Virtual Memory Wrap-Up

CSE 351 Autumn 2017

Instructor:

Justin Hsia

Teaching Assistants:

Lucas Wotton

Michael Zhang

Parker DeWilde

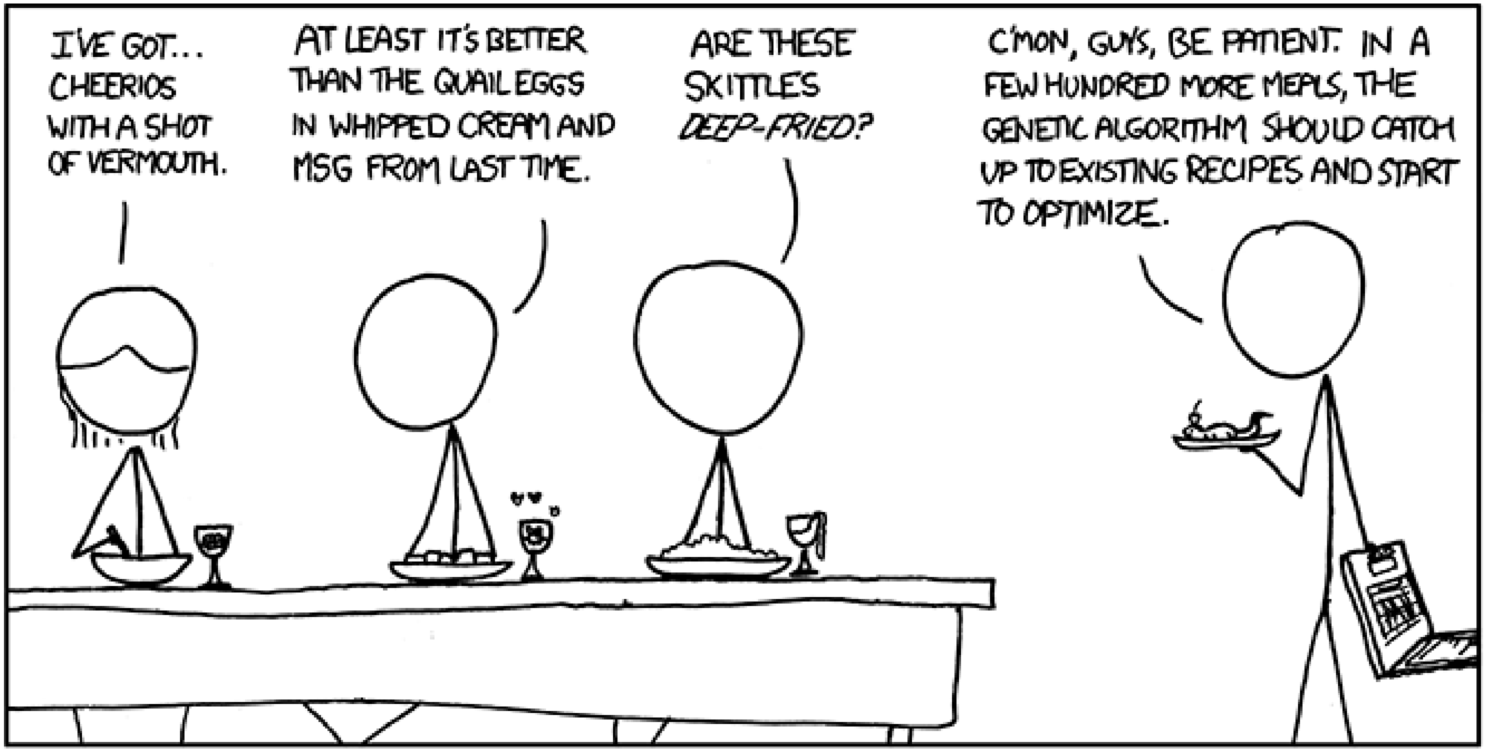
Ryan Wong

Sam Gehman

Sam Wolfson

Savanna Yee

Vinny Palaniappan



WE'VE DECIDED TO DROP THE CS DEPARTMENT FROM OUR WEEKLY DINNER PARTY HOSTING ROTATION.

