Virtual Memory: Systems

14-513/18-613:
Computer Systems
18th Lecture, June 25th, 2020
A page table contains page table entries (PTEs) that map virtual pages to physical pages.
Translating with a k-level Page Table

- Having multiple levels greatly reduces page table size

![Diagram of page table with multiple levels]

VIRTUAL ADDRESS

PHYSICAL ADDRESS

Page table base register (part of the process’ context)
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

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

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

Address of word:

<table>
<thead>
<tr>
<th>t bits</th>
<th>s bits</th>
<th>b bits</th>
</tr>
</thead>
</table>

$CT$ tag
$CI$ index
$CO$ offset

Data begins at this offset
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

(bits per field for our simple example)
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

![Diagram showing virtual and physical addresses]

- **VPN** Virtual Page Number
- **VPO** Virtual Page Offset
- **PPN** Physical Page Number
- **PPO** Physical Page Offset
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

\[
\begin{array}{cccccccc}
0 & 19 & 1 & 99 & 11 & 23 & 11 \\
1 & 15 & 0 & - & - & - & - \\
2 & 1B & 1 & 00 & 02 & 04 & 08 \\
3 & 36 & 0 & - & - & - & - \\
4 & 32 & 1 & 43 & 6D & 8F & 09 \\
5 & 0D & 1 & 36 & 72 & F0 & 1D \\
6 & 31 & 0 & - & - & - & - \\
7 & 16 & 1 & 11 & C2 & DF & 03 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
8 & 24 & 1 & 3A & 00 & 51 & 89 \\
9 & 2D & 0 & - & - & - & - \\
A & 2D & 1 & 93 & 15 & DA & 3B \\
B & 0B & 0 & - & - & - & - \\
C & 12 & 0 & - & - & - & - \\
D & 16 & 1 & 04 & 96 & 34 & 15 \\
E & 13 & 1 & 83 & 77 & 1B & D3 \\
F & 14 & 0 & - & - & - & - \\
\end{array}
\]
Address Translation Example

Virtual Address: 0x03D4

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

<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

PPN 0x01 11  0x01 10  0x01 09  0x01 08  0x01 07  0x01 06  0x01 05  0x01 04  0x01 03  0x01 02  0x01 01  0x01 00
Address Translation Example

Physical Address

```
<table>
<thead>
<tr>
<th>CO</th>
<th>Cl</th>
<th>CT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x5</td>
<td>0x0D</td>
</tr>
</tbody>
</table>
```

Hit? Y
Byte: 0x36

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

<table>
<thead>
<tr>
<th>VPN</th>
<th>TLBI</th>
<th>TLBT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Physical Address

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

CT  \[\begin{array}{cc}
11 & 10 \\
9 & 8 \\
7 & 6 \\
5 & 4 \\
3 & 2 \\
1 & 0
\end{array}\]  CO

Cl  \[\begin{array}{cc}
1 & 0 \\
1 & 0 \\
0 & 0 \\
1 & 0 \\
0 & 0 \\
0 & 0
\end{array}\]  CO

PPN  \[\begin{array}{cc}
0 & 0 \\
0 & 0 \\
1 & 0 \\
0 & 0 \\
0 & 0 \\
0 & 0
\end{array}\]  PPO

CO: 0  Cl: 0x8  CT: 0x28  Hit?: N  Byte: Mem
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>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</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>

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

- Instruction fetch
  - L1 d-TLB 64 entries, 4-way
  - L1 i-TLB 128 entries, 4-way
  - L2 unified TLB 512 entries, 4-way

- MMU (addr translation)

- L2 unified cache 256 KB, 8-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)

- QuickPath interconnect
  - 4 links @ 25.6 GB/s each

- Main memory

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

CPU

Virtual address (VA)

VPN VPO

36 12

TLBT TLBI

32 4

TLB miss

L1 TLB (16 sets, 4 entries/set)

VPN1 VPN2 VPN3 VPN4

9 9 9 9

CR3

Page tables

PTE PTE PTE PTE

L2, L3, and main memory

Result

32/64

L1 hit

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

40 12

CT CI CO

Physical address (PA)

L1 miss

Physical address (PA)

CPU

Virtual address (VA)

VPN VPO

36 12

TLBT TLBI

32 4

TLB miss

L1 TLB (16 sets, 4 entries/set)

VPN1 VPN2 VPN3 VPN4

9 9 9 9

CR3

Page tables

PTE PTE PTE PTE

L2, L3, and main memory

Result

32/64

L1 hit

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

40 12

CT CI CO

Physical address (PA)

