Virtual Memory: Systems

15-213/18-213/14-513/15-513: Introduction to Computer Systems
18th Lecture, March 26, 2019
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>
<th>Physical memory (DRAM)</th>
<th>Virtual memory (disk)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTE 0</td>
<td>Valid 0, null</td>
<td>VP 1, VP 2, VP 4</td>
<td>VP 1, VP 2</td>
</tr>
<tr>
<td></td>
<td>Valid 1</td>
<td>VP 7</td>
<td>VP 3</td>
</tr>
<tr>
<td></td>
<td>Valid 1</td>
<td></td>
<td>VP 6</td>
</tr>
<tr>
<td></td>
<td>Valid 0, null</td>
<td></td>
<td>VP 7</td>
</tr>
<tr>
<td></td>
<td>Physical memory (DRAM)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Memory resident page table</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(DRAM)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Valid 0: Page not in memory.
- Valid 1: Page in memory.
- null: Page does not exist in memory.
- Physical page number or disk address:
  - 0: Page 0
  - 1: Page 1
  - 2: Page 2
  - 3: Page 3
  - 4: Page 4
  - 5: Page 5
  - 6: Page 6
  - 7: Page 7
Translating with a k-level Page Table

- Having multiple levels greatly reduces page table size

Page table base register (CR3) 
part of process context

![Diagram showing page table levels and translation process]

- 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

E = 2^e lines per set

S = 2^s sets

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

Address of word:

CT | CI | CO
---|----|----
t bits | s bits | b bits

data begins at this offset

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 → 0x2D
Simple Memory System Cache

- 16 lines, 4-byte cache line size
- Physically addressed
- Direct mapped

V[0b00001101101001] = 0x15
P[0b101101101001] = 0x15

<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>

# Address Translation Example

**Virtual Address:** 0x03D4

---

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

## 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>

## Physical Address

<table>
<thead>
<tr>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---
Address Translation Example

Physical Address

Cache

<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>
Address Translation Example: **TLB/Cache Miss**

**Virtual Address: 0x0020**

**Physical Address**

### 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>
Address Translation Example: TLB/Cache Miss

Cache

<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>OD</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>

Physical Address

CO 0  CI 0x8  CT 0x28  Hit? N  Byte: Mem

Physical Address Example:

<table>
<thead>
<tr>
<th>PPO</th>
<th>PPN</th>
<th>PPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Cache Address Translation Example:

Physical Address:

<table>
<thead>
<tr>
<th>CT</th>
<th>CI</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

TLB/Cache Miss:

Physical Address:

<table>
<thead>
<tr>
<th>CO</th>
<th>CI</th>
<th>CT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x8</td>
<td>0x28</td>
</tr>
</tbody>
</table>
Quiz Time!

Check out:

https://canvas.cmu.edu/courses/8555
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
- L1 i-cache: 32 KB, 8-way
- L2 unified cache: 256 KB, 8-way
- L1 d-TLB: 64 entries, 4-way
- L1 i-TLB: 128 entries, 4-way
- L2 unified TLB: 512 entries, 4-way
- 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)
- MMU (addr translation)
- QuickPath interconnect: 4 links @ 25.6 GB/s each
  - To other cores
  - To I/O bridge

Main memory
End-to-end Core i7 Address Translation

Virtual address (VA) → CPU

VPN → VPO

TLBT → TLBI

TLB hit

TLB (16 sets, 4 entries/set)

VPN1 → VPN2 → VPN3 → VPN4

TLB miss

L1 TLB (16 sets, 4 entries/set)

VPN1 → VPN2 → VPN3 → VPN4

32/64

Result → L2, L3, and main memory

L1 hit

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

L1 miss

Physical address (PA)

CR3 → Page tables

PTE

L1 cache (64 sets, 8 lines/set)

Physical address (PA)

PPN → PPO

CT → CI → CO
## 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 |

| 50 | 49 | 48 | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 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)**

| 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=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.

**G:** Process global – pages readable by all processes, subject to access restrictions
## Core i7 Level 4 Page Table Entries

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>52</th>
<th>51</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>XD</td>
<td>Unused</td>
<td>Page physical base address</td>
<td>Unused</td>
<td>G</td>
<td>D</td>
<td>A</td>
<td>CD</td>
<td>WT</td>
<td>U/S</td>
<td>R/W</td>
<td>P=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
- (40 out of 52 bits; 12 remaining bits address the page & force 4KB alignment)

**XD**: Disable or enable instruction fetches from this page.
Core i7 Page Table Translation

CR3
Physical address of L1 PT

VPN 1
L1 PT
Page global directory
512 GB region per entry

VPN 2
L2 PT
Page upper directory
1 GB region per entry

VPN 3
L3 PT
Page middle directory
2 MB region per entry

VPN 4
L4 PT
Page table
4 KB region per entry

VPN 0
PPO
Physical address

L1 PTE
L2 PTE
L3 PTE
L4 PTE

Physical address of page

512 GB region per entry
1 GB region per entry
2 MB region per entry
4 KB region per entry

Offset into physical and virtual page

VPN 1
VPN 2
VPN 3
VPN 4
VPO
Virtual address

PPN

40
12
12
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

- **Process-specific data structs** (ptables, task and mm structs, kernel stack)
- **Kernel code and data**
- **Physical memory**
- **Kernel virtual memory**
- **Program text (.text)**
- **Uninitialized data (.bss)**
- **Initialized data (.data)**
- **Runtime heap (malloc)**
- **Memory mapped region for shared libraries**
- **User stack**

- **Different for each process**
- **Identical for each process**

 процессы

- **Process virtual memory**

Брк
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 \textit{memory mapping}

Area can be \textit{backed by} (i.e., get its initial values from):

- \textit{Regular file} on disk (e.g., an executable object file)
  - Initial page bytes come from a section of a file

- \textit{Anonymous file} (e.g., nothing)
  - First fault will allocate a physical page full of 0's (\textit{demand-zero page})
  - Once the page is written to (\textit{dirtied}), it is like any other page

Dirty pages are copied back and forth between memory and a special \textit{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 for Process 2:

Address translation

Physical Address Space (DRAM)

(e.g., read-only library code)
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.
- But, difference must be multiple of page size
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.

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Private, demand-zero</th>
<th>Shared, file-backed</th>
</tr>
</thead>
<tbody>
<tr>
<td>libc.so</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a.out</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Runtime heap</td>
<td>Private, demand-zero</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(via malloc)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Uninitialized</td>
<td>Private, demand-zero</td>
<td></td>
</tr>
<tr>
<td></td>
<td>data (.bss)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Initialized</td>
<td>Private, file-backed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>data (.data)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Program text</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(.text)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)
- Disk file specified by file descriptor `fd`
- Offset (bytes)
- 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(STDOUT_FILENO,
         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);
}
```
Some Uses of mmap

- **Reading big files**
  - Uses paging mechanism to bring files into memory

- **Shared data structures**
  - When call with `MAP_SHARED` flag
    - Multiple processes have access to same region of memory
    - Risky!

- **File-based data structures**
  - E.g., database
  - Give `prot` argument `PROT_READ | PROT_WRITE`
  - When unmap region, file will be updated via write-back
  - Can implement load from file / update / write back to file
Summary

- **VM requires hardware support**
  - Exception handling mechanism
  - TLB
  - Various control registers

- **VM requires OS support**
  - Managing page tables
  - Implementing page replacement policies
  - Managing file system

- **VM enables many capabilities**
  - Loading programs from memory
  - Forking processes
  - Providing memory protection