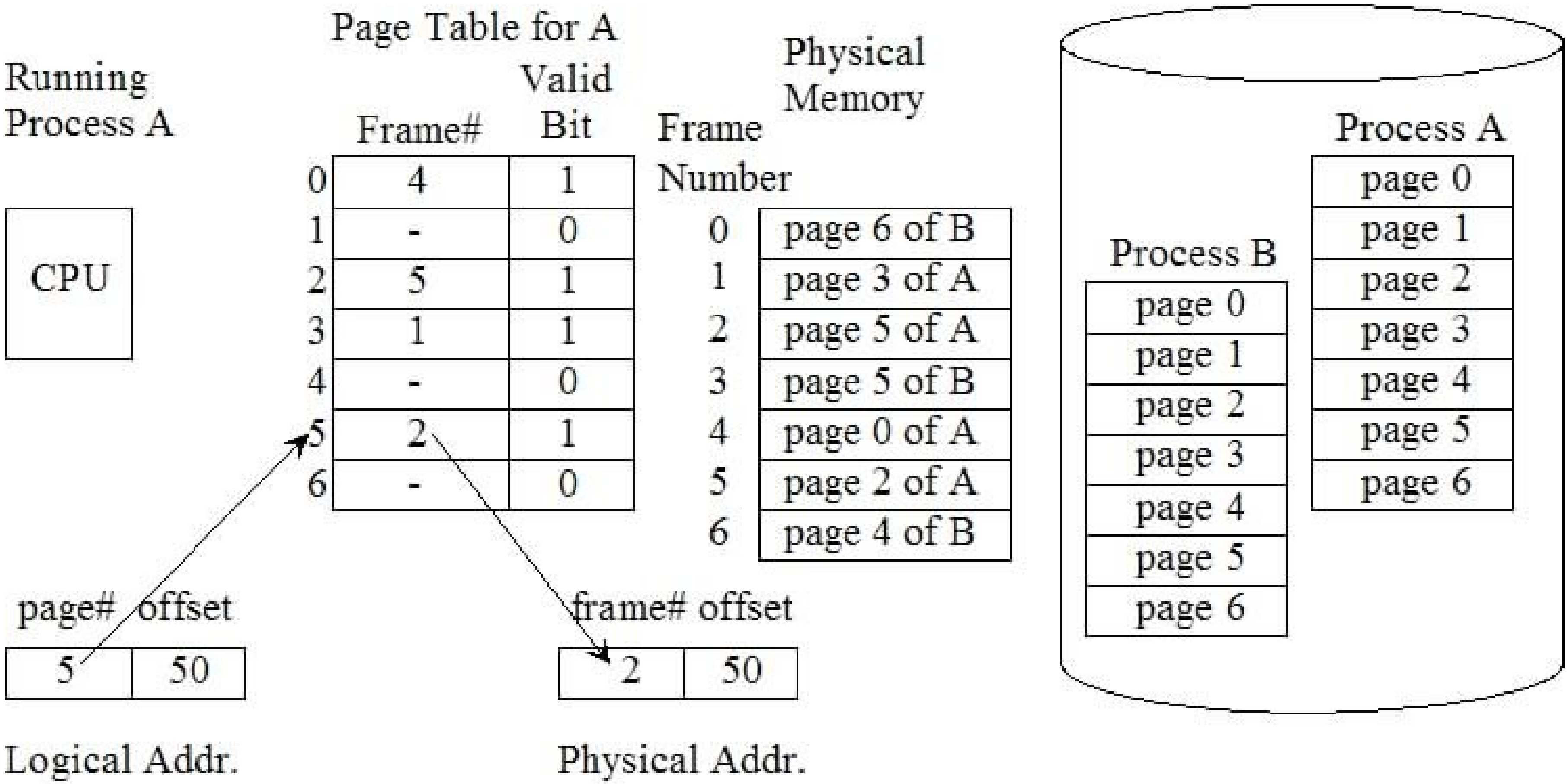
Virtual Memory, Rowhammer, and Meltdown

CSE 536 Spring 2024

jedimaestro@asu.edu

Virtual memory...



