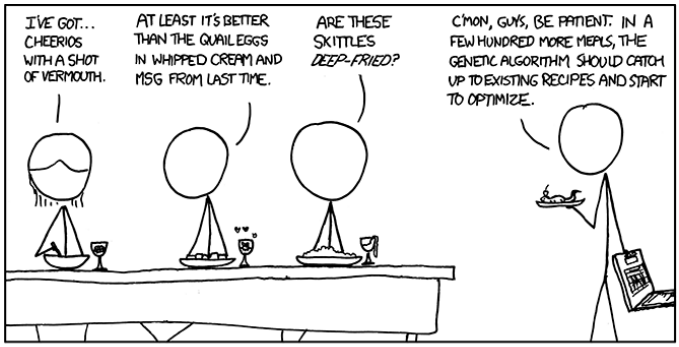
# Virtual Memory III

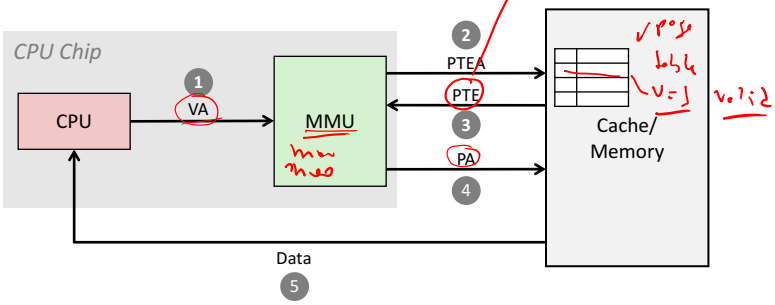
CSE 351 Winter 2017




https://xkcd.com/720/



# Address Translation: Page Hit




**1)** Processor sends *virtual* address to MMU (*memory management unit*)

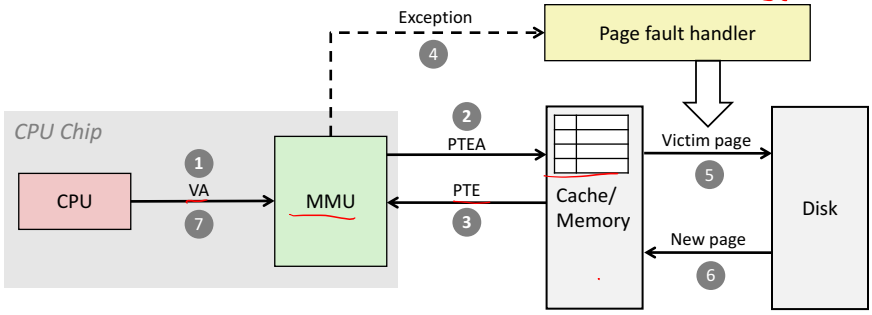
**2-3)** MMU fetches PTE from page table in cache/memory
(Uses PTBR to find beginning of page table for current process)

**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



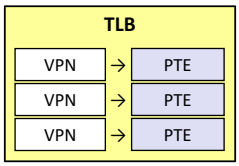


# Hmm… Translation Sounds Slow

- 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

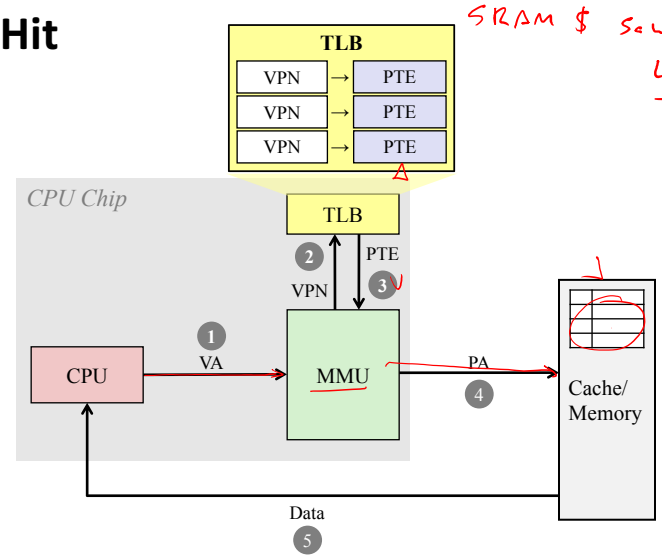
- *Translation Lookaside Buffer* (TLB):
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete *page table entries* for small number of pages
    - Modern Intel processors have 128 or 256 entries in TLB
  - Much faster than a page table lookup in cache/memory





