

Virtual Memory: Concepts and System

CS-281: Introduction to Computer Systems

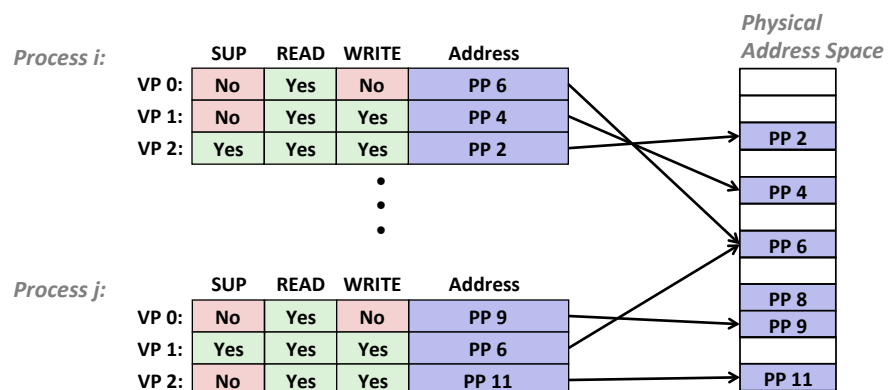
Instructor:

Dr. Thomas C. Bressoud

1

VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- Page fault handler checks these before remapping
 - If violated, send process SIGSEGV (segmentation fault)



2

VM Address Translation

- **Virtual Address Space**

- $V = \{0, 1, \dots, N-1\}$

- **Physical Address Space**

- $P = \{0, 1, \dots, M-1\}$

- **Address Translation**

- $MAP: V \rightarrow P \cup \{\emptyset\}$
 - For virtual address a :
 - $MAP(a) = a'$ if data at virtual address a is at physical address a' in P
 - $MAP(a) = \emptyset$ if data at virtual address a is not in physical memory
 - Either invalid or stored on disk

3

Summary of Address Translation 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)

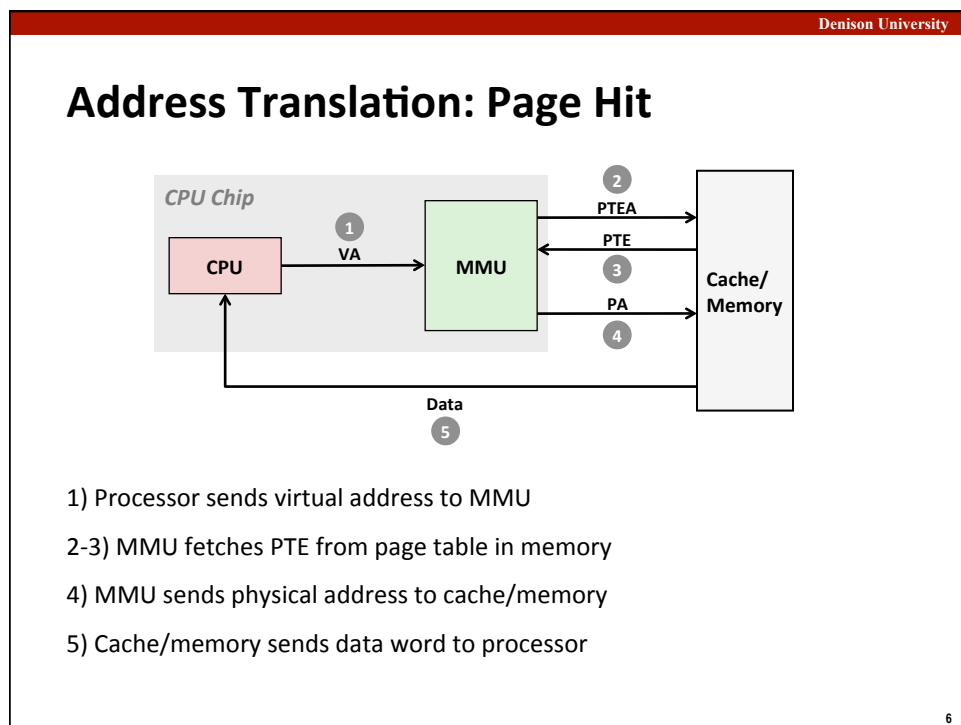
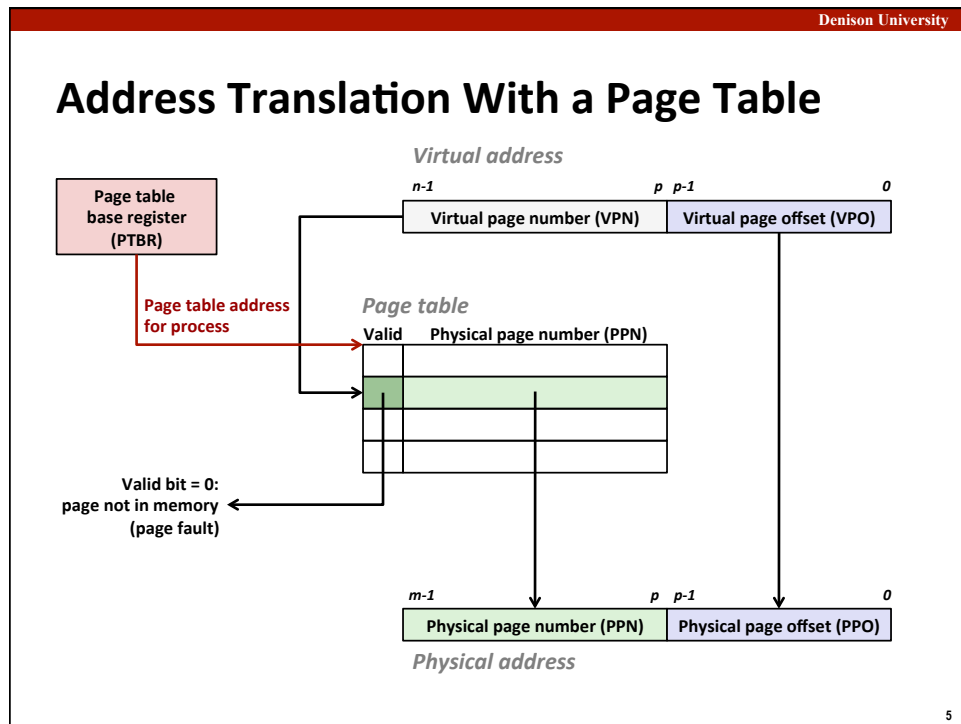
- **Components of the virtual address (VA)**