Plagiarized from http://www.cs.uni.edu/~fienup/cs142f05/lectures/lec20_OS_virtual_memory.HTM

```
jedi@tortuga:~$ head -n 16 /proc/`echo $$`/maps | sed "s/   / /g"
5eb8e1d11000-5eb8e1d40000 r--p 00000000 fc:01 72220755       /usr/bin/bash
5eb8e1d40000-5eb8e1e1f000 r-xp 0002f000 fc:01 72220755       /usr/bin/bash
5eb8e1e1f000-5eb8e1e59000 r--p 0010e000 fc:01 72220755       /usr/bin/bash
5eb8e1e5a000-5eb8e1e5e000 r--p 00148000 fc:01 72220755       /usr/bin/bash
5eb8e1e5e000-5eb8e1e67000 rw-p 0014c000 fc:01 72220755       /usr/bin/bash
5eb8e1e67000-5eb8e1e72000 rw-p 00000000 00:00 0
5eb8e2c62000-5eb8e2e09000 rw-p 00000000 00:00 0          [heap]
739311400000-739312306000 r--p 00000000 fc:01 72221250       /usr/lib/locale/locale-archive
739312400000-739312428000 r--p 00000000 fc:01 72286240       /usr/lib/x86_64-linux-gnu/libc.so.6
739312428000-7393125bd000 r-xp 00028000 fc:01 72286240       /usr/lib/x86_64-linux-gnu/libc.so.6
7393125bd000-739312615000 r--p 001bd000 fc:01 72286240       /usr/lib/x86_64-linux-gnu/libc.so.6
739312615000-739312616000 ---p 00215000 fc:01 72286240       /usr/lib/x86_64-linux-gnu/libc.so.6
739312616000-73931261a000 r--p 00215000 fc:01 72286240       /usr/lib/x86_64-linux-gnu/libc.so.6
73931261a000-73931261c000 rw-p 00219000 fc:01 72286240       /usr/lib/x86_64-linux-gnu/libc.so.6
73931261c000-739312629000 rw-p 00000000 00:00 0
7393127fc000-7393127ff000 rw-p 00000000 00:00 0
```

```
jedi@tortuga:~$ tail -n 16 /proc/`echo $$`/maps | sed "s/   / /g"
7393127ff000-73931280d000 r--p 00000000 fc:01 72300422      /usr/lib/x86_64-linux-gnu/libtinfo.so
73931280d000-73931281e000 r-xp 0000e000 fc:01 72300422      /usr/lib/x86_64-linux-gnu/libtinfo.so
73931281e000-73931282c000 r--p 0001f000 fc:01 72300422      /usr/lib/x86_64-linux-gnu/libtinfo.so
73931282c000-739312830000 r--p 0002c000 fc:01 72300422      /usr/lib/x86_64-linux-gnu/libtinfo.so
739312830000-739312831000 rw-p 00030000 fc:01 72300422      /usr/lib/x86_64-linux-gnu/libtinfo.so
739312842000-739312849000 r--s 00000000 fc:01 72286510      /usr/lib/x86_64-linux-gnu/gconv/gconv-
739312849000-73931284b000 rw-p 00000000 00:00 0
73931284b000-73931284d000 r--p 00000000 fc:01 72286234      /usr/lib/x86_64-linux-gnu/ld-linux-x8
73931284d000-739312877000 r-xp 00002000 fc:01 72286234      /usr/lib/x86_64-linux-gnu/ld-linux-x8
739312877000-739312882000 r--p 0002c000 fc:01 72286234      /usr/lib/x86_64-linux-gnu/ld-linux-x8
739312883000-739312885000 r--p 00037000 fc:01 72286234      /usr/lib/x86_64-linux-gnu/ld-linux-x8
739312885000-739312887000 rw-p 00039000 fc:01 72286234      /usr/lib/x86_64-linux-gnu/ld-linux-x8
7ffe8c20a000-7ffe8c22b000 rw-p 00000000 00:00 0          [stack]
7ffe8c335000-7ffe8c339000 r--p 00000000 00:00 0          [vvar]
7ffe8c339000-7ffe8c33b000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0      [vsyscall]
```

