Virtual Memory III
CSE 351 Winter 2020

Instructor:
Ruth Anderson

Teaching Assistants:
Jonathan Chen
Justin Johnson
Porter Jones
Josie Lee
Jeffery Tian
Callum Walker
Eddy (Tianyi) Zhou

https://xkcd.com/648/
Administrivia

- Lab 4 – Due Tonight! (3/02)
  - Cache parameter puzzles and code optimizations

- hw19 due Wednesday (3/04)

- hw20 due Friday (3/06)

— Wash Your Hands
— Don’t Touch Your Face
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   PTE= Page Table Entry
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
A TLB hit eliminates a memory access!
TLB Miss
(page has not been used recently)

- A TLB miss incurs an additional memory access (the PTE)
  - Fortunately, TLB misses are rare
Fetching Data on a Memory Read

1) Check TLB  \((\text{translate } VA \rightarrow PA)\)
   - \textbf{Input:} VPN, \textbf{Output:} PPN
   - \textit{TLB Hit:} Fetch translation, return PPN
   - \textit{TLB Miss:} Check page table (in memory)
     - \textit{Page Table Hit:} Load page table entry into TLB
     - \textit{Page Fault:} Fetch page from disk to memory, update corresponding page table entry, then load entry into TLB

2) Check cache  \((\text{fetch requested data})\)
   - \textbf{Input:} physical address, \textbf{Output:} data
   - \textit{Cache Hit:} Return data value to processor
   - \textit{Cache Miss:} Fetch data value from memory, store it in cache, return it to processor
Address Translation

Virtual Address

1. TLB Lookup

   TLB Miss

   1. Check the Page Table
      - Val ≠ 0: Page not in Mem
      - Val = 1: Page in Mem

      1. Page Fault (OS loads page)
         - Find in Disk

      2. Update TLB
         - Find in Mem

   2. TLB Hit

      Protection Check
      - Access Denied
        - SIGSEGV
      - Access Permitted
        - Physical Address

      Check cache
      - Miss
      - Hit
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

$\textit{m}$-bit physical address:

split to access cache: Physical Page Number Page offset

Cache Tag Cache Index 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: Physically Indexed
    - Can leave alone because storing based on PAs – good for shared data
Summary of Address Translation 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)**
  - VPO  Virtual page offset
  - VPN  Virtual page number
  - TLBI  TLB index
  - TLBT  TLB tag

- **Components of the physical address (PA)**
  - PPO  Physical page offset (same as VPO)
  - PPN  Physical page number
Simple Memory System Example (small)

- **Addressing**
  - 14-bit virtual addresses \( n = 14 \) bits \( \iff \) \( N = 16 \) KiB VA space
  - 12-bit physical address \( m = 12 \) bits \( \iff \) \( M = 4 \) KiB PA space
  - Page size = 64 bytes \( \rightarrow P = 64 \) B \( \iff \) \( p = 6 \) bits
Simple Memory System: Page Table

- Only showing first 16 entries (out of \(2^8 = 256\) one for every virtual page)
  - **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>0x13</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>

\(D, R, W, X\)
Simple Memory System: TLB

- 16 entries total
- 4-way set associative

Why does the TLB ignore the page offset?

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

Circled #s refer to Memory Request Example #

**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>
</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>

**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>
</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>V</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>
Memory Request Example #1

- **Virtual Address:** 0x03D4

  - **VPN:** 0xF
  - **TLBT:** 0x03
  - **TLBI:** 3
  - **TLB Hit?** Y
  - **Page Fault?** N
  - **PPN:** 0x0D

  - **Note:** It is just coincidence that the PPN is the same width as the cache Tag

- **Physical Address:**

  - **CT:** 0x0D
  - **CI:** 5
  - **CO:** 0
  - **Cache Hit?** Y
  - **Data (byte):** 0x36
Memory Request Example #2

- **Virtual Address**: 0x038F

  

  ![Virtual Address Diagram](image)

  - VPN 0x0E
  - TLBT 0x03
  - TLBI 2
  - TLB Hit? N
  - Page Fault? Y
  - PPN n/a

- **Physical Address**:

  

  ![Physical Address Diagram](image)

  - 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 #3

- **Virtual Address:** \(0x0020\)

  ![Virtual Address Diagram]

- **Physical Address:**

  ![Physical Address Diagram]

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

- **Virtual Address:** \(0x0020\)

  - TLBT
  - TLBI
  - VPN
  - VPO

- **Physical Address:**

  - CT
  - CI
  - CO
  - PPN
  - PPO

**Cache Hit?** \(N\)

**Data (byte)** \(n/a\)
Memory Request Example #4

- **Virtual Address:** \(0x036B\)

  ![Virtual Address Diagram]

  - TLBT: 00001110
  - TLBI: 11011001
  - VPN: 00
  - VPO: 0

- **Physical Address:**

  ![Physical Address Diagram]

  - CT: 002D
  - CI: A
  - CO: 3
  - PPN: 00
  - PPO: 0

  - Cache Hit?: Y
  - Data (byte): \(0x3B\)

**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?
    - 1 PTE for every virtual page
      \[ 2^{n-p} = 2^{51} \text{ PTEs} \]
  - About how long is each PTE?
    - PPN width + management bits = 20 + 5 = 25 bits \( \approx \) 3 bytes

- 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
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:

- \(2^{17}\) Entries in a page table
- \(2^{n-p}\) \(\leq\) # of virtual pages
- \(17\) TLBT bits
- \(\frac{V_{PTN}}{2^n} = TLBT/TLBI\) here TLBI = 0
- 20 Minimum bit-width of PTBR \(\leftarrow\) physical address of PT
- \(2^5\) Max # of valid entries in a page table \(\leftarrow\) # of pages in physical memory
- \(2^{m-p}\)
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)?

  The stride is always 2049 because the array index is incremented by 2049 for each access.

  ```c
  i
  0 0
  1 2049
  2 2*2049
  ; ;
  ```
Practice VM Question

\[ \text{page size} = 32 \text{ KiB} = 2^{15} \text{ B} \]

- One process uses a page-aligned square matrix \( \text{mat}[\cdot] \) 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 \( \text{mat}[\cdot] \) starts on disk, what are the following hit rates for the execution of the for-loop?

  - TLB Hit Rate: \( \frac{3}{4} = 75\% \)
  - Page Table Hit Rate: 0%

  **access pattern:** single write to index
  never revisit indices (always increasing)
  we access every row of matrix exactly once
  each page holds \( 2^{15}/2^{13} = 4 \) rows of matrix
  within each page: MTHH

  **only access PT on TLB Miss**
  because \( \text{mat}[\cdot] \) on disk, each first access to page causes page fault.
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
Memory System Summary

Memory Caches (L1/L2/L3)
- Purely a speed-up technique
- Behavior invisible to application programmer and (mostly) OS
- Implemented totally in hardware

Virtual Memory
- Supports many OS-related functions
  - Process creation, task switching, protection
- Operating System (software)
  - Allocates/shares physical memory among processes
  - Maintains high-level tables tracking memory type, source, sharing
  - Handles exceptions, fills in hardware-defined mapping tables
- Hardware
  - Translates virtual addresses via mapping tables, enforcing permissions
  - Accelerates mapping via translation cache (TLB)