Virtual Memory III
CSE 351 Autumn 2018

Instructor:
Justin Hsia

Teaching Assistants:
Akshat Aggarwal
An Wang
Andrew Hu
Brian Dai
Britt Henderson
James Shin
Kevin Bi
Kory Watson
Riley Germundson
Sophie Tian
Teagan Horkan

https://xkcd.com/648/
Administrivia

- Lab 4 due Monday (11/26)
- Homework 5 due next Friday (11/30)

- “Virtual section” on virtual memory released
  - 3 PDFs: VM overview, worksheet, and solutions
  - Linked in the code section of today’s lecture
  - See Piazza post for links and videos
Quick Review

- What do Page Tables map?
- Where are Page Tables located?
- How many Page Tables are there?
- Can your process tell if a page fault has occurred?
- True / False: Virtual Addresses that are contiguous will always be contiguous in physical memory
- TLB stands for ________________________ and stores ________________
Address Translation

- VM is complicated, but also elegant and effective
  - Level of indirection to provide isolated memory & caching
  - TLB as a cache of page tables avoids two trips to memory for every memory access
Simple Memory System Example (small)

- **Addressing**
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes
Simple Memory System: Page Table

- Only showing first 16 entries (out of ____)
  - **Note**: showing 2 hex digits for PPN even though only 6 bits
  - **Note**: other management bits not shown, but part of PTE

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>7</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>8</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>
Simple Memory System: TLB

- 16 entries total
- 4-way set associative

**Diagram:**

- TLB tag
- TLB index
- virtual page number
- virtual page offset

**Table: TLB Entries**

<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>

**Question:** Why does the TLB ignore the page offset?
Simple Memory System: Cache

- Direct-mapped with $K = 4$ B, $C/K = 16$
- Physically addressed

**Note:** It is just coincidence that the PPN is the same width as the cache Tag
# Current State of Memory System

## TLB:

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>V</th>
<th>Tag</th>
<th>PPN</th>
<th>V</th>
<th>Tag</th>
<th>PPN</th>
<th>V</th>
<th>Tag</th>
<th>PPN</th>
<th>V</th>
<th>Tag</th>
<th>PPN</th>
<th>V</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>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Cache:

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>V</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>

## Page table (partial):

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>V</th>
<th>VPN</th>
<th>PPN</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>28</td>
<td>1</td>
<td>8</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>–</td>
<td>0</td>
<td>9</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>1</td>
<td>A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>02</td>
<td>1</td>
<td>B</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>0</td>
<td>C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>1</td>
<td>D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>–</td>
<td>0</td>
<td>E</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>–</td>
<td>0</td>
<td>F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>
Memory Request Example #1

- **Virtual Address:** 0x03D4

![Diagram of TLB entries with virtual and physical addresses]

- **Physical Address:**

![Diagram of cache and page table with virtual and physical addresses]

**Note:** It is just coincidence that the PPN is the same width as the cache Tag.
Memory Request Example #2

- **Virtual Address:** 0x038F

![Diagram of memory request example](image)

- **Physical Address:**

![Diagram of physical address](image)

Note: It is just coincidence that the PPN is the same width as the cache Tag.
Memory Request Example #3

- **Virtual Address:** `0x0020`

- **Physical Address:**

  - `CT` ______
  - `CI` ______
  - `CO` ______
  - `Cache Hit?` ___
  - `Data (byte)` _______

---

**Note:** It is just coincidence that the PPN is the same width as the cache Tag.
Memory Request Example #4

- **Virtual Address:** 0x036B
  - VPN _______  TLBT _____  TLBI _____  TLB Hit? ____  Page Fault? ____  PPN _______

- **Physical Address:**
  - CT _______  CI _______  CO _______  Cache Hit? ____  Data (byte) ________

Note: It is just coincidence that the PPN is the same width as the cache Tag
Memory Overview

- `movl 0x8043ab, %rdi`
Page Table Reality

- Just one issue... the numbers don’t work out for the story so far!

- The problem is the page table for each process:
  - Suppose 64-bit VAs, 8 KiB pages, 8 GiB physical memory
  - How many page table entries is that?
  - About how long is each PTE?

- Moral: Cannot use this naïve implementation of the virtual→physical page mapping – it’s way too big
A Solution: Multi-level Page Tables

This is called a **page walk**

**Virtual Address**

- Level 1 page table
- Level 2 page table
- Level k page table

**Physical Address**

- **VPN** → **PTE**
- **VPN** → **PTE**
- **VPN** → **PTE**

---

---

This is extra (non-testable) material
Multi-level Page Tables

- A tree of depth \( k \) where each node at depth \( i \) has up to \( 2^j \) children if part \( i \) of the VPN has \( j \) bits
- Hardware for multi-level page tables inherently more complicated
  - But it’s a necessary complexity – 1-level does not fit
- Why it works: Most subtrees are not used at all, so they are never created and definitely aren’t in physical memory
  - Parts created can be evicted from cache/memory when not being used
  - Each node can have a size of \( \sim 1\text{-}100\text{KB} \)
- But now for a \( k \)-level page table, a TLB miss requires \( k + 1 \) cache/memory accesses
  - Fine so long as TLB misses are rare – motivates larger TLBs

