Virtual Memory III
CSE 351 Autumn 2016

Instructor:
Justin Hsia

Teaching Assistants:
Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun

https://xkcd.com/720/
Administrivia

- Lab 4 due Monday, Nov. 28

- “Virtual section” on virtual memory over the break
  - Worksheet and solutions will be posted for extra practice
  - Expect to see Justin’s past exam questions posted as well
    - Find it at the bottom of the course schedule under the Final date
Quick Review

- What do Page Tables map?
  \[ \text{VPN} \rightarrow \text{PPN or disk address} \]

- Where are Page Tables located?
  physical memory

- How many Page Tables are there?
  one per process

- Can your program tell if a page fault has occurred?
  \( \text{No. MMM throws page fault exception, process just waits} \)

- What is thrashing?
  excessive paging in/out

- True / False: Virtual Addresses that are contiguous will always be contiguous in physical memory
  \[ x \mid x+1 \] page boundary

- TLB stands for \underline{translation lookaside buffer} and stores \underline{page table entries}
Quick Review Answers

- What do Page Tables map?
  - VPN → PPN or disk address

- Where are Page Tables located?
  - In physical memory

- How many Page Tables are there?
  - One per process

- Can your program tell if a page fault has occurred?
  - Nope, but it has to wait a long time

- What is thrashing?
  - Constantly paging out and paging in

- True / False: Virtual Addresses that are contiguous will always be contiguous in physical memory
  - Could fall across a page boundary

- TLB stands for Translation Lookaside Buffer and stores page table entries
Review: 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
Memory Overview

- `movl 0x8043ab, %rdi`
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
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
    \[ 2^{n-p} = 2^{51} \text{ PTE} \]
  - About how long is each PTE?
    PPN + management bits
    \[ m - p = 20 \text{ bits} \rightarrow 24 \text{ bits} = 3 \text{B} \]

- **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**

- **Page table base register (PTBR)**
- **VPN 1**
- **VPN 2**
- **...**
- **VPN k**

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

- **VPN → PTE → PPN**

---

**Physical Address**

- **PPN**
- **PPO**
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 \( m = 20 \)
  - 4 GiB of virtual address space \( n = 32 \)
  - 32 KiB page size \( p = 15 \)
  - 4-entry fully associative TLB with LRU replacement

a) Fill in the following blanks:

\[
\begin{align*}
\frac{2^{17}}{2^{n-p}} & \leq \# \text{ of virtual pages} \\
17 & \text{ TLBT bits} \\
20 & \text{ Minimum bit-width of PTBR} \\
\frac{2^5}{2^{n-p}} & \text{ Max # of valid entries in a page table}
\end{align*}
\]
Practice VM Question

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

  ```
  #define MAT_SIZE = 2048 = 2^n
  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)?

<table>
<thead>
<tr>
<th>i</th>
<th>index</th>
<th>2049 ints = 2049*4 B = stride</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2049</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2*2049</td>
<td></td>
</tr>
</tbody>
</table>

accesses diagonal of matrix
Practice VM Question

- One process uses a page-aligned square matrix \( \text{mat[]} \) 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;
```

- What are the following hit rates for the first execution of the for loop? (assume all of mat[] starts on disk)
  - **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: \( \text{MTHHH} \)

- page size = 32 KiB = \( 2^{15} \) B
- \( 2^n \) integers = \( 2^{13} \) B
For Fun: DRAMMER Security Attack

- Why are we talking about this?
  - Current: Announced in October 2016; Google released Android patch on November 8
  - 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/wiki/index.php?curid=38868341
Row Hammer Exploit

- Force memory access by constantly reading and then flushing 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