Carnegie Mellon

Virtual Memory: Systems

15-213 / 18-213: Introduction to Computer Systems
17th Lecture, Mar. 22, 2012

Instructors:
Todd C. Mowry & Anthony Rowe

1

Carnegie Mellon

Today

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

2

Carnegie Mellon

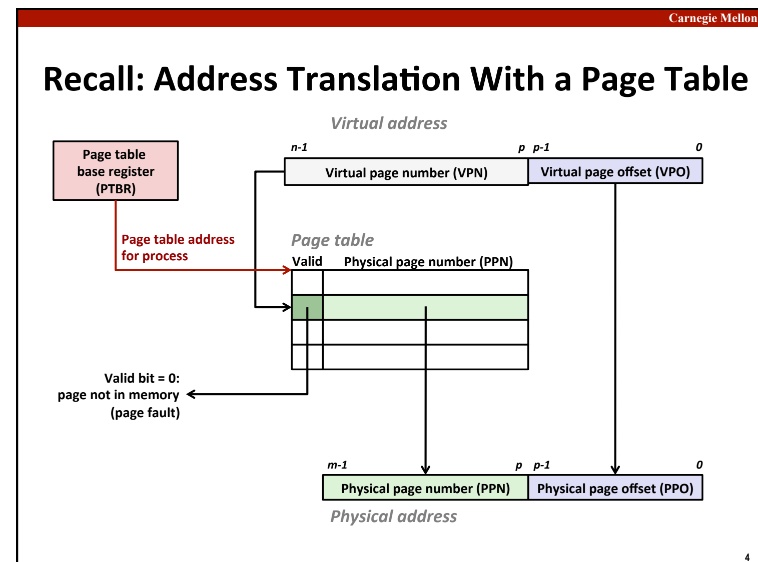
Virtual memory reminder/review

- **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

3

Carnegie Mellon

Recall: Address Translation With a Page Table



4

©R.A. Rutenbar, 2001

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

©R.A. Rutenbar, 2001

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

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

- **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**

©R.A. Rutenbar, 2001

Carnegie Mellon

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)

13

Carnegie Mellon

A Two-Level Page Table Hierarchy

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

14

Carnegie Mellon

Translating with a k-level Page Table

15

Carnegie Mellon

Question #4

- **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**

16

Carnegie Mellon

Sharing Revisited: Shared Objects

- Process 1 maps the shared object.

17

Carnegie Mellon

Sharing Revisited: Shared Objects

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

18

Carnegie Mellon

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

19

Carnegie Mellon

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!

20

The `fork` Function Revisited

- **`fork` provides private address space for each process**
- **To create virtual address for new process**
  - Create exact copies of parent page tables
  - Flag each page in both processes (parent and child) as read-only
  - Flag writeable areas in both processes as private COW
- **On return, each process has exact copy of virtual memory**
- **Subsequent writes create new physical pages using COW mechanism**
- **Perfect approach for common case of fork() followed by exec()**
  - Why?

Today

- Virtual memory questions and answers
- **Simple memory system example**
- Bonus: Case study: Core i7/Linux memory system
- Bonus: 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)**
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
- **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

©R.A. Rutenbar, 2001

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

Simple Memory System TLB

- 16 entries
- 4-way associative

TLBT ← bits 13–8 →, TLBI ← bits 7–6 →

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

VPN ← bits 13–6 →, VPO ← bits 5–0 →

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

CT ← bits 11–6 →, CI ← bits 5–2 →, CO ← bits 1–0 →

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

PPN ← bits 11–6 →, PPO ← bits 5–0 →

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 ← bits 13–8 →, TLBI ← bits 7–6 →

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 ← bits 13–6 →, VPO ← bits 5–0 →

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

Physical Address

CT ← bits 11–6 →, CI ← bits 5–2 →, CO ← bits 1–0 →

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 ← bits 11–6 →, PPO ← bits 5–0 →

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

©R.A. Rutenbar, 2001

Carnegie Mellon

Address Translation Example #2

Virtual Address: `0x01CF`

TLBT ← → TLBI

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

VPN ← → VPO

VPN 0x7 TLBI 0x3 TLBT 0x1 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: ____