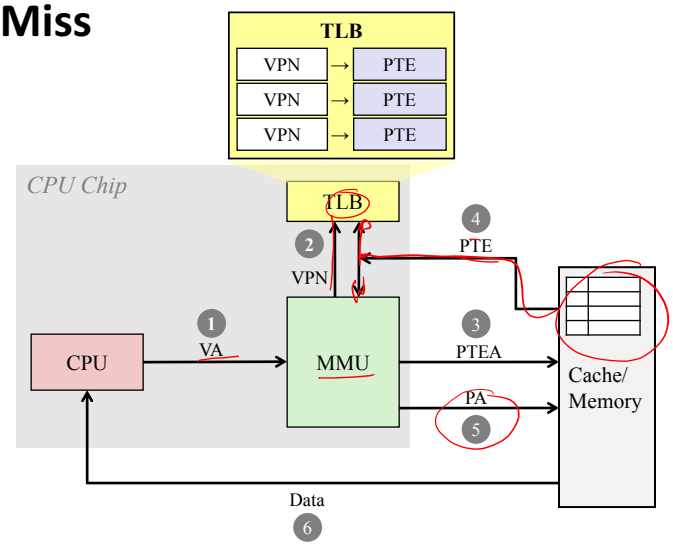



# TLB Hit




- A TLB hit eliminates a memory access!

# TLB Miss




- A TLB miss incurs an additional memory access (the PTE)
  - Fortunately, TLB misses are rare

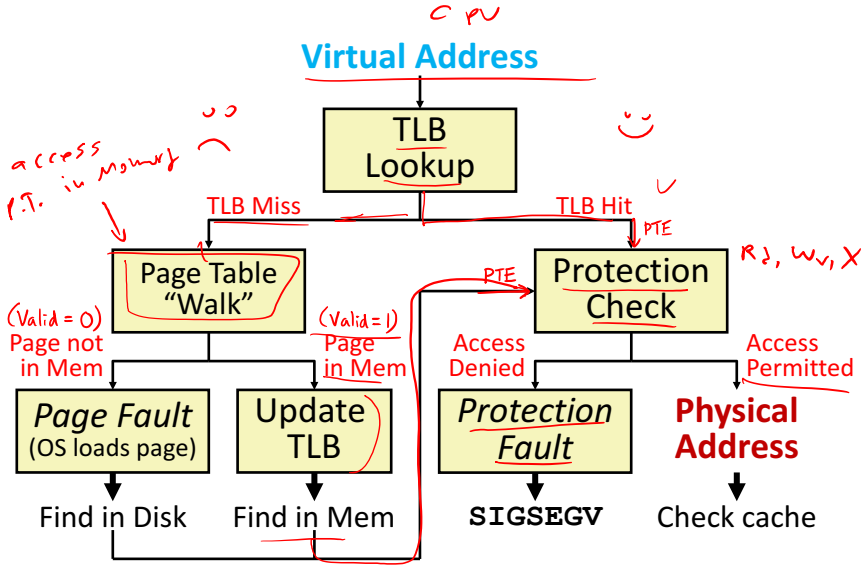




# Fetching Data on a Memory Read

1) Check TLB
   - Input: VPN, Output: PPN
   - *TLB Hit:* Fetch translation, return PPN
   - *TLB Miss:* Check page table (in memory)
     - *Page Table Hit:* Load page table entry into TLB
     - *Page Fault:* Fetch page from disk to memory, update corresponding page table entry, then load entry into TLB
2) Check cache
   - Input: physical address, Output: data
   - *Cache Hit:* Return data value to processor
   - *Cache Miss:* Fetch data value from memory, store it in cache, return it to processor

## Address Translation





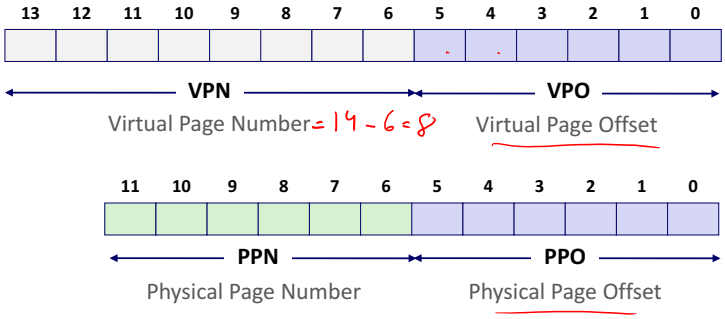



## Summary of Address Translation Symbols

- 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
  - 12-bit physical address
  - Page size = 64 bytes ⇒ P.O. = $\log_2 64 = 6$ bits








# Simple Memory System: Page Table

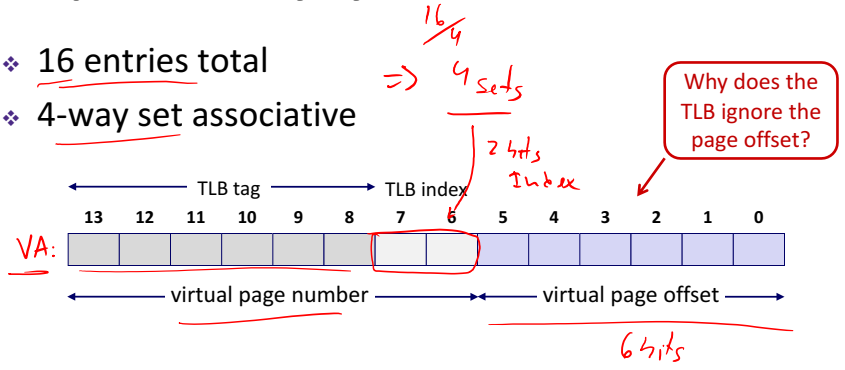
- Only showing first 16 entries (out of 256)
  - **Note:** showing 2 hex digits for PPN even though only 6 bits