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

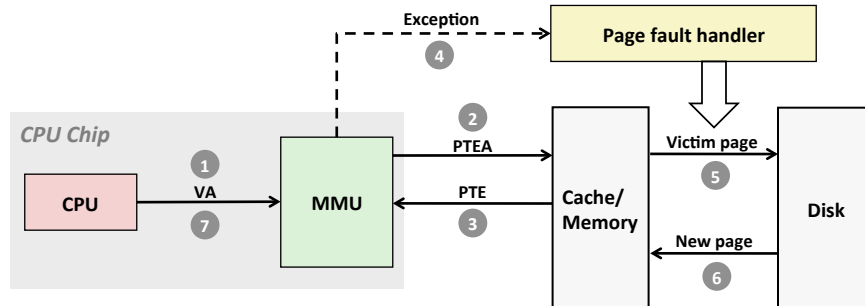
- **Components of the physical address (PA)**

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

4



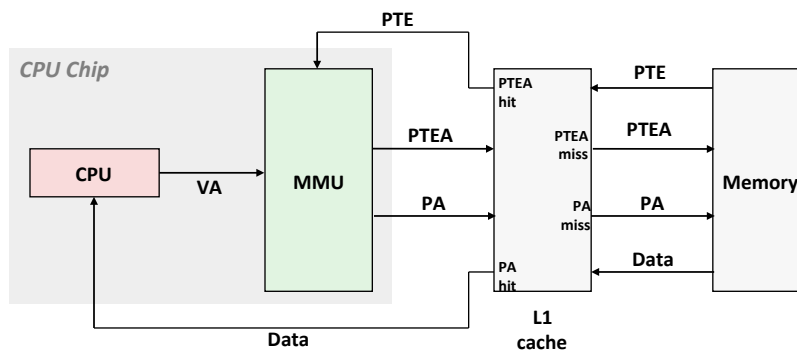
Address Translation: Page Fault



- 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

7

Integrating VM and Cache



VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

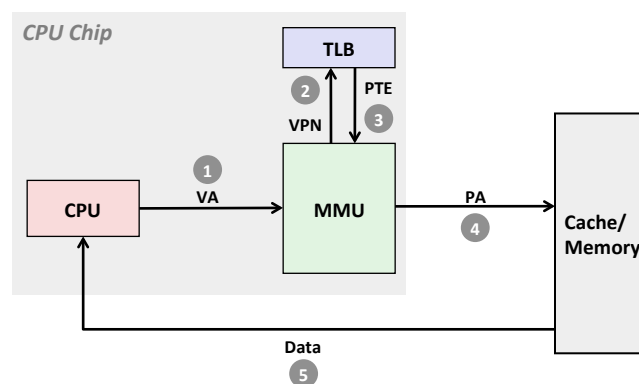
8

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 hardware cache in MMU
 - Maps virtual page numbers to physical page numbers
 - Contains complete page table entries for small number of pages

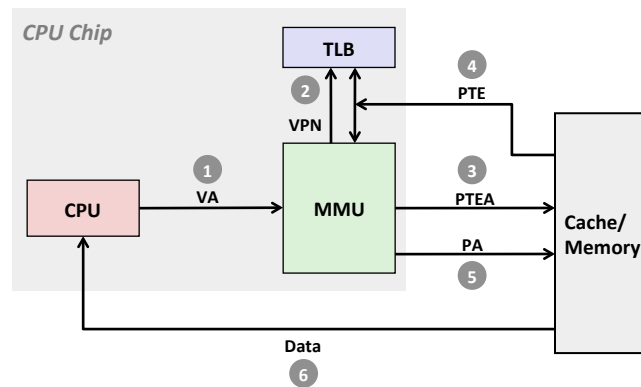
9

TLB Hit



10

TLB Miss



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

Fortunately, TLB misses are rare. Why?

11

Multi-Level Page Tables

■ Suppose:

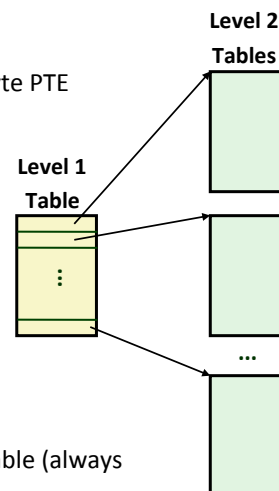
- 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE

■ Problem:

- Would need a 512 GB page table!
 - $2^{48} * 2^{-12} * 2^3 = 2^{39}$ bytes

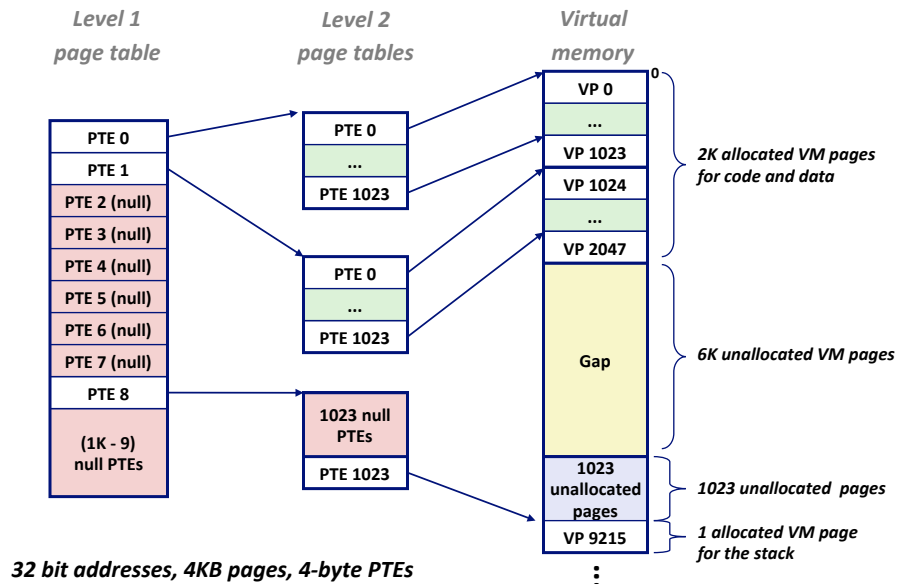
■ Common solution:

- 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)



12

A Two-Level Page Table Hierarchy



13

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

14

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)

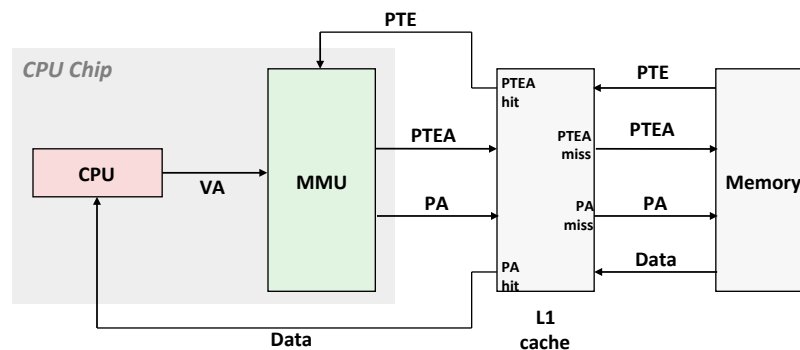
- TLBI: TLB index
- TLBT: TLB tag
- VPO: Virtual page offset
- VPN: Virtual page number

■ 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

15

Integrating VM and Cache



VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

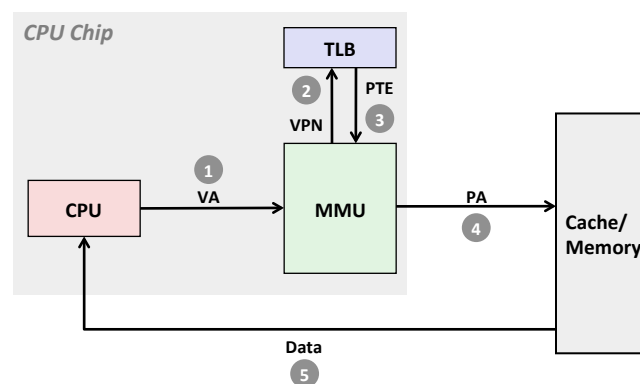
16

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 hardware cache in MMU
 - Maps virtual page numbers to physical page numbers
 - Contains complete page table entries for small number of pages

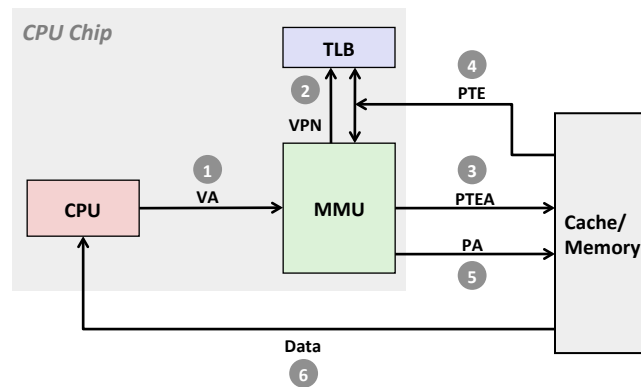
17

TLB Hit



18

TLB Miss



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

Fortunately, TLB misses are rare. Why?

19

Multi-Level Page Tables

■ Suppose:

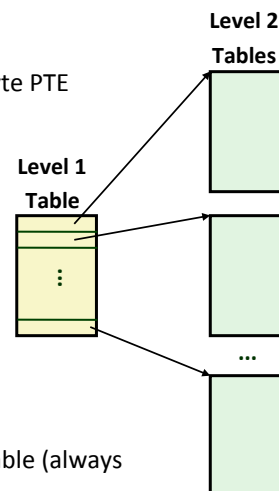
- 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE

■ Problem:

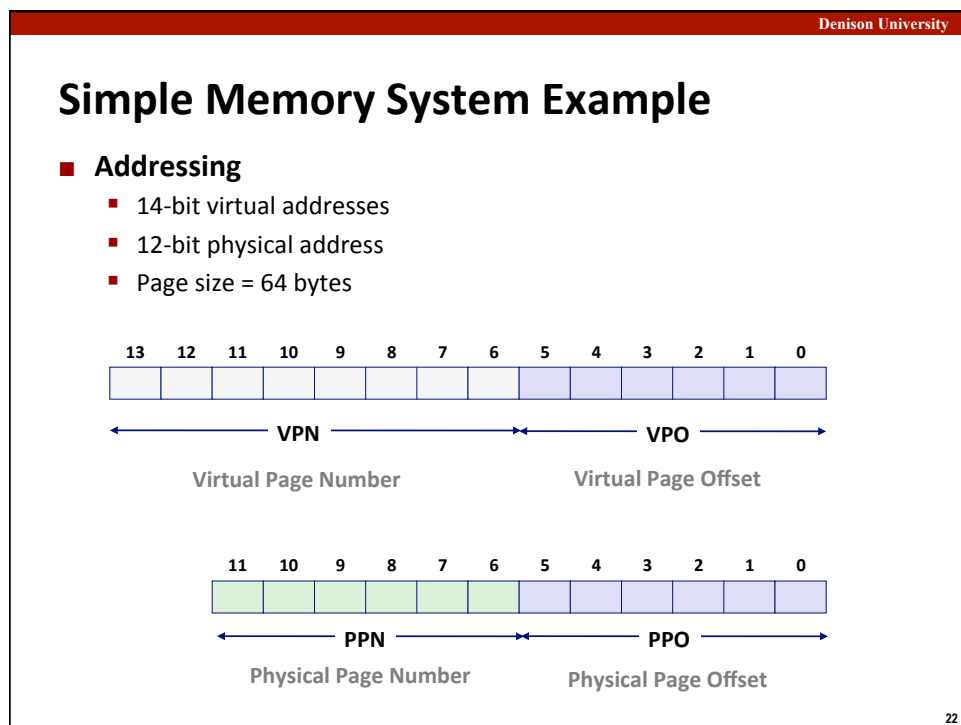
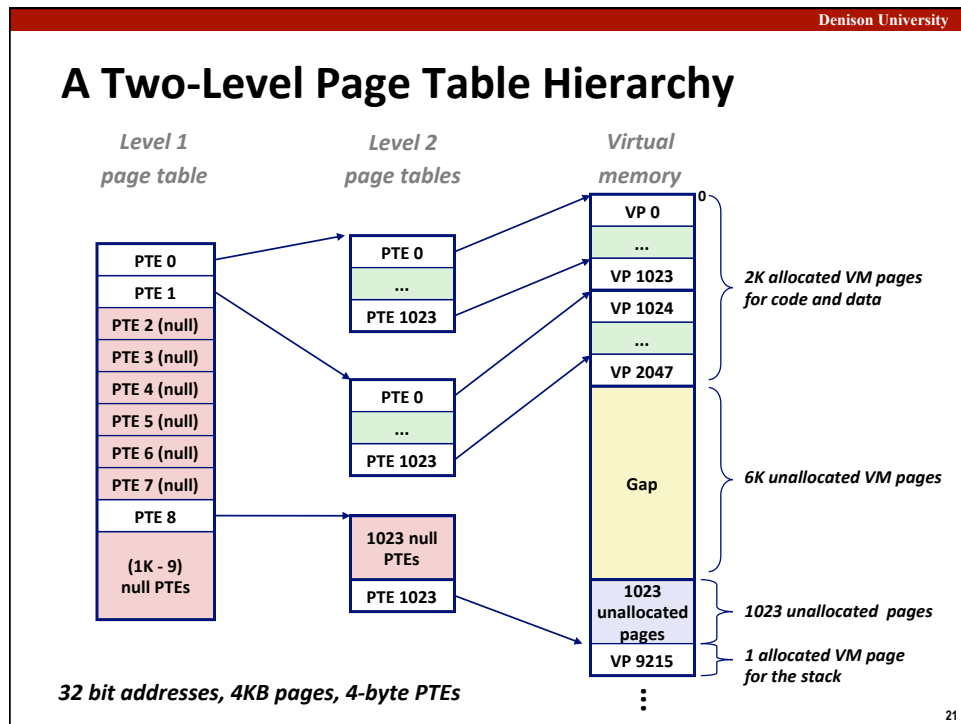
- Would need a 512 GB page table!
 - $2^{48} * 2^{-12} * 2^3 = 2^{39}$ bytes

■ 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)



20



Denison University

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

23

Denison University

Simple Memory System TLB

- 16 entries
- 4-way associative

TLBT

TLBI

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

VPN

VPO

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

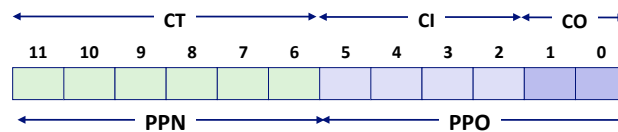
24

©R.A. Rutenbar, 2001

12

Simple Memory System Cache

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



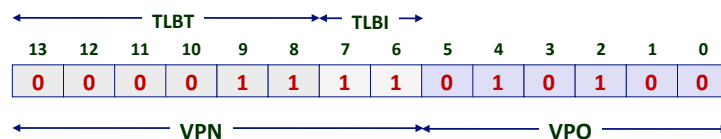
Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	-	-	-	-
2	18	1	00	02	04	08
3	36	0	-	-	-	-
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	-	-	-	-
7	16	1	11	C2	DF	03

