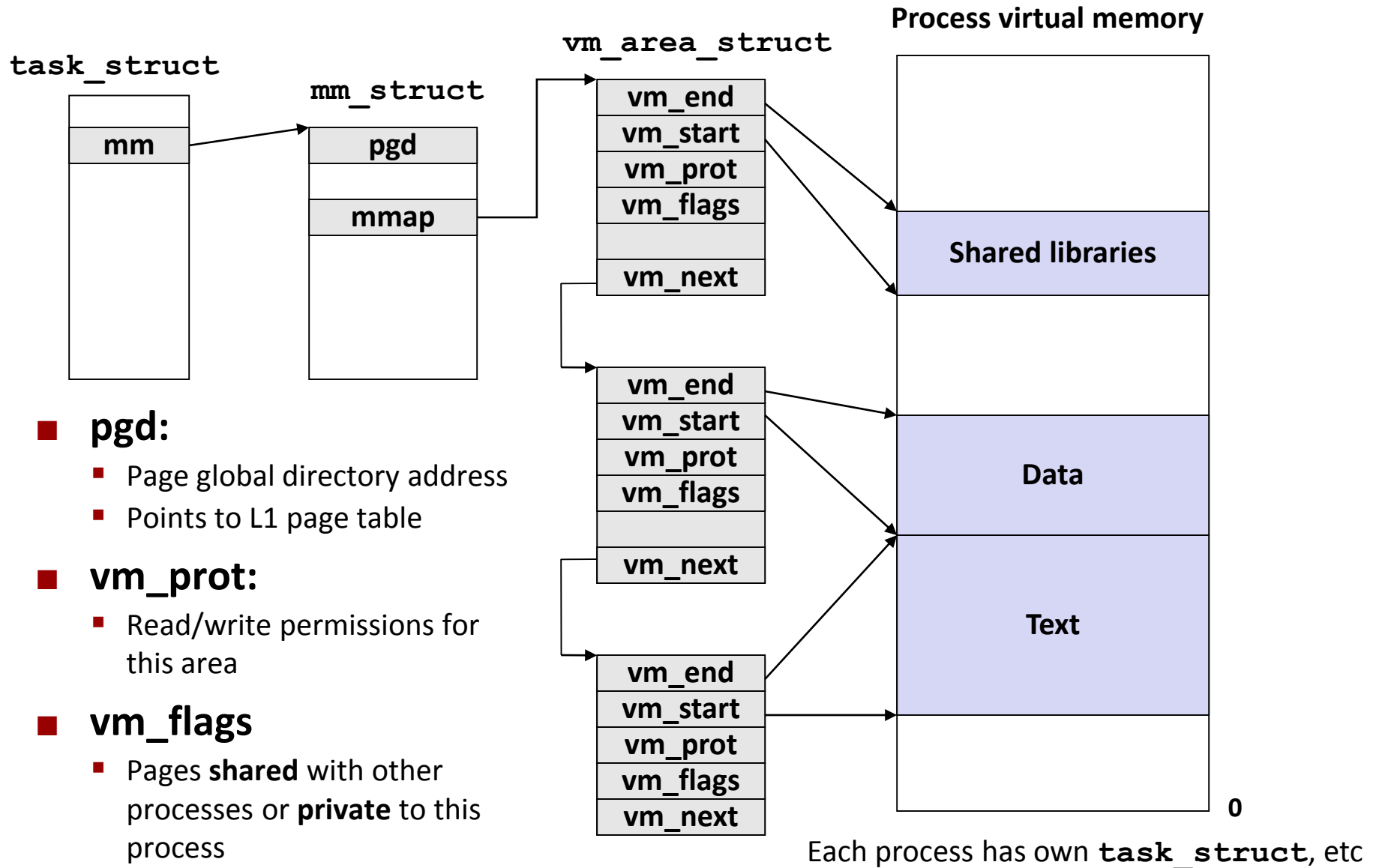
■ Observation

- Bits that determine CI identical in virtual and physical address
- Can index into cache while address translation taking place
- Generally we hit in TLB, so PPN bits (CT bits) available next
- ***“Virtually indexed, physically tagged”***
- Cache carefully sized to make this possible