P.T.E.

| VPN | PPN | Valid | VPN | PPN | Valid |
|---|---|---|---|---|---|
| 0 | 28 | 1 | 8 | 13 | 1 |
| 1 | – | 0 | 9 | 17 | 1 |
| 2 | 33 | 1 | A | 09 | 1 |
| 3 | 02 | 1 | B | – | 0 |
| 4 | – | 0 | C | – | 0 |
| 5 | 16 | 1 | D | 2D | 1 |
| 6 | – | 0 | E | – | 0 |
| 7 | – | 0 | F | 0D | 1 |

# Simple Memory System: TLB

- 16 entries total
- 4-way set associative




| Set | Tag | PPN | Valid | Tag | PPN | Valid | Tag | PPN | Valid | Tag | PPN | Valid |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 03 | – | 0 | 09 | 0D | 1 | 00 | – | 0 | 07 | 02 | 1 |
| 1 | 03 | 2D | 1 | 02 | – | 0 | 04 | – | 0 | 0A | – | 0 |
| 2 | 02 | – | 0 | 08 | – | 0 | 06 | – | 0 | 03 | – | 0 |
| 3 | 07 | – | 0 | 03 | 0D | 1 | 0A | 34 | 1 | 02 | – | 0 |

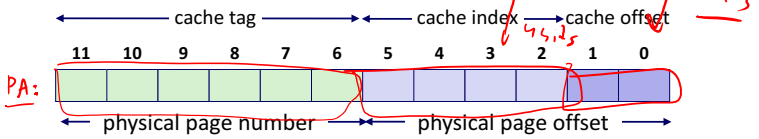




# Simple Memory System: Cache

- 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




| Index | Tag | Valid | B0 | B1 | B2 | B3 | Index | Tag | Valid | B0 | B1 | B2 | B3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 19 | 1 | 99 | 11 | 23 | 11 | 8 | 24 | 1 | 3A | 00 | 51 | 89 |
| 1 | 15 | 0 | – | – | – | – | 9 | 2D | 0 | – | – | – | – |
| 2 | 1B | 1 | 00 | 02 | 04 | 08 | A | 2D | 1 | 93 | 15 | DA | 3B |
| 3 | 36 | 0 | – | – | – | – | B | 0B | 0 | – | – | – | – |
| 4 | 32 | 1 | 43 | 6D | 8F | 09 | C | 12 | 0 | – | – | – | – |
| 5 | 0D | 1 | 36 | 72 | F0 | 1D | D | 16 | 1 | 04 | 96 | 34 | 15 |
| 6 | 31 | 0 | – | – | – | – | E | 13 | 1 | 83 | 77 | 1B | D3 |
| 7 | 16 | 1 | 11 | C2 | DF | 03 | F | 14 | 0 | – | – | – | – |

# Current State of Memory System

TLB:

| Set | Tag | PPN | V | Tag | PPN | V | Tag | PPN | V | Tag | PPN | V |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 03 | – | 0 | 09 | 0D | 1 | 00 | – | 0 | 07 | 02 | 1 |
| 1 | 03 | 2D | 1 | 02 | – | 0 | 04 | – | 0 | 0A | – | 0 |
| 2 | 02 | – | 0 | 08 | – | 0 | 06 | – | 0 | 03 | – | 0 |
| 3 | 07 | – | 0 | 03 | 0D | 1 | 0A | 34 | 1 | 02 | – | 0 |

Page table (partial):

| VPN | PPN | V | VPN | PPN | V |
|---|---|---|---|---|---|
| 0 | 28 | 1 | 8 | 13 | 1 |
| 1 | – | 0 | 9 | 17 | 1 |
| 2 | 33 | 1 | A | 09 | 1 |
| 3 | 02 | 1 | B | – | 0 |
| 4 | – | 0 | C | – | 0 |
| 5 | 16 | 1 | D | 2D | 1 |
| 6 | – | 0 | E | – | 0 |
| 7 | – | 0 | F | 0D | 1 |

Cache:

| Index | Tag | V | B0 | B1 | B2 | B3 | Index | Tag | V | B0 | B1 | B2 | B3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 19 | 1 | 99 | 11 | 23 | 11 | 8 | 24 | 1 | 3A | 00 | 51 | 89 |
| 1 | 15 | 0 | – | – | – | – | 9 | 2D | 0 | – | – | – | – |
| 2 | 1B | 1 | 00 | 02 | 04 | 08 | A | 2D | 1 | 93 | 15 | DA | 3B |
| 3 | 36 | 0 | – | – | – | – | B | 0B | 0 | – | – | – | – |
| 4 | 32 | 1 | 43 | 6D | 8F | 09 | C | 12 | 0 | – | – | – | – |
| 5 | 0D | 1 | 36 | 72 | F0 | 1D | D | 16 | 1 | 04 | 96 | 34 | 15 |
| 6 | 31 | 0 | – | – | – | – | E | 13 | 1 | 83 | 77 | 1B | D3 |
| 7 | 16 | 1 | 11 | C2 | DF | 03 | F | 14 | 0 | – | – | – | – |