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	-	-	-	-

25

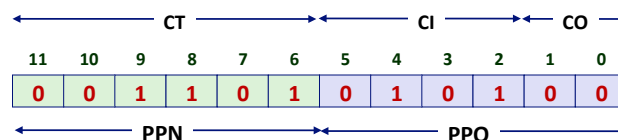
Address Translation Example #1

Virtual Address: 0x03D4



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

Physical Address

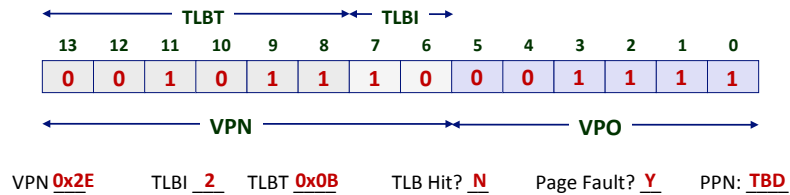


CO 0 CI 0x5 CT 0x0D Hit? Y Byte: 0x36

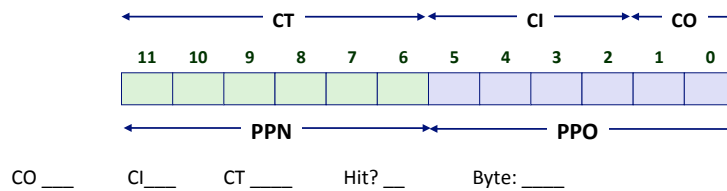
26

Address Translation Example #2

Virtual Address: 0x0B8F



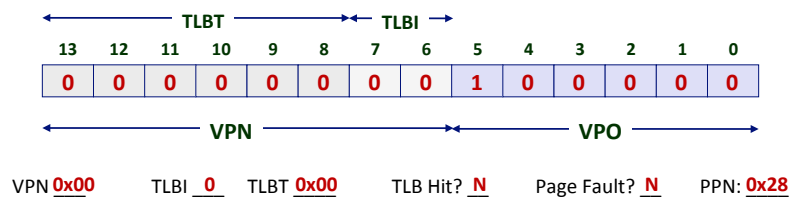
Physical Address



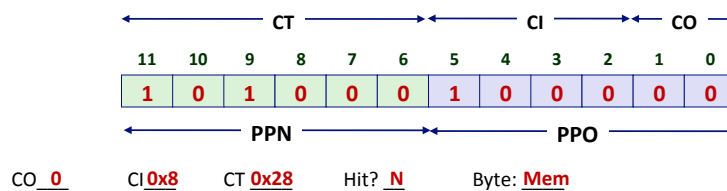
27

Address Translation Example #3

Virtual Address: 0x0020



Physical Address



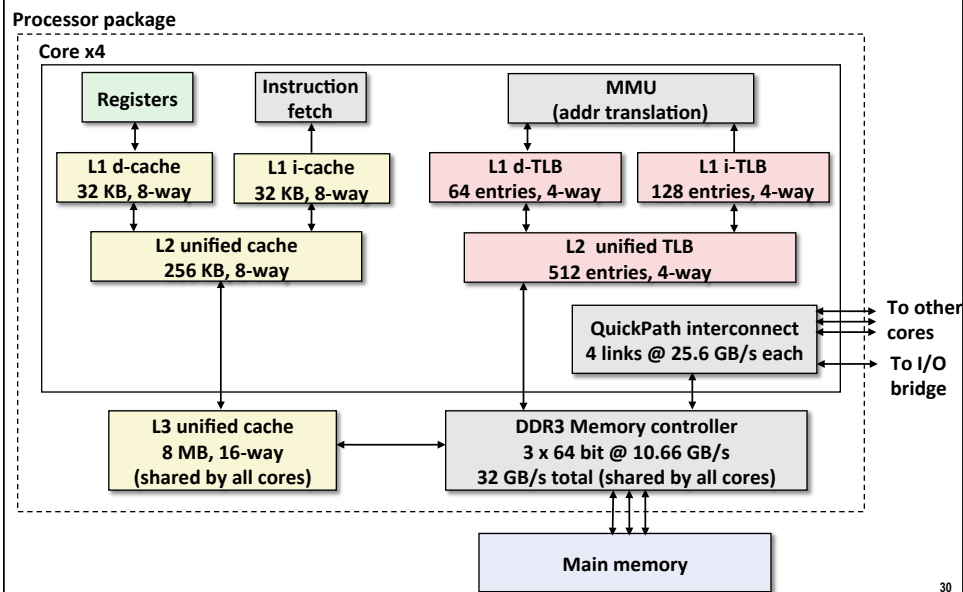
28

Today

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

29

Intel Core i7 Memory System



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)

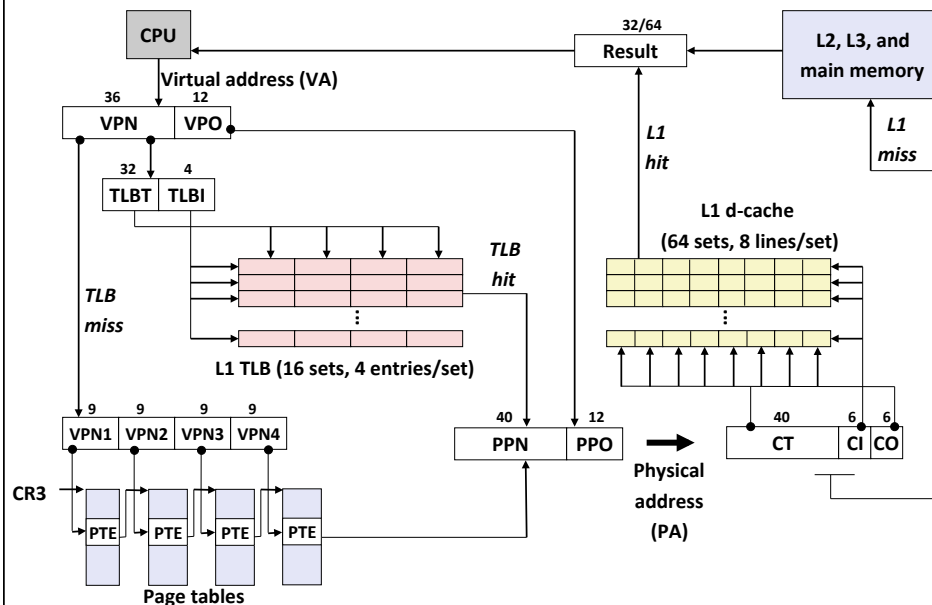
- TLBI: TLB index
- TLBT: TLB tag
- VPO: Virtual page offset
- VPN: Virtual page number

■ 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

31

End-to-end Core i7 Address Translation



32

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=
Available for OS (page table location on disk)															P=

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 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)

33

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=
Available for OS (page location on disk)															P=

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)