29

Carnegie Mellon

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

30

Carnegie Mellon

Today

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

31

Carnegie Mellon

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

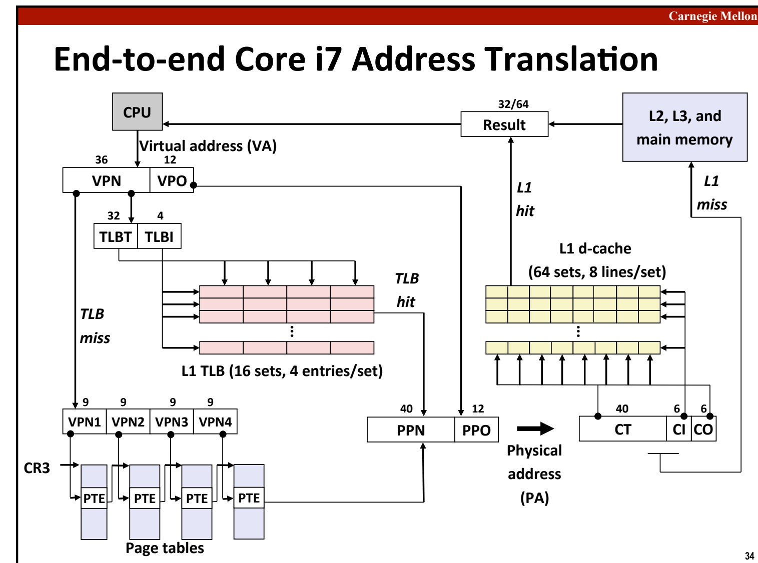
32

©R.A. Rutenbar, 2001

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 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)

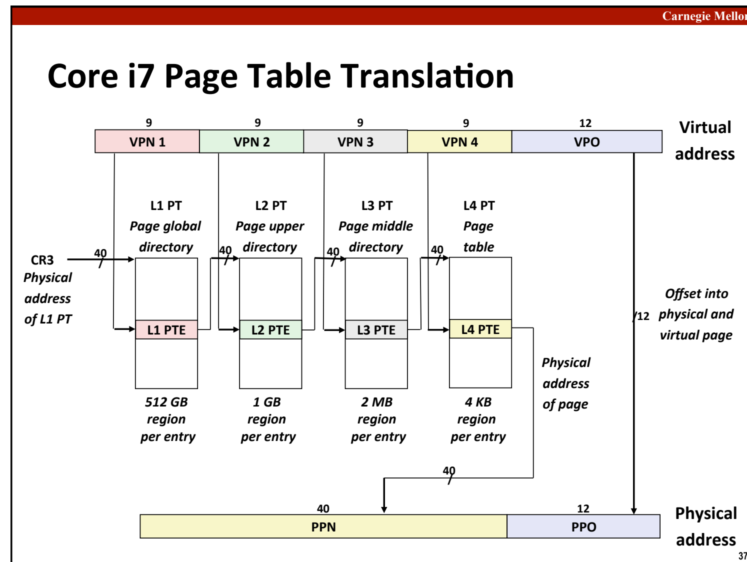
A: Reference bit (set by MMU on reads and writes, cleared by software)

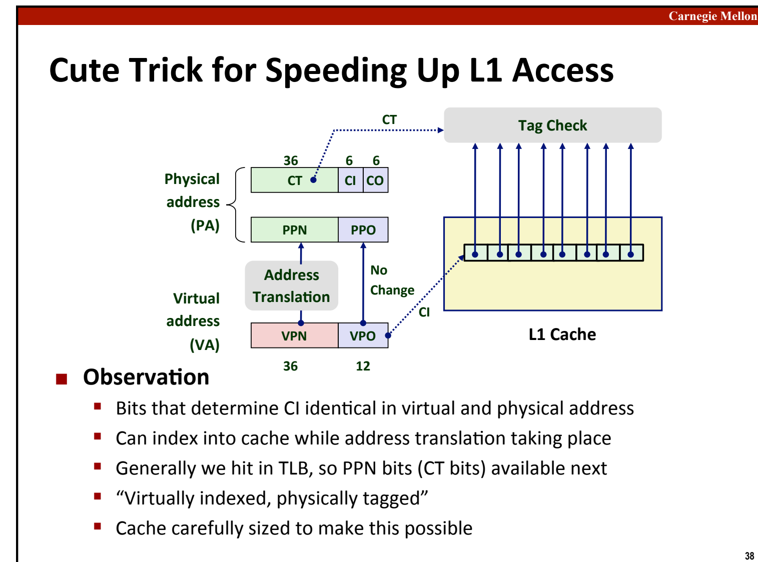
D: Dirty bit (set by MMU on writes, cleared by software)

