Virtual Memory: Systems

Kai Shen

Outline

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

Simple Memory System Example

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

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

VPN (Virtual Page Number): bits 13–6; VPO (Virtual Page Offset): bits 5–0

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

PPN (Physical Page Number): bits 11–6; PPO (Physical Page Offset): bits 5–0

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

Virtual Address: `0x03D4`

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

Physical Address

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

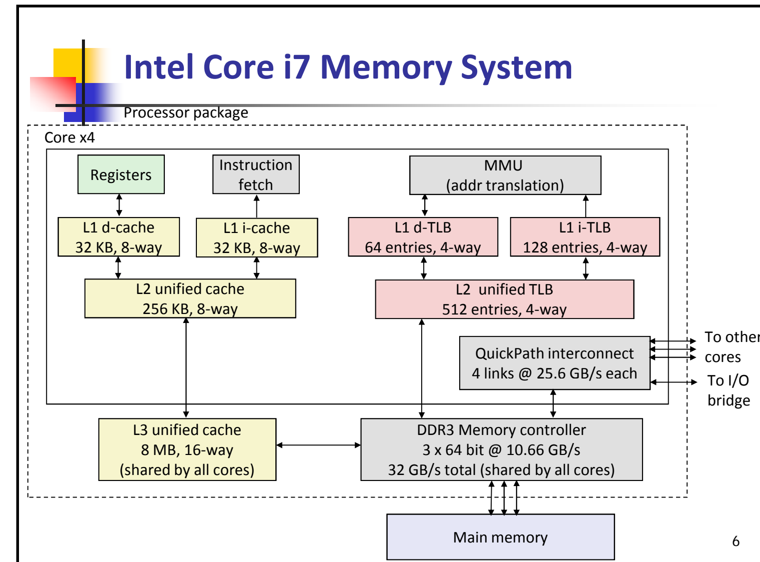
PPN: bits 11–6; PPO: bits 5–0

Outline

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

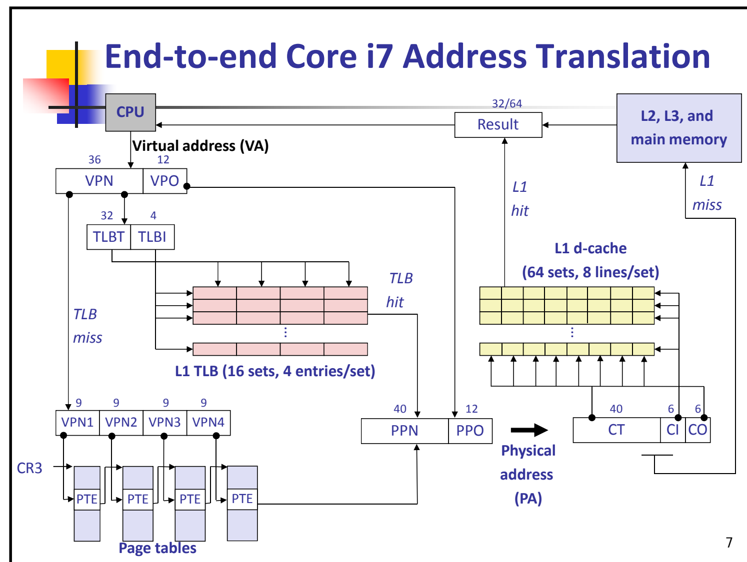
5

Intel Core i7 Memory System



6

End-to-end Core i7 Address Translation



7

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 4KB 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)

8

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 4KB 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)