https://xkcd.com/720/

Administrivia

- Lab 4 due Monday (11/27)
- Homework 5 due next Friday (12/1)

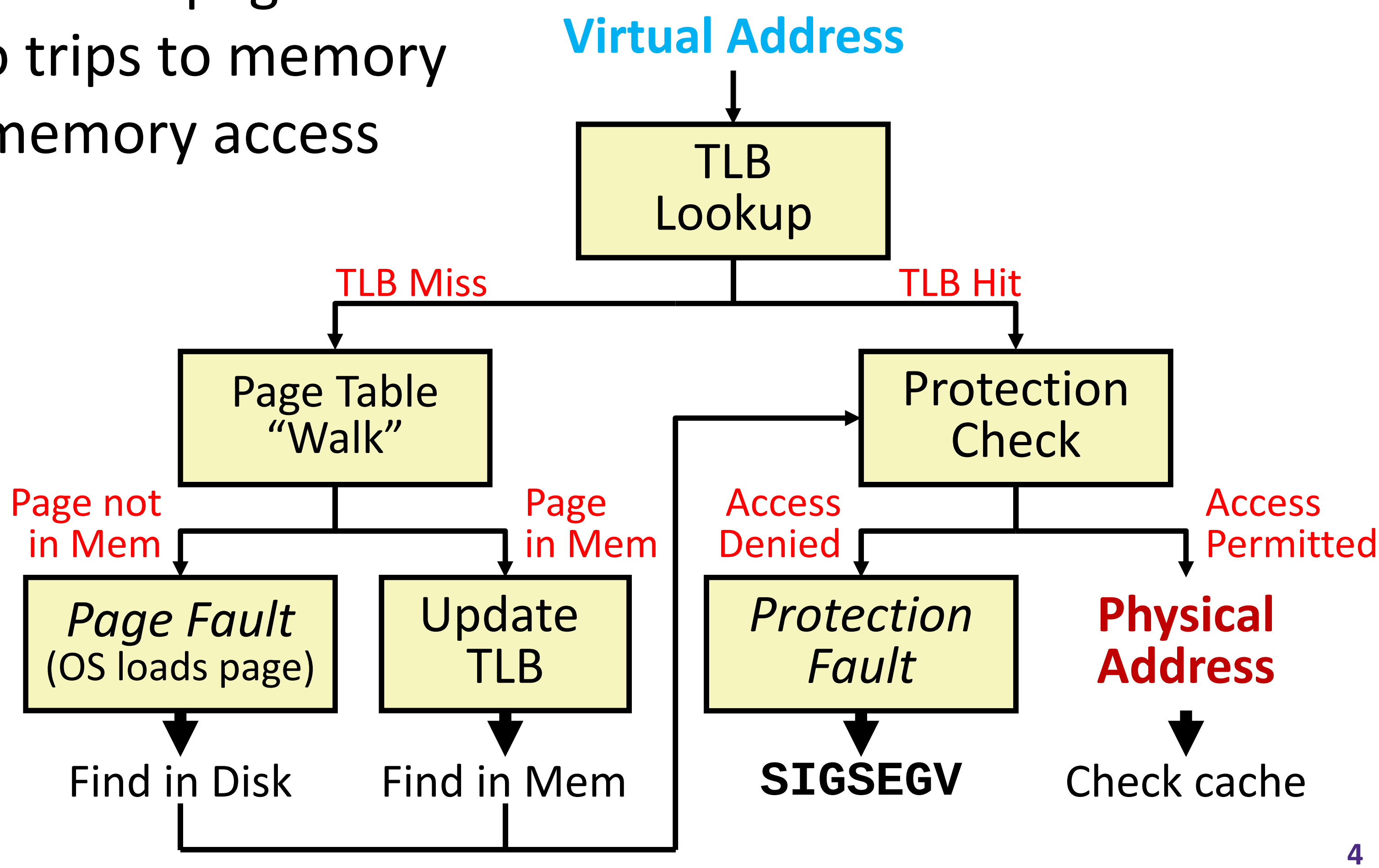
- “Virtual section” on virtual memory released
  - 3 PDFs: VM overview, worksheet, and solutions
  - Linked in the code section of today’s lecture
  - See Piazza post for links and videos