Virtual Address Space of a Linux Process

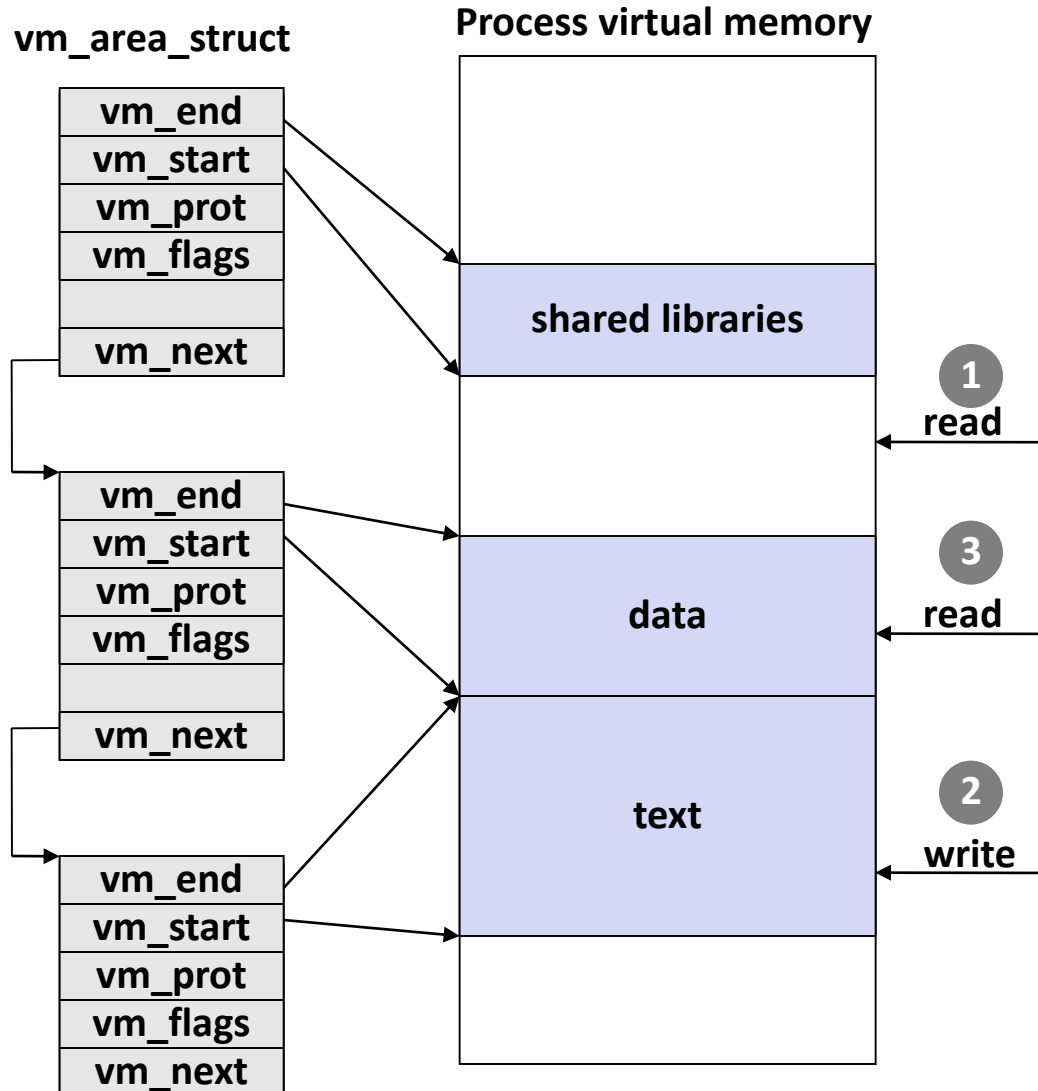


Linux Organizes VM as Collection of “Areas”



