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?
  VPN → PPN or disk address

- Where are Page Tables located?
  physical memory

- How many Page Tables are there?
  one per process

- Can your process tell if a page fault has occurred?
  No. MMU/OS throws page fault; process just waits for data

- True / False: Virtual Addresses that are contiguous will always be contiguous in physical memory
  True / False: Virtual Addresses that are contiguous will always be contiguous in physical memory
  page boundary: \[ x \] \[ x+1 \]
  pages can be mapped to any slot in physical mem

- TLB stands for translation lookaside buffer and stores page table entries
  British for “cache”
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: \( n = 14 \) bits \( \iff \) \( N = 16 \text{ KiB} \) \( \text{VA space} \)
  - 12-bit physical address: \( m = 12 \) bits \( \iff \) \( M = 4 \text{ KiB} \) \( \text{PA space} \)
  - Page size = 64 bytes: \( P = 64 \text{ B} \) \( \iff \) \( p = 6 \) bits

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

VPN width = \( n-p \)

Virtual Page Number

VPO

Virtual Page Offset

2\(^{n-p}\) pages in VA space

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

PPN width = \( m-p \)

Physical Page Number

PPN

Physical Page Offset

2\(^{m-p}\) pages in PA space
Simple Memory System: Page Table

- Only showing first 16 entries (out of $2^n = 256$)
  - **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>1</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>1</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>1</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>1</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>1</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

\[ \frac{16}{4} = 4 \text{ sets} \]

Why does the TLB ignore the page offset?

not part of its job!

(address translation)

Valid

PPN

Tag

Set

0

1

2

3

Way 0

Way 1

Way 2

Way 3

03  
03  
02  
07  

09 0D  
02 –  
08 –  
03  

00 –  
04 –  
06 –  
0A –  

07 02 1

04 02 1

06 03 – 0

0A 34 1

virtual page number

virtual page offset
Simple Memory System: Cache

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

```
<table>
<thead>
<tr>
<th>Index</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>Index</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>
```

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>② 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>④ 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 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>① 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>① 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>②E</td>
<td>–</td>
<td>0X</td>
</tr>
<tr>
<td>F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>
Memory Request Example #1

- **Virtual Address:** \(0x03D4\)

  - TLBT
  - TLBI

<table>
<thead>
<tr>
<th>13</th>
<th>12</th>
<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>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</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>

  - VPN
  - VPO

  - TLB Hit? \(\text{Y}\)
  - Page Fault? \(\text{N}\)
  - PPN \(0x0D\)

- **Physical Address:**

  - CT
  - CI
  - CO

<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>

  - PPN
  - PPO

  - CT \(0xD\)
  - CI \(5\)
  - CO \(0\)

  - Cache Hit? \(\text{Y}\)
  - Data (byte) \(0x36\)

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

- **Virtual Address:** 0x038F

  ```
  TLBT   TLBI
  13 12 11 10  9  8  7  6  5  4  3  2  1  0
  0  0  0  0  1  1  1  0  0  0  1  1  1  1
  ```

- **Physical Address:**

  ```
  CT   CI   CO
  11 10  9  8  7  6  5  4  3  2  1  0
  ```

  ```
  PPN    PPO
  ```

  **VPN** 0x0E, **TLBT** 0x03, **TLBI** 2, **TLB Hit?** Y, **Page Fault?** Y, **PPN** na

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

- **Physical Address:**

  ```
  CT   CI   CO
  11 10  9  8  7  6  5  4  3  2  1  0
  ```

  ```
  PPN    PPO
  ```

  **CT** _____, **CI** _____, **CO** _____, **Cache Hit?** ___, **Data (byte)** ______
Memory Request Example #3

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

  - TLBT: 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  - TLBI: 0 0 0 0 0 0 0 0 1 0 0 0 0 0
  - VPN: \(0x00\)
  - VPO: \(0x0\)

- **Physical Address:**

  - CT: 11 10 9 8 7 6 5 4 3 2 1 0
  - CI: \(0x28\)
  - CO: \(0\)
  - PPN: \(0x28\)
  - PPO: \(0\)

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

**Physical Address:**

- CT: \(0x28\)
  - CI: \(0\)
  - CO: \(0\)
  - Cache Hit? \(N\)
  - Data (byte) n/a
Memory Request Example #4

- **Virtual Address**: \(0x036B\)
  - TLBT: \(0000\)
  - TLBI: \(1101\)
  - VPN: \(0x0D\)
  - VPO: \(0x0D\)
  - TLBT: \(0x03\)
  - TLBI: \(1\)
  - TLB Hit? \(Y\)
  - Page Fault? \(N\)
  - PPN: \(0x2D\)

- **Physical Address**:
  - CT: \(0x2D\)
  - CI: \(A\)
  - CO: \(3\)
  - 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?
    \[ 2^{n-p} = 2^{51} \text{ PTEs} \]
  - About how long is each PTE?
    \[ m-p \text{ bits} \approx 3 \text{ bytes} \]
  - Moral: Cannot use this naïve implementation of the virtual→physical page mapping – it’s way too big

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

This is called a page walk

Virtual Address

n-1

VPN 1  VPN 2  ...  VPN k

Level 1 page table  Level 2 page table  ...  Level k page table

arrays of pointers

m-1

VPN

PTE

VPN

PTE

VPN

PTE

TLB

PPN

PPO

Physical Address

This is extra (non-testable) material

Page table base register (PTBR)

VPN → PTE

VPN → PTE

VPN → PTE

VPN → PTE

This is an example.

8 virtual pages,

split VPN into 1-bit fields

VPO

VPNs:

VPN → PTE

VPN → PTE

VPN → PTE

PTBR

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

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:

- \(2^{17}\) Entries in a page table
- \(2^{n-p}\) \(\leq\) \# of virtual pages
- \(17\) TLBT bits
  - VPN→TLBT/TLBI
  - here TLBI = 0
- \(20\) Minimum bit-width of PTBR
  - physical address of PT
- \(2^5\) Max # of valid entries in a page table
  - \# of pages in physical memory
- \(2^{15}\)
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 integers, which is equal to 2049 * 4 bytes.
Practice VM Question

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

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

  ```c
  #define MAT_SIZE = 2048
  int3 = 2^{13} \text{B}
  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**: \(\frac{3}{4} = 75\%\)
- **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: M THTHH

only access PT on TLB Miss because `mat[]` on disk, each first access to page causes page fault.
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

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
  - `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

```
hammertime:
  mov (X), %eax
  mov (Y), %ebx
  clflush (X)
  clflush (Y)
  jmp hammertime
```
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 determination 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