This is extra (non-testable) material
Practice VM Question

- Our system has the following properties
  - 1 MiB of physical address space
  - 4 GiB of virtual address space
  - 32 KiB page size
  - 4-entry fully associative TLB with LRU replacement

a) Fill in the following blanks:

<table>
<thead>
<tr>
<th>Entries in a page table</th>
<th>Minimum bit-width of PTBR</th>
</tr>
</thead>
<tbody>
<tr>
<td>_______</td>
<td>_______</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TLBT bits</th>
<th>Max # of valid entries in a page table</th>
</tr>
</thead>
<tbody>
<tr>
<td>_______</td>
<td>_______</td>
</tr>
</tbody>
</table>
Practice VM Question

- One process uses a page-aligned square matrix mat[] of 32-bit integers in the code shown below:

```c
#define MAT_SIZE = 2048
for(int i = 0; i < MAT_SIZE; i++)
    mat[i*(MAT_SIZE+1)] = i;
```

b) What is the largest stride (in bytes) between successive memory accesses (in the VA space)?
Practice VM Question

- One process uses a page-aligned square matrix `mat[]` of 32-bit integers in the code shown below:

```c
#define MAT_SIZE = 2048
for(int i = 0; i < MAT_SIZE; i++)
    mat[i*(MAT_SIZE+1)] = i;
```

c) Assuming all of `mat[]` starts on disk, what are the following hit rates for the execution of the for-loop?

- _________ TLB Hit Rate
- _________ Page Table Hit Rate
For Fun: DRAMMER Security Attack

- Why are we talking about this?
  - **Recent:** Announced in October 2016; Google released Android patch on November 8, 2016
  - **Relevant:** Uses your system’s memory setup to gain elevated privileges
    - Ties together some of what we’ve learned about virtual memory and processes
  - **Interesting:** It’s a software attack that uses only hardware vulnerabilities and requires no user permissions
Underlying Vulnerability: Row Hammer

- Dynamic RAM (DRAM) has gotten denser over time
  - DRAM cells physically closer and use smaller charges
  - More susceptible to “disturbance errors” (interference)
- DRAM capacitors need to be “refreshed” periodically (~64 ms)
  - Lose data when loss of power
  - Capacitors accessed in rows
- Rapid accesses to one row can flip bits in an adjacent row!
  - ~ 100K to 1M times
Row Hammer Exploit

- Force constant memory access
  - Read then flush the cache
  - `clflush` – flush cache line
    - Invalidates cache line containing the specified address
    - Not available in all machines or environments
  - Want addresses X and Y to fall in activation target row(s)
    - Good to understand how banks of DRAM cells are laid out

- The row hammer effect was discovered in 2014
  - Only works on certain types of DRAM (2010 onwards)
  - These techniques target x86 machines
Consequences of Row Hammer

- Row hammering process can affect another process via memory
  - Circumvents virtual memory protection scheme
  - Memory needs to be in an adjacent row of DRAM

- Worse: privilege escalation
  - Page tables live in memory!
  - Hope to change PPN to access other parts of memory, or change permission bits
  - **Goal**: gain read/write access to a page containing a page table, hence granting process read/write access to *all of physical memory*
Effectiveness?

- Doesn’t seem so bad – random bit flip in a row of physical memory
  - Vulnerability affected by system setup and physical condition of memory cells

- Improvements:
  - Double-sided row hammering increases speed & chance
  - Do system identification first (e.g. Lab 4)
    - Use timing to infer memory row layout & find “bad” rows
    - Allocate a huge chunk of memory and try many addresses, looking for a reliable/repeatable bit flip
  - Fill up memory with page tables first
    - fork extra processes; hope to elevate privileges in any page table
What’s DRAMMER?

- No one previously made a huge fuss
  - **Prevention:** error-correcting codes, target row refresh, higher DRAM refresh rates
  - Often relied on special memory management features
  - Often crashed system instead of gaining control

- Research group found a *deterministic* way to induce row hammer exploit in a non-x86 system (ARM)
  - Relies on predictable reuse patterns of standard physical memory allocators
  - Universiteit Amsterdam, Graz University of Technology, and University of California, Santa Barbara
DRAMMER Demo Video

- It’s a shell, so not that sexy-looking, but still interesting
  - Apologies that the text is so small on the video
How did we get here?

- Computing industry demands more and faster storage with lower power consumption
- Ability of user to circumvent the caching system
  - `clflush` is an unprivileged instruction in x86
  - Other commands exist that skip the cache
- Availability of virtual to physical address mapping
  - Example: `/proc/self/pagemap` on Linux (not human-readable)

- Google patch for Android (Nov. 8, 2016)
  - Patched the ION memory allocator
More reading for those interested

- DRAMMER paper: 
- Google Project Zero: 
  https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html
- First row hammer paper: 
- Wikipedia: 
  https://en.wikipedia.org/wiki/Row_hammer