Virtual Memory III
CSE 351 Autumn 2021

Instructor:
Justin Hsia

Teaching Assistants:
Allie Pfleger
Anirudh Kumar
Assaf Vayner
Atharva Deodhar
Celeste Zeng
Dominick Ta
Francesca Wang
Hamsa Shankar
Isabella Nguyen
Joy Dang
Julia Wang
Maggie Jiang
Monty Nitschke
Morel Fotsing
Sanjana Chintalapati

https://xkcd.com/648/
Relevant Course Information

❖ hw21 due Friday (11/26)
❖ Lab 4 due Monday (11/29)
❖ hw22 due Wednesday (12/1)

❖ “Virtual Section” on Virtual Memory
  ▪ Worksheet and solutions released on Wednesday or Thursday
  ▪ Videos will be released of material review and problem solutions

❖ Final Dec. 13-15, regrade requests Dec. 18-19
Reading Review

❖ Terminology:
  ▪ Address translation: page hit, page fault
  ▪ Translation Lookaside Buffer (TLB): TLB Hit, TLB Miss

❖ Questions from the Reading?
Address Translation: Page Hit

1) Processor sends *virtual* address to MMU (*memory management unit*)
2-3) MMU fetches PTE from page table in cache/memory
   (Uses PTBR to find beginning of page table for current process)
4) MMU sends *physical* address to cache/memory requesting data
5) Cache/memory sends data to processor

VA = Virtual Address  
PTEA = Page Table Entry Address  
PA = Physical Address  
Data = Contents of memory stored at VA originally requested by CPU
Address Translation: Page Fault

1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in cache/memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction
Hmm... Translation Sounds Slow

- The MMU accesses memory *twice*: once to get the PTE for translation, and then again for the actual memory request
  - The PTEs *may* be cached in L1 like any other memory word
    - But they may be evicted by other data references
    - And a hit in the L1 cache still requires 1-3 cycles

- *What can we do to make this faster?*
  - **Solution:** add another cache!
Speeding up Translation with a TLB

Translation Lookaside Buffer (TLB):
- Small hardware cache in MMU
  - Split VPN into TLB Tag and TLB Index based on # of sets in TLB
- Maps virtual page numbers to physical page numbers
- Stores page table entries for a small number of pages
  - Modern Intel processors have 128 or 256 entries in TLB
- Much faster than a page table lookup in cache/memory

![Diagram of TLB and page table entries]
A TLB hit eliminates a memory access!
A TLB miss incurs an additional memory access (the PTE)
- Fortunately, TLB misses are rare
Fetching Data on a Memory Read

1) Check TLB
   - **Input**: VPN, **Output**: PPN
   - **TLB Hit**: Fetch translation, return PPN
   - **TLB Miss**: Check page table (in memory)
     - *Page Table Hit*: Load page table entry into TLB
     - *Page Fault*: Fetch page from disk to memory, update corresponding page table entry, then load entry into TLB

2) Check cache
   - **Input**: physical address, **Output**: data
   - **Cache Hit**: Return data value to processor
   - **Cache Miss**: Fetch data value from memory, store it in cache, return it to processor
Address Translation

Virtual Address

TLB Lookup

- **TLB Miss**: Check the Page Table
  - Page not in Mem: Page Fault (OS loads page)
    - Find in Disk
  - Page in Mem: Update TLB
    - Find in Mem

- **TLB Hit**: Protection Check
  - Access Denied
    - SIGSEGV
  - Access Permitted
    - Check cache
      - Miss
      - Hit

Physical Address
Address Manipulation

request from CPU: \( n \)-bit virtual address

split to access TLB: TLB Tag TLB Index Page Offset

(on TLB miss) access PT: Virtual Page Number Page offset

\( m \)-bit physical address: Physical Page Number Page offset

split to access cache: Cache Tag Cache Index Block offset

TRANSLATION
Context Switching Revisited

❖ What needs to happen when the CPU switches processes?

▪ Registers:
  • Save state of old process, load state of new process
  • Including the Page Table Base Register (PTBR)

▪ Memory:
  • Nothing to do! Pages for processes already exist in memory/disk and protected from each other

▪ TLB:
  • *Invalidate* all entries in TLB – mapping is for old process’ VAs

▪ Cache:
  • Can leave alone because storing based on PAs – good for shared data
Summary of Address Translation Symbols

❖ Basic Parameters
- N = 2^n \quad \text{Number of addresses in virtual address space}
- M = 2^m \quad \text{Number of addresses in physical address space}
- P = 2^p \quad \text{Page size (bytes)}

❖ Components of the virtual address (VA)
- VPO \quad \text{Virtual page offset}
- VPN \quad \text{Virtual page number}
- TLBI \quad \text{TLB index}
- TLBT \quad \text{TLB tag}

❖ Components of the physical address (PA)
- PPO \quad \text{Physical page offset (same as VPO)}
- PPN \quad \text{Physical page number}
Simple Memory System Example (small)

- **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

<table>
<thead>
<tr>
<th>VPO</th>
<th>VPN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Virtual Page Number (VPN) and Virtual Page Offset (VPO)

Physical Page Number (PPN) and Physical Page Offset (PPO)
# 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>
<th>VPN</th>
<th>PPN</th>
<th>Valid</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>
Simple Memory System: TLB

- 16 entries total
- 4-way set associative

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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>03</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>07</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

### Page table (partial):

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>V</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>

### Cache:

#### Index

<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>

<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>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>
Memory Request Example #1

- **Virtual Address:** 0x03D4

![Virtual Address Diagram]

- **Physical Address:**

![Physical Address Diagram]

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

- **Virtual Address:** 0x038F

![Virtual Address Diagram]

- **Physical Address:**

![Physical Address Diagram]

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

❖ Virtual Address: \(0x0020\)

\[
\begin{array}{cccccccccccccc}
13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

VPN _____ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

❖ Physical Address:

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

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

![Virtual Address Diagram]

- **Physical Address:**

![Physical Address Diagram]

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

- `movl 0x8043ab, %rdi`
Virtual Memory Summary

❖ Programmer’s view of virtual memory
  ▪ Each process has its own private linear address space
  ▪ Cannot be corrupted by other processes

❖ System view of virtual memory
  ▪ Uses memory efficiently by caching virtual memory pages
    • Efficient only because of locality
  ▪ Simplifies memory management and sharing
  ▪ Simplifies protection by providing permissions checking
❖ Multi-level Page Tables
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

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 \(~1\)-100KB
- 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
For Fun: DRAMMER Security Attack

- Why are we talking about this?
  - **Recent**: First announced in October 2016; latest attack variant announced November 2021
  - **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

By Dsimic (modified), CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=38868341
Row Hammer Exploit

- Force constant memory access
  - Read then flush the cache
  - \texttt{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 \textit{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

\begin{verbatim}
hammertime:
  mov (X), %eax
  mov (Y), %ebx
  clflush (X)
  clflush (Y)
  jmp hammertime
\end{verbatim}
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 rowhammer paper:

❖ Latest non-uniform, frequency-based exploit:
https://comsec.ethz.ch/research/dram/blacksmith/