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
  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
  (lookaside buffer: British for "cache")

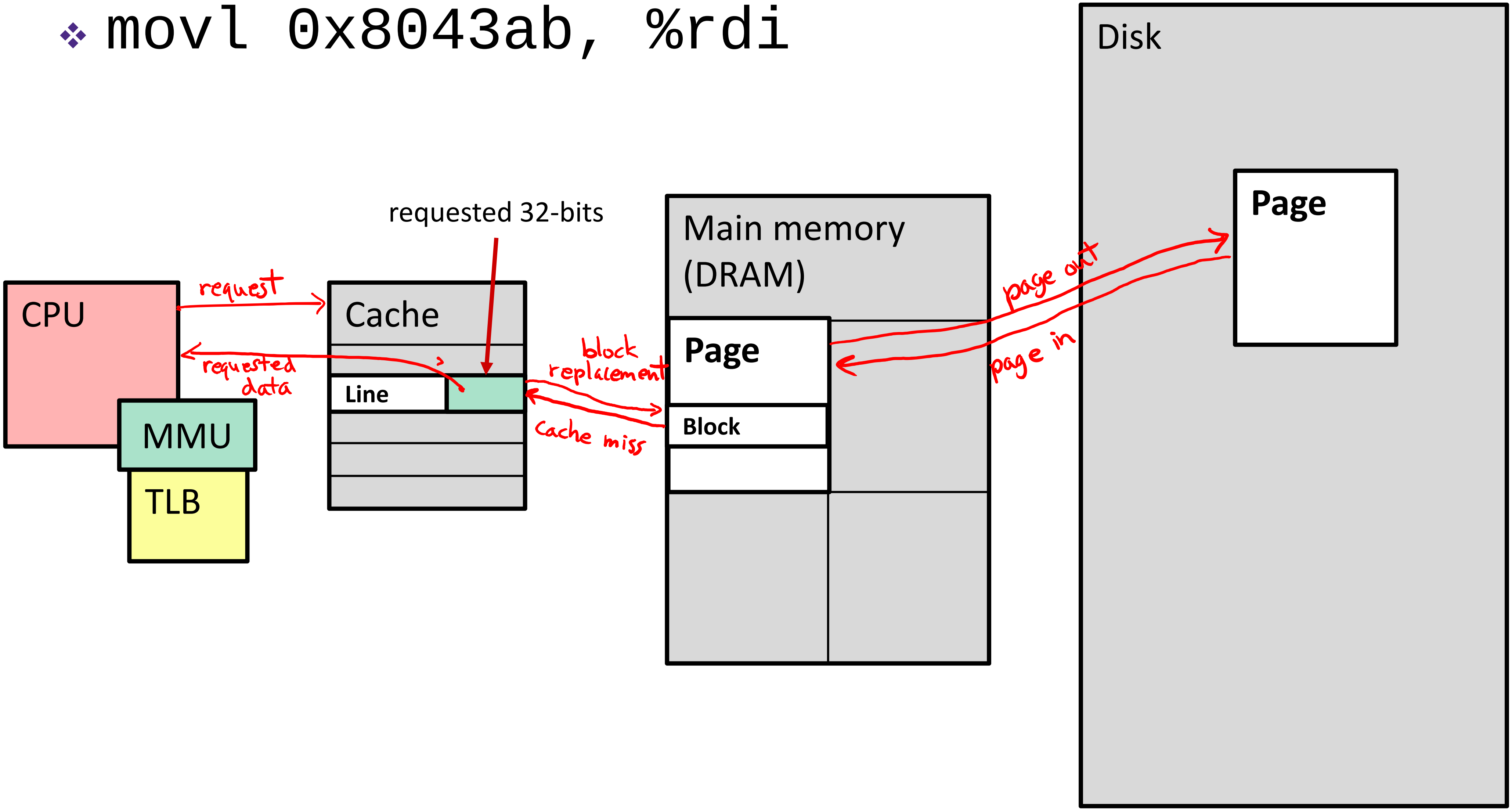
Address Translation

- 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



Memory Overview (data flow)