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)

©R.A. Rutenbar, 2001





Today

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

©R.A. Rutenbar, 2001

Carnegie Mellon

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

41

Carnegie Mellon

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`)

42

Carnegie Mellon

User-Level Memory Mapping

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

len bytes

start
(or address chosen by kernel)

len bytes

offset
(bytes)

0

0

Disk file specified by file descriptor `fd`

Process virtual memory

43

Carnegie Mellon

Using `mmap` to Copy Files

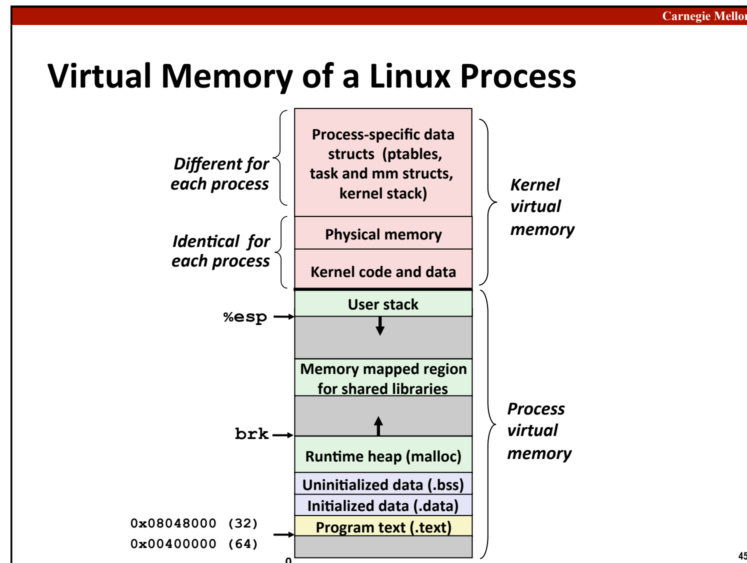
- Copying without transferring data to user space .

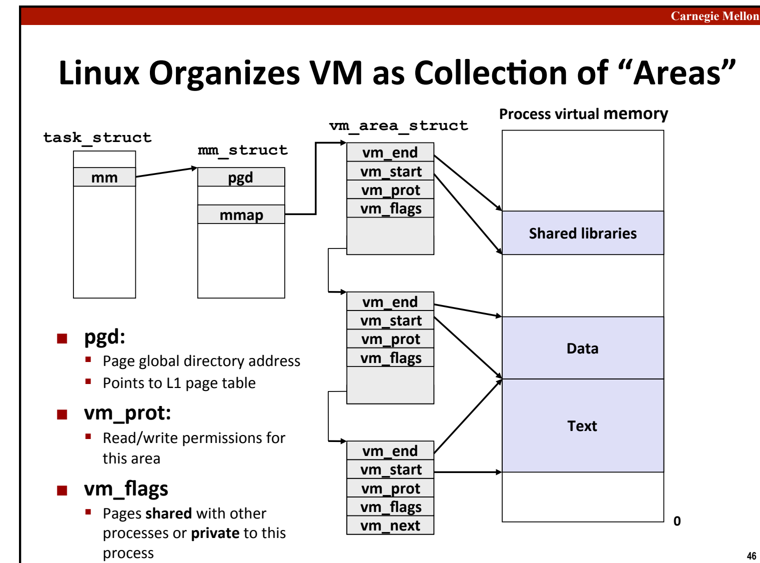
```
#include "csapp.h"

/*
 * 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;
}
```