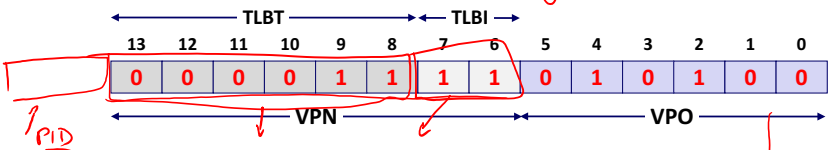




# Memory Request Example #1

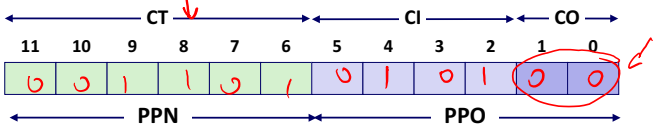
**Note:** It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: 0x03D4




VPN _____ TLBT _____ TLBI _____ TLB Hit? ____ Page Fault? ____ PPN _____

- Physical Address:

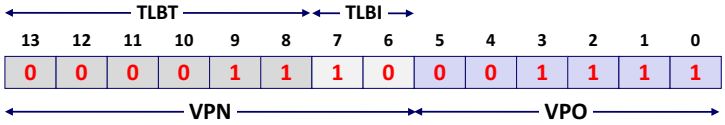



CT _____ CI _____ CO _____ Cache Hit? ____ Data (byte) _____

# Memory Request Example #2

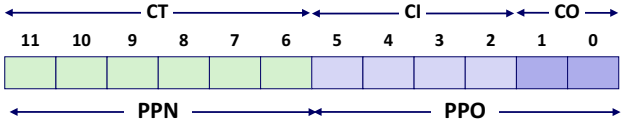
**Note:** It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: `0x038F`




VPN _____ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:




CT _____ CI _____ CO _____ Cache Hit? ___ Data (byte) _____

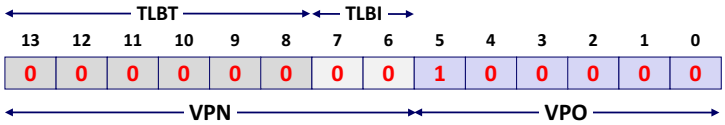




# Memory Request Example #3

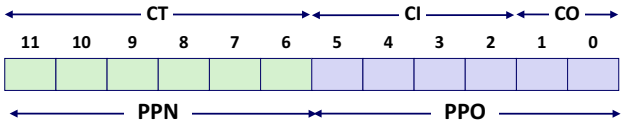
**Note:** It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: `0x0020`




VPN _____ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:

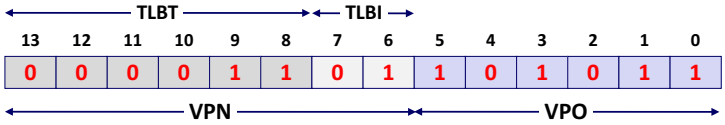



CT _____ CI _____ CO _____ Cache Hit? ___ Data (byte) _____

# Memory Request Example #4

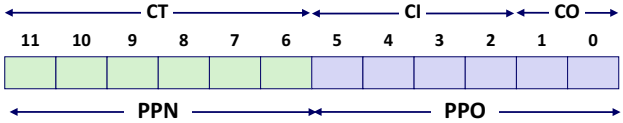
**Note:** It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: `0x036B`




VPN _____ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:




CT _____ CI _____ CO _____ Cache Hit? ___ Data (byte) _____





# 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)



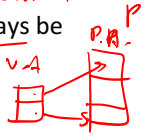


# Memory System – Who controls what?

- Memory Caches (L1/L2/L3)
  - Controlled by hardware
  - Programmer cannot control it
  - Programmer ***can*** write code to take advantage of it
- Virtual Memory
  - Controlled by OS and hardware
  - Programmer cannot control mapping to physical memory
  - Programmer can control sharing and some protection
    - via OS functions (not in CSE 351)

# Quick Review

- What do Page Tables map? V.A. ⇒ P.A. PT. entry
- Where are Page Tables located? Memory (cached in TLB, not BLT)
- How many Page Tables are there? as many as processes.
- Can your program tell if a page fault has occurred? P.T.E is invalid, or not existent
- What is thrashing? MEM ⇐ DSK using more mem than phys.
- True / ~~False~~: Virtual Addresses that are contiguous will always be contiguous in physical memory