Abstractions

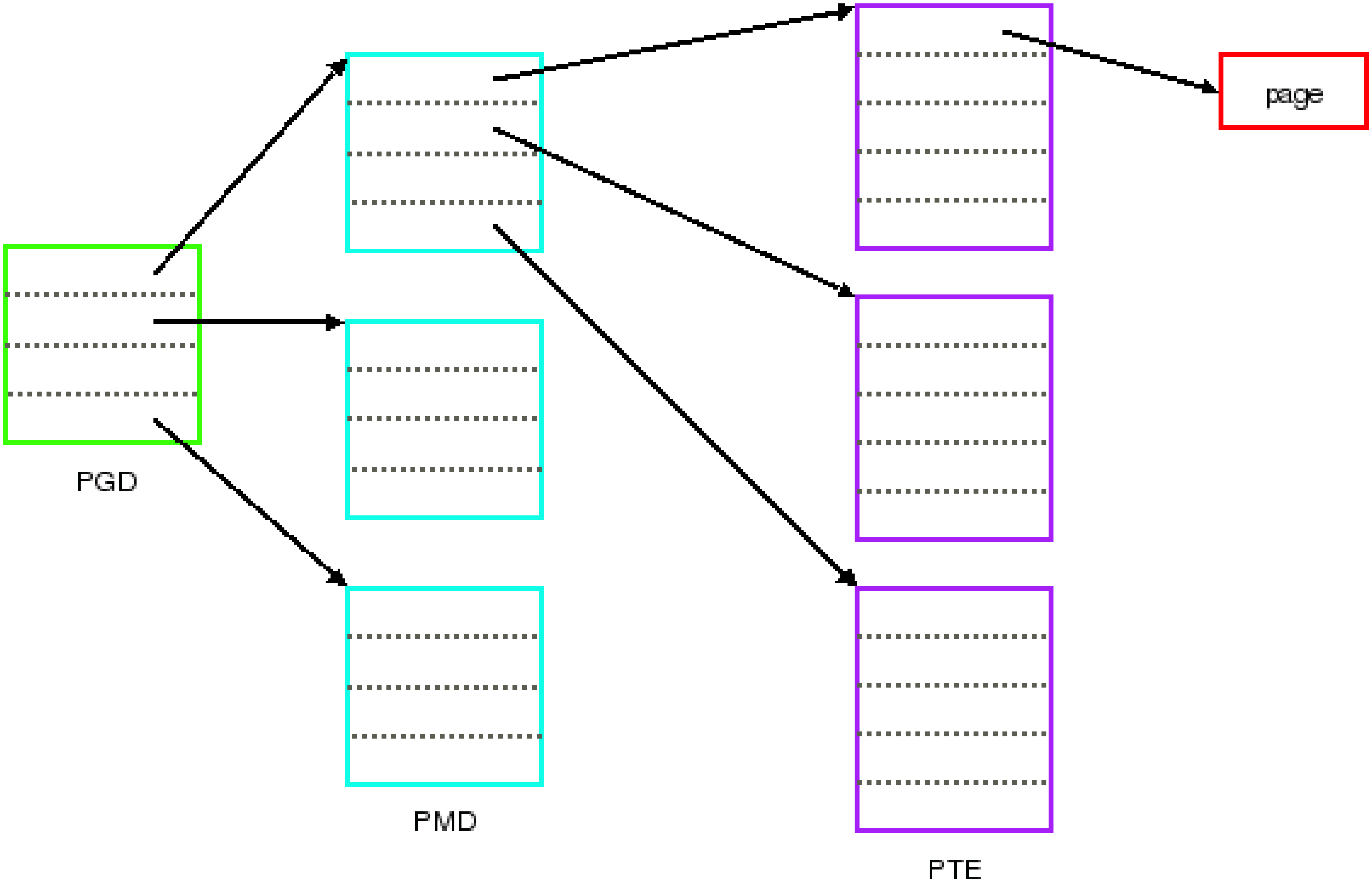
- Process hierarchy
- Filesystem and filesystem hiearchy
- Virtual address spaces