- **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

Linux Page Fault Handling



Today

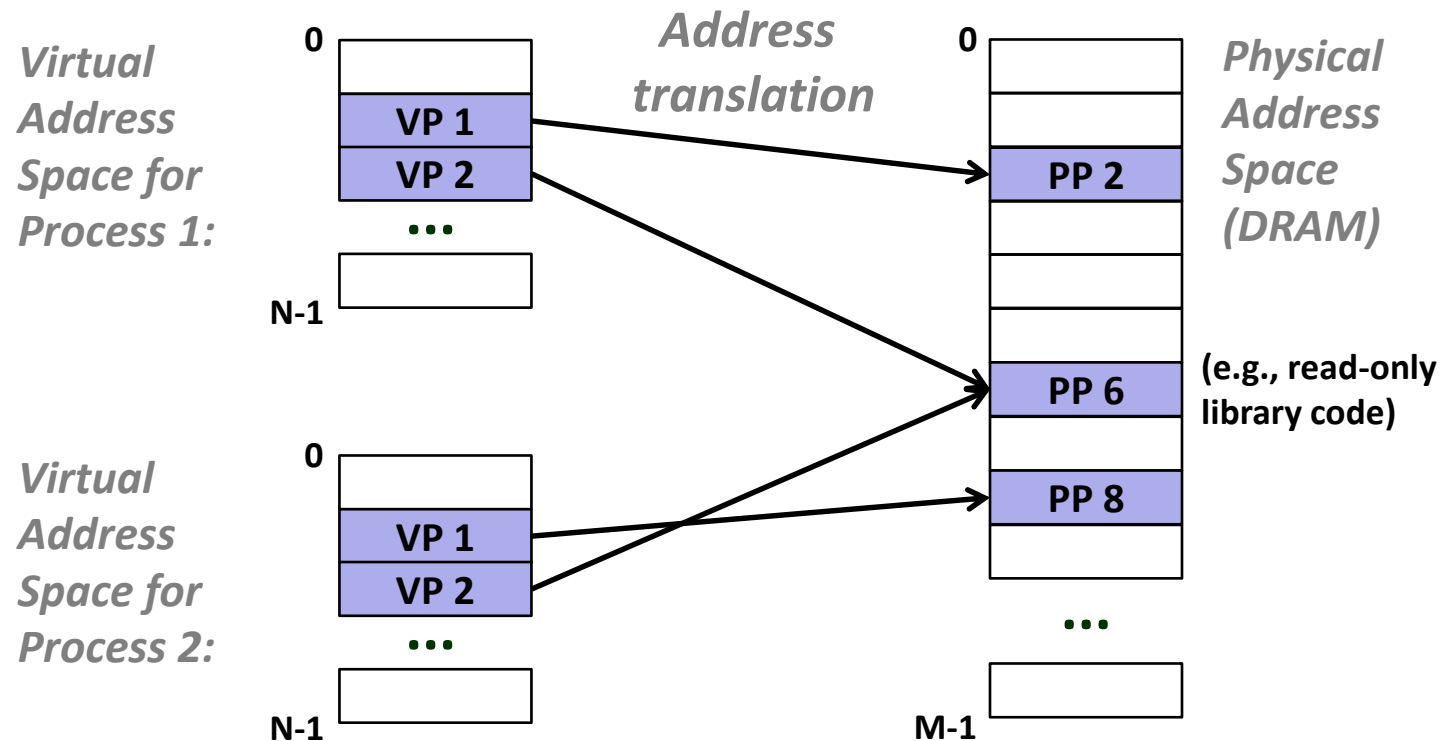
- Virtual memory questions and answers
- Simple memory system example
- Case study: Core i7/Linux memory system
- **Memory mapping**