- TLB stands for _____ and stores _____



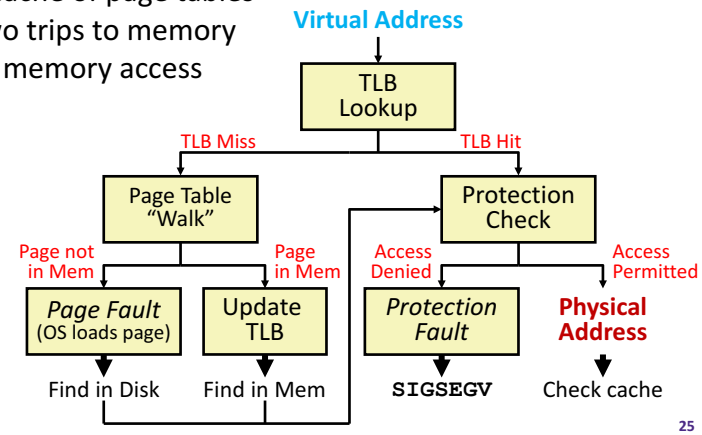


# Quick Review Answers

- 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

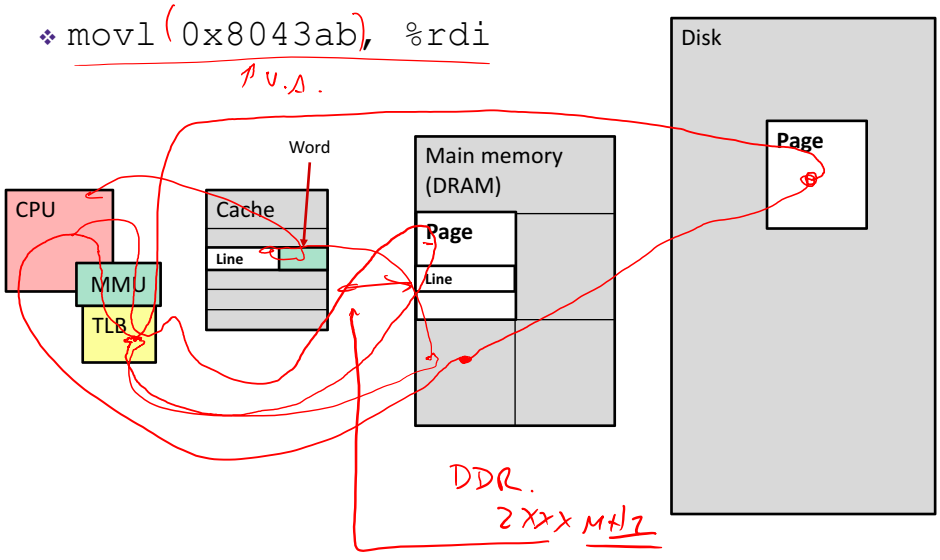







# Memory Overview: Putting it all together

- `movl 0x8043ab, %rdi`



# Context Switching Revisited

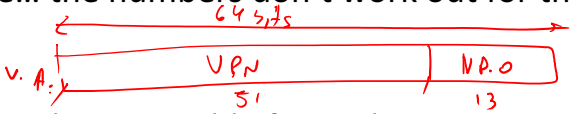
- What needs to happen when the CPU switches processes?
  - Registers:
    - Save state of old process, load state of new process
    - Including the Page Table Base Register (PTBR)
  - Memory:
    - Nothing to do! Pages for processes already exist in memory/disk and protected from each other
  - TLB:
    - *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!

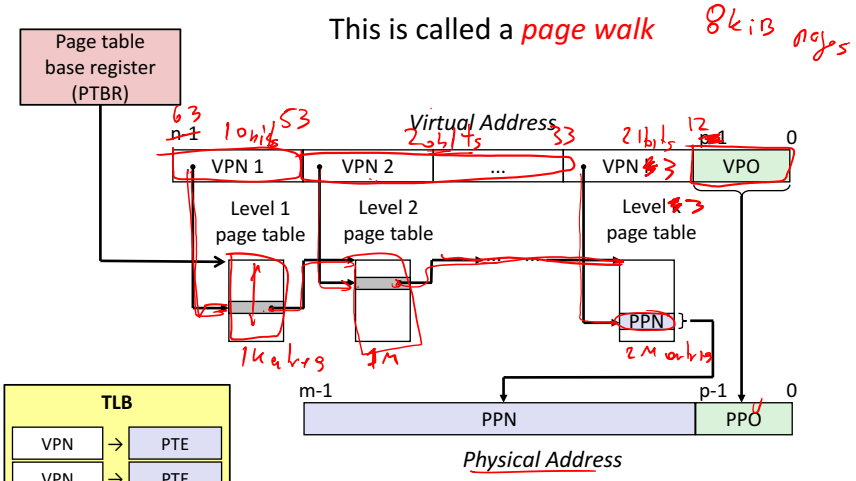



- The problem is the page table for each process:
  - Suppose 64-bit VAs, 8 KiB pages, 8 GiB physical memory
  - How many page table entries is that?
  - About how long is each PTE?
  - **Moral:** Cannot use this naïve implementation of the virtual→physical-page mapping – it's *way* too big

# A Solution: Multi-level Page Tables

This is extra (non-testable) material

This is called a *page walk*




Why does this work?