Reality

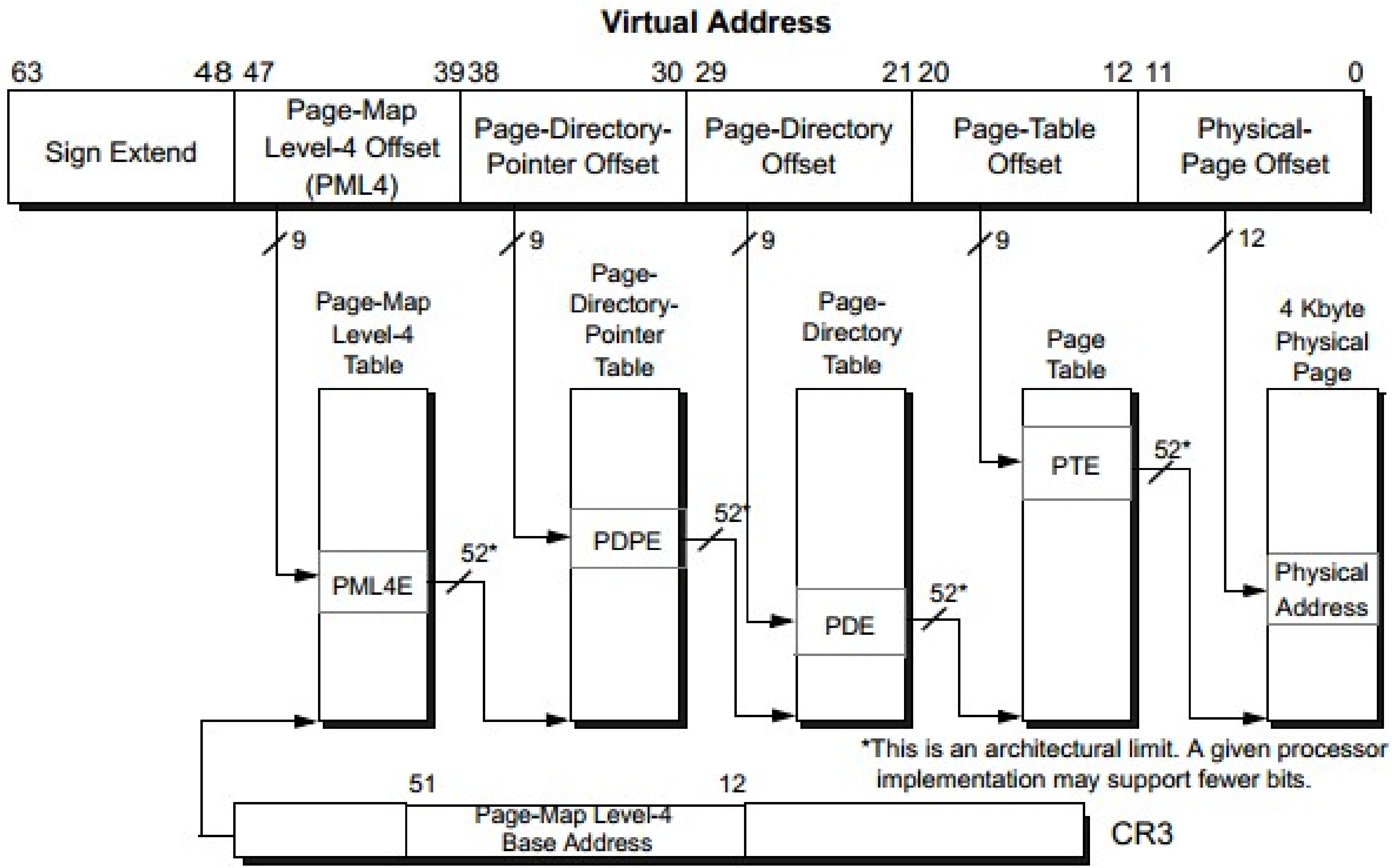
- Files are spread all over disk and the memory, namespace is shared by many processes
- Virtual address spaces are spread all over the disk and memory, physical pages are shared by multiple processes
- Processes are an abstraction of an architectural state, but the CPU is physically implemented as a microarchitecture