Memory Mapping

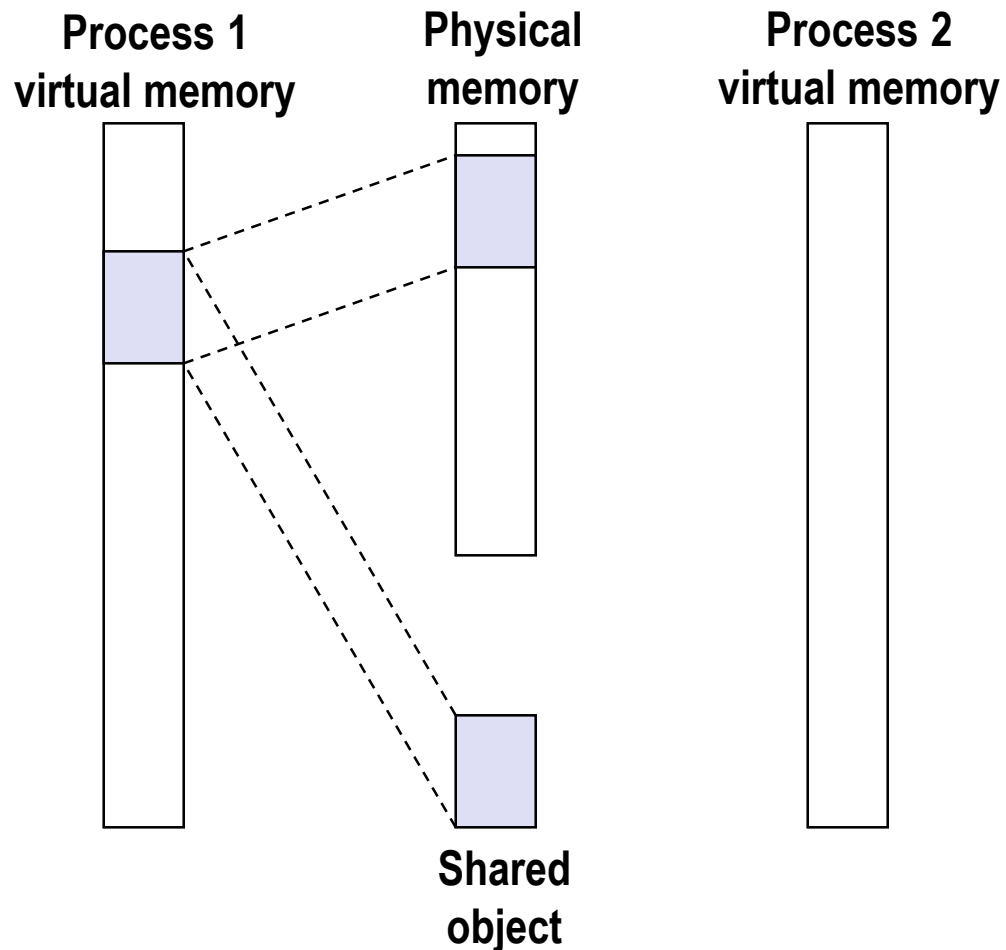
- VM areas initialized by associating them with disk objects.
 - Process is known as *memory mapping*.
- Area can be *backed by* (i.e., get its initial values from) :
 - *Regular file* on disk (e.g., an executable object file)
 - Initial page bytes come from a section of a file
 - *Anonymous file* (e.g.,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