# 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 (1 set) TLB with LRU replacement

a) Fill in the following blanks:

$2^{17}$ / $2^{n-p}$ ← # of virtual pages — Entries in page table

20 — Minimum bit-width of PTBR ← physical address of PT ($m$)

17 — TLBT bits (VPN→TLBT/TLBI, here TLBI = 0)

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





# 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 = 2^11
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)?

| i | index |
|---|---|
| 0 | 0 |
| 1 | 2049 |
| 2 | 2*2049 |

2049 ints = 2049*4 B = stride

accesses diagonal of matrix

# 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;
```

2048 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: MHHH

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





# Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
10001101000001000000010
1000100111000010
11000001111110100011111
```

OS: Windows 8, Mac

Computer system:

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

# Multiple Ways to Store Program Data

- Static global data
  - *Fixed size* at compile-time
  - Entire *lifetime of the program* (loaded from executable)
  - Portion is read-only (e.g. string literals)
- Stack-allocated data
  - Local/temporary variables
    - *Can* be dynamically sized (in some versions of C)
  - *Known lifetime* (deallocated on `return`)
- **Dynamic (heap) data**
  - Size known only at runtime (i.e. based on user-input)
  - Lifetime known only at runtime (long-lived data structures)

```
int array[1024];

void foo(int n) {
  int tmp;
  int local_array[n];

  int* dyn =
    (int*)malloc(n*sizeof(int));
}
```



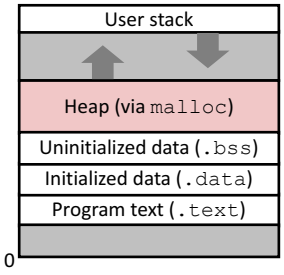


# Memory Allocation

- **Dynamic memory allocation**
  - **Introduction and goals**
  - **Allocation and deallocation (free)**
  - **Fragmentation**
- Explicit allocation implementation
  - Implicit free lists
  - Explicit free lists (Lab 5)
  - Segregated free lists
- Implicit deallocation: garbage collection
- Common memory-related bugs in C

# Dynamic Memory Allocation

- Programmers use dynamic memory allocators to acquire virtual memory at run time
  - For data structures whose size (or lifetime) is known only at runtime
  - Manage the heap of a process' virtual memory:



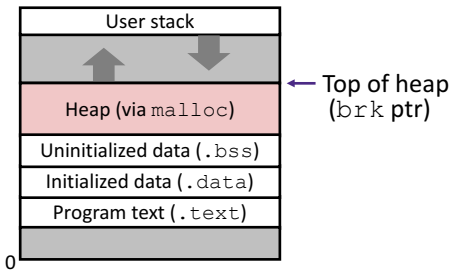

- Types of allocators
  - ***Explicit* allocator:** programmer allocates and frees space
    - Example: `malloc` and `free` in C
  - ***Implicit* allocator:** programmer only allocates space (no free)
    - Example: garbage collection in Java, Caml, and Lisp





# Dynamic Memory Allocation

- Allocator organizes heap as a collection of variable-sized *blocks*, which are either *allocated* or *free*
  - Allocator requests pages in the heap region; virtual memory hardware and OS kernel allocate these pages to the process
  - Application objects are typically smaller than pages, so the allocator manages blocks *within* pages
    - (Larger objects handled too; ignored here)



# Allocating Memory in C

- Need to `#include <stdlib.h>`
- **`void*`** `malloc(`**`size_t`** `size)`
  - Allocates a continuous block of `size` bytes of uninitialized memory
  - Returns a pointer to the beginning of the allocated block; NULL indicates failed request
    - Typically aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
    - Returns `NULL` if allocation failed (also sets `errno`) or `size==0`
  - Different blocks not necessarily adjacent
- Good practices:
  - `ptr = (int*) malloc(n*sizeof(int));`
    - `sizeof` makes code more portable
    - `void*` is implicitly cast into any pointer type; explicit typecast will help you catch coding errors when pointer types don't match





# Allocating Memory in C

- Need to `#include <stdlib.h>`
- **`void*`** `malloc(`**`size_t`** `size)`
  - Allocates a continuous block of `size` bytes of uninitialized memory
  - Returns a pointer to the beginning of the allocated block; NULL indicates failed request
    - Typically aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
    - Returns `NULL` if allocation failed (also sets `errno`) or `size==0`
  - Different blocks not necessarily adjacent
- Related functions:
  - **`void*`** `calloc(`**`size_t`** `nitems, `**`size_t`** `size)`
    - "Zeros out" allocated block
  - **`void*`** `realloc(`**`void*`** `ptr, `**`size_t`** `size)`
    - Changes the size of a previously allocated block (if possible)
  - **`void*`** `sbrk(`**`intptr_t`** `increment)`
    - Used internally by allocators to grow or shrink the heap