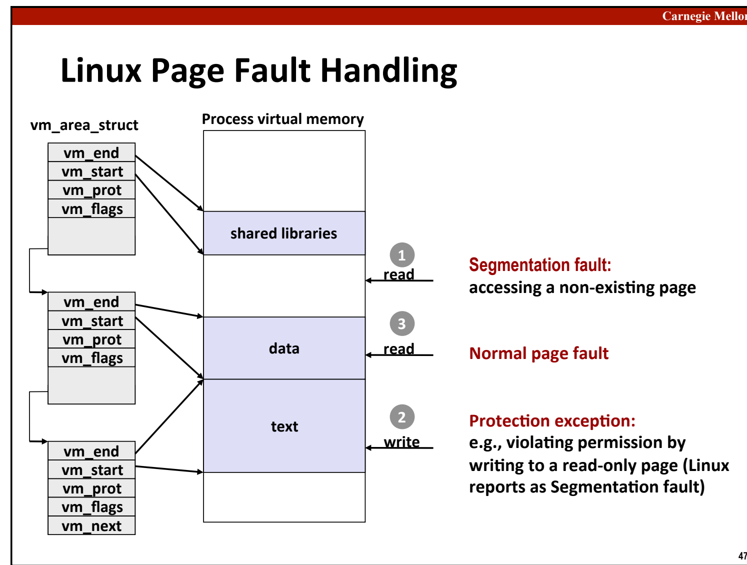
```
/* mmapcopy driver */
int main(int argc, char **argv)
{
    struct stat stat;
    int fd;

    /* 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);
}
```

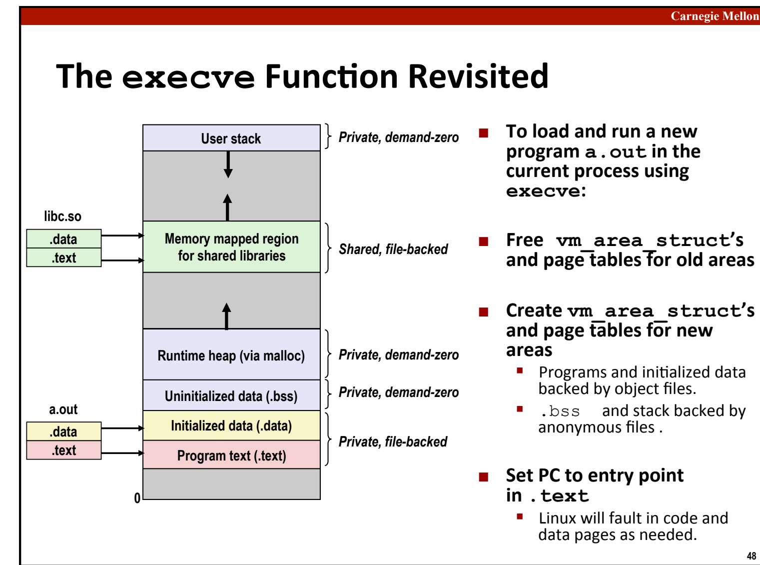
44

©R.A. Rutenbar, 2001


Carnegie Mellon
Virtual Memory of a Linux Process
Different for each process
Identical for each process
Process-specific data structs (ptables, task and mm structs, kernel stack)
Physical memory
Kernel code and data
Kernel virtual memory
%esp
User stack
Memory mapped region for shared libraries
brk
Runtime heap (malloc)
Uninitialized data (.bss)
Initialized data (.data)
Program text (.text)
Process virtual memory
0x08048000 (32)
0x00400000 (64)
0
45


Carnegie Mellon
Linux Organizes VM as Collection of "Areas"
task_struct
mm
mm_struct
pgd
mmap
vm_area_struct
vm_end
vm_start
vm_prot
vm_flags
vm_next
Process virtual memory
Shared libraries
Data
Text
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
46


Carnegie Mellon
Linux Page Fault Handling
vm_area_struct
vm_end
vm_start
vm_prot
vm_flags
vm_next
Process virtual memory
shared libraries
data
text
1 read
3 read
2 write
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)
47


Carnegie Mellon
The execve Function Revisited
User stack
Private, demand-zero
libc.so
.data
.text
Memory mapped region for shared libraries
Shared, file-backed
Runtime heap (via malloc)
Private, demand-zero
Uninitialized data (.bss)
Private, demand-zero
a.out
.data
.text
Initialized data (.data)
Program text (.text)
Private, file-backed
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.
48