❖ `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

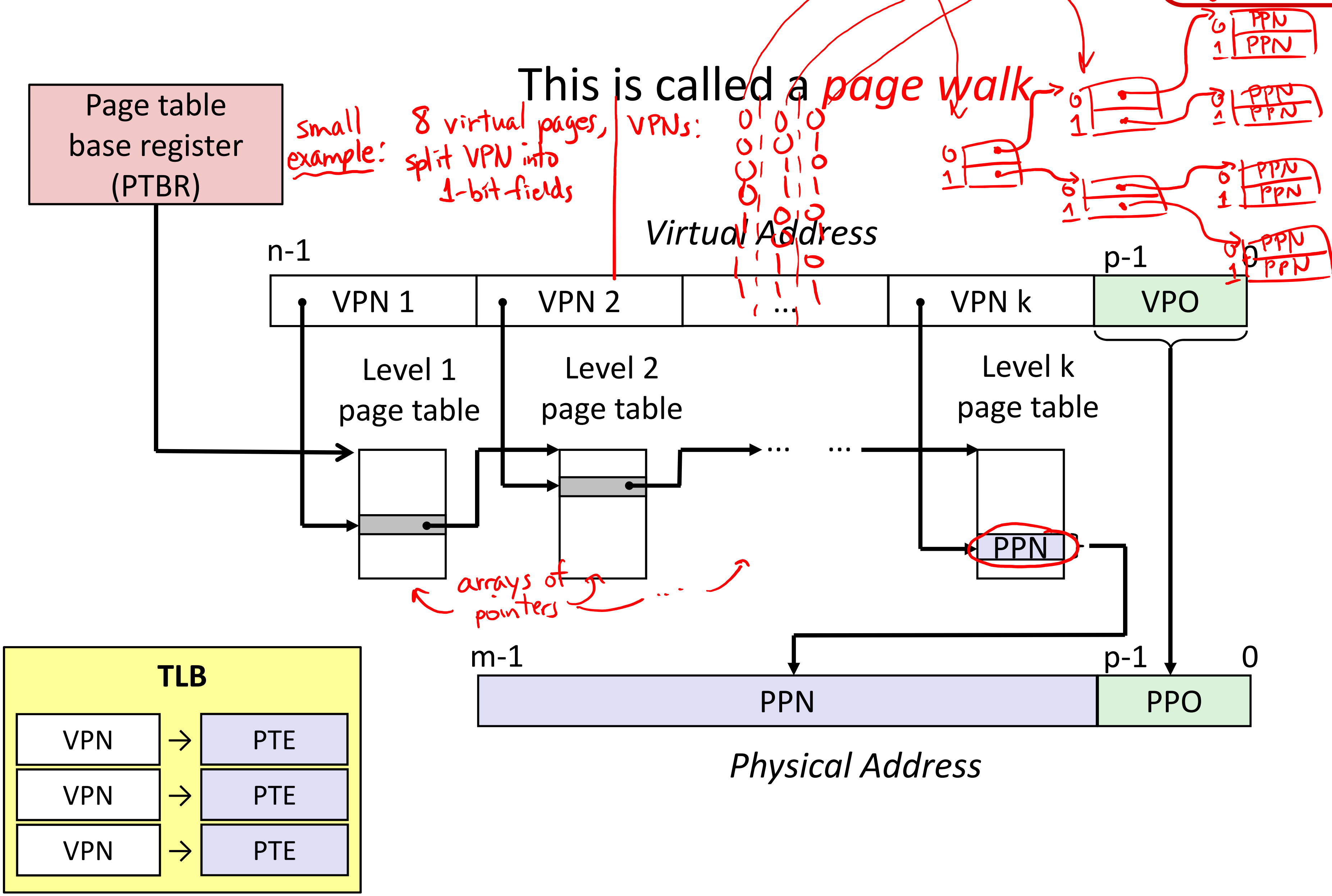
- Just one issue… the numbers don't work out for the story so far!

