

# Introduction to Computer Systems

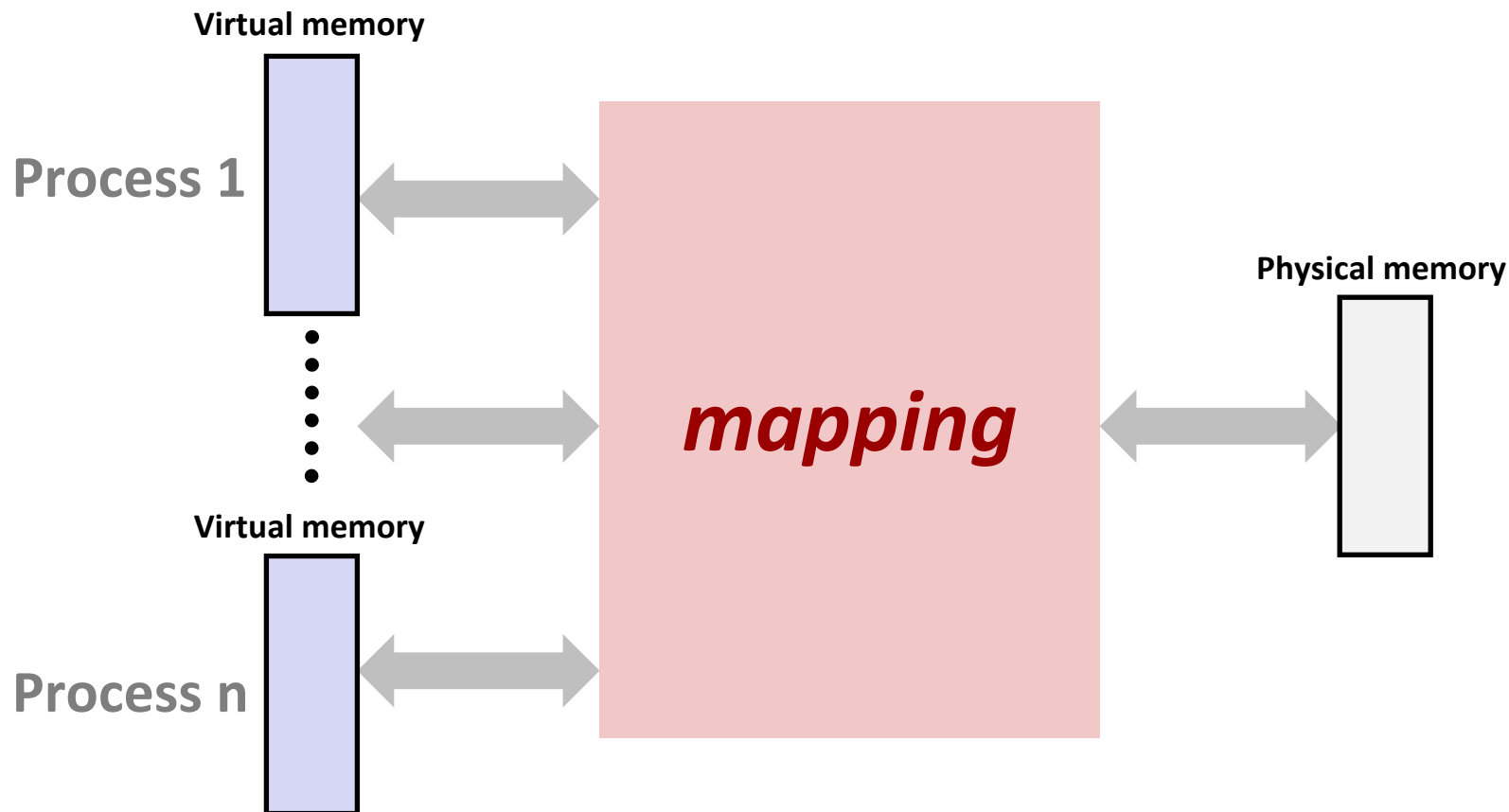
15-213, fall 2009

17<sup>th</sup> Lecture, Oct. 26<sup>th</sup>

## **Instructors:**

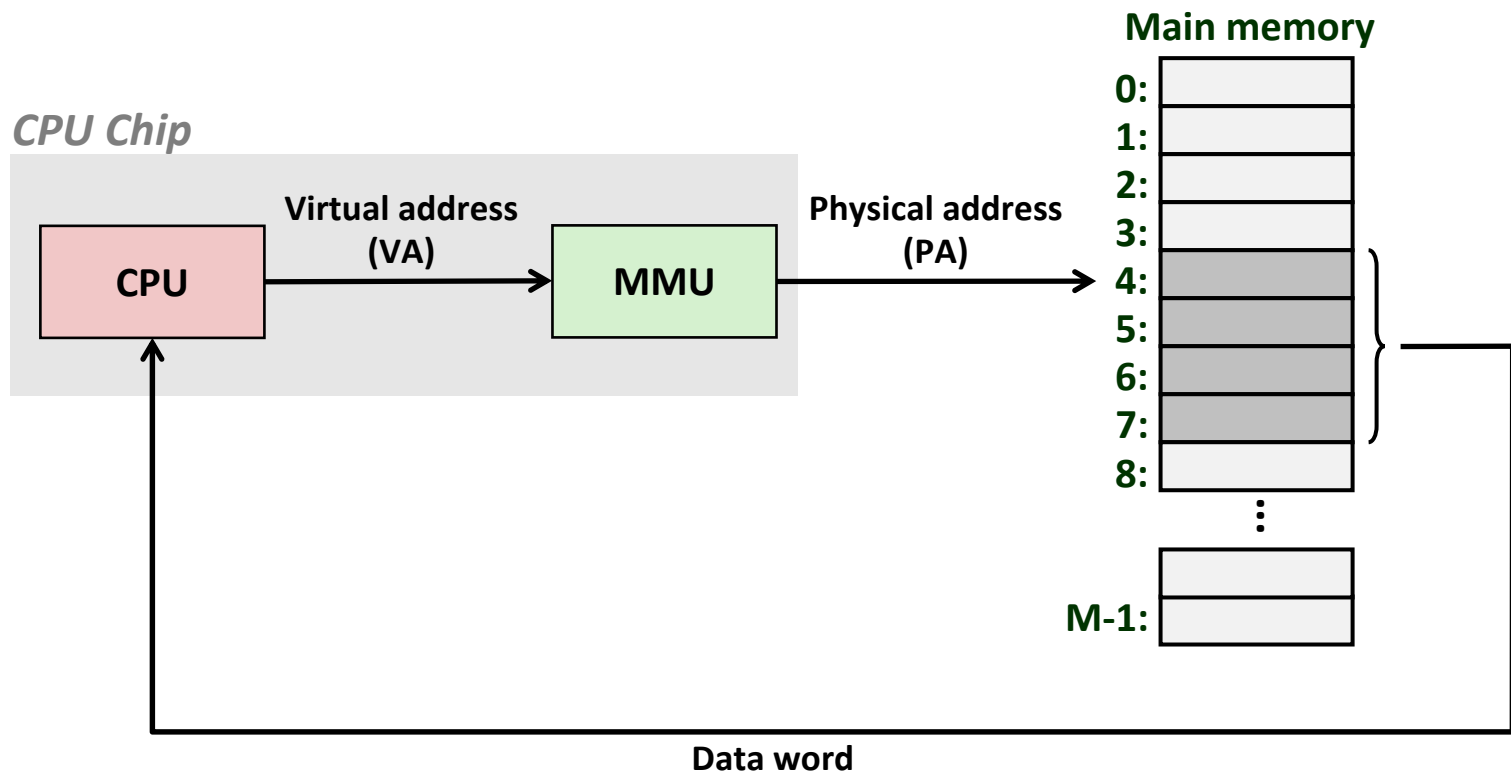
Majd Sakr and Khaled Harras

# Last Time: Virtual Memory



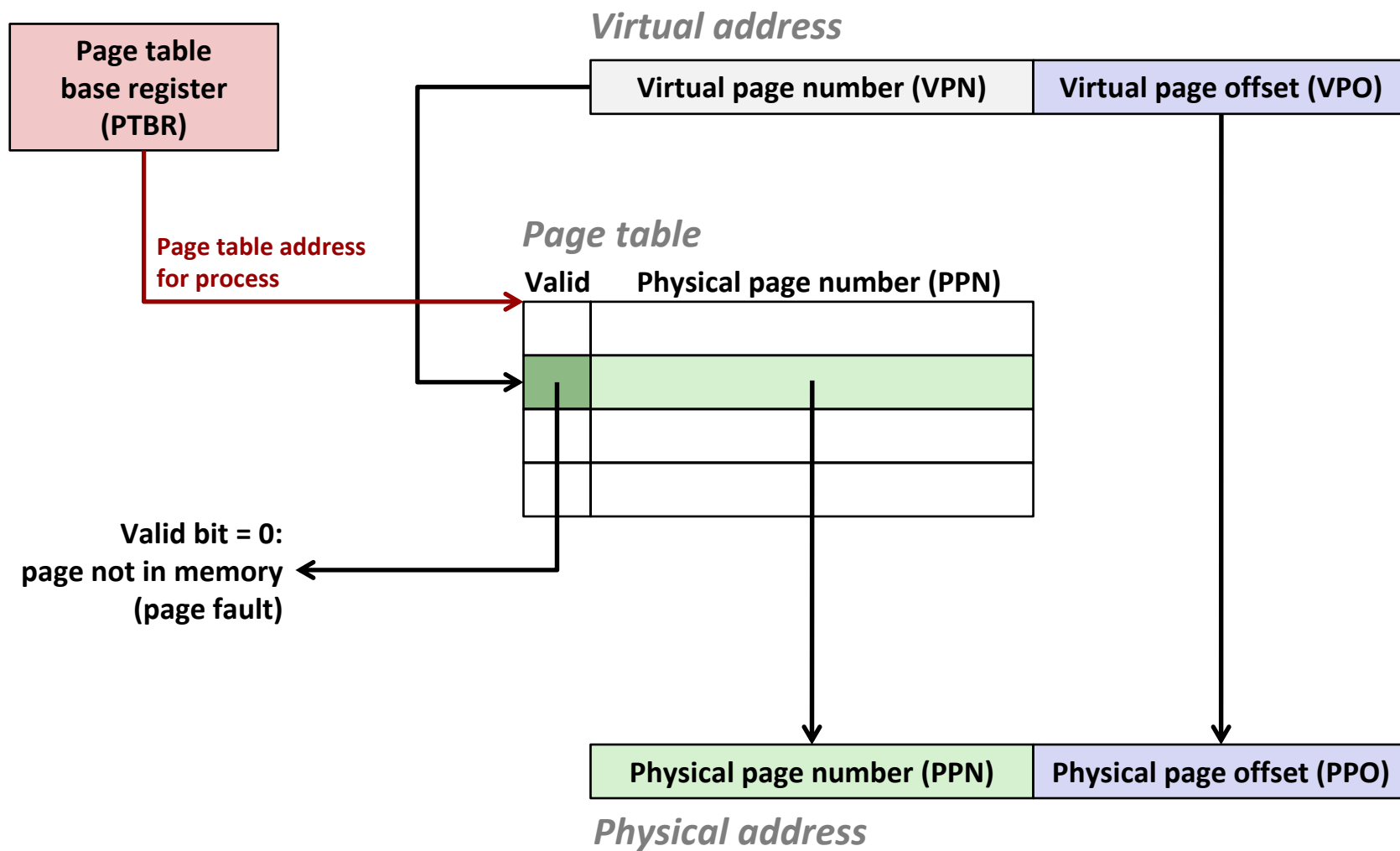
- Each process gets its own private memory space
- Solves the previous problems

# A System Using Virtual Addressing



- Used in all modern desktops, laptops, workstations
- One of the great ideas in computer science
- *MMU checks the cache*

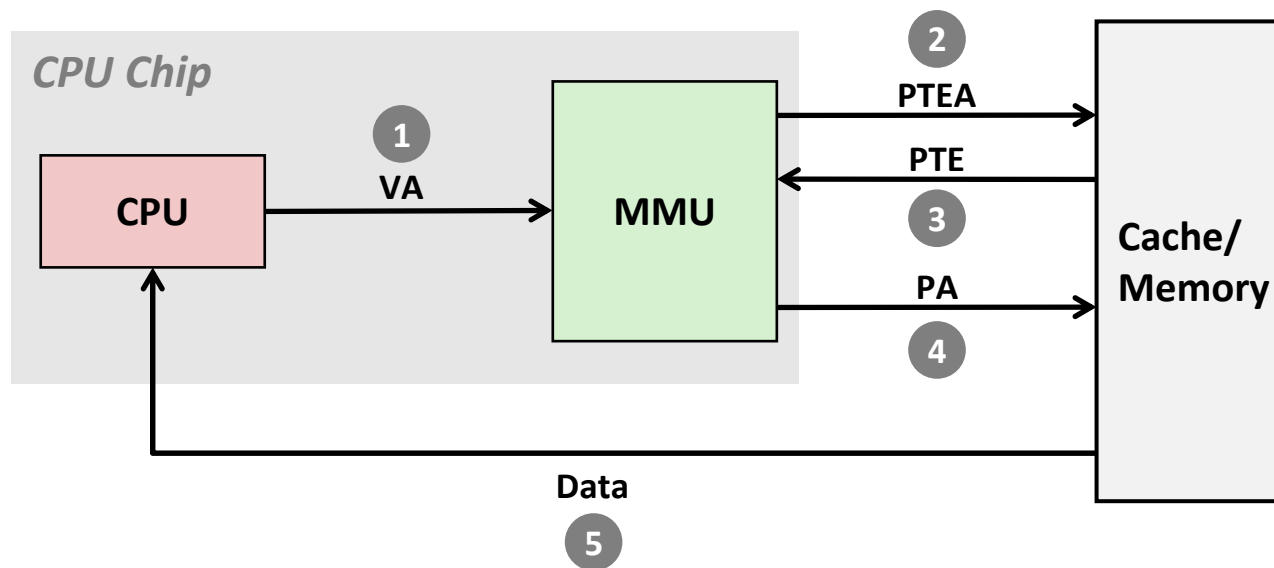
# Last Time: Address Translation



# Today

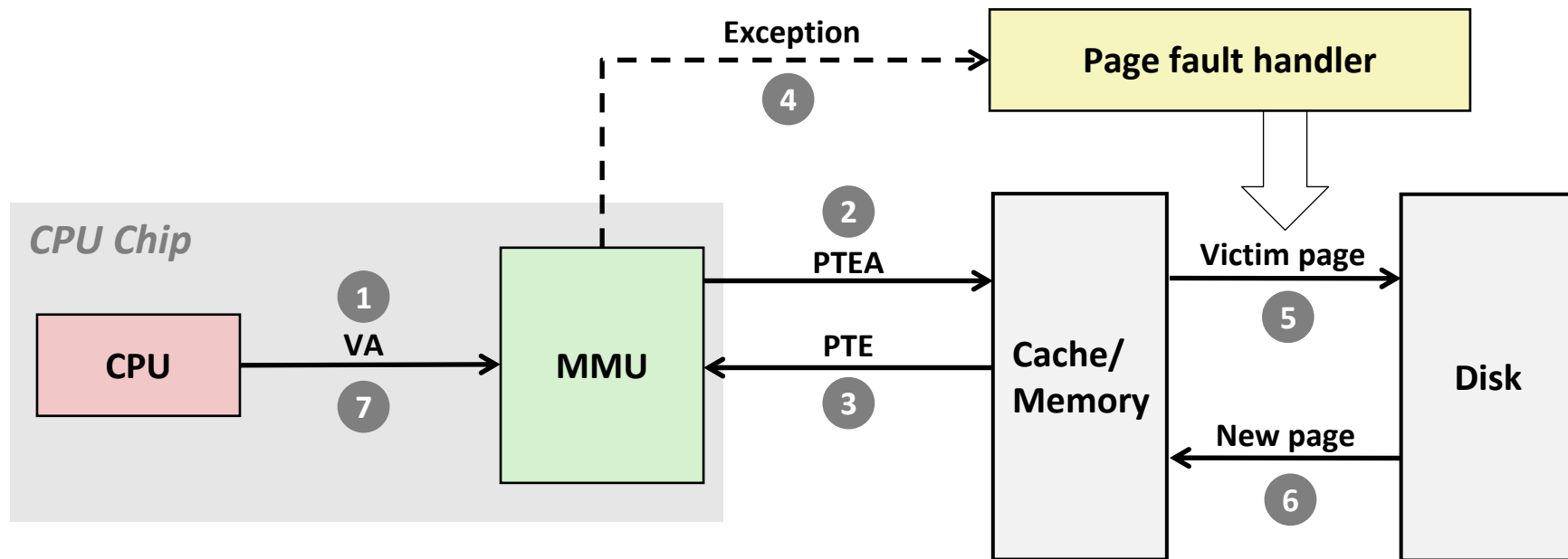
- **Virtual memory (VM)**
  - **Address translation**
  - Allocation
  - Multi-level page tables
- **Linux VM system**

