CS 61C: Great Ideas in Computer Architecture

Lecture 23: *Virtual Memory*

Bernhard Boser & Randy Katz

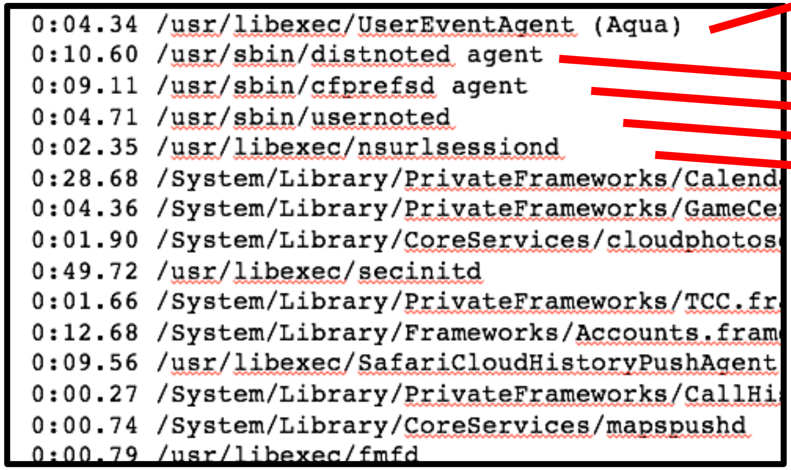
http://inst.eecs.berkeley.edu/~cs61c

Agenda

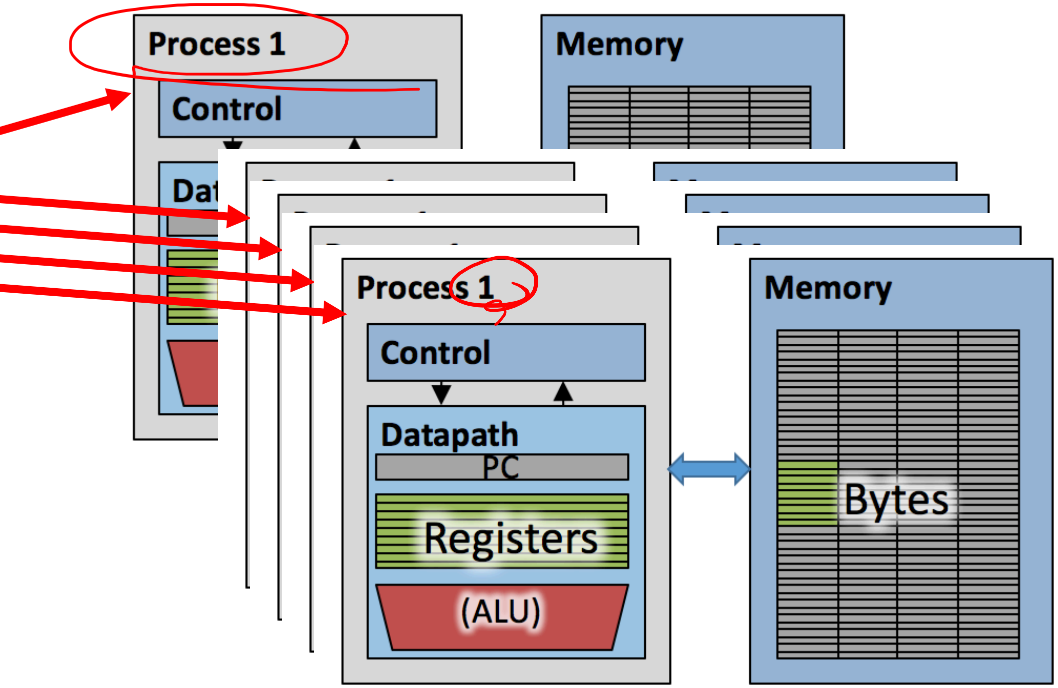
- Virtual Memory
- Paged Physical Memory
- Swap Space
- Page Faults
- Hierarchical Page Tables
- Caching Page Table Entries (TLB)

Virtual Machine

100+ Processes, managed by OS

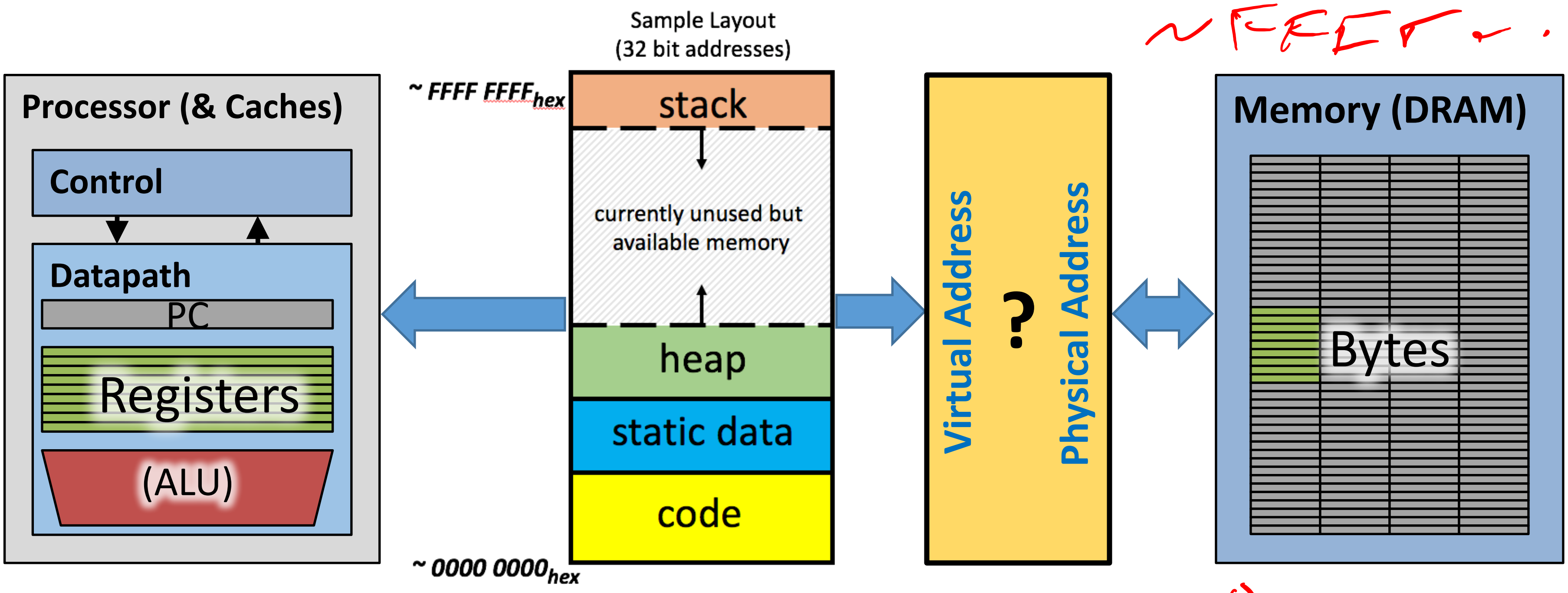


```
0:04.34 /usr/libexec/UserEventAgent (Aqua)
0:10.60 /usr/sbin/distnoted agent
0:09.11 /usr/sbin/cfprefsd agent
0:04.71 /usr/sbin/usernoted
0:02.35 /usr/libexec/nsurlsessiond
0:28.68 /System/Library/PrivateFrameworks/Calend
0:04.36 /System/Library/PrivateFrameworks/GameCe
0:01.90 /System/Library/CoreServices/cloudphotos
0:49.72 /usr/libexec/secinitd
0:01.66 /System/Library/PrivateFrameworks/TCC.fr
0:12.68 /System/Library/Frameworks/Accounts.fram
0:09.56 /usr/libexec/SafariCloudHistoryPushAgent
0:00.27 /System/Library/PrivateFrameworks/CallHi
0:00.74 /System/Library/CoreServices/mapspushd
0:00.79 /usr/libexec/fmfd
```



- 100’s of processes
  - OS multiplexes these over available cores
- But what about memory?
  - There is only one!
  - We cannot just ”save” its contents in a context switch …

Virtual versus Physical Addresses



Many of these (software & hardware cores)

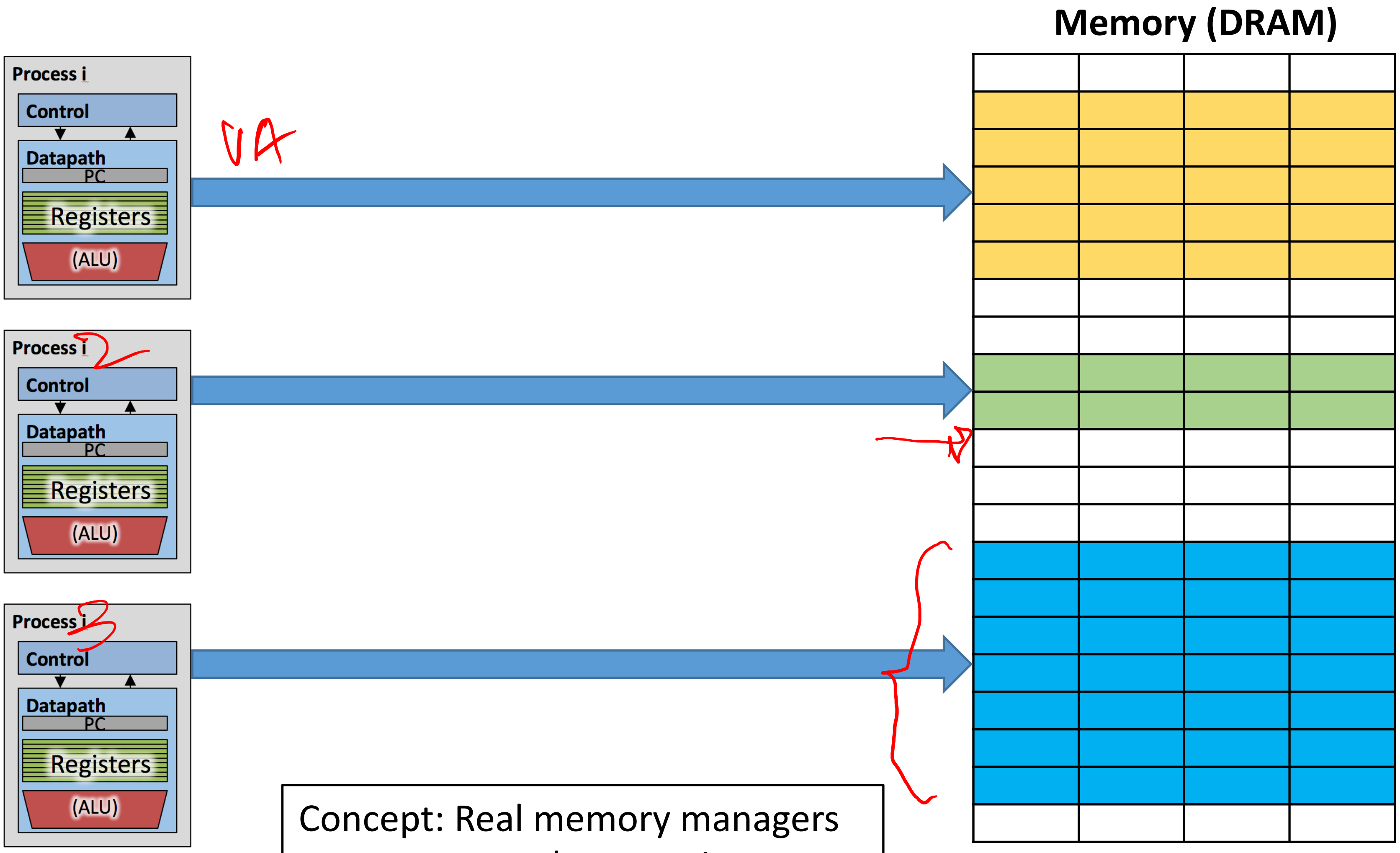
One main memory

- Processes use virtual addresses, e.g. 0 … 0xffff,ffff
  - Many processes, all using same (conflicting) addresses
- Memory uses physical addresses (also, e.g. 0 ... 0xffff,ffff)
- ***Memory manager maps virtual to physical addresses***

Address Spaces

- Address space
  = set of addresses for all available memory locations
- Now, 2 kinds of memory addresses:
  - **Virtual Address Space**
    - set of addresses that the user program knows about
  - **Physical Address Space**
    - set of addresses that map to actual physical cells in memory
    - hidden from user applications
- Memory manager maps between these two address spaces

Conceptual Memory Manager



Concept: Real memory managers use more complex mappings.

Responsibilities of Memory Manager

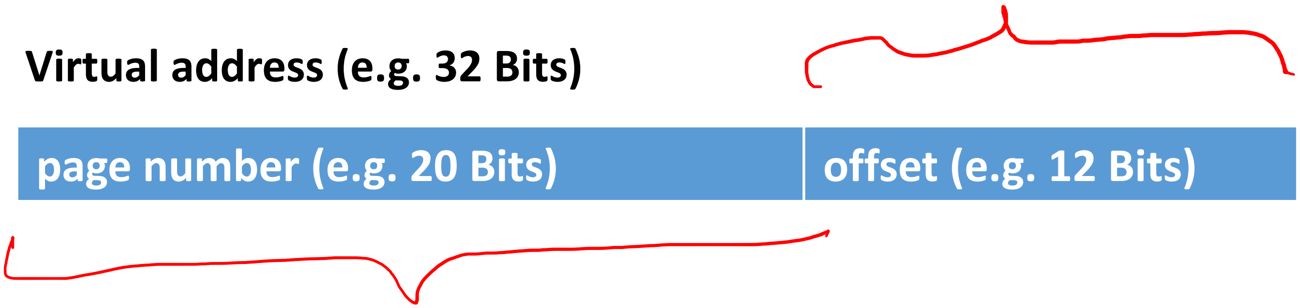
1) Map virtual to physical addresses
2) Protection:
   - Isolate memory between processes
   - Each process gets dedicate ”private” memory
   - Errors in one program won’t corrupt memory of other program
   - Prevent user programs from messing with OS’ memory
3) Swap memory to disk
   - Give illusion of larger memory by storing some content on disk
   - Disk is usually much larger & slower than DRAM
     - Use “clever” caching strategies

Agenda

- Virtual Memory
- **Paged Physical Memory**
- Swap Space
- Hierarchical Page Tables
- Caching Page Table Entries (TLB)

Memory Manager

- Several options
- Today “paged memory” dominates
  - Physical memory (DRAM) is broken into pages
  - Typical page size: 4 KiB+