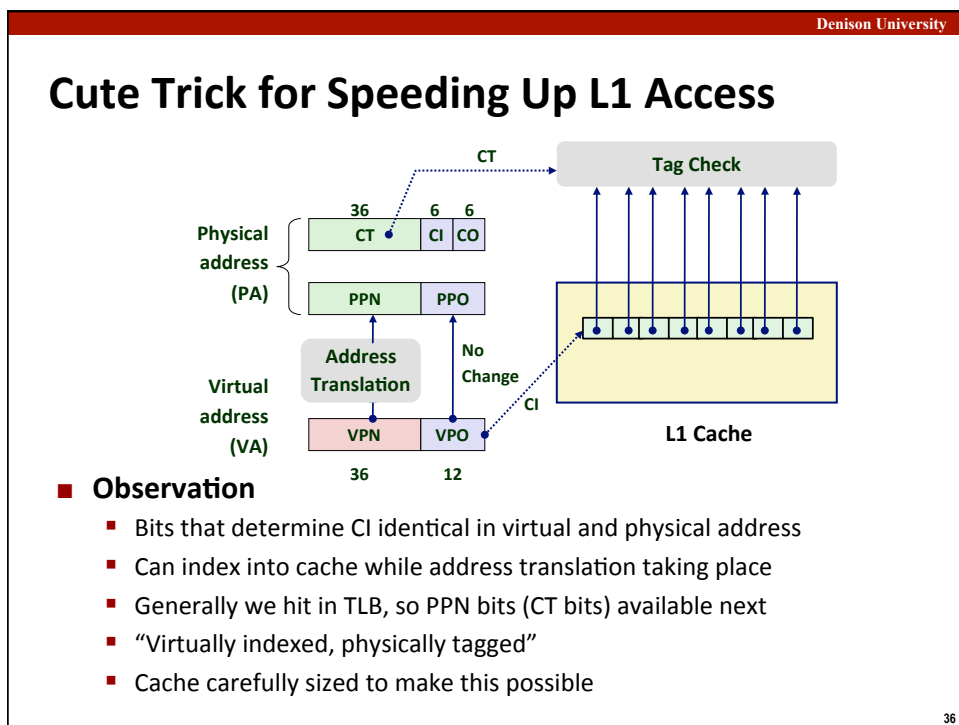
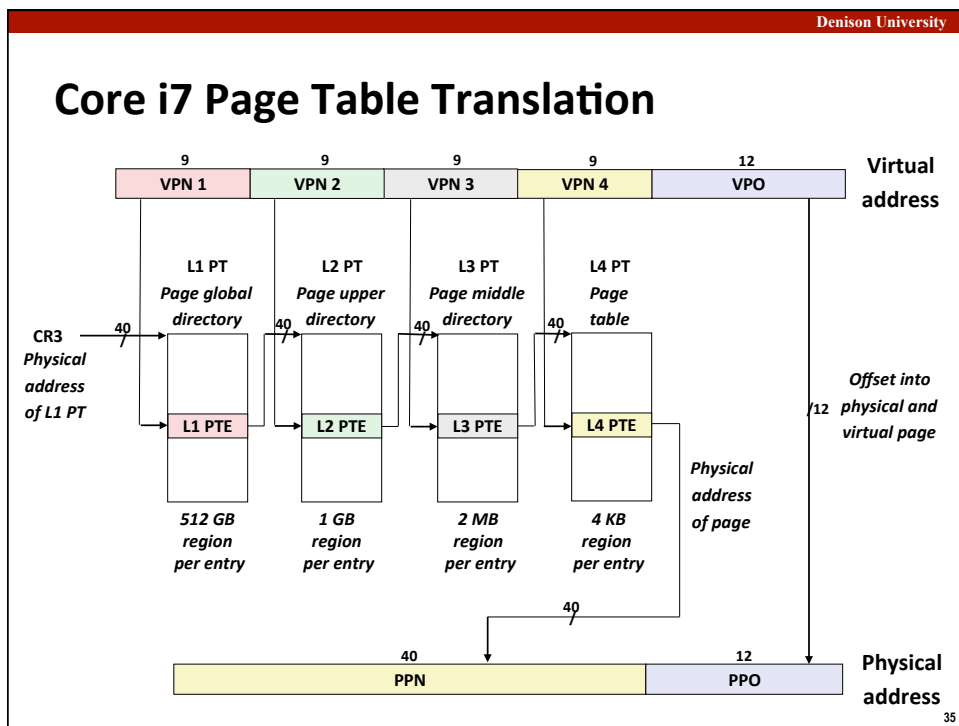
A: Reference bit (set by MMU on reads and writes, cleared by software)

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

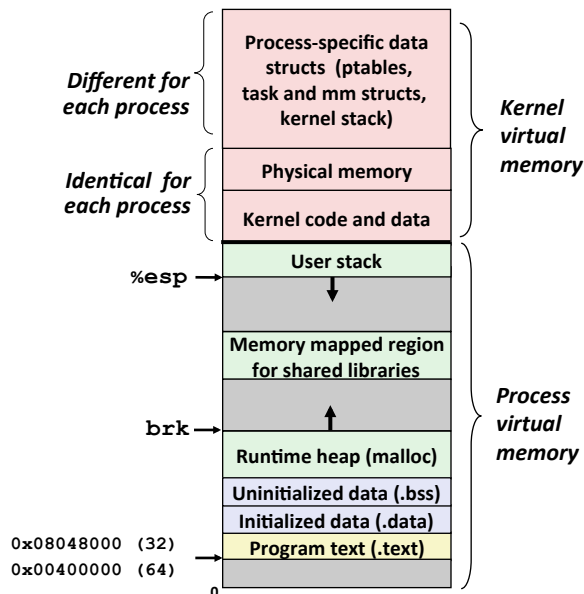
G: Global page (don't evict from TLB on task switch)