# 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

# Address Translation: Page Fault



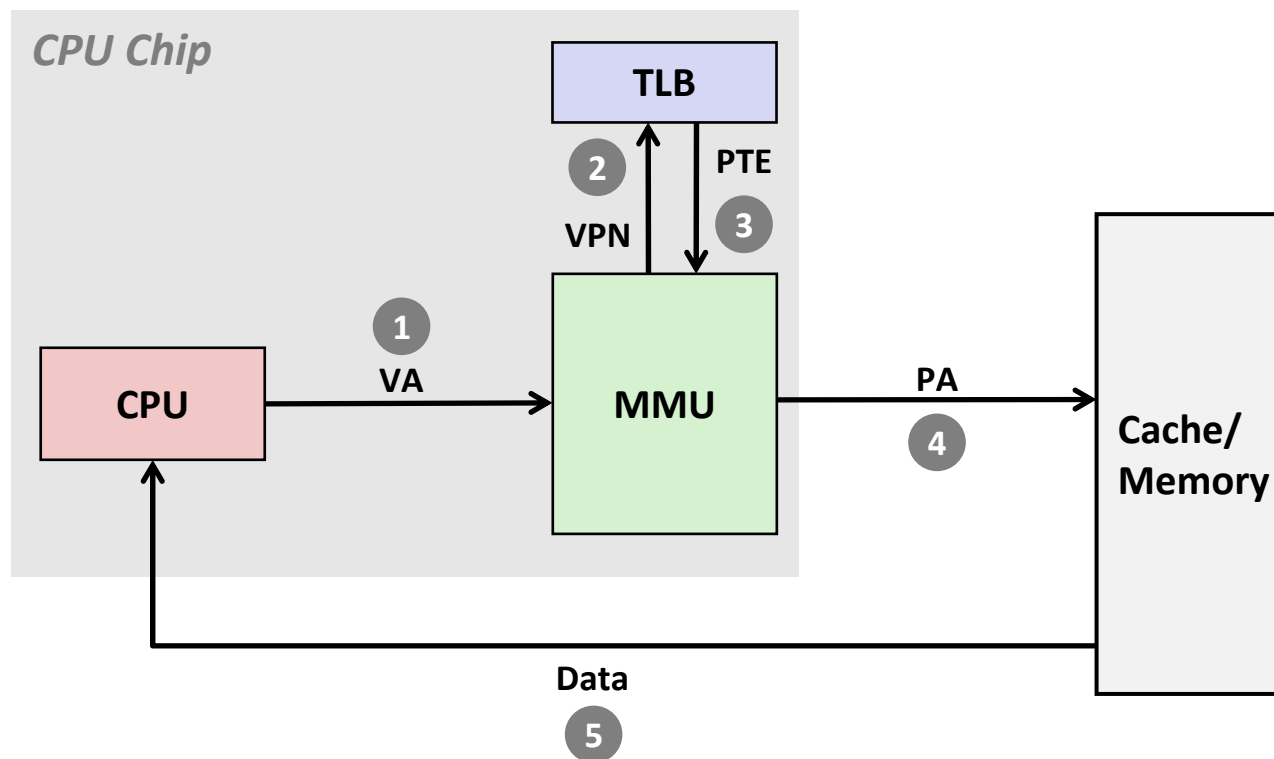
- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim (and, if dirty, pages it out to disk)
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns to original process, restarting faulting instruction

# 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 1-cycle delay
- **Solution: *Translation Lookaside Buffer* (TLB)**
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

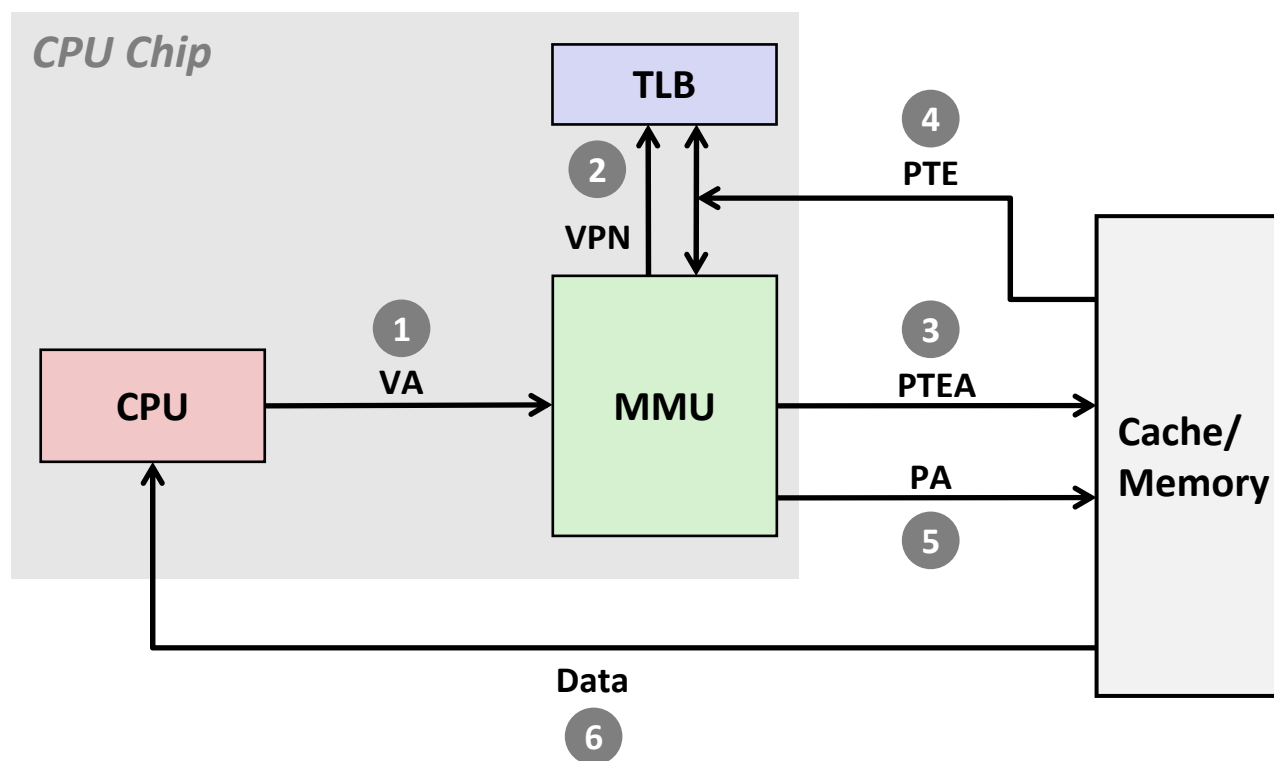


# TLB Hit



**A TLB hit eliminates a memory access**

# TLB Miss



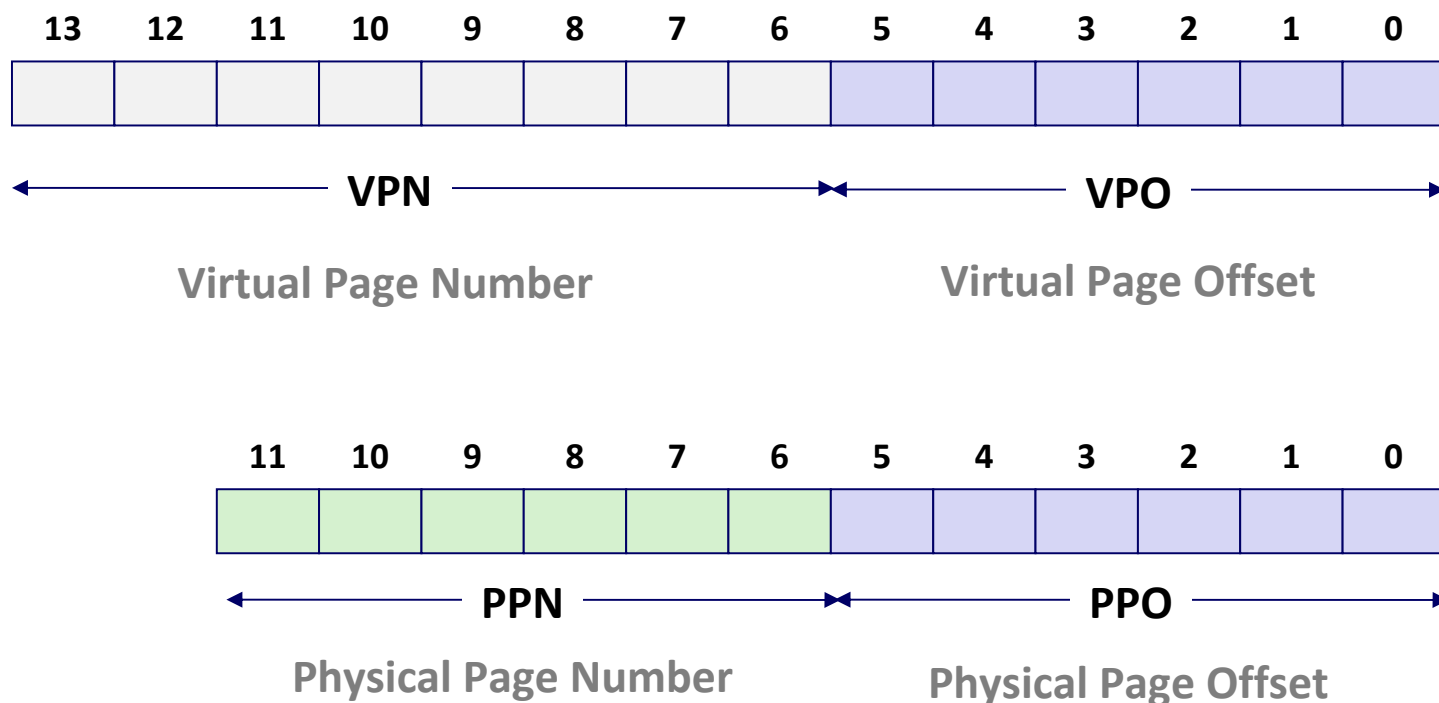
**A TLB miss incurs an add'l memory access (the PTE)**

Fortunately, TLB misses are rare

# Simple Memory System Example

## ■ Addressing

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



# Simple Memory System Page Table

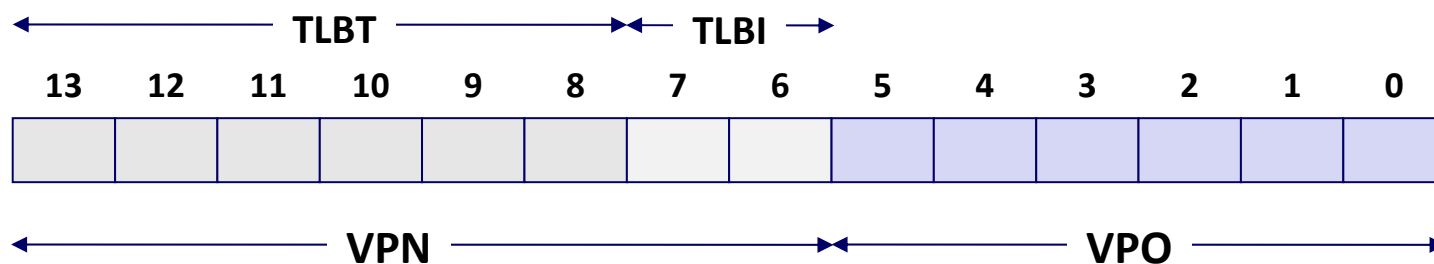
Only show first 16 entries (out of 256)

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

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

# Simple Memory System TLB

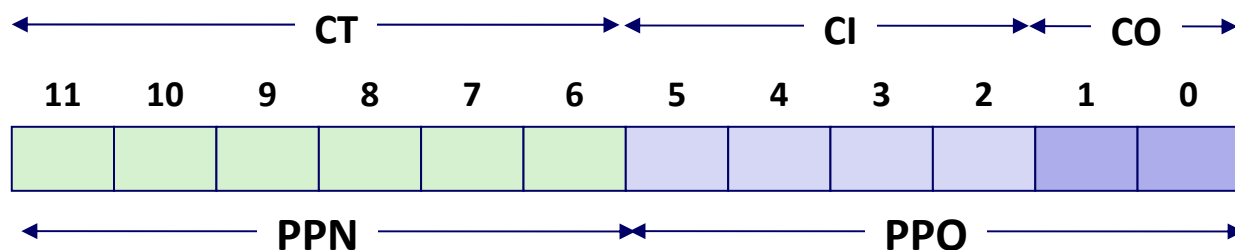
- 16 entries
- 4-way associative



<i>Set</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>
<b>0</b>	03	–	0	09	0D	1	00	–	0	07	02	1
<b>1</b>	03	2D	1	02	–	0	04	–	0	0A	–	0
<b>2</b>	02	–	0	08	–	0	06	–	0	03	–	0
<b>3</b>	07	–	0	03	0D	1	0A	34	1	02	–	0

# Simple Memory System Cache

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

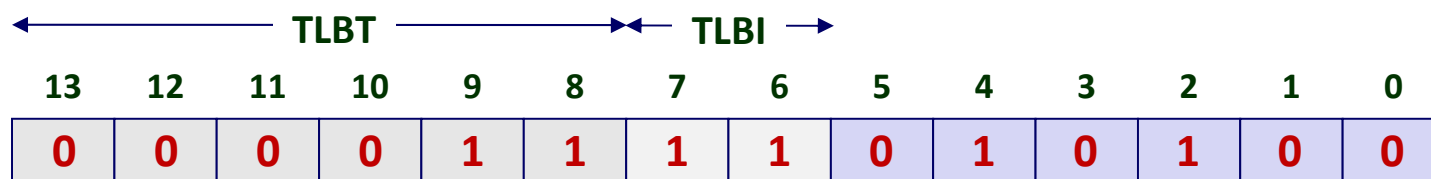


<i>Idx</i>	<i>Tag</i>	<i>Valid</i>	<i>B0</i>	<i>B1</i>	<i>B2</i>	<i>B3</i>
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

