Virtual Memory: Systems

15-213: Introduction to Computer Systems
18th Lecture, October 27, 2016

Instructor:
Phil Gibbons
Your Academic Integrity

■ What is cheating?
  ▪ *Sharing code:* by copying, retyping, *looking at*, or supplying a file
  ▪ *Describing:* verbal description of code from one person to another.
  ▪ *Coaching:* helping your friend to write a lab, line by line
  ▪ *Searching the Web* for solutions
  ▪ *Copying code* from a previous course or online solution
    ▪ You are only allowed to use code we supply, or from the CS:APP website

■ What is NOT cheating?
  ▪ Explaining how to use systems or tools
  ▪ Helping others with high-level design issues

■ See the course syllabus & Lecture 1 slides for details.
  ▪ Ignorance is not an excuse
A page table contains page table entries (PTEs) that map virtual pages to physical pages.

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Physical page number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTE 0</td>
<td>Valid: 0, null</td>
</tr>
<tr>
<td></td>
<td>Valid: 1</td>
</tr>
<tr>
<td></td>
<td>Valid: 1</td>
</tr>
<tr>
<td></td>
<td>Valid: 0</td>
</tr>
<tr>
<td></td>
<td>Valid: 1</td>
</tr>
<tr>
<td></td>
<td>Valid: 0, null</td>
</tr>
<tr>
<td></td>
<td>Valid: 1</td>
</tr>
<tr>
<td>PTE 7</td>
<td>Valid: 1</td>
</tr>
</tbody>
</table>

Memory resident page table (DRAM)

Physical memory (DRAM): VP 1, VP 2, VP 7, VP 4, VP 3, VP 6, VP 7

Virtual memory (disk): VP 1, VP 2, VP 3, VP 4, VP 6, VP 7

Review: Virtual Memory & Physical Memory
Translating with a k-level Page Table

- Having multiple levels greatly reduces page table size
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.
Set Associative Cache: Read

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

Address of word:
- Address of word: \[ t \text{ bits} \ | \ s \text{ bits} \ | \ b \text{ bits} \]
- Tag (CT, t bits)
- Index (CI, s bits)
- Offset (CO, b bits)

B = 2^b bytes per cache block (the data)
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
Today

- 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
Simple Memory System TLB

- 16 entries
- 4-way associative

Translation Lookaside Buffer (TLB)

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
<td>09</td>
<td>0D</td>
<td>1</td>
<td>00</td>
<td>–</td>
<td>0</td>
<td>07</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>03</td>
<td>2D</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>04</td>
<td>–</td>
<td>0</td>
<td>0A</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>08</td>
<td>–</td>
<td>0</td>
<td>06</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>07</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>0D</td>
<td>1</td>
<td>0A</td>
<td>34</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

VPN = 0b1101 = 0x0D
# Simple Memory System Page Table

Only showing the first 16 entries (out of 256)

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>09</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>0B</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>0F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>