- The problem is the page table for each process:
  - Suppose 64-bit VAs, 8 KiB pages, 8 GiB physical memory
    - $n = 64$ bits, $p = 13$ bits, $m = 33$ bits
  - How many page table entries is that?
    - 1 PTE for every virtual page: $2^{n-p} = 2^{51}$ PTEs
  - About how long is each PTE?
    - PPN width ($m-p$) + management bits (V,D,R,W,X) = 20+5 = 25 bits ≈ 3 bytes
    - $\approx 2^{52}+2^{51}$ bytes per page table!
  - **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 extra (non-testable) material

This is called a *page walk*



Multi-level Page Tables

This is extra (non-testable) material

- 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
    (fully associative: 1 set)

a) Fill in the following blanks:

2^{17} Total entries in page table
2^{n-p} ← # of virtual pages

20 Minimum bit-width of PTBR ← physical address of PT
m

17 TLBT bits
VPN → TLBT / TLBI
here TLBI = 0

2^5 Max # of valid entries in a page table ← # of pages in physical memory
2^{m-p}

Practice VM Question

starting address of matrix is at page offset of 0

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

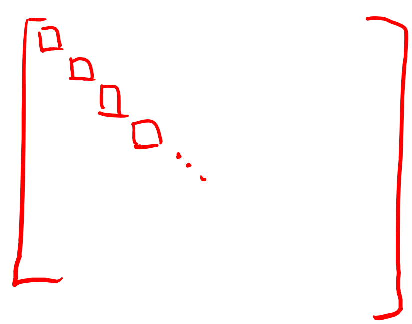
```
#define MAT_SIZE = 2048
for(int i=0; i<MAT_SIZE; i++)
      mat[i*(MAT_SIZE+1)] = i;
```

updating diagonal entries

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

*stride is always 2049 ints = 2049*4 bytes*

i	array index accessed
0	0
1	2049
2	2*2049
⋮	⋮



Practice VM Question

page size = 32 KiB = 2^{15} B

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

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

2^{11} ints = 2^{13} B

c) What are the following hit rates for the *first* execution of the for loop? (assume all of mat[] starts on disk)

3/4 = 75% TLB Hit Rate   0% Page Table Hit Rate

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 H H H

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

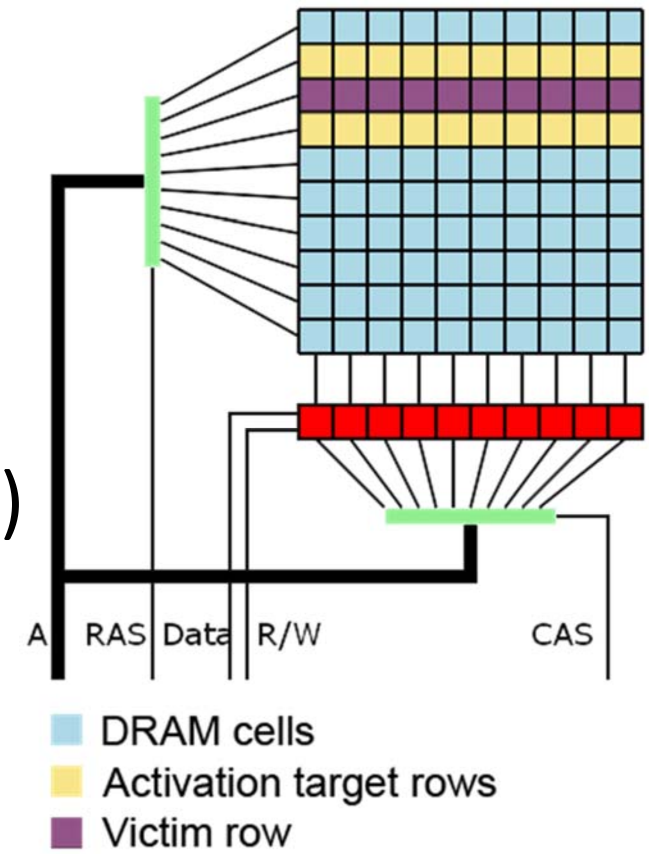
BONUS SLIDES

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

```
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 *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: https://vvdveen.com/publications/drammer.pdf
- Google Project Zero: https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html
- First row hammer paper: https://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf
- Wikipedia: https://en.wikipedia.org/wiki/Row_hammer