<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	–	–	–	–

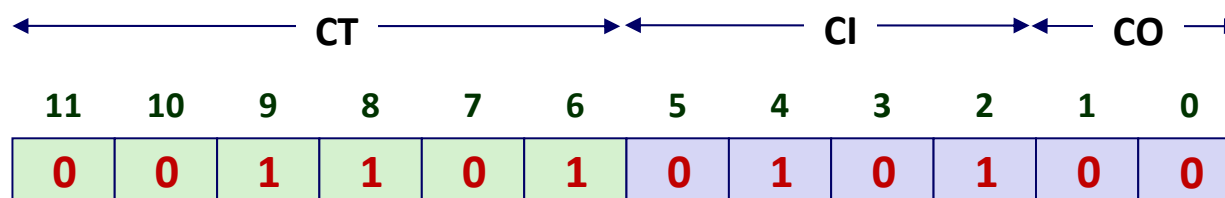
# Address Translation Example #1

Virtual Address: 0x03D4



VPN 0x0F    TLBI 3    TLBT 0x03    TLB Hit? Y    Page Fault? N    PPN: 0x0D

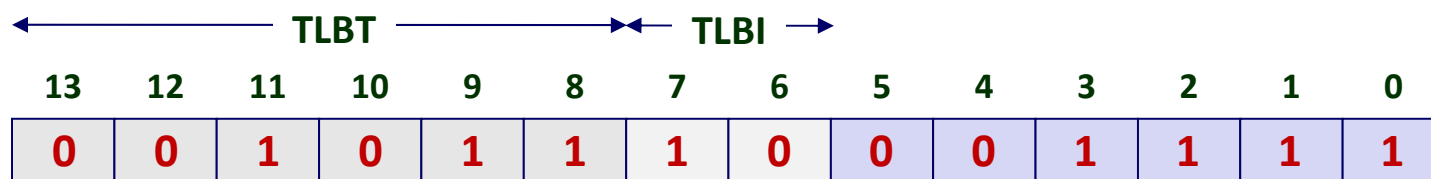
## Physical Address



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

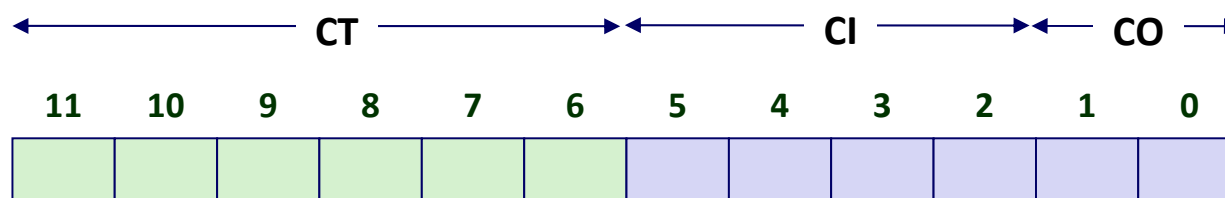
# Address Translation Example #2

Virtual Address: 0x0B8F



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

## Physical Address

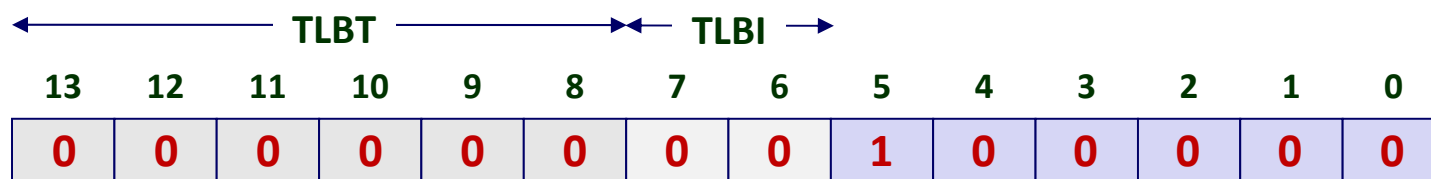


CO \_\_\_    CI \_\_\_    CT \_\_\_    Hit? \_\_\_    Byte: \_\_\_



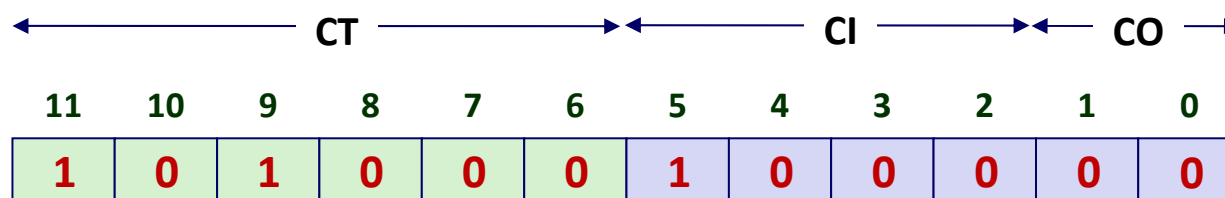
# Address Translation Example #3

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

# 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

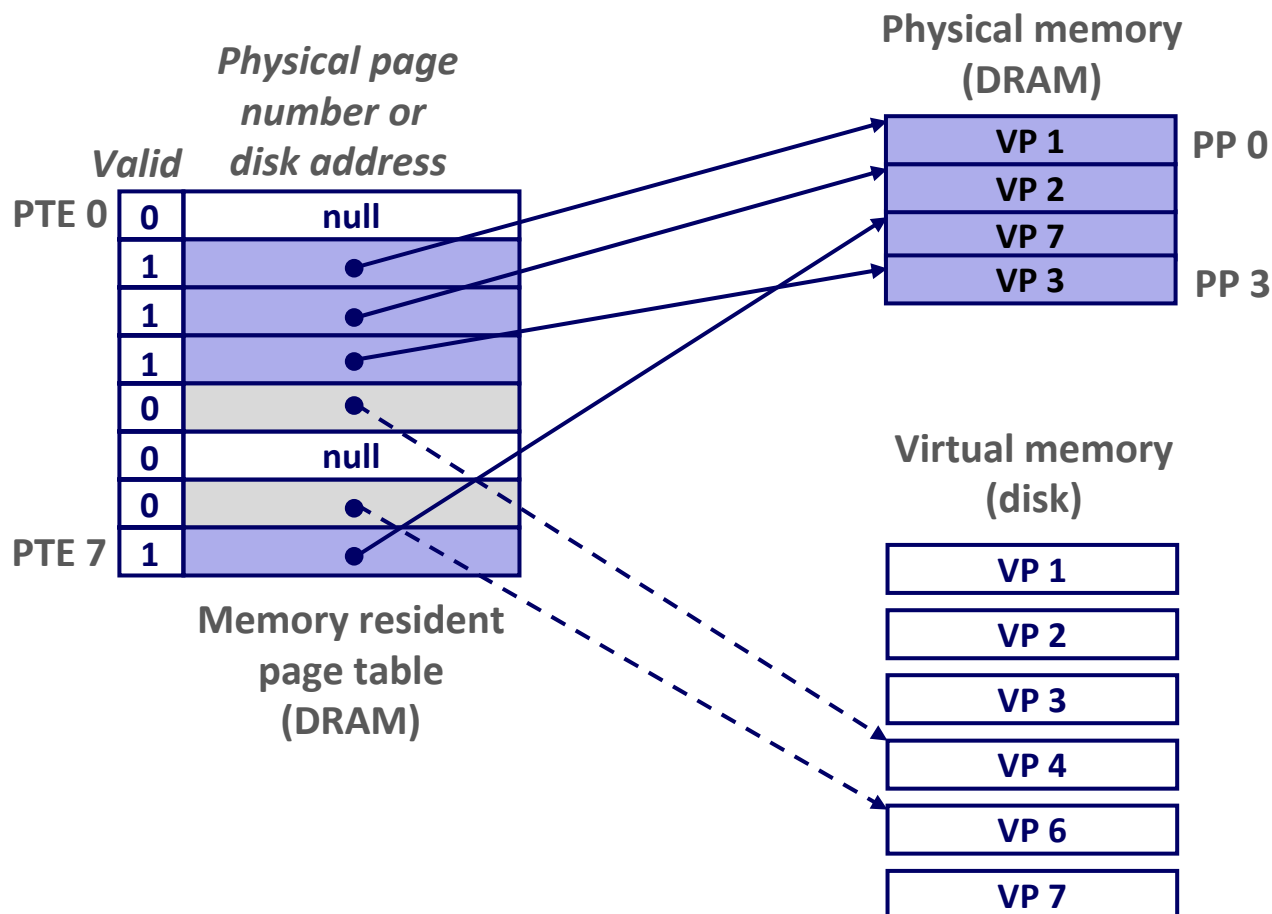
- Uses memory efficiently by caching virtual memory pages
  - Efficient only because of locality
- Simplifies memory management and programming
- Simplifies protection by providing a convenient interpositioning point to check permissions

# Today

- **Virtual memory (VM)**
  - Address translation
  - **Allocation**
  - Multi-level page tables
- Linux VM system

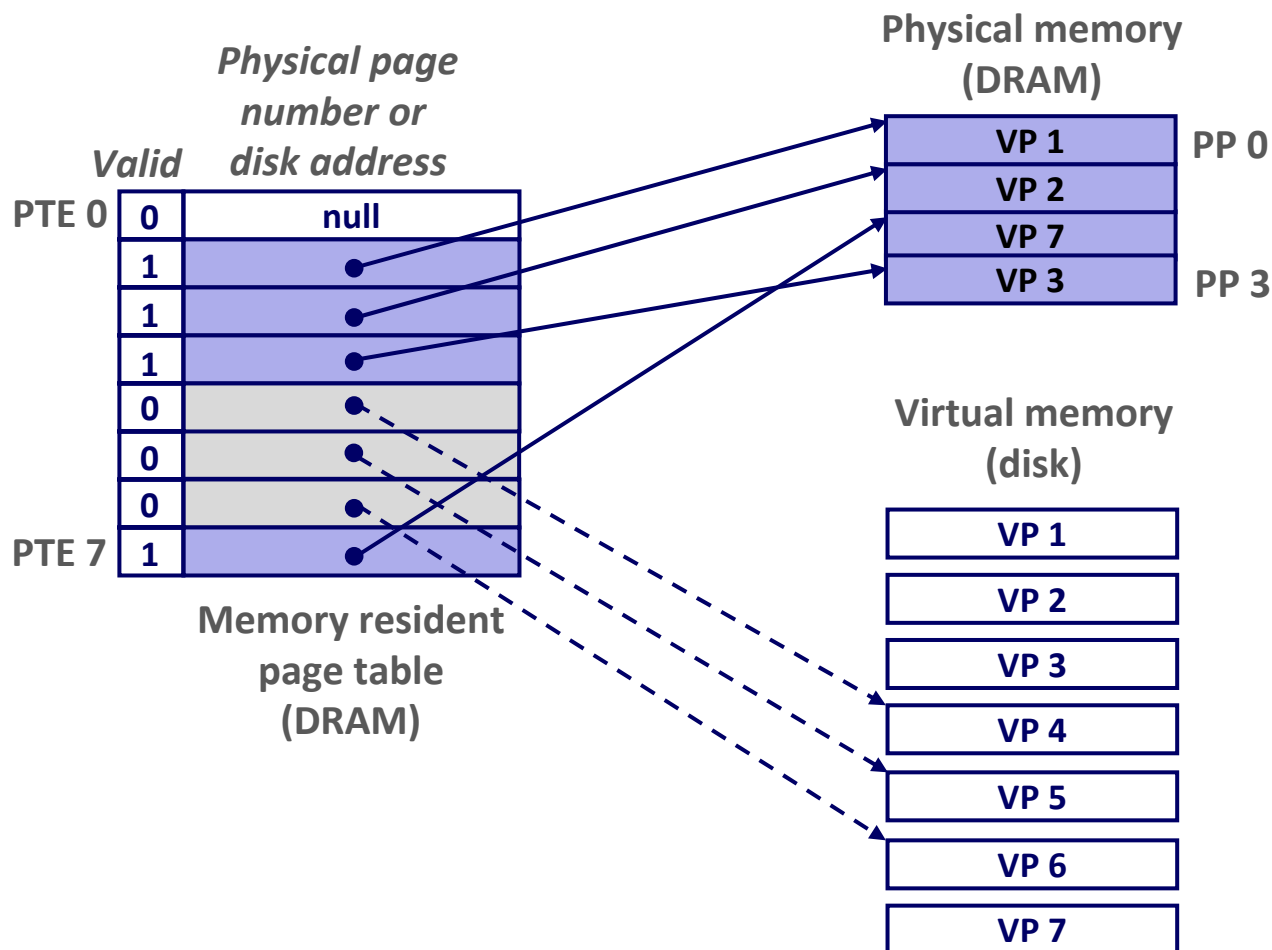
# Allocating Virtual Pages

## ■ Example: Allocating VP5



# Allocating Virtual Pages

- Example: Allocating VP 5
- Kernel allocates VP 5 on disk and points PTE 5 to it



# Today

- **Virtual memory (VM)**
  - Address translation
  - Allocation
  - Multi-level page tables
- **Linux VM system**

# Multi-Level Page Tables

## ■ Given:

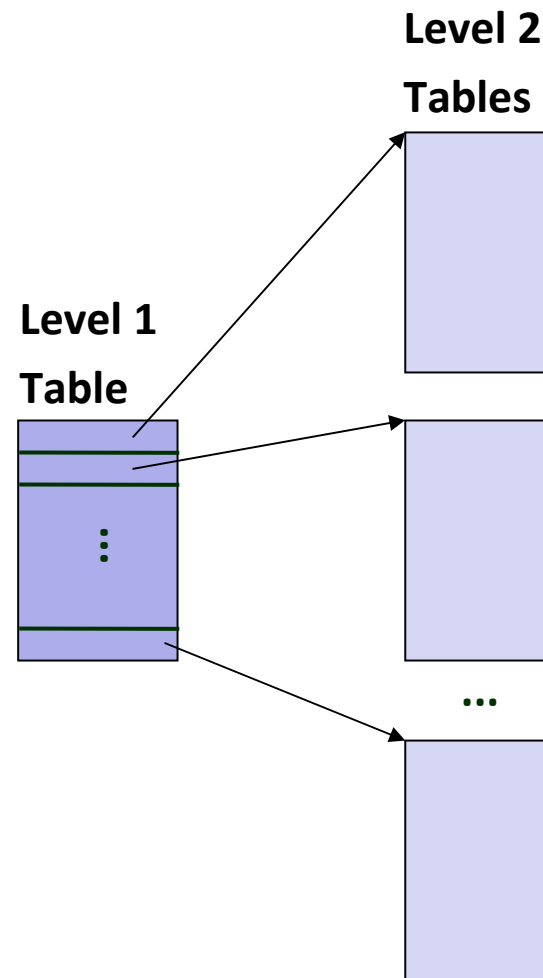
- 4KB ( $2^{12}$ ) page size
- 48-bit address space
- 4-byte PTE

## ■ Problem:

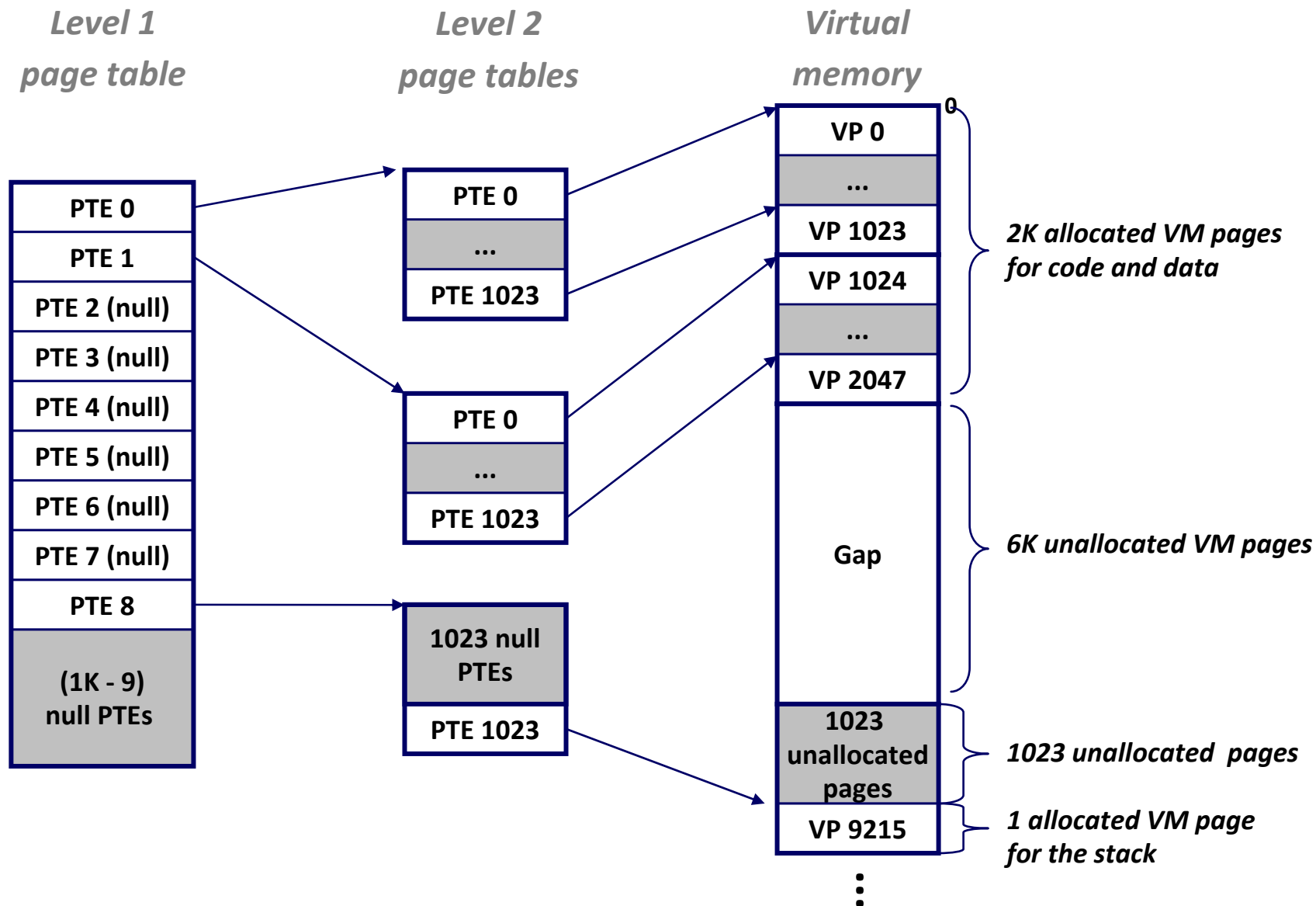
- Would need a 256 GB page table!
  - $2^{48} * 2^{-12} * 2^2 = 2^{38}$  bytes

## ■ Common solution

- Multi-level page tables
- Example: 2-level page table
- Level 1 table: each PTE points to a page table
- Level 2 table: each PTE points to a page (paged in and out like other data)
  