L1 miss

Physical address (PA)
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

<table>
<thead>
<tr>
<th>XD</th>
<th>Unused</th>
<th>Page physical base address</th>
<th>Unused</th>
<th>G</th>
<th>D</th>
<th>A</th>
<th>CD</th>
<th>WT</th>
<th>U/S</th>
<th>R/W</th>
<th>P=1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>XD</th>
<th>Unused</th>
<th>P=0</th>
<th>Available for OS (page location on disk)</th>
</tr>
</thead>
</table>

#### 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

Virtual address

VPN 1 | VPN 2 | VPN 3 | VPN 4 | VPO

L1 PT | L2 PT | L3 PT | L4 PT

Page global directory | Page upper directory | Page middle directory | Page table

CR3

Physical address of L1 PT

L1 PTE | L2 PTE | L3 PTE | L4 PTE

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

VPN 1 | VPN 2 | VPN 3 | VPN 4

VPO

40

PPN

12

Offset into physical and virtual page

Physical address

512 GB region per entry

1 GB region per entry

2 MB region per entry

4 KB region per entry
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)
- Physical memory
- Kernel code and data

Identical for each process:
- User stack
- Memory mapped region for shared libraries
- Runtime heap (malloc)
- Uninitialized data (.bss)
- Initialized data (.data)
- Program text (.text)

Kernel virtual memory:

Process virtual memory:

physical memory

Identical for each process
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 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!
Finding Shareable Pages

- **Kernel Same-Page Merging**
  - OS scans through all of physical memory, looking for duplicate pages
  - When found, merge into single copy, marked as copy-on-write
  - Implemented in Linux kernel in 2009
  - Limited to pages marked as likely candidates
  - Especially useful when processor running many virtual machines
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
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
Example: Using `mmap` to Support Attack Lab

- **Problem**
  - Want students to be able to perform code injection attacks
  - Shark machine stacks are not executable

- **Solution**
  - Suggested by Sam King (now at UC Davis)
  - Use `mmap` to allocate region of memory marked executable
  - Divert stack to new region
  - Execute student attack code
  - Restore back to original stack
  - Remove mapped region
Using `mmap` to Support Attack Lab

Kernel virtual memory

User stack (created at runtime)

Memory-mapped region for shared libraries

Run-time heap (created by `malloc`)

Read/write segment (`.data`, `.bss`)

Read-only segment (`.init`, `.text`, `.rodata`)

Unused

Memory invisible to user code

%rsp (stack pointer)

0x40000000

0
Using `mmap` to Support Attack Lab

- Kernel virtual memory
  - User stack (created at runtime)
  - Memory-mapped region for shared libraries
  - Region created by `mmap`
  - Run-time heap (created by `malloc`)
  - Read/write segment (.data, .bss)
  - Read-only segment (.init, .text, .rodata)
  - Unused

Memory mapped region for shared libraries

- Memory invisible to user code
- `%rsp` (stack pointer)

0x55586000

Run-time heap (created by `malloc`)

0x40000000

Read/write segment (.data, .bss)

0x40000000

Read-only segment (.init, .text, .rodata)
Using mmap to Support Attack Lab

- **Kernel virtual memory**
- **User stack** (created at runtime)
- **Memory-mapped region for shared libraries**
- **Region created by mmap**
- **Run-time heap** (created by malloc)
- **Read/write segment** (.data, .bss)
- **Read-only segment** (.init, .text, .rodata)
- **Unused**

Memory invisible to user code

%rsp (stack pointer)

Frame for launch
Frame for test
Frame for getbuf

0x55586000

0x40000000

0x40000000
Using `mmap` to Support Attack Lab

- Kernel virtual memory
- User stack (created at runtime)
- Memory-mapped region for shared libraries
- Run-time heap (created by `malloc`)
- Read/write segment (`.data`, `.bss`)
- Read-only segment (`.init`, `.text`, `.rodata`)
- Unused

Memory invisible to user code

%rsp (stack pointer)
Using **mmap** to Support Attack Lab

Allocate new region

```c
void *new_stack = mmap(START_ADDR, STACK_SIZE, PROT_EXEC|PROT_READ|PROT_WRITE,
    MAP_PRIVATE | MAP_GROWSDOWN | MAP_ANONYMOUS | MAP_FIXED,
    0, 0);
if (new_stack != START_ADDR) {
    munmap(new_stack, STACK_SIZE);
    exit(1);
}
```

Divert stack to new region & execute attack code

```c
stack_top = new_stack + STACK_SIZE - 8;
asm("movq %%rsp,%%rax ; movq %1,%%rsp ;
   movq %%rax,%0"
   : "=r" (global_save_stack) // %0
   : "r" (stack_top) // %1
);
launch(global_offset);
```

Restore stack and remove region

```c
asm("movq %0,%%rsp"
    :
    : "r" (global_save_stack) // %0
);
munmap(new_stack, STACK_SIZE);
```
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
  - Providing memory protection