Linux page cache

- Demand paging
  - Only bring blocks into memory, or allocate physical pages, when they are used
    - *Copy-on-write*
    - Zero page
  - rwx permissions are hardware enforced
  - Pages can be *dirty* or *clean*
    - Dirty pages should eventually be written back to disk
      - Easier to *evict* clean pages
- Page replacement algorithm
  - *E.g.*, Least Recently Used or LRU



https://lwn.net/Articles/106177/



https://superuser.com/questions/1740680/are-page-tables-under-utilized-in-x86-systems

Page Table Entry

31 ... 12	11... 9	8	7	6	5	4	3	2	1	0
Bits 31-12 of address	AVL	G	PAT	D	A	PCD	PWT	U/S	R/W	P

P: Present
R/W: Read/Write
U/S: User/Supervisor
PWT: Write-Through
PCD: Cache Disable
A: Accessed
D: Dirty
G: Global
AVL: Available
PAT: Page Attribute Table

https://wiki.osdev.org/File:Page_table_entry.png

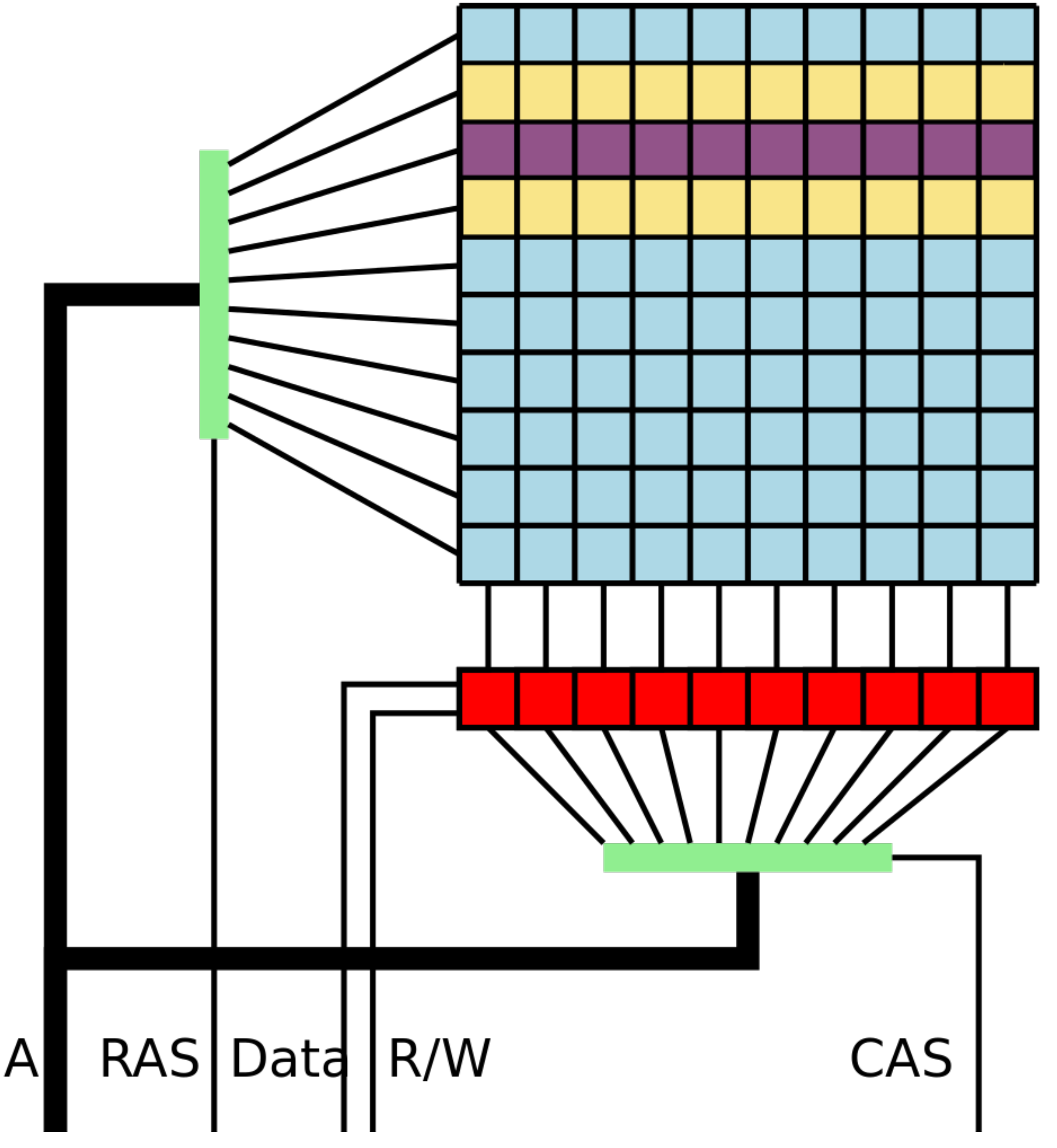
Page faults

- Trap to the kernel whenever there is any issue with translating a page or accessing the memory
  - Kernel diagnoses what the problem is
    - Fix the problem and let the process continue?
    - Or, send SIGSEGV
- There's a special cache for page table entries called a translation lookaside buffer (TLB)

Rowhammer...

Food for thought

- Information is inherently physical
- Information only has meaning in that it is subject to interpretation