0x0D $\rightarrow$ 0x2D
Simple Memory System Cache

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

![Cache Diagram]

```
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

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

<table>
<thead>
<tr>
<th>Idx</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>24</td>
<td>1</td>
<td>3A</td>
<td>00</td>
<td>51</td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td>2D</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A</td>
<td>2D</td>
<td>1</td>
<td>93</td>
<td>15</td>
<td>DA</td>
<td>3B</td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>D</td>
<td>16</td>
<td>1</td>
<td>04</td>
<td>96</td>
<td>34</td>
<td>15</td>
</tr>
<tr>
<td>E</td>
<td>13</td>
<td>1</td>
<td>83</td>
<td>77</td>
<td>1B</td>
<td>D3</td>
</tr>
<tr>
<td>F</td>
<td>14</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### Physical Address

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>-</td>
<td>0</td>
<td>09</td>
<td>0D</td>
<td>1</td>
<td>00</td>
<td>-</td>
<td>0</td>
<td>07</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>03</td>
<td>2D</td>
<td>1</td>
<td>02</td>
<td>-</td>
<td>0</td>
<td>04</td>
<td>-</td>
<td>0</td>
<td>0A</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>-</td>
<td>0</td>
<td>08</td>
<td>-</td>
<td>0</td>
<td>06</td>
<td>-</td>
<td>0</td>
<td>03</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>07</td>
<td>-</td>
<td>0</td>
<td>03</td>
<td>0D</td>
<td>1</td>
<td>0A</td>
<td>34</td>
<td>1</td>
<td>02</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
Address Translation Example: TLB/Cache Miss

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

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

Physical Address

<table>
<thead>
<tr>
<th>CO</th>
<th>PPN</th>
<th>PPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>Mem</td>
</tr>
<tr>
<td>0x8</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>0x28</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Hit?</td>
<td>N</td>
<td>–</td>
</tr>
</tbody>
</table>

Page table

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>TLB</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>Tag</td>
<td>Frame Number</td>
<td>Valid</td>
</tr>
<tr>
<td>0</td>
<td>0x13</td>
<td>0x30</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0x34</td>
<td>0x58</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0x1F</td>
<td>0x80</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0x2A</td>
<td>0x72</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0x1F</td>
<td>0x95</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0x20</td>
<td>0xCA</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0x3F</td>
<td>0x20</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0x3E</td>
<td>0xFF</td>
<td>0</td>
</tr>
</tbody>
</table>

A. Use the TLB to fill in the table. Strike out anything that you don’t have enough information to fill in.

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7E85</td>
<td>0x9585</td>
</tr>
<tr>
<td>0xD301</td>
<td>--------</td>
</tr>
<tr>
<td>0x4C20</td>
<td>0x3020</td>
</tr>
<tr>
<td>0xD040</td>
<td>--------</td>
</tr>
<tr>
<td>--------</td>
<td>0x5830</td>
</tr>
</tbody>
</table>

0x7E85 = 0x0111111010000101

Cl = 0x2
CT = 0x1F

0x7E85 → 0x9585

Exam: [http://www.cs.cmu.edu/~213/oldexams/exam2b-s11.pdf (solution)]
Today

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

Processor package

Core x4

Registers
L1 d-cache 32 KB, 8-way

Instruction fetch
L1 i-cache 32 KB, 8-way

L2 unified cache 256 KB, 8-way

MMU (addr translation)
L1 d-TLB 64 entries, 4-way

L1 i-TLB 128 entries, 4-way

L2 unified TLB 512 entries, 4-way

QuickPath interconnect 4 links @ 25.6 GB/s each

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

To other cores

To I/O bridge
End-to-end Core i7 Address Translation

Virtual address (VA) → CPU

VPN VPO

TLBT TLBI

TLB

VPN1 VPN2 VPN3 VPN4

L1 TLB (16 sets, 4 entries/set)

L1 TLB hit

L1 d-cache (64 sets, 8 lines/set)

L1 hit

L1 miss

Physical address (PA)

VPN1 VPN2 VPN3 VPN4

PTE PTE PTE PTE

Page tables

L2, L3, and main memory

Result

32/64

CT CI CO

Physical address (PA)

CPU

VPN VPO
## 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 1 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

- VPN 1
- VPN 2
- VPN 3
- VPN 4
- VPN Offset (VPO)

CR3

Physical address of L1 PT

L1 PT
Page global directory
L1 PTE
512 GB region per entry

L2 PT
Page upper directory
L2 PTE
1 GB region per entry

L3 PT
Page middle directory
L3 PTE
2 MB region per entry

L4 PT
Page table
L4 PTE
4 KB region per entry

PPN

Physical address

VPO

Offset into physical and virtual page

VPN 1
VPN 2
VPN 3
VPN 4

Virtual address

Bryant and O'Hallaron, Computer Systems: A Programmer’s Perspective, Third Edition
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

Different for each process:
- Process-specific data structs (ptables, task and mm structs, kernel stack)

Identical for each process:
- Physical memory
- Kernel code and data

Kernel virtual memory:
- User stack
- Memory mapped region for shared libraries
- Runtime heap (malloc)
- Uninitialized data (.bss)
- Initialized data (.data)
- Program text (.text)

Process virtual memory:
- brk
- %rsp
- 0x00400000
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

Each process has own `task_struct`, etc.
Linux Page Fault Handling

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)
Today

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

- VM areas initialized by associating them with disk objects.
  - Called *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

Virtual Address Space for Process 1:
- Virtual Address Space: VP 1, VP 2
- Physical Address Space (DRAM): 0 to N-1 (e.g., read-only library code)

Virtual Address Space for Process 2:
- Virtual Address Space: VP 1, VP 2
- Physical Address Space (DRAM): 0 to M-1

Address translation:
- Mapping from virtual to physical addresses.
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.
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.

- To create virtual address for 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.
User-Level Memory Mapping

```c
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, PROT_EXEC, ...
  - `flags`: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...

- Return a pointer to start of mapped area (may not be `start`)
User-Level Memory Mapping

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

- `len` bytes
- Start (or address chosen by kernel)
- Offset (bytes)

Disk file specified by file descriptor `fd`

Process virtual memory
Example: Using mmap to Copy Files

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

```c
#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 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
Today: Virtual Memory Systems

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

Next Lecture

- Dynamic Memory Allocation