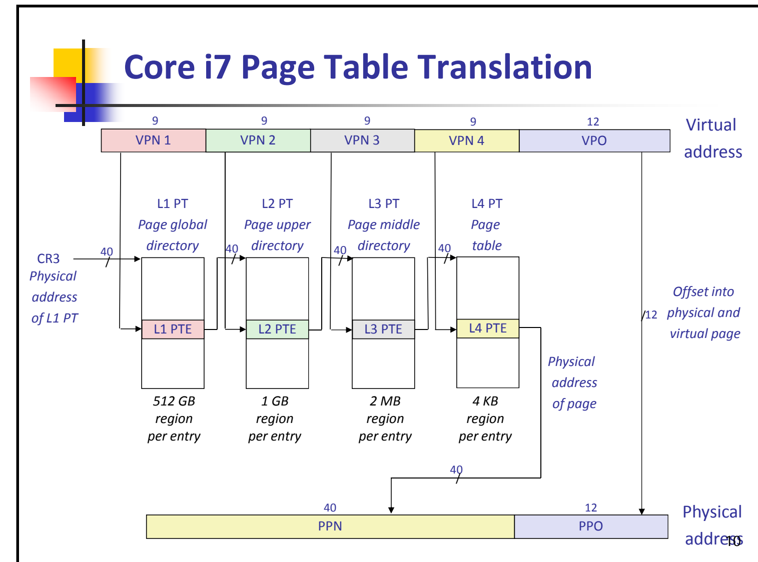
A: Reference bit (set by MMU on reads and writes, cleared by software)

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)

9



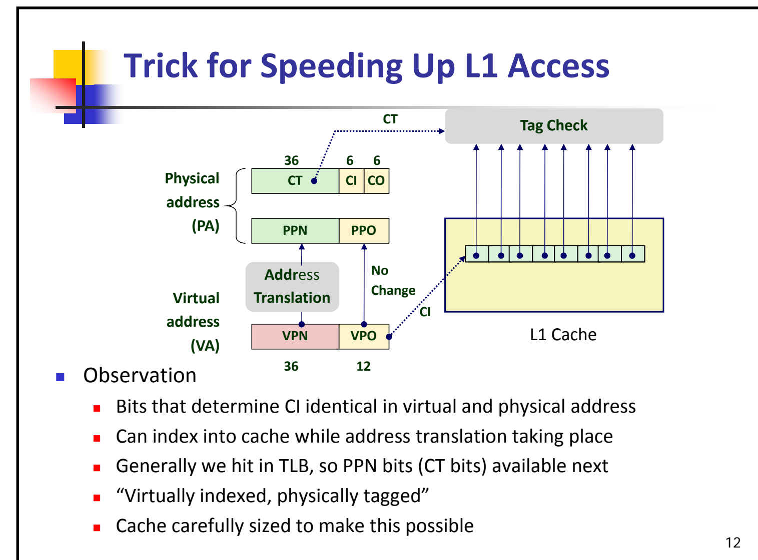
Address Translation and Memory Access

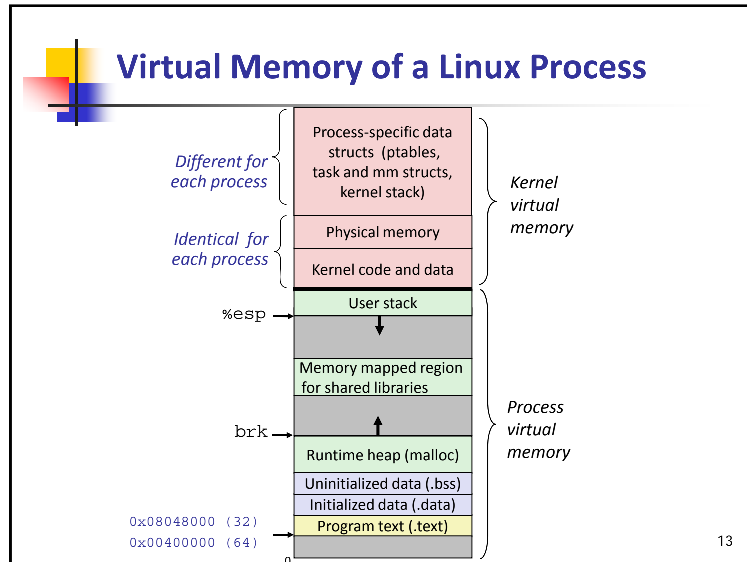
- Address translation must proceed memory access. Why?
- Can we parallelize?
  - Specifically, parallelize address translation with the first step of memory access.
  - What is the first step of memory access?
- A helpful address layout