- Management information stored in-band with regular information
- Programming the weird machine



Plagiarized from:
https://en.wikipedia.org/wiki/Row_hammer#/media/File:Row_hammer.svg

Step #1: Find aggressor and victim

- Allocate a large chunk of memory, like 1GB or more
- Aggressors X and Y must be different rows in the same bank
  - DRAM row is typically >4K and <2MB
  - Rows in a bank activated in lockstep
- Pick X and Y as random virtual addresses
  - Check if hammering X and Y flips a bit in Z
  - If you find that Z (have to check the whole block), that’s your victim
- Hope that you can flip, e.g., the 12^{th} bit in a 64-bit word rather than, e.g., the 51^{st}
- munmap() all but these three pages (two aggressors, one victim)

Step #2: Randomize physical memory

- Why? So a small change in where a PTE points will not go from one data page to another.
- Allocate a huge chunk of memory with mmap() with MAP_POPULATE
- Throughout the exploit, release a random 4KB at a time with madvise + MADV_DONTNEED

Step #3: Spray physical memory with page tables

- Keep mmap()ing a file with markers in it, 2MB aligned
  - Why 2MB? One page table has 512 entries times 4K = 2MB
  - Try to have more page tables in memory than data
    - When victim is released it's likely to be a page table
    - When bit is flipped new value is likely to point to a page table