Review: Memory Management & Protection

- Code and data can be isolated or shared among processes

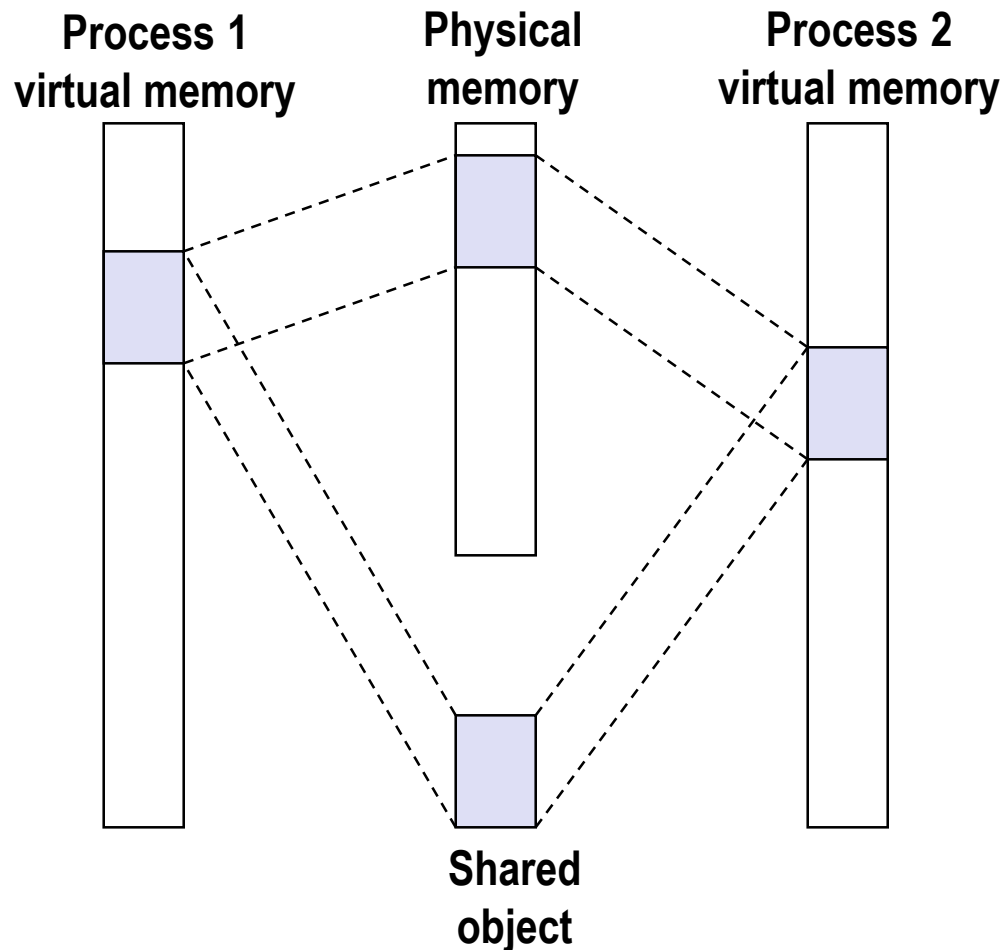


Sharing Revisited: Shared Objects



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

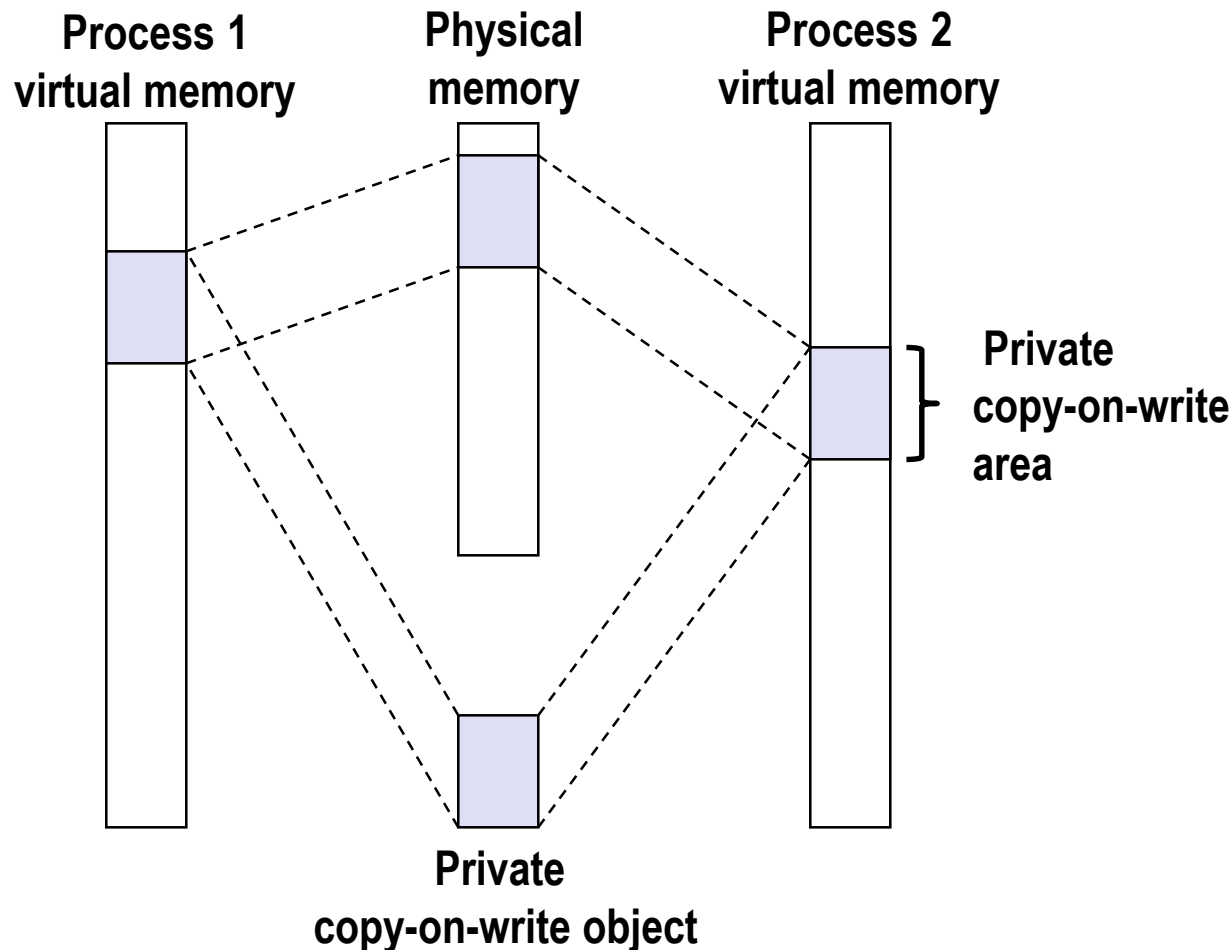
Sharing Revisited: Shared Objects



- **Process 2 maps the same shared object.**
- **Notice how the virtual addresses can be different.**
- **But, difference must be multiple of page size**

Sharing Revisited:

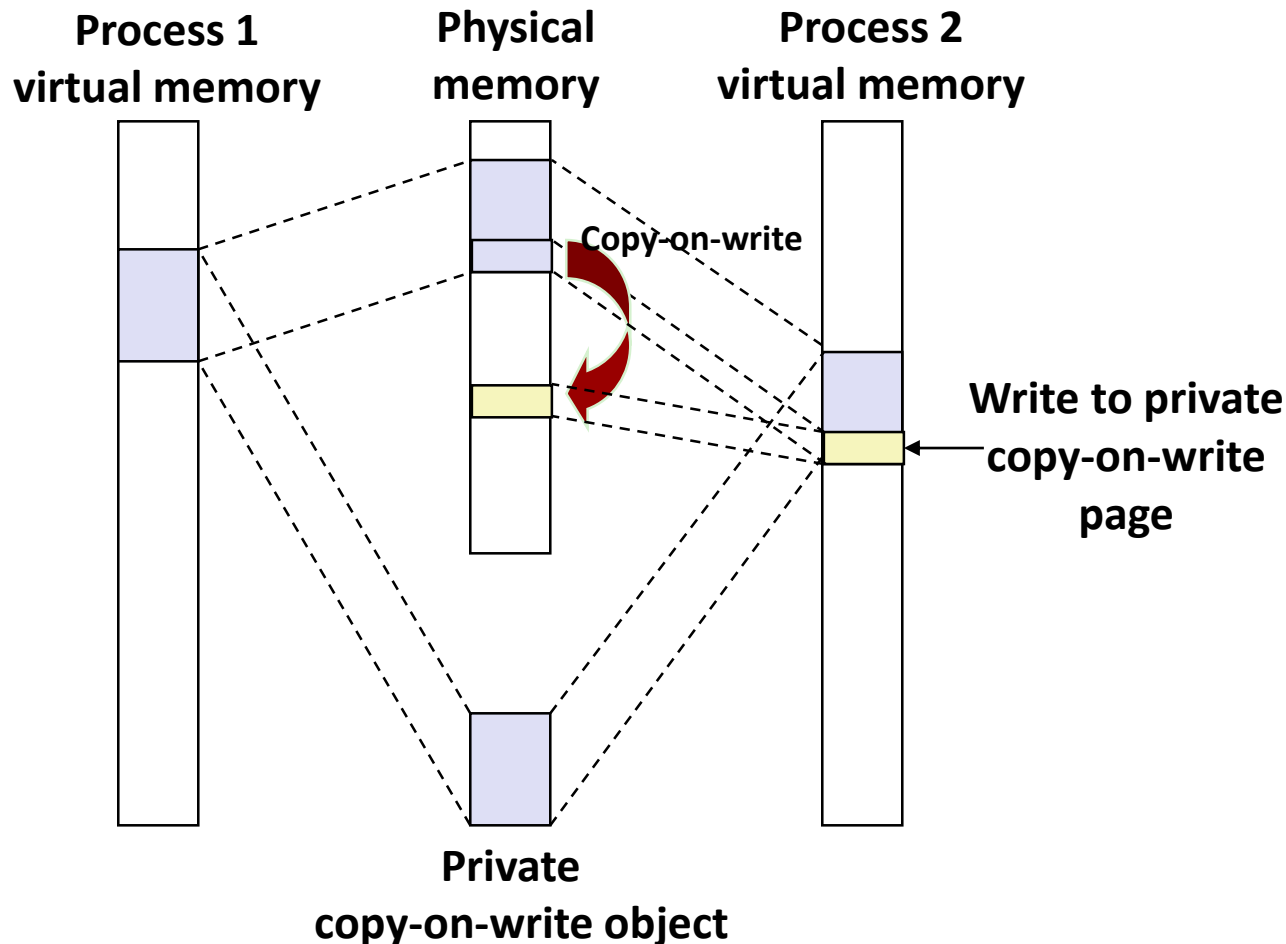
Private Copy-on-write (COW) Objects



- 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

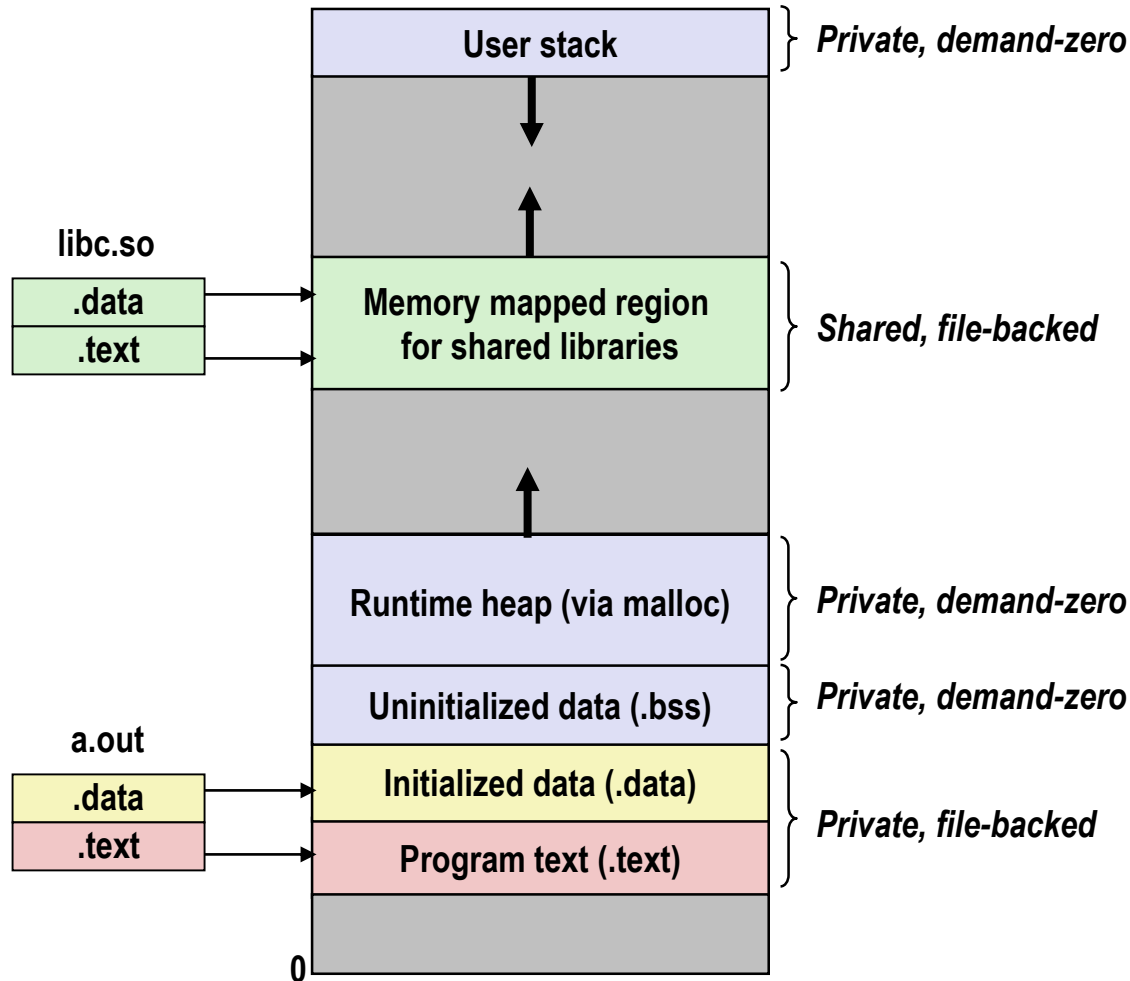


- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

The `fork` Function Revisited

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



- 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 create 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