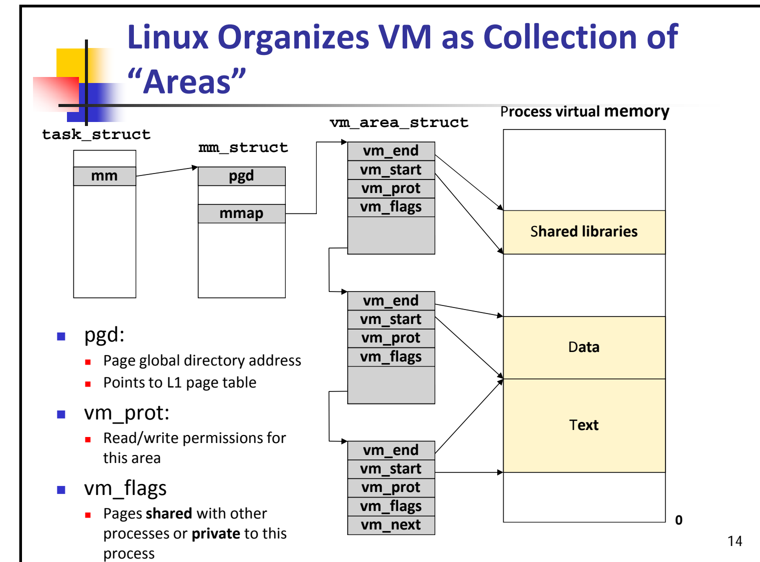
Cache tag | Cache Index

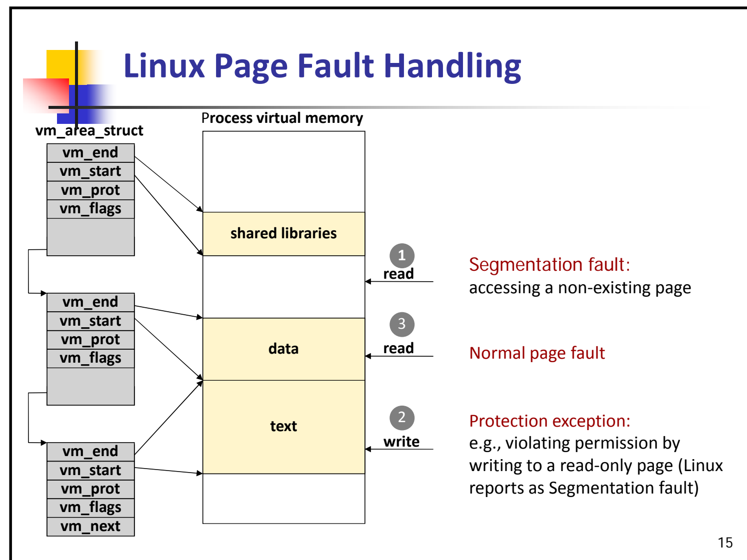
Page number | Page offset

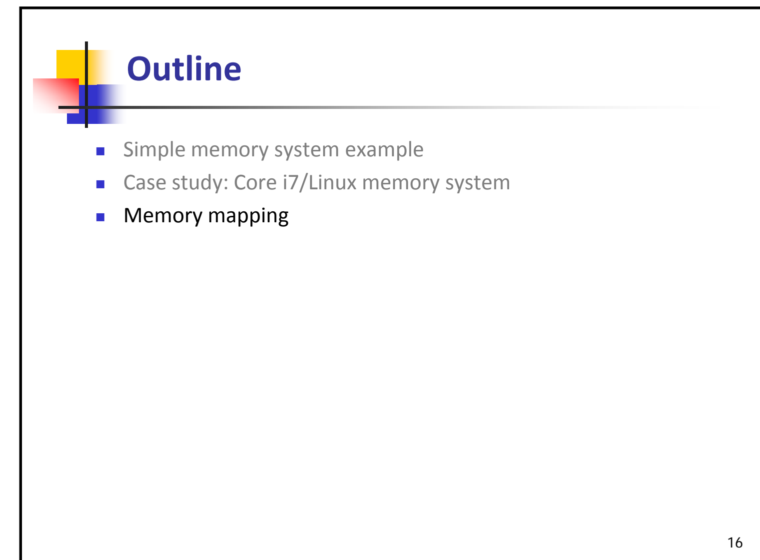
11




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
13


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
14


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)
15


Outline
Simple memory system example
Case study: Core i7/Linux memory system
Memory mapping
16