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

34

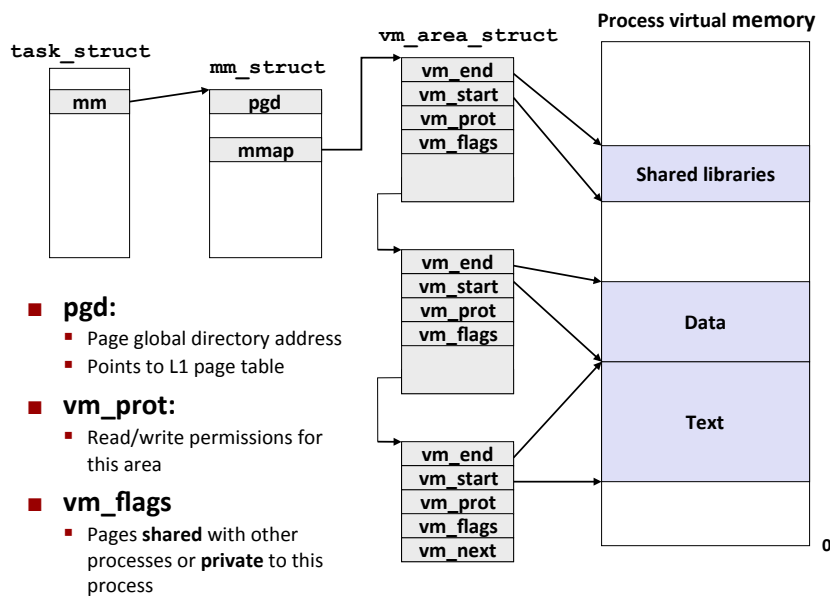


Virtual Memory of a Linux Process

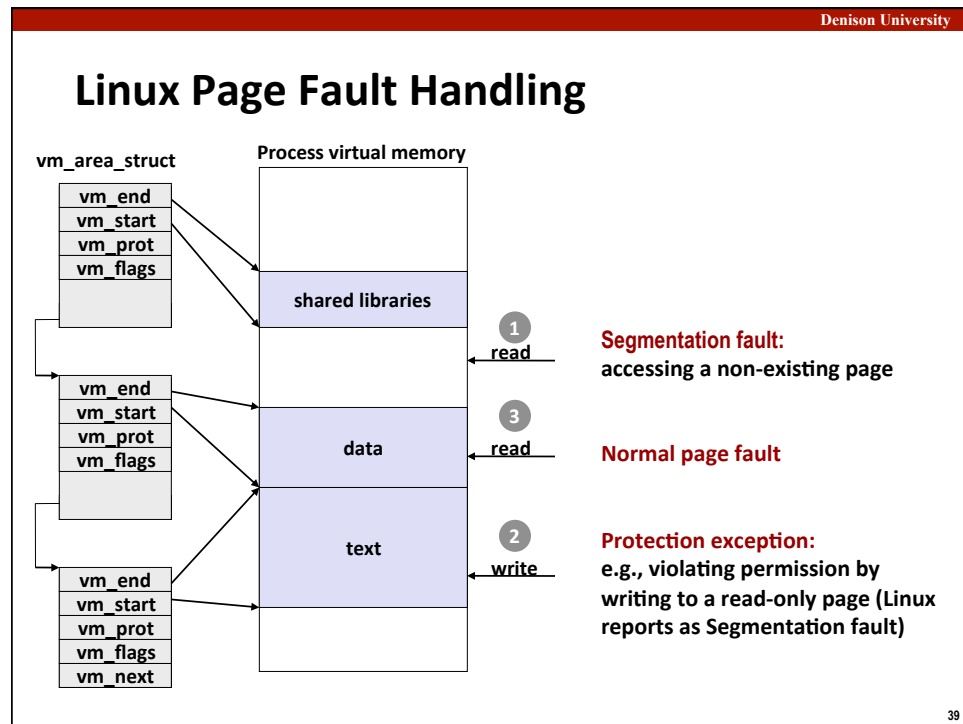


37

Linux Organizes VM as Collection of “Areas”