# Freeing Memory in C

- Need to `#include <stdlib.h>`
- **`void`** `free(`**`void*`** `p)`
  - Releases whole block pointed to by `p` to the pool of available memory
  - Pointer `p` must be the address *originally* returned by `m/c/realloc` (i.e. beginning of the block), otherwise throws system exception
  - Don't call `free` on a block that has already been released or on `NULL`



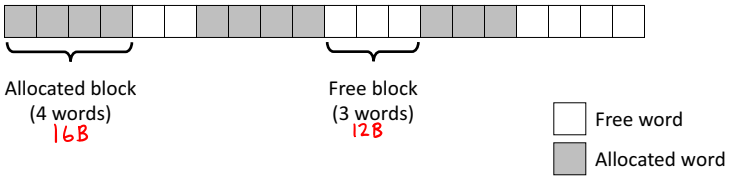


# Memory Allocation Example in C

```
void foo(int n, int m) {
  int i, *p;
  p = (int*) malloc(n*sizeof(int)); /* allocate block of n ints */
  if (p == NULL) {                  /* check for allocation error */
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++)               /* initialize int array */
    p[i] = i;
                                    /* add space for m ints to end of p block */
  p = (int*) realloc(p,(n+m)*sizeof(int));
  if (p == NULL) {                  /* check for allocation error */
    perror("realloc");
    exit(0);
  }
  for (i=n; i < n+m; i++)           /* initialize new spaces */
    p[i] = i;
  for (i=0; i<n+m; i++)             /* print new array */
    printf("%d\n", p[i]);
  free(p);                          /* free p */
}
```

# Notation Node (these slides, book, videos)

- Memory is drawn divided into *words*
  - Each *word* can hold an `int` (32 bits/4 bytes)
  - Allocations will be in sizes that are a multiple of words, i.e. multiples of 4 bytes
  - In pictures in slides, book, videos : ☐ = one word, 4 bytes

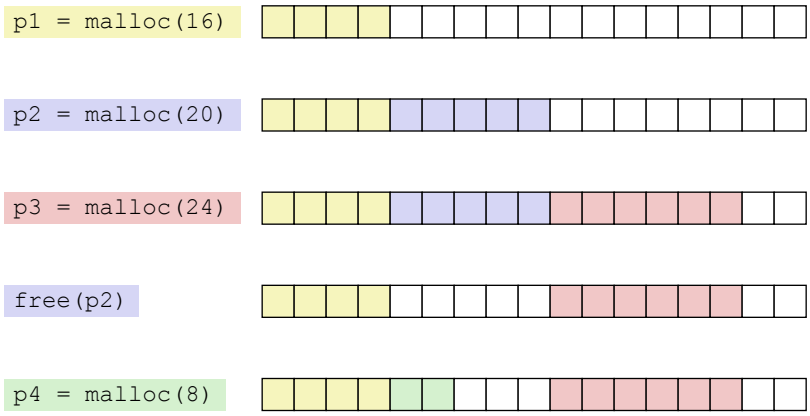







# Allocation Example

☐ = 4-byte word



# Constraints (interface/contract)

- **Applications**
  - Can issue arbitrary sequence of `malloc` and `free` requests
  - Must never access memory not currently allocated
  - Must never free memory not currently allocated
    - Also must only use `free` with previously `malloc`'ed blocks (not, e.g., stack data)
- **Allocators**
  - Can't control number or size of allocated blocks
  - Must respond immediately to `malloc` (*i.e.* can't reorder or buffer)
  - Must allocate blocks from free memory (*i.e.* blocks can't overlap – *Why not?*)
  - Must align blocks so they satisfy all alignment requirements
  - Can't move the allocated blocks (*i.e.* compaction/defragmentation is not allowed – *Why not?*)





# Performance Goals

- **Goals:** Given some sequence of `malloc` and `free` requests $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$, maximize throughput and peak memory utilization
  - These goals are often conflicting

**1) Throughput**

- Number of completed requests per unit time
- Example:
  - If 5,000 `malloc` calls and 5,000 `free` calls completed in 10 seconds, then throughput is 1,000 operations/second

# Performance Goals

- Definition: *Aggregate payload* $P_k$
  - `malloc(p)` results in a block with a *payload* of `p` bytes
  - After request $R_k$ has completed, the *aggregate payload* $P_k$ is the sum of currently allocated payloads
- Definition: *Current heap size* $H_k$
  - Assume $H_k$ is monotonically non-decreasing
    - Allocator can increase size of heap using `sbrk`

**2) Peak Memory Utilization**

- Defined as $U_k = (\max_{i \le k} P_i)/H_k$ after $k$+1 requests
- Goal: maximize utilization for a sequence of requests
- Why is this hard? And what happens to throughput?