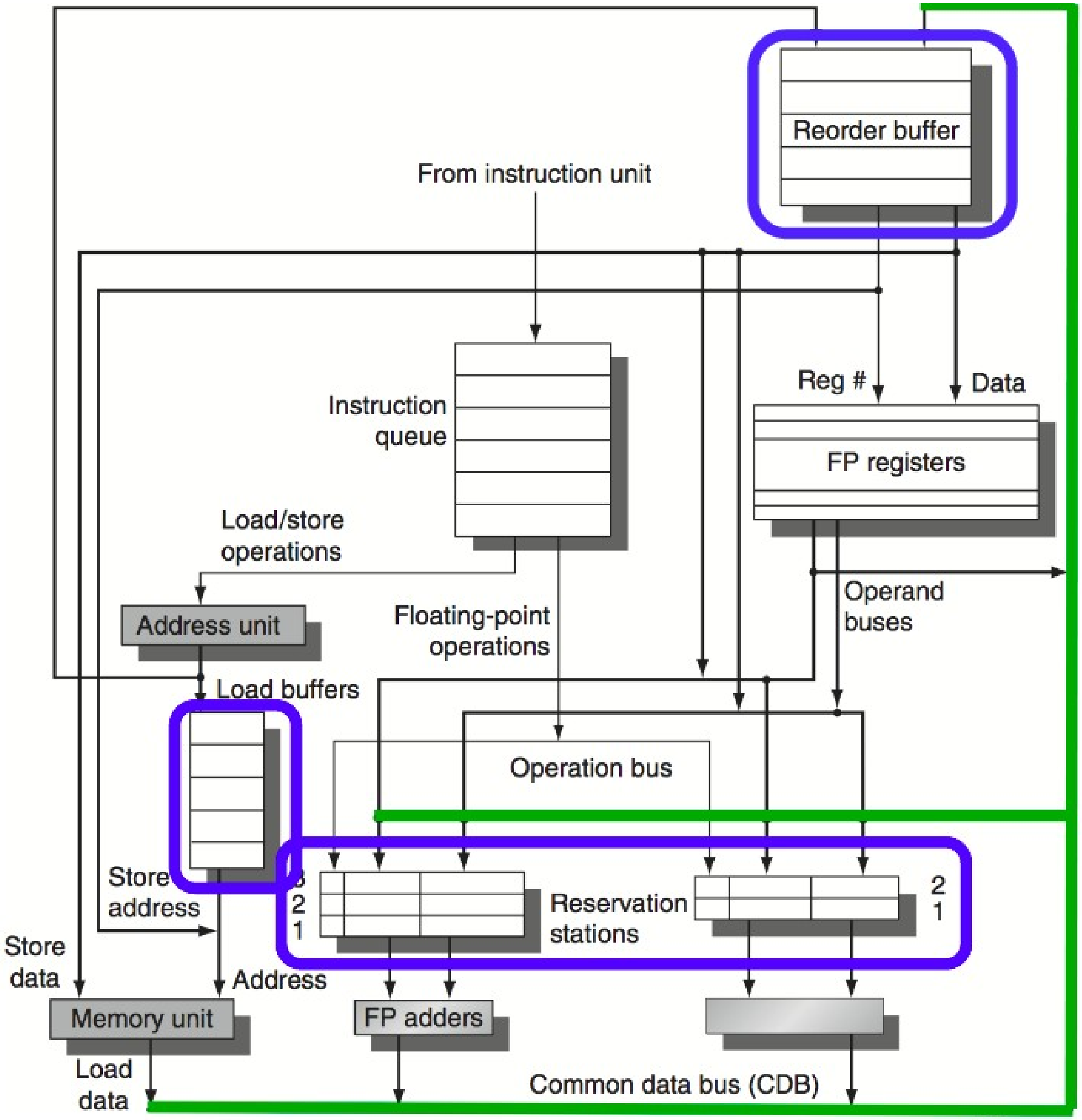
Step #4: Hammer time

- Check if bit flip changed a mapping in the page table to point to another page table
  - Only have to check the Nth page within each 2MB chunk
- If it's not pointing to the file, then it's likely pointing to another page table. Which one?
  - Can change it arbitrarily, then scan our virtual address space to fine another page that now doesn't point to the file

Step #5: Exploit

- mmap() a setuid binary, like ping
  - Kernel won’t set write bit in your PTE for ping’s code section
  - Modify your writable page table to give yourself write permissions to the physical page where ping’s code section gets cached
  - Execute it as root

MELTDOWN...



Plagiarized from:
https://passlab.github.io/CSCE513/notes/lecture18_ILP_SuperscalarAdvancedARMIntel.pdf

Overly simplified MELTDOWN

```
int a[256 * cachelinesize] // cache aligned
char *p = &SomethingICantReadInKernel
int x = a[*p * cachelinesize]
```

- Side channel: whatever gets cached speculatively reveals *p

What does this mean?

- Supervisor bit is useless, because microarchitectural state can be visibly changed based on speculative execution that ignores the supervisor bit
- Can no longer put the kernel at the top of the virtual address space of every process

- https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html
- https://www.usenix.org/conference/usenixsecurity18/presentation/lipp
-