38



Denison University

Today

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

40

Memory Mapping

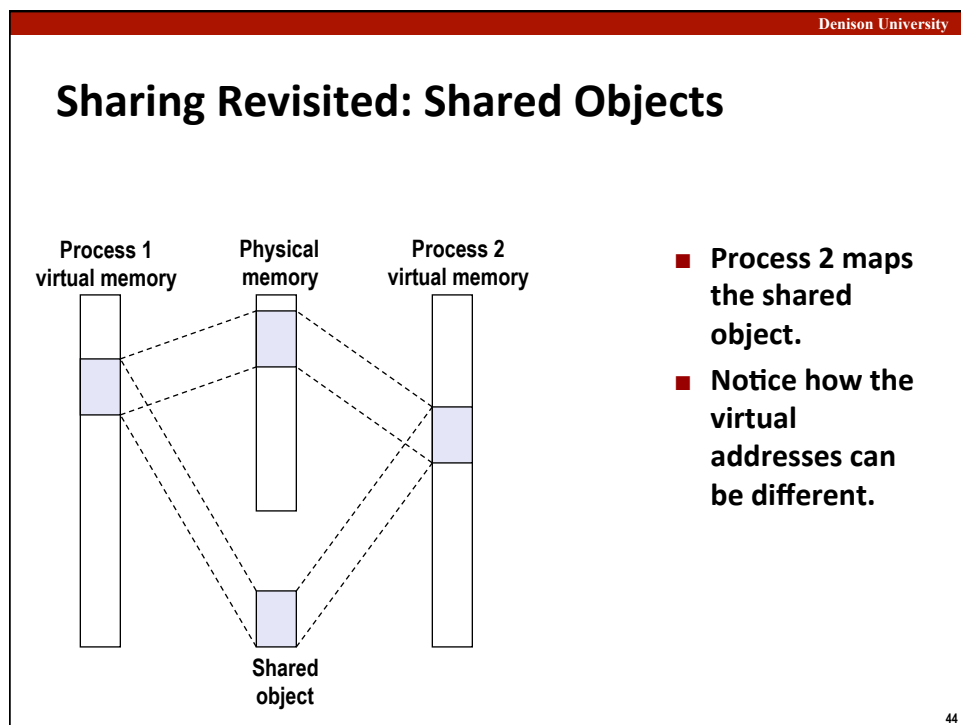
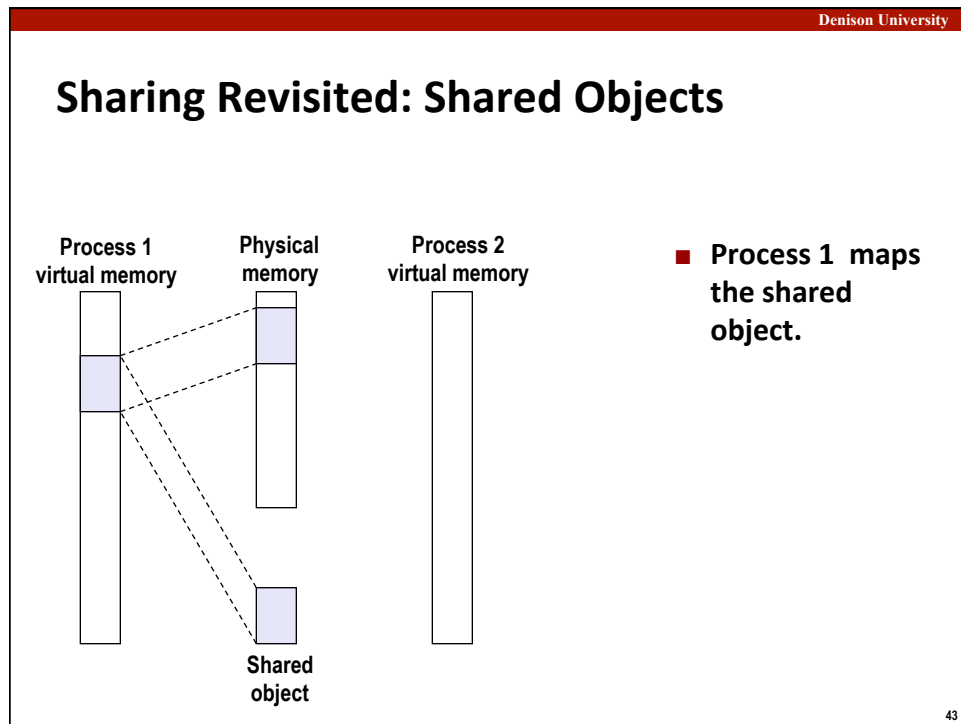
- VM areas initialized by associating them with disk objects.
 - Process is known as *memory mapping*.
- Area can be backed by (i.e., get its initial values from) :
 - *Regular file* on disk (e.g., an executable object file)
 - Initial page bytes come from a section of a file
 - *Anonymous file* (e.g.,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

41

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

42



Denison University

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

45

Denison University

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!

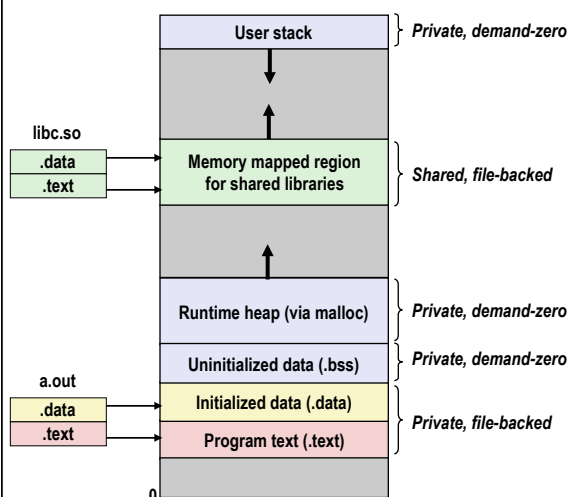
46

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.

47

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.

48

User-Level Memory Mapping

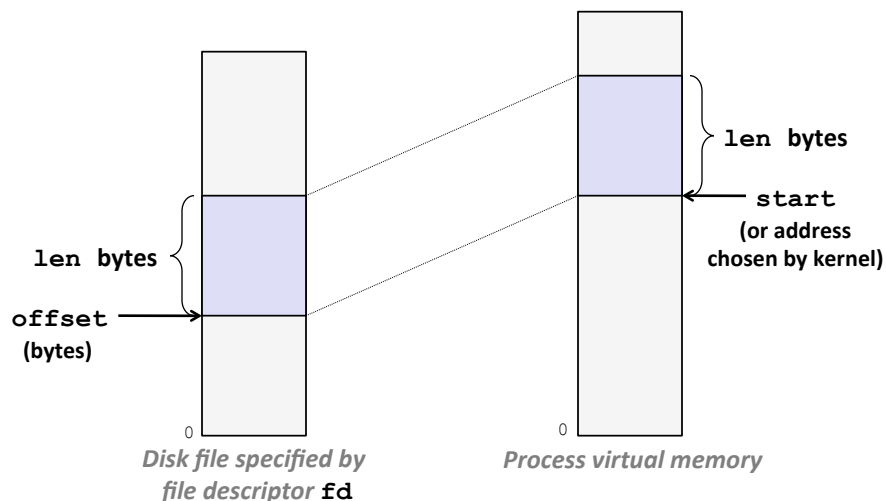
```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- 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`)

49

User-Level Memory Mapping

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



50

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);
}
```