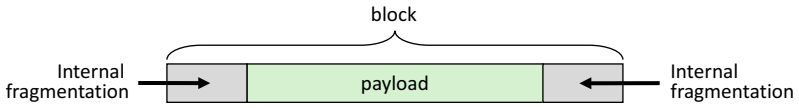


# Fragmentation

- Poor memory utilization is caused by *fragmentation*
  - Sections of memory are not used to store anything useful, but cannot satisfy allocation requests
  - Two types: *internal* and *external*
- **Recall:** Fragmentation in structs
  - Internal fragmentation was wasted space *inside* of the struct (between fields) due to alignment
  - External fragmentation was wasted space *between* struct instances (e.g. in an array) due to alignment
- Now referring to wasted space in the heap *inside* or *between* allocated blocks

# Internal Fragmentation

- For a given block, *internal fragmentation* occurs if payload is smaller than the block




- **Causes:**
  - Padding for alignment purposes
  - Overhead of maintaining heap data structures (inside block, outside payload)
  - Explicit policy decisions (e.g., to return a big block to satisfy a small request)
- Easy to measure because only depends on past requests

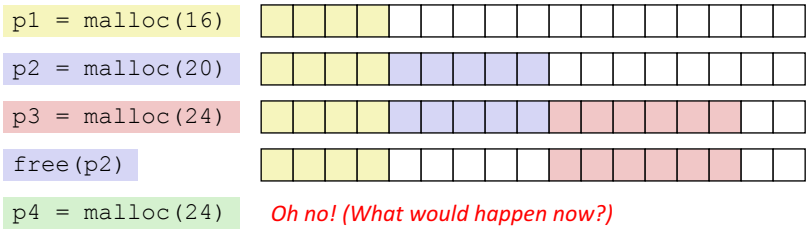




# External Fragmentation

= 4-byte word

- For the heap, *external fragmentation* occurs when allocation/free pattern leaves "holes" between blocks
  - That is, the aggregate payload is non-continuous
  - Can cause situations where there is enough aggregate heap memory to satisfy request, but no single free block is large enough




- Don't know what future requests will be
  - Difficult to impossible to know if past placements will become problematic

# Implementation Issues

- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- How do we pick a block to use for allocation (when many might fit)?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we reinsert a freed block into the heap?

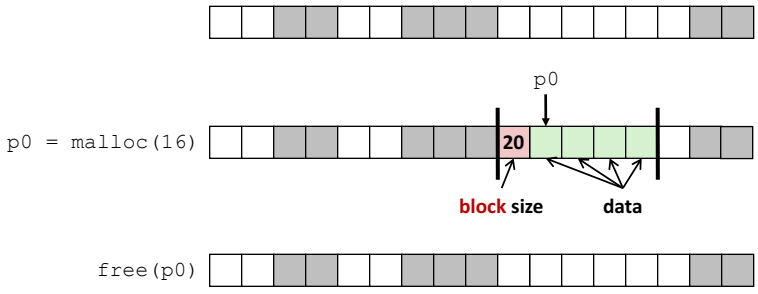




# Knowing How Much to Free

= 4-byte word (free)
= 4-byte word (allocated)

- Standard method
  - Keep the length of a block in the word preceding the block
    - This word is often called the ***header field*** or ***header***
  - Requires an extra word for every allocated block



# BONUS SLIDES

## For Fun: DRAMMER Security Attack

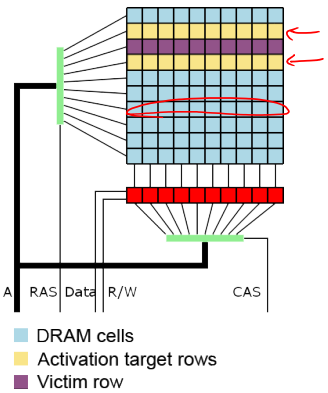
- Why are we talking about this?
  - **Current:** Announced in October 2016; Google released Android patch on November 8
  - **Relevant:** Uses your system's memory setup to gain elevated privileges
    - Ties together some of what we've learned about virtual memory and processes
  - **Interesting:** 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/w/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

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

- 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



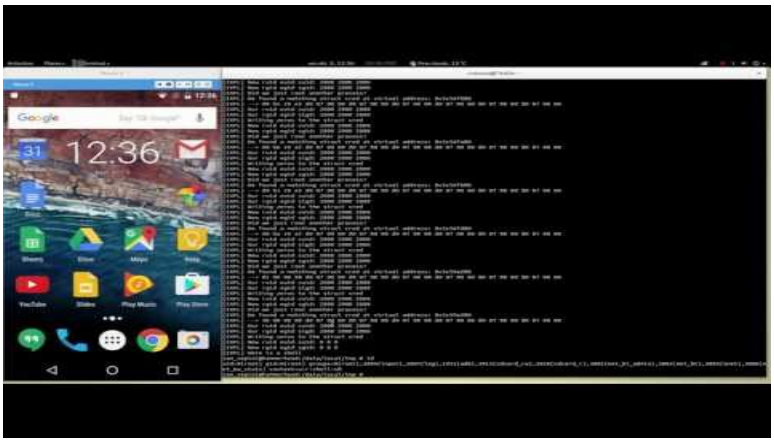


## What's DRAMMER?

- 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: 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