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
    - Can be ***dirtied*** later.
  - ***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 to the disk lazily.

17

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

18

Sharing Revisited: Shared Objects

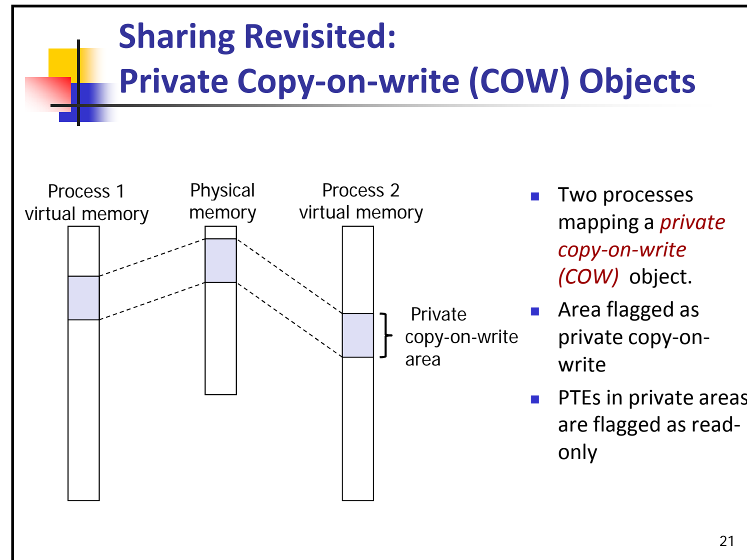
- Process 1 maps the shared object.

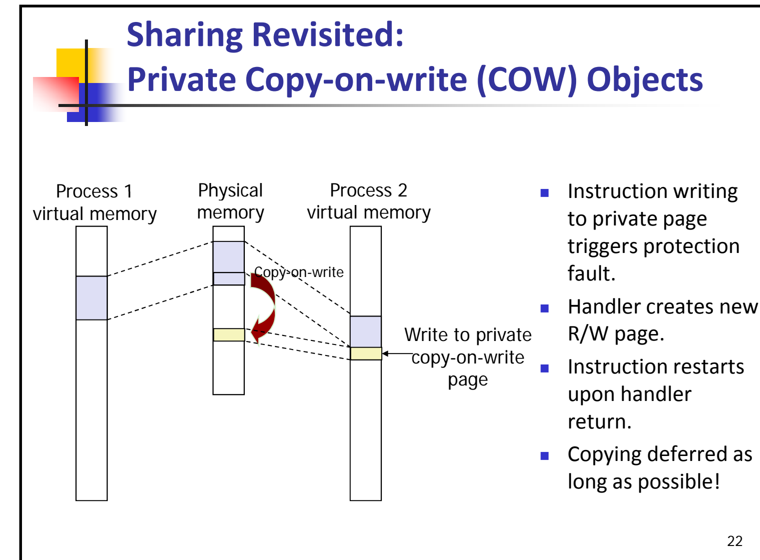
19

Sharing Revisited: Shared Objects

- Process 2 maps the shared object.
- Shared objects are typically read only.

20





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 memory area structures and page tables.
  - Flag each page in both processes as read-only
  - Flag each memory area in both processes as private COW
- On return, each process has exact copy of virtual memory.
- Subsequent writes create new pages using COW mechanism.

23

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

24

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

25

Using `mmap` to Read File

- `mmap` →lazy file read to buffer.
  - Fast start;
  - Real read is only performed on-demand.

26

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

27