- Level 1 table stays in memory
- Level 2 tables paged in and out

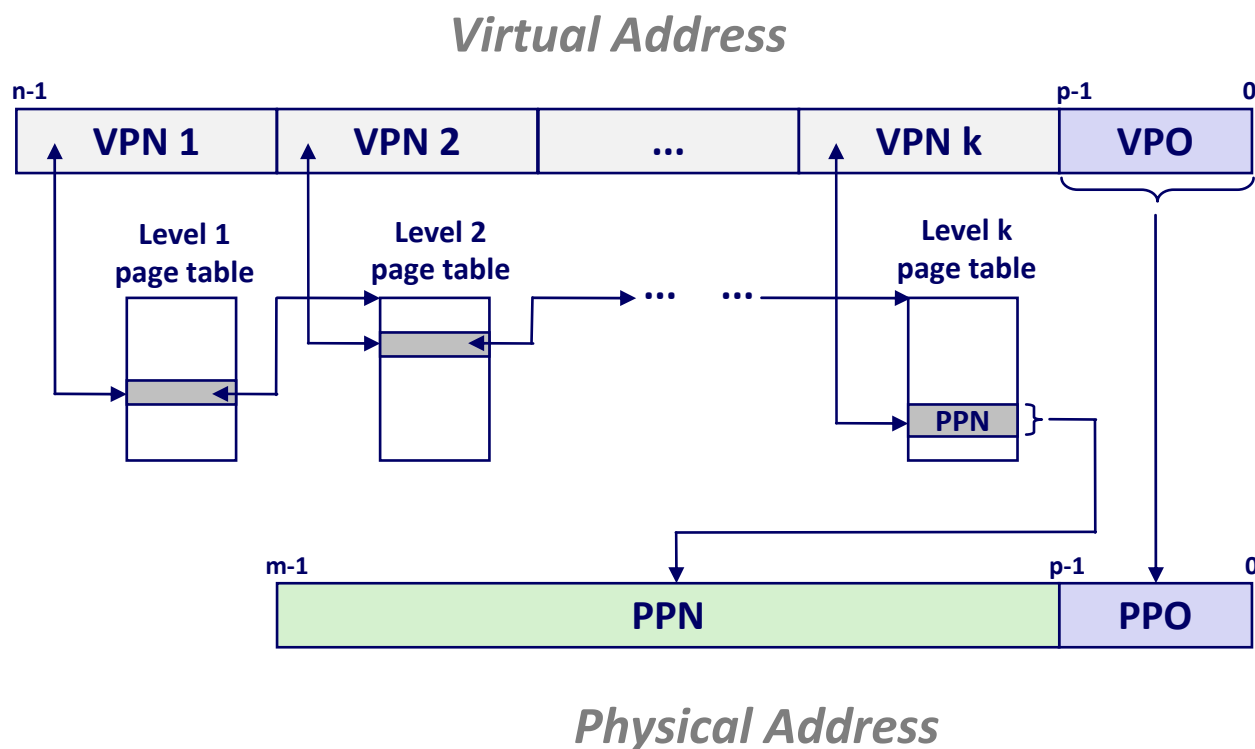


# A Two-Level Page Table Hierarchy

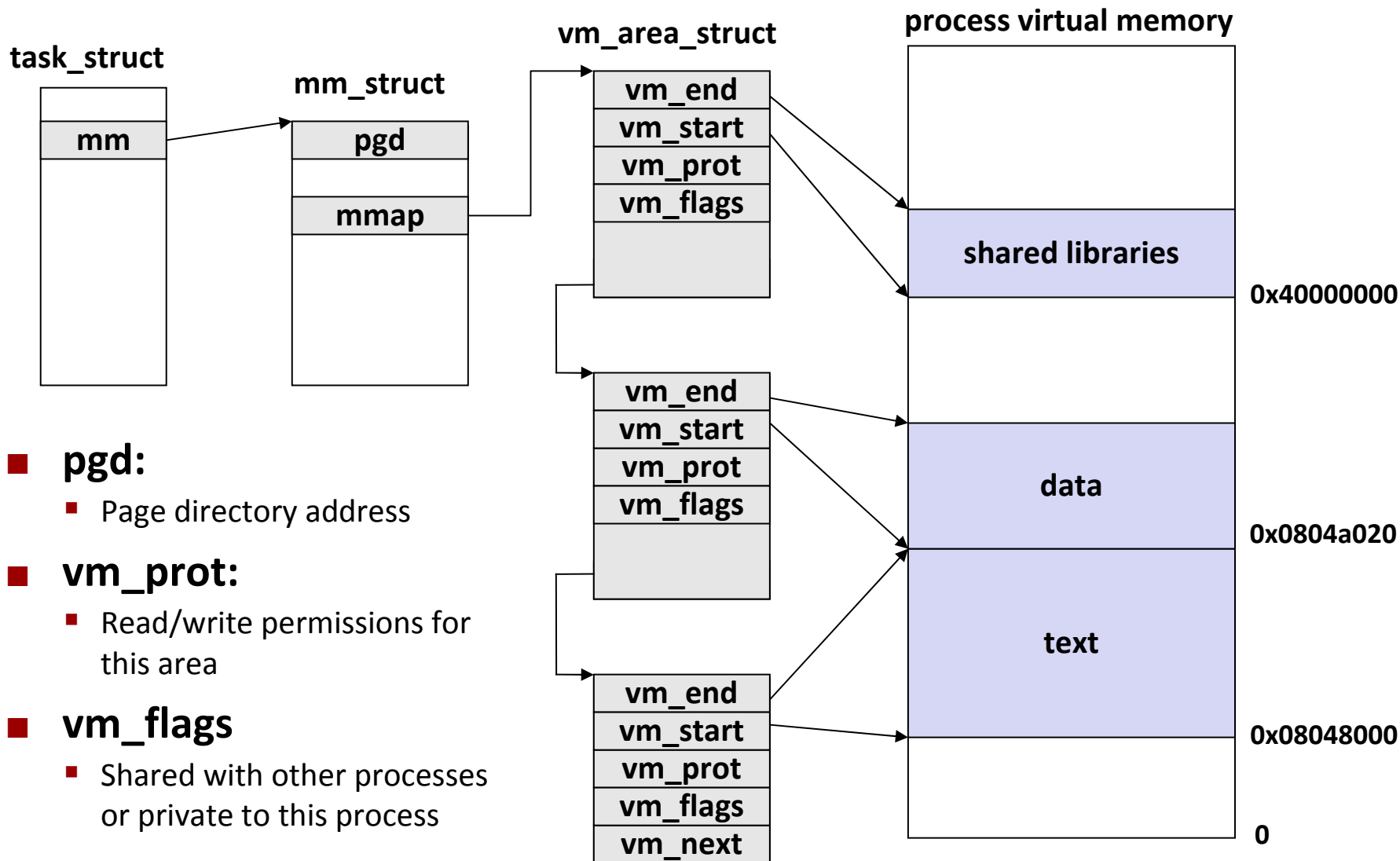




# Translating with a k-level Page Table

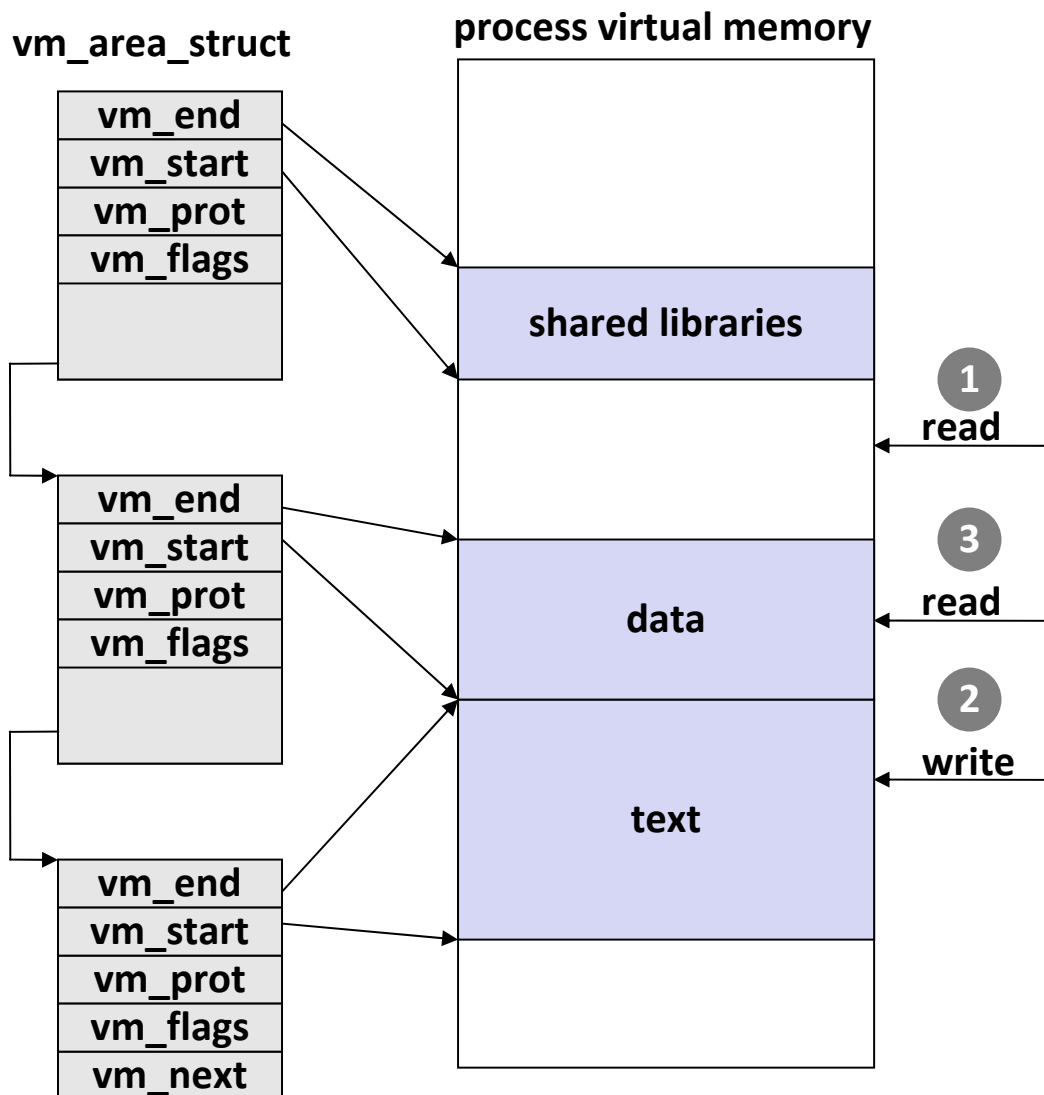


# Linux Organizes VM as Collection of “Areas”



- **pgd:**
  - Page directory address
- **vm\_prot:**
  - Read/write permissions for this area
- **vm\_flags**
  - Shared with other processes or private to this process

# Linux Page Fault Handling



## ■ Is the VA legal?

- = Is it in an area defined by a `vm_area_struct`?
- If not (#1), then signal segmentation violation

## ■ Is the operation legal?

- i.e., Can the process read/write this area?
- If not (#2), then signal protection violation

## ■ Otherwise

- Valid address (#3): handle fault

# Today

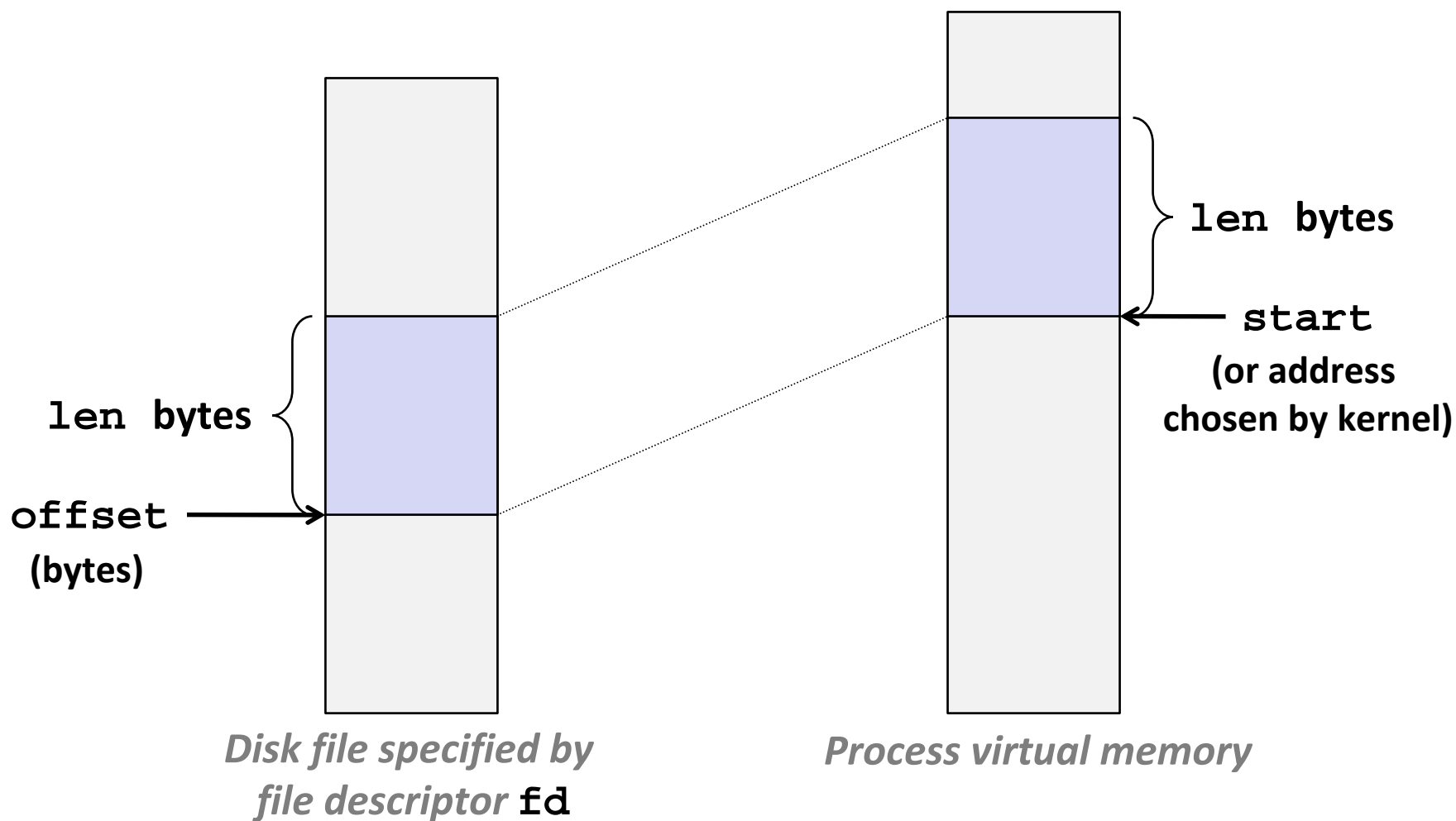
- **Virtual memory (VM)**
  - Address translation
  - Allocation
  - Multi-level page tables
- **Linux VM system**
- **Case study: VM system on P6**

# Memory Mapping

- **Creation of new VM *area* done via “memory mapping”**
  - Create new `vm_area_struct` and page tables for area
- **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
  - Nothing (e.g., `.bss`)
    - First fault will allocate a physical page full of 0's (demand-zero)
    - Once the page is written to (dirtyed), it is like any other page
- **Dirty pages are swapped back and forth between a special swap file.**
- ***Key point:* no virtual pages are copied into physical memory until they are referenced!**
  - Known as “demand paging”
  - Crucial for time and space efficiency

# User-Level Memory Mapping

```
void *mmap(void *start, int len,  
           int prot, int flags, int fd, int offset)
```



# 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_PRIVATE`, `MAP_SHARED`, ...
- **Return a pointer to start of mapped area (may not be `start`)**
- **Example: fast file-copy**
  - Useful for applications like Web servers that need to quickly copy files.
  - `mmap( )` allows file transfers without copying into user space.

# mmap() Example: Fast File Copy

```
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/*
 * a program that uses mmap to copy
 * the file input.txt to stdout
 */
```

```
int main() {
    struct stat stat;
    int i, fd, size;
    char *bufp;

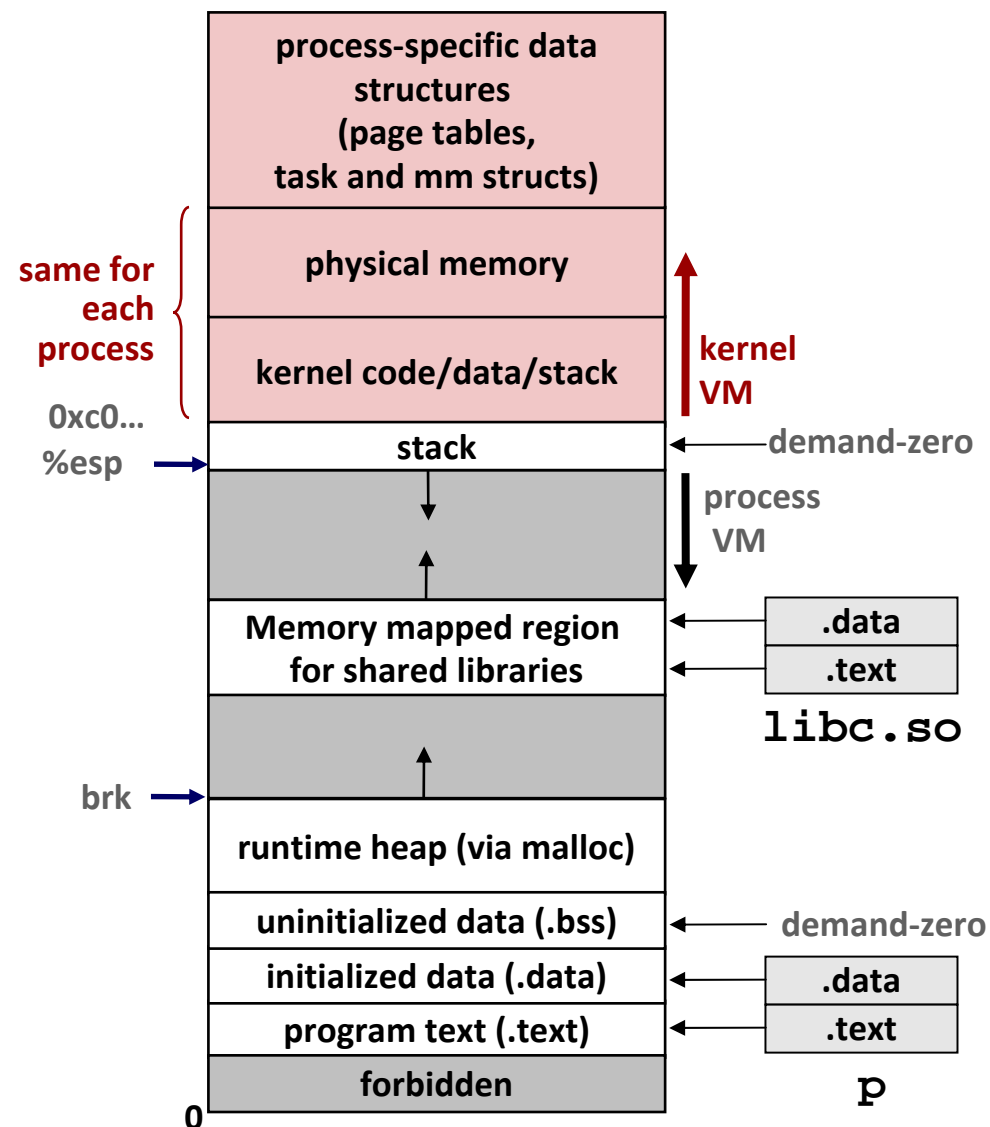
    /* open the file & get its size*/
    fd = open("./input.txt", O_RDONLY);
    fstat(fd, &stat);
    size = stat.st_size;

    /* map the file to a new VM area */
    bufp = mmap(0, size, PROT_READ,
                MAP_PRIVATE, fd, 0);

    /* write the VM area to stdout */
    write(1, bufp, size);
    exit(0);
}
```



# Exec() Revisited



To run a new program `p` in the current process using `exec()`:

- Free `vm_area_struct`'s and page tables for old areas
- Create new `vm_area_struct`'s and page tables for new areas
  - Stack, BSS, data, text, shared libs.
  - Text and data backed by ELF executable object file
  - BSS and stack initialized to zero
- Set PC to entry point in `.text`
  - Linux will fault in code, data pages as needed

# Fork() Revisited

## ■ To create a new process using `fork()`:

- Make copies of the old process's `mm_struct`, `vm_area_struct`'s, and page tables.
  - At this point the two processes share all of their pages.
  - How to get separate spaces without copying all the virtual pages from one space to another?
    - “Copy on Write” (COW) technique.
- Copy-on-write
  - Mark PTE's of writeable areas as read-only
  - Writes by either process to these pages will cause page faults
  - Flag `vm_area_struct`'s for these areas as private “copy-on-write”
    - Fault handler recognizes copy-on-write, makes a copy of the page, and restores write permissions.

## ■ Net result:

- Copies are deferred until absolutely necessary (i.e., when one of the processes tries to modify a shared page).

# Memory System Summary

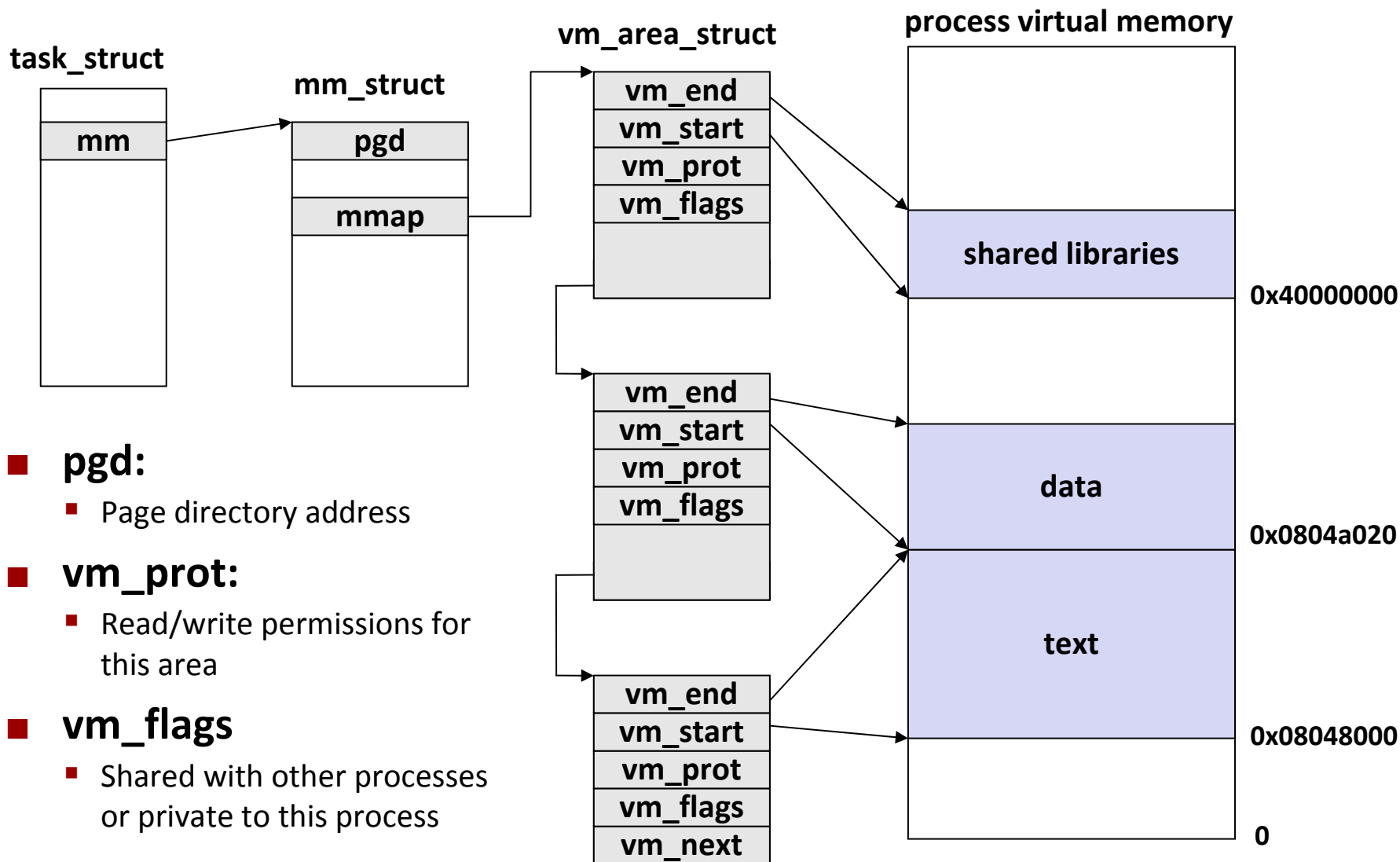
## ■ L1/L2 Memory Cache

- Purely a speed-up technique
- Behavior invisible to application programmer and (mostly) OS
- Implemented totally in hardware

## ■ Virtual Memory

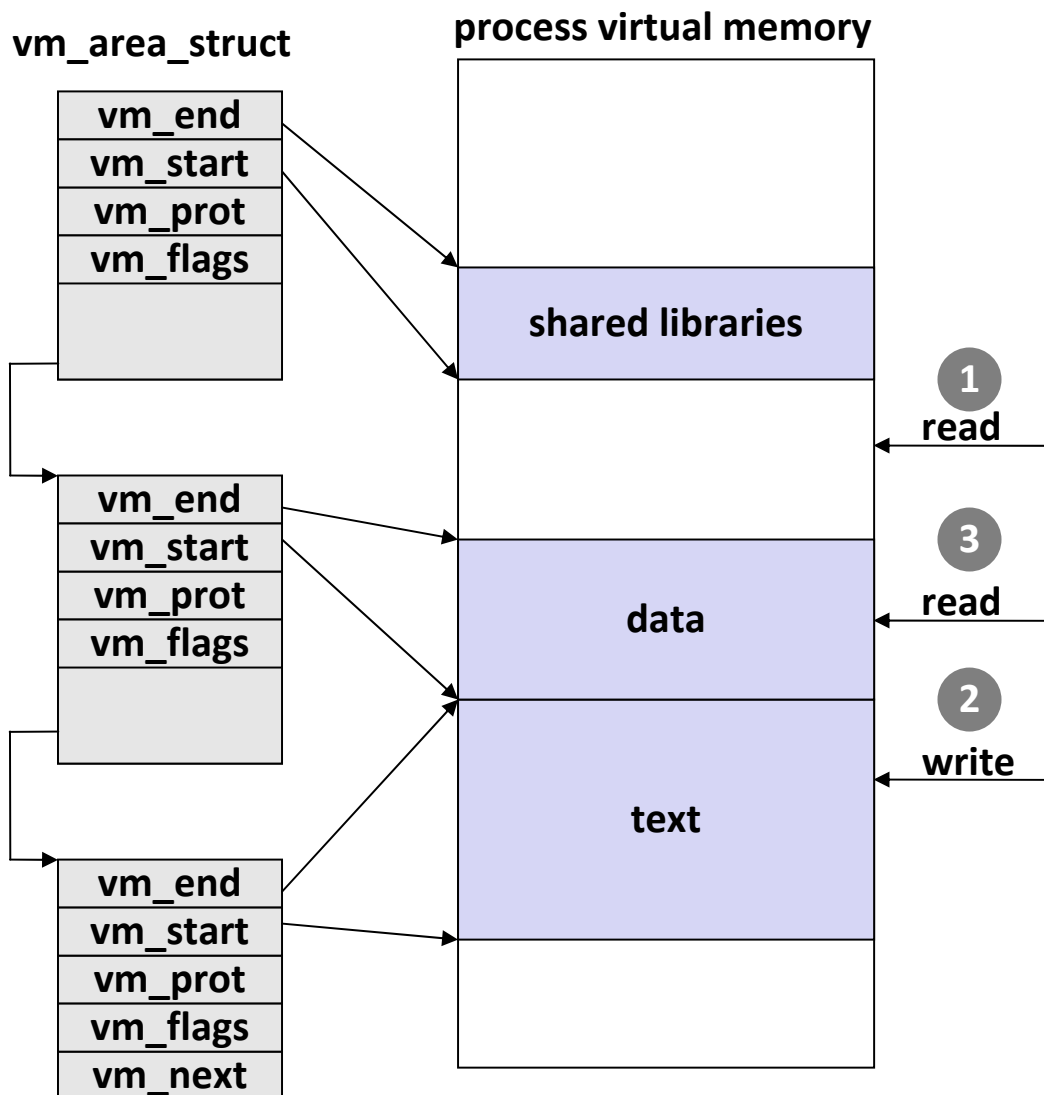
- Supports many OS-related functions
  - Process creation, task switching, protection
- Software
  - Allocates/shares physical memory among processes
  - Maintains high-level tables tracking memory type, source, sharing
  - Handles exceptions, fills in hardware-defined mapping tables
- Hardware
  - Translates virtual addresses via mapping tables, enforcing permissions
  - Accelerates mapping via translation cache (TLB)

# Linux Organizes VM as Collection of “Areas”



- **pgd:**
  - Page directory address
- **vm\_prot:**
  - Read/write permissions for this area
- **vm\_flags**
  - Shared with other processes or private to this process

# Linux Page Fault Handling



## ■ Is the VA legal?

- = Is it in an area defined by a `vm_area_struct`?
- If not (#1), then signal segmentation violation

## ■ Is the operation legal?

- i.e., Can the process read/write this area?
- If not (#2), then signal protection violation

## ■ Otherwise

- Valid address (#3): handle fault

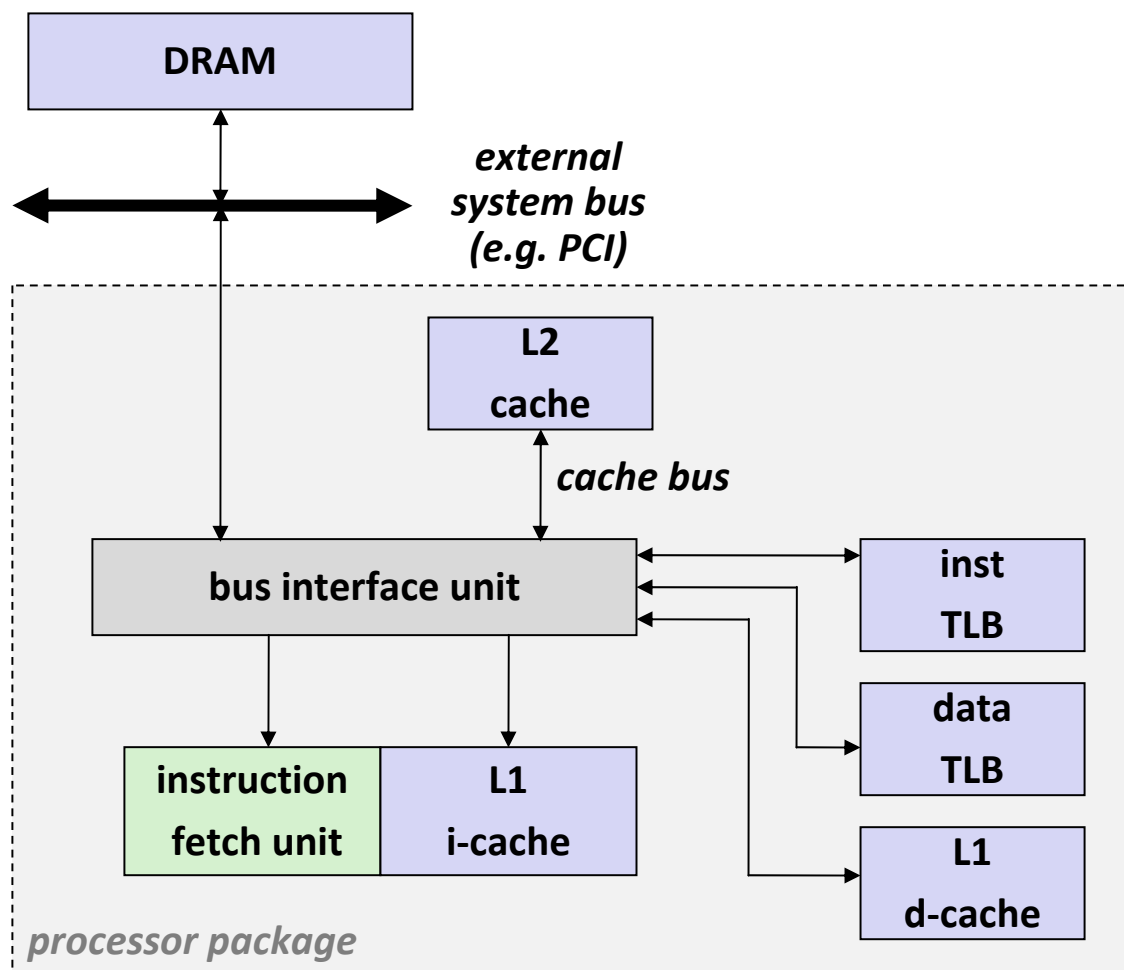
# Today

- **Virtual memory (VM)**
  - Address translation
  - Allocation
  - Multi-level page tables
- **Linux VM system**
- **Case study: VM system on P6**

# Intel P6 (Bob Colwell's Chip, CMU Alumni)

- **Internal designation for successor to Pentium**
  - Which had internal designation P5
- **Fundamentally different from Pentium**
  - Out-of-order, superscalar operation
- **Resulting processors**
  - Pentium Pro (1996)
  - Pentium II (1997)
    - L2 cache on same chip
  - Pentium III (1999)
    - The freshwater fish machines
- **Saltwater fish machines: Pentium 4**
  - Different operation, but similar memory system
  - Abandoned by Intel in 2005 for P6-based Core 2 Duo

# P6 Memory System



**32 bit address space**

**4 KB page size**

**L1, L2, and TLBs**

- 4-way set associative

**Inst TLB**

- 32 entries
- 8 sets

**Data TLB**

- 64 entries
- 16 sets

**L1 i-cache and d-cache**

- 16 KB
- 32 B line size
- 128 sets

**L2 cache**

- unified
- 128 KB–2 MB



# Review of Abbreviations

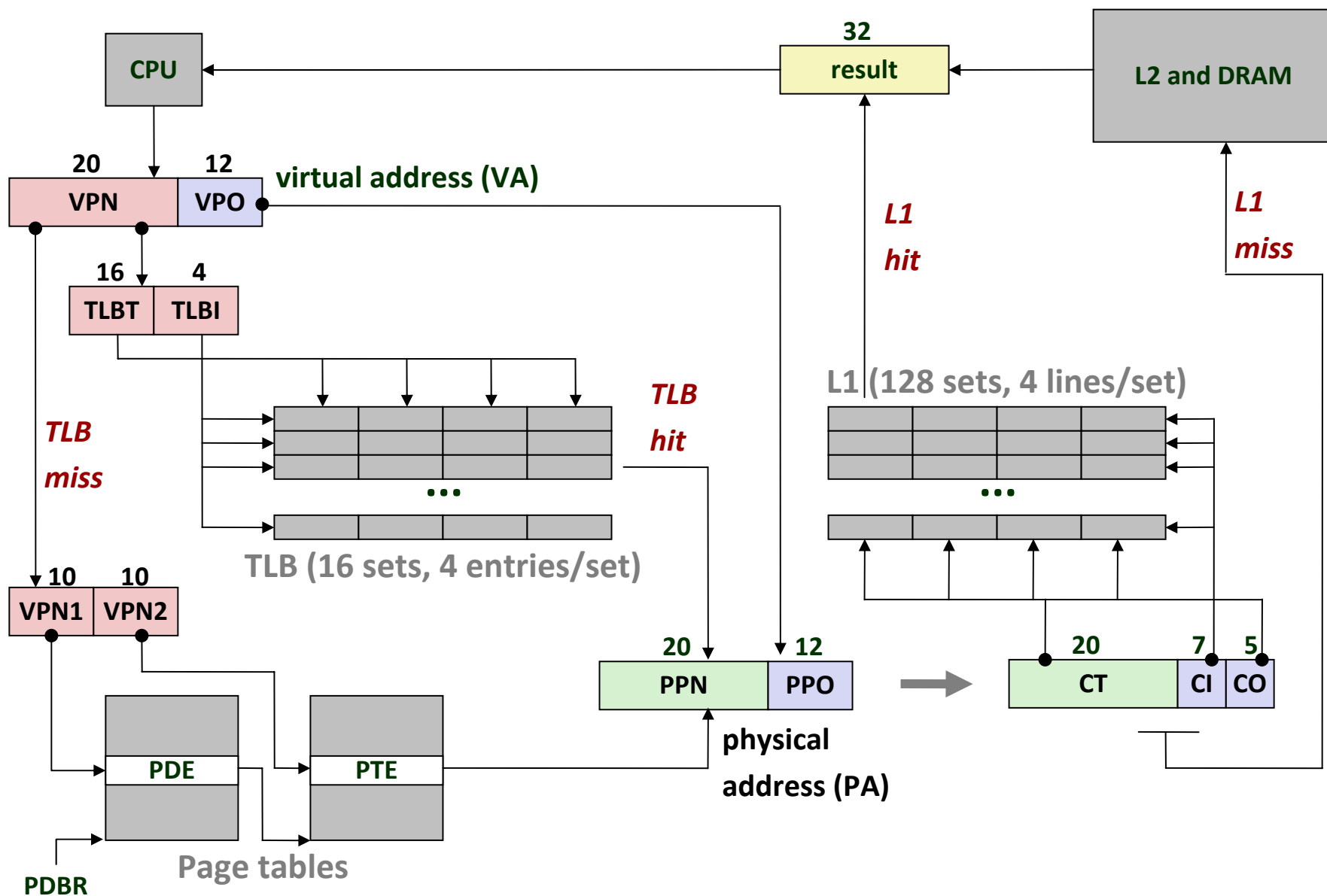
## ■ Components of the virtual address (VA)

- 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

# Overview of P6 Address Translation



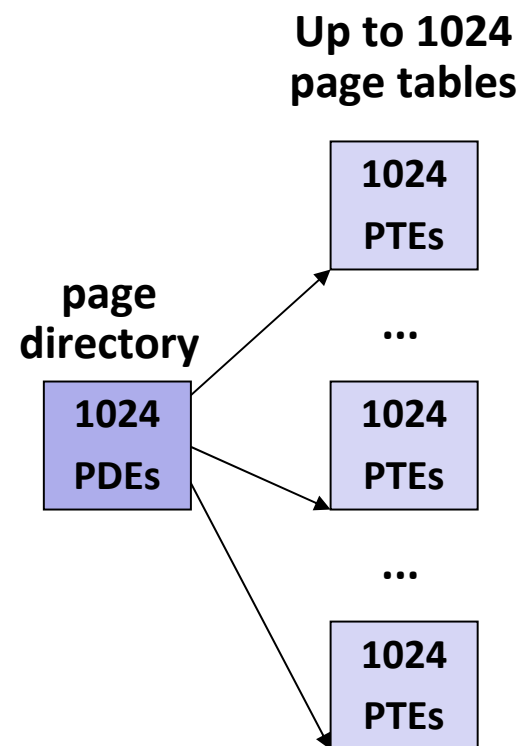
# P6 2-level Page Table Structure

## ■ Page directory

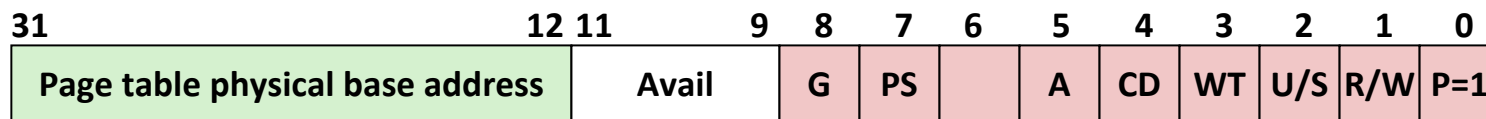
- 1024 4-byte page directory entries (PDEs) that point to page tables
- One page directory per process
- Page directory must be in memory when its process is running
- Always pointed to by PD BR
- Large page support:
  - Make PD the page table
  - Fixes page size to 4MB (why?)

## ■ Page tables:

- 1024 4-byte page table entries (PTEs) that point to pages
- Size: exactly one page
- Page tables can be paged in and out



# P6 Page Directory Entry (PDE)



**Page table physical base address:** 20 most significant bits of physical page table address (forces page tables to be 4KB aligned)

**Avail:** These bits available for system programmers

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

**PS:** page size 4K (0) or 4M (1)

**A:** accessed (set by MMU on reads and writes, cleared by software)

**CD:** cache disabled (1) or enabled (0)

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

**U/S:** user or supervisor mode access

**R/W:** read-only or read-write access

**P:** page table is present in memory (1) or not (0)



# P6 Page Table Entry (PTE)

31	12 11	9	8	7	6	5	4	3	2	1	0
Page physical base address		Avail	G	0	D	A	CD	WT	U/S	R/W	P=1

**Page base address:** 20 most significant bits of physical page address (forces pages to be 4 KB aligned)

**Avail:** available for system programmers

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

**D:** dirty (set by MMU on writes)

**A:** accessed (set by MMU on reads and writes)

**CD:** cache disabled or enabled

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

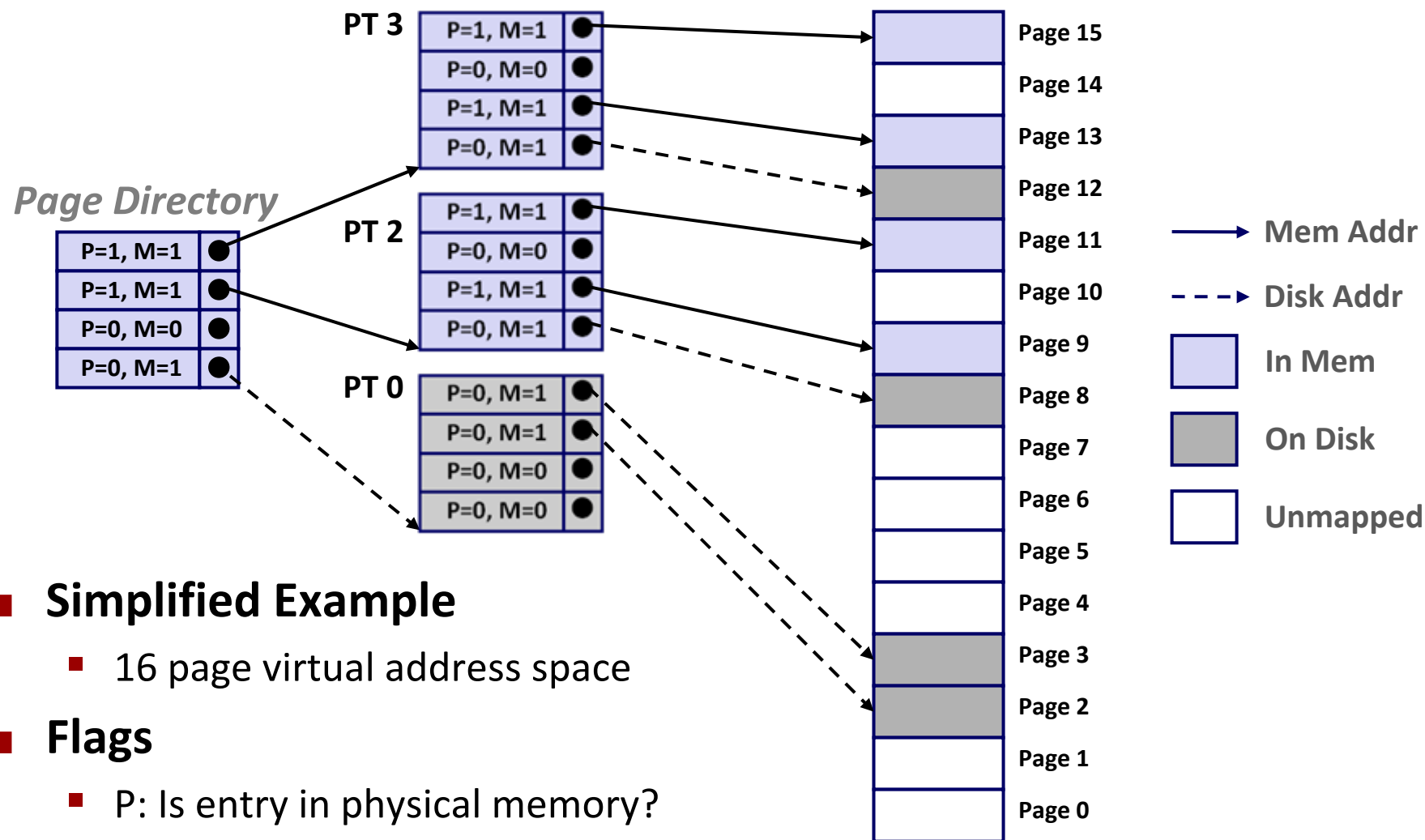
**U/S:** user/supervisor

**R/W:** read/write

**P:** page is present in physical memory (1) or not (0)

31	1	0
Available for OS (page location in secondary storage)		P=0

# Representation of VM Address Space



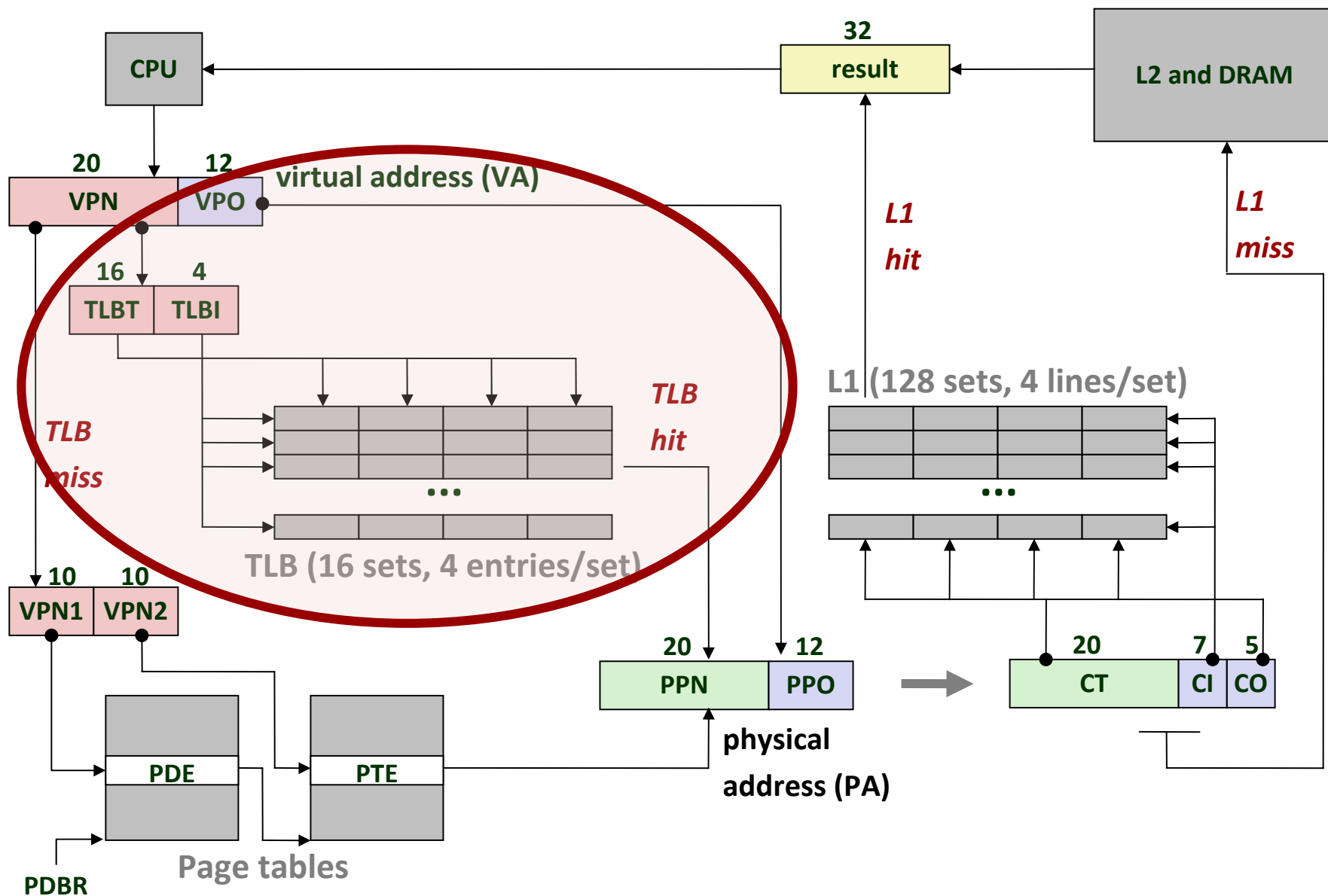
## ■ Simplified Example

- 16 page virtual address space

## ■ Flags

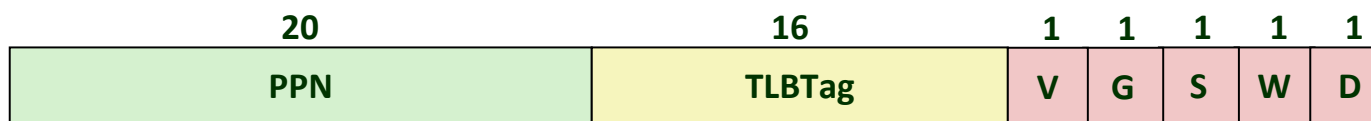
- P: Is entry in physical memory?
- M: Has this part of VA space been mapped?

# P6 TLB Translation



# P6 TLB

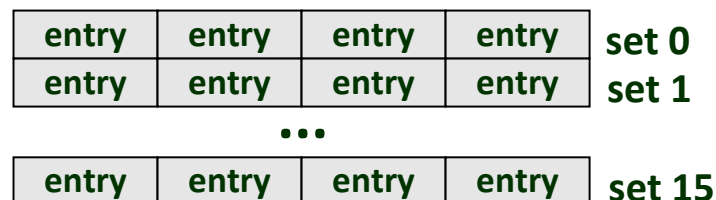
## ■ TLB entry (not all documented, so this is speculative):



- **V:** indicates a valid (1) or invalid (0) TLB entry
- **TLBTag:** disambiguates entries cached in the same set
- **PPN:** translation of the address indicated by index & tag
- **G:** page is “global” according to PDE, PTE
- **S:** page is “supervisor-only” according to PDE, PTE
- **W:** page is writable according to PDE, PTE
- **D:** PTE has already been marked “dirty” (once is enough)

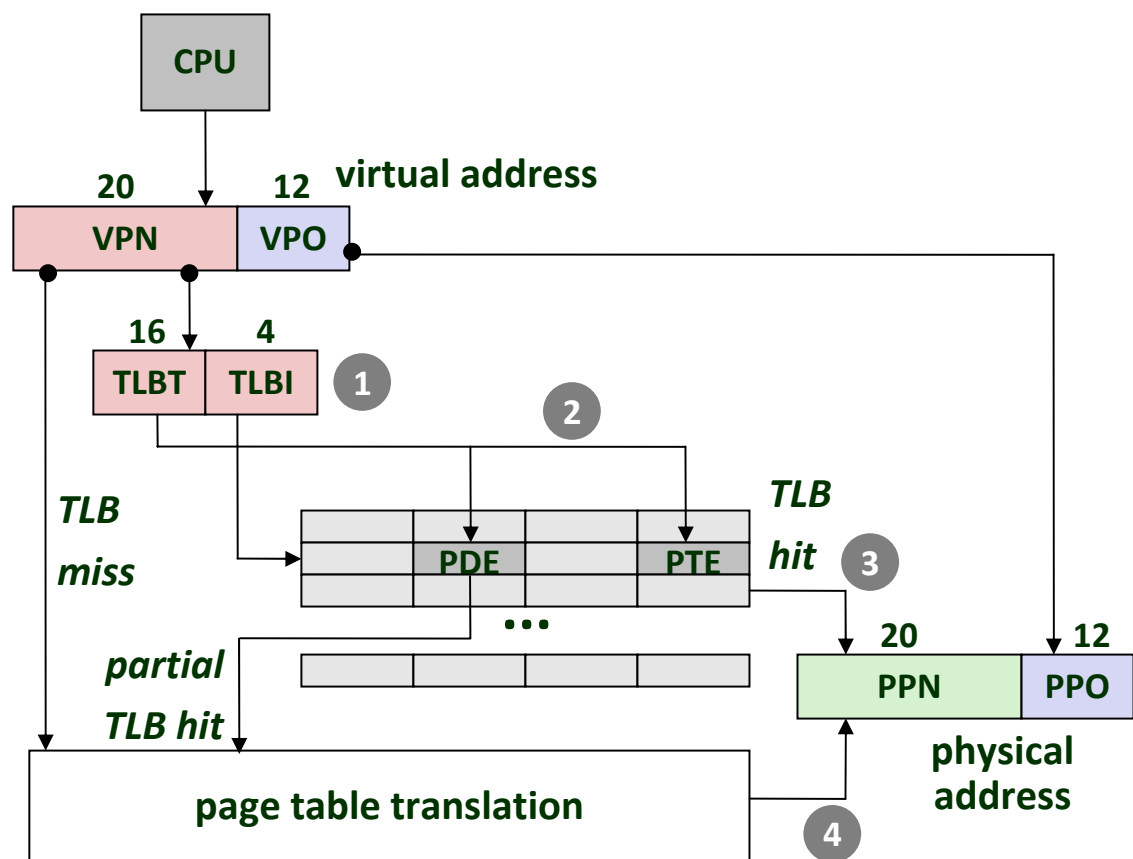
## ■ Structure of the data TLB:

- 16 sets, 4 entries/set





# Translating with the P6 TLB



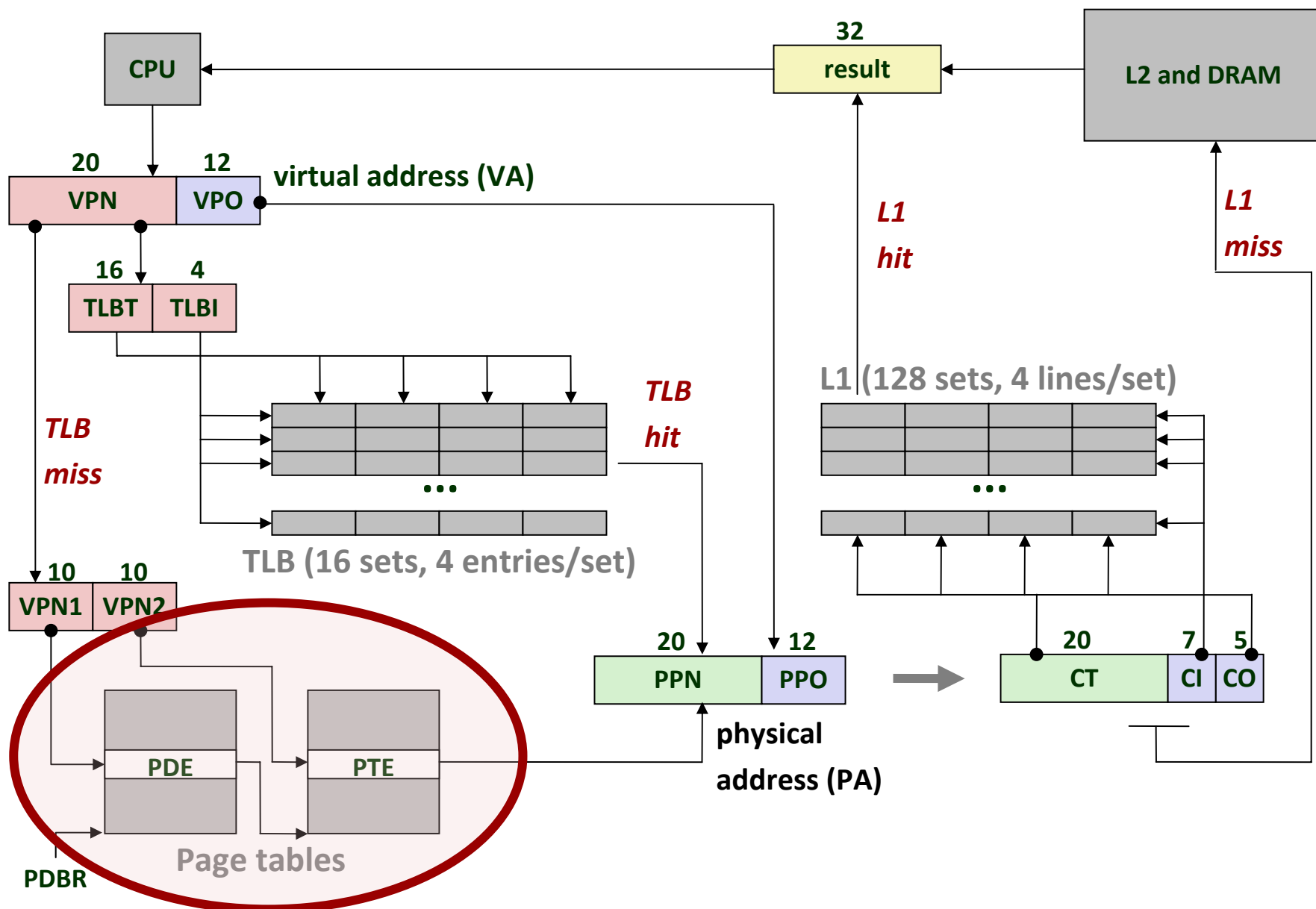
1. Partition VPN into TLBT and TLBI.

2. Is the PTE for VPN cached in set TLBI?

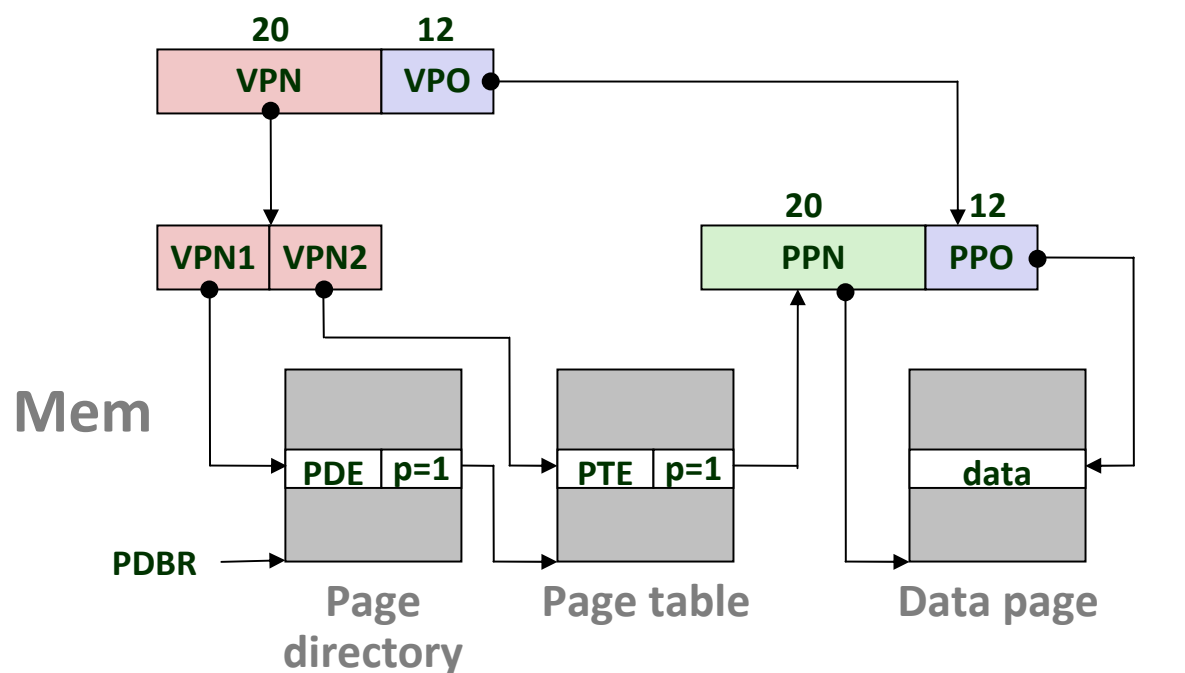
3. **Yes:** Check permissions, build physical address

4. **No:** Read PTE (and PDE if not cached) from memory and build physical address

# P6 TLB Translation



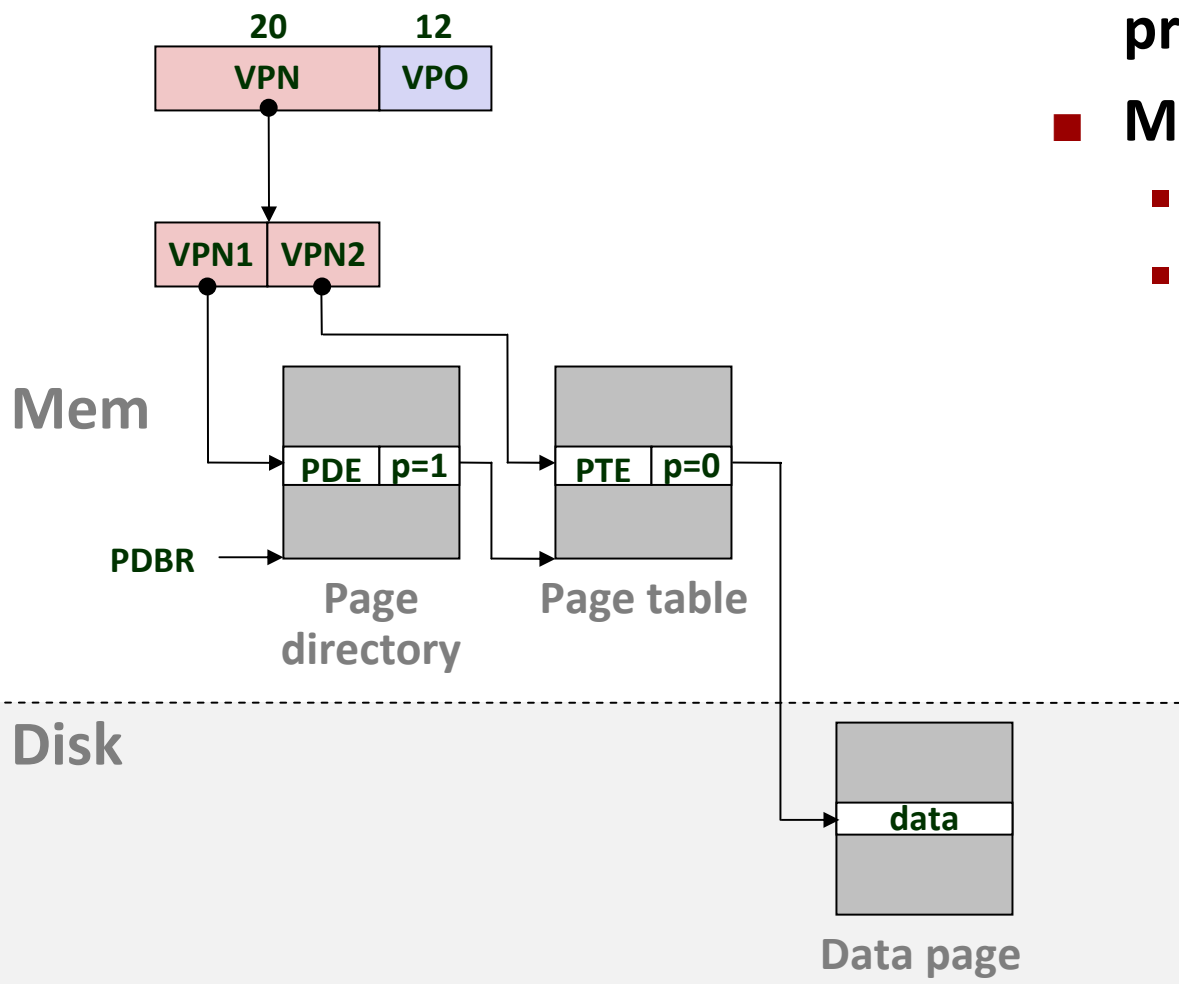
# Translating with the P6 Page Tables (case 1/1)



**Disk**

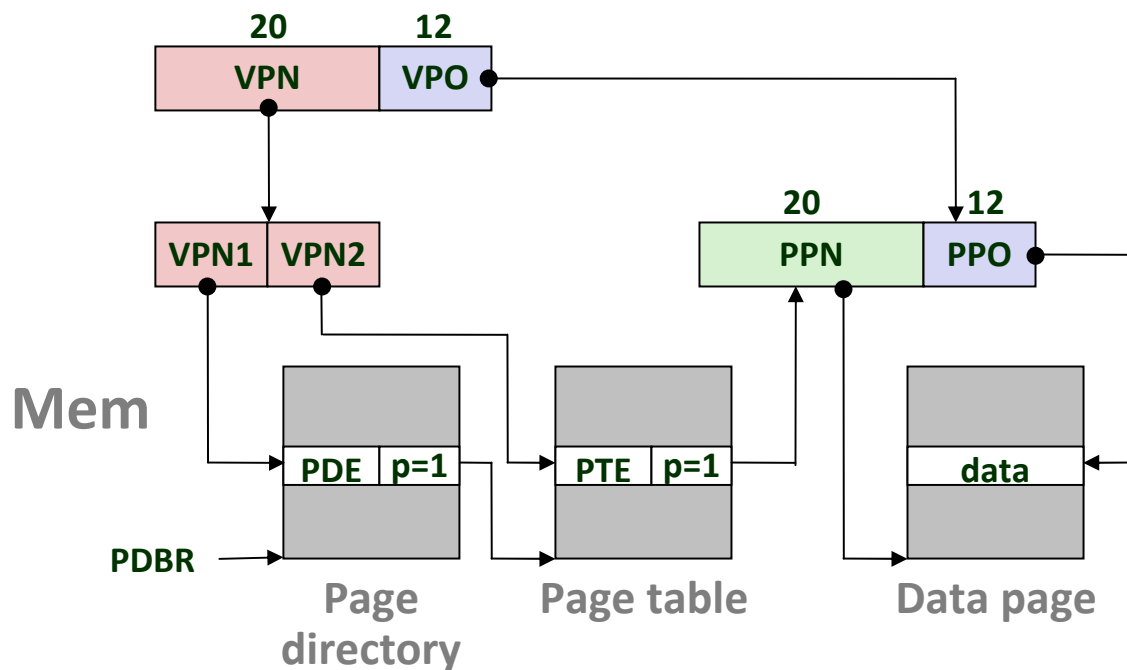
- **Case 1/1: page table and page present**
- **MMU Action:**
  - MMU builds physical address and fetches data word
- **OS action**
  - None

# Translating with the P6 Page Tables (case 1/0)



- **Case 1/0: page table present, page missing**
- **MMU Action:**
  - Page fault exception
  - Handler receives the following args:
    - %eip that caused fault
    - VA that caused fault
    - Fault caused by non-present page or page-level protection violation
      - Read/write
      - User/supervisor

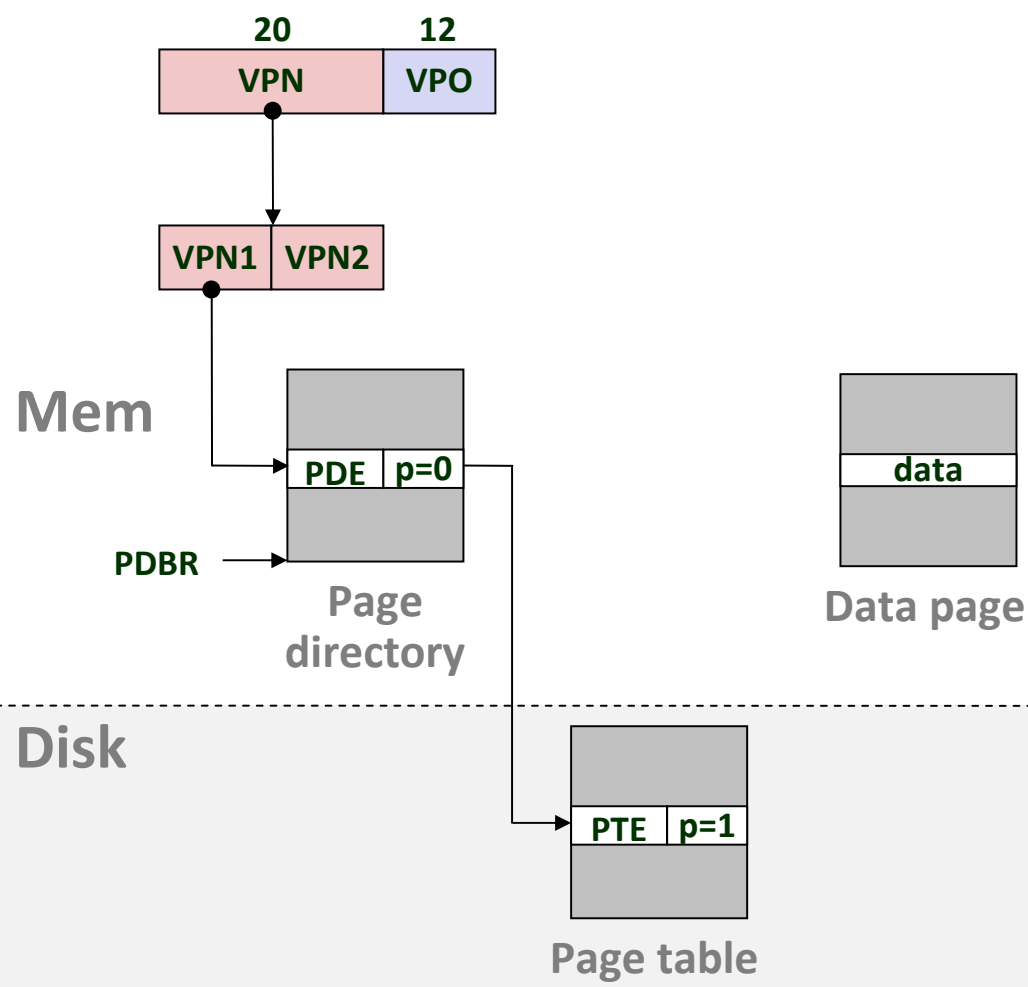
# Translating with the P6 Page Tables (case 1/0, cont.)



## ■ OS Action:

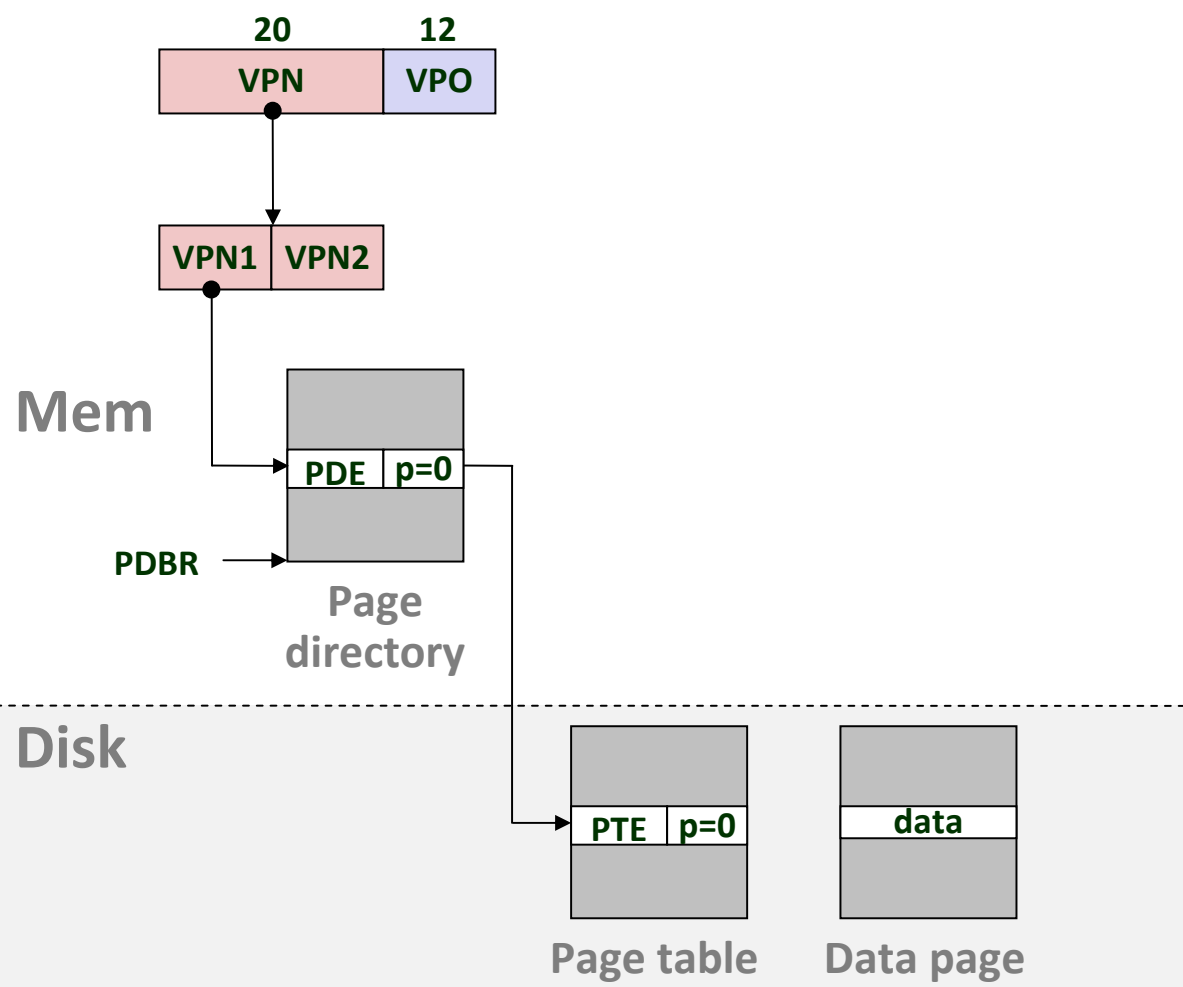
- Check for a legal virtual address.
- Read PTE through PDE.
- Find free physical page (swapping out current page if necessary)
- Read virtual page from disk into physical page
- Adjust PTE to point to physical page, set  $p=1$
- Restart faulting instruction by returning from exception handler

# Translating with the P6 Page Tables (case 0/1)



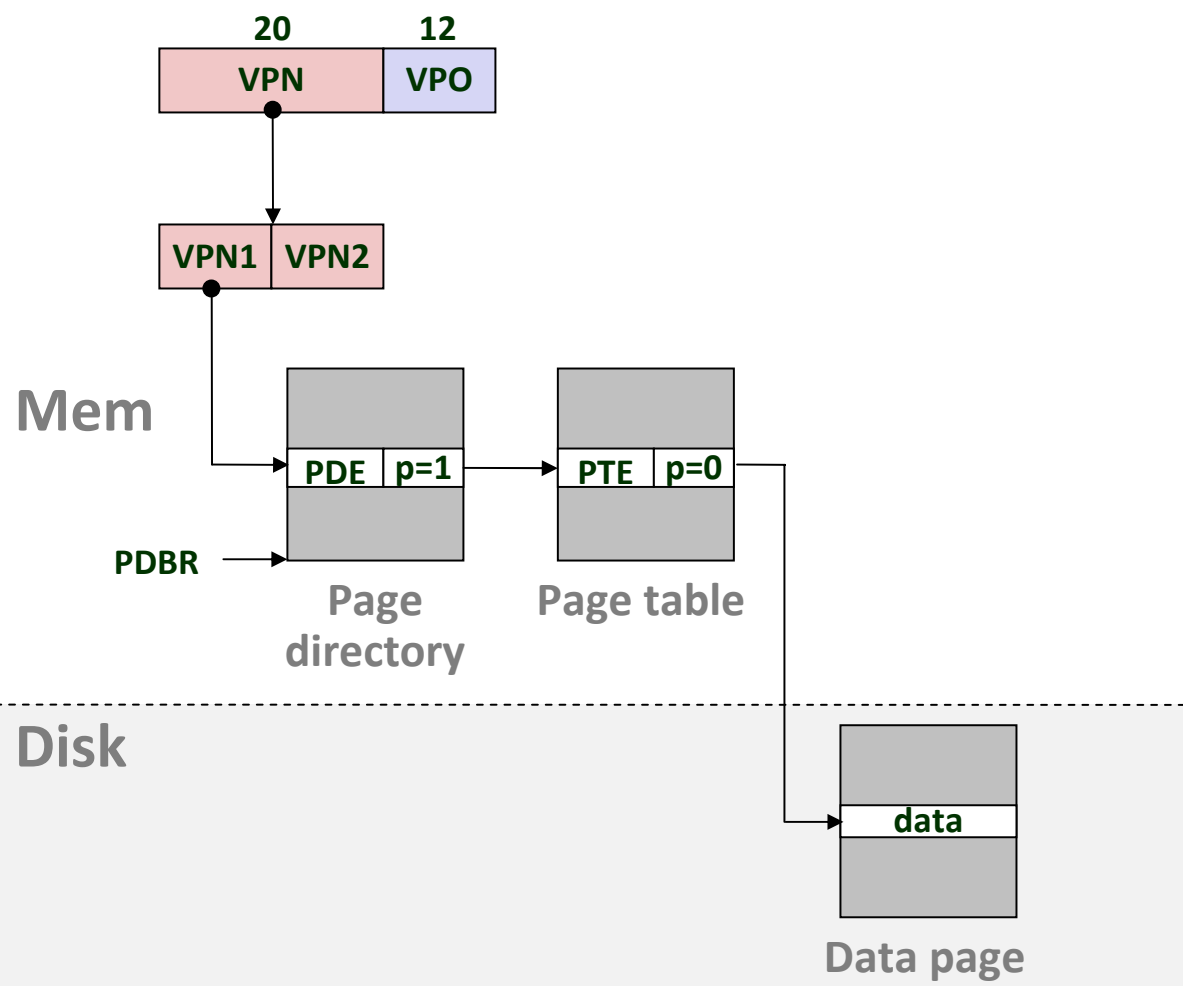
- **Case 0/1: page table missing, page present**
- **Introduces consistency issue**
  - Potentially every page-out requires update of disk page table
- **Linux disallows this**
  - If a page table is swapped out, then swap out its data pages too

# Translating with the P6 Page Tables (case 0/0)



- **Case 0/0: page table and page missing**
- **MMU Action:**
  - Page fault

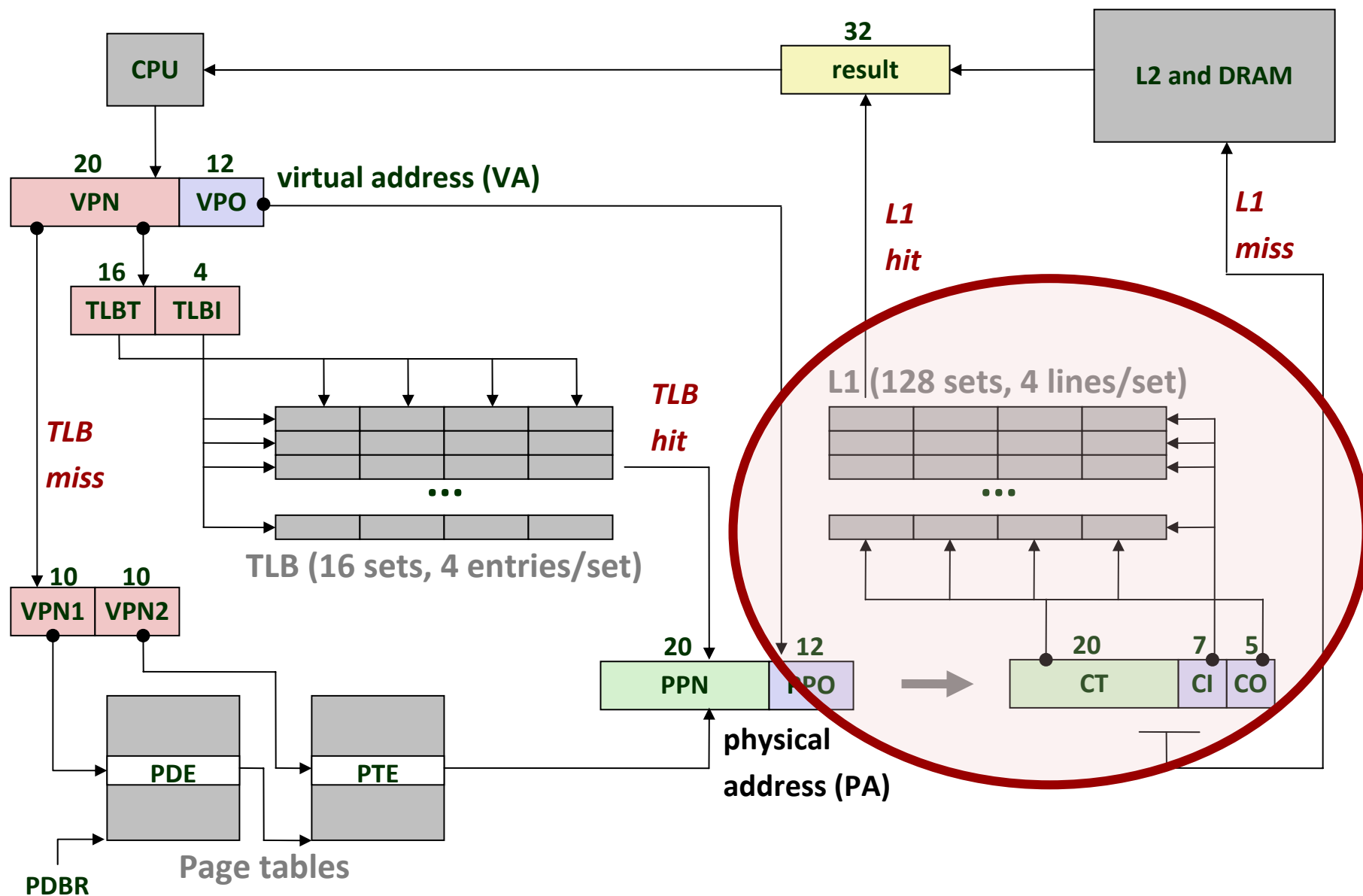
# Translating with the P6 Page Tables (case 0/0, cont.)



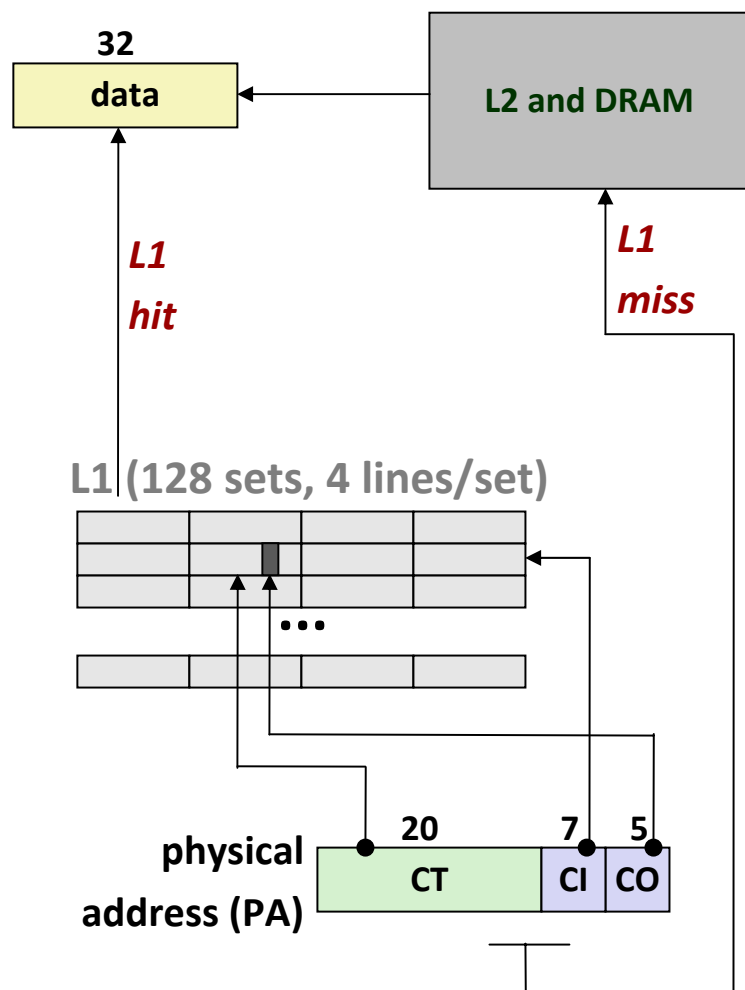
- **OS action:**
  - Swap in page table
  - Restart faulting instruction by returning from handler
- **Like case 0/1 from here on.**
  - Two disk reads



# P6 L1 Cache Access

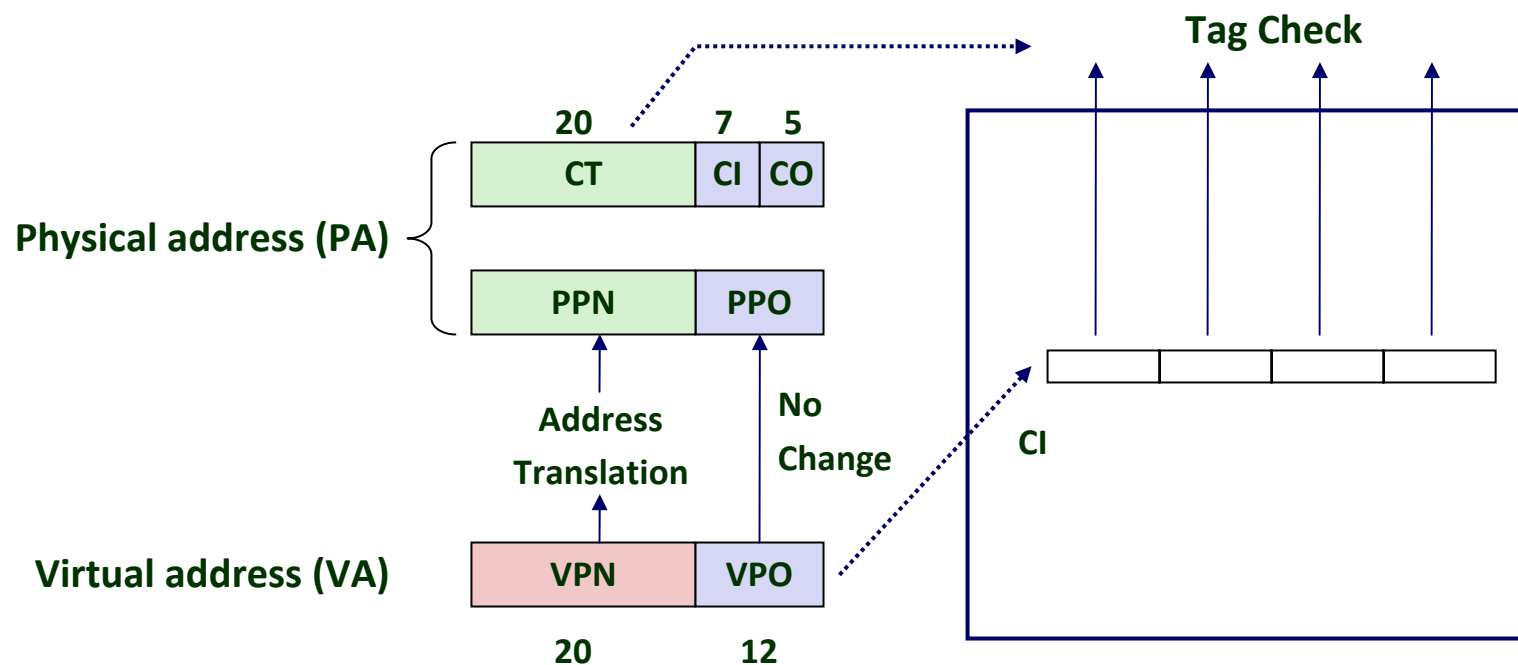


# L1 Cache Access



- Partition physical address: CO, CI, and CT
- Use CT to determine if line containing word at address PA is cached in set CI
- **No:** check L2
- **Yes:** extract word at byte offset CO and return to processor

# 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

# x86-64 Paging

## ■ Origin

- AMD's way of extending x86 to 64-bit instruction set
- Intel has followed with "EM64T"

## ■ Requirements

- 48-bit virtual address
  - 256 terabytes (TB)
  - Not yet ready for full 64 bits
    - Nobody can buy that much DRAM yet
    - Mapping tables would be huge
    - Multi-level array map may not be the right data structure
- 52-bit physical address = 40 bits for PPN
  - Requires 64-bit table entries
- Keep traditional x86 4KB page size, and same size for page tables
  - $(4096 \text{ bytes per PT}) / (8 \text{ bytes per PTE}) = \text{only } 512 \text{ entries per page}$

# x86-64 Paging