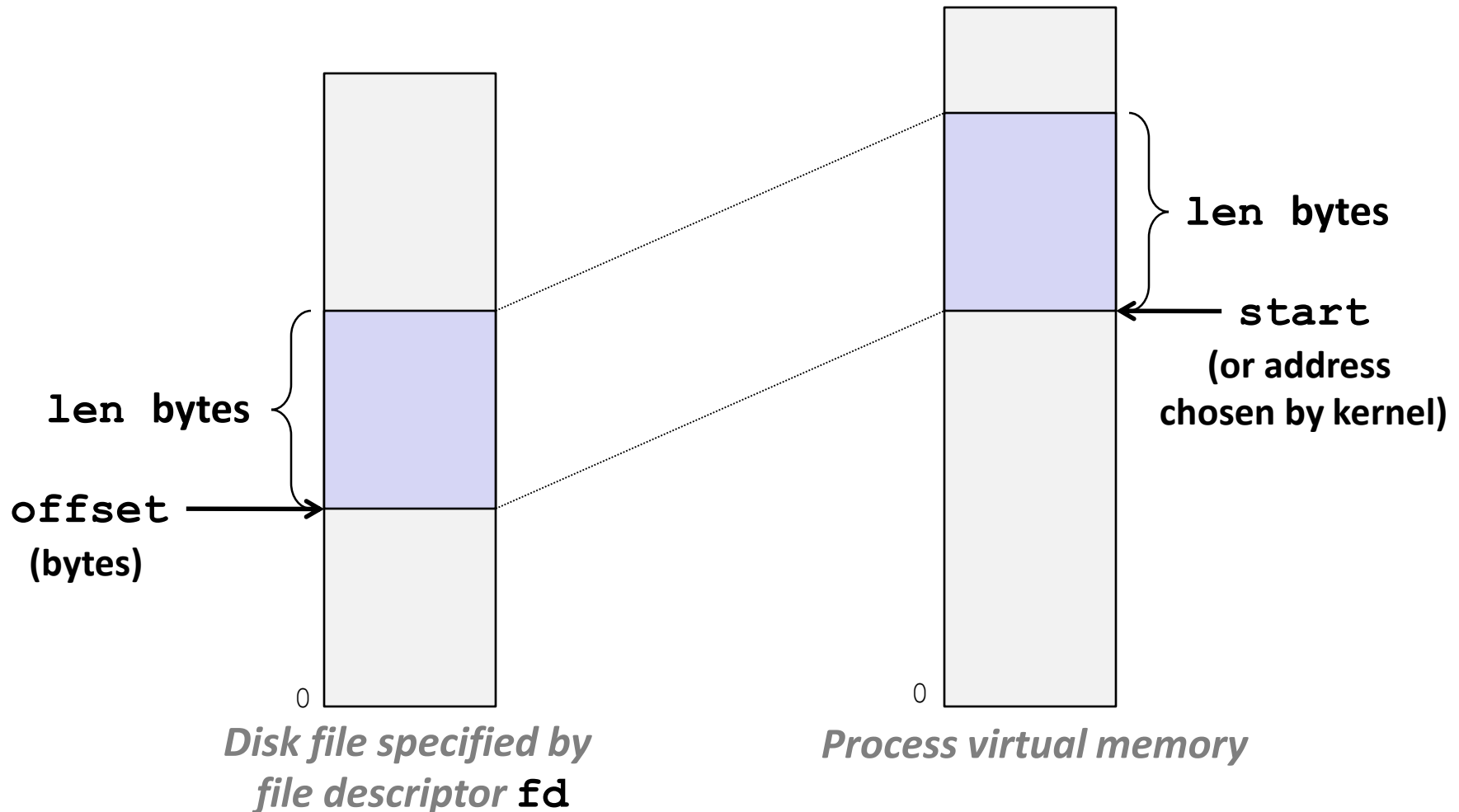
User-Level Memory Mapping

```
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`, ...
 - `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)
```



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

Example: Using mmap to Copy Files

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

```
#include "csapp.h"

void mmapcopy(int fd, int size)
{

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

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

mmapcopy.c

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

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

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

mmapcopy.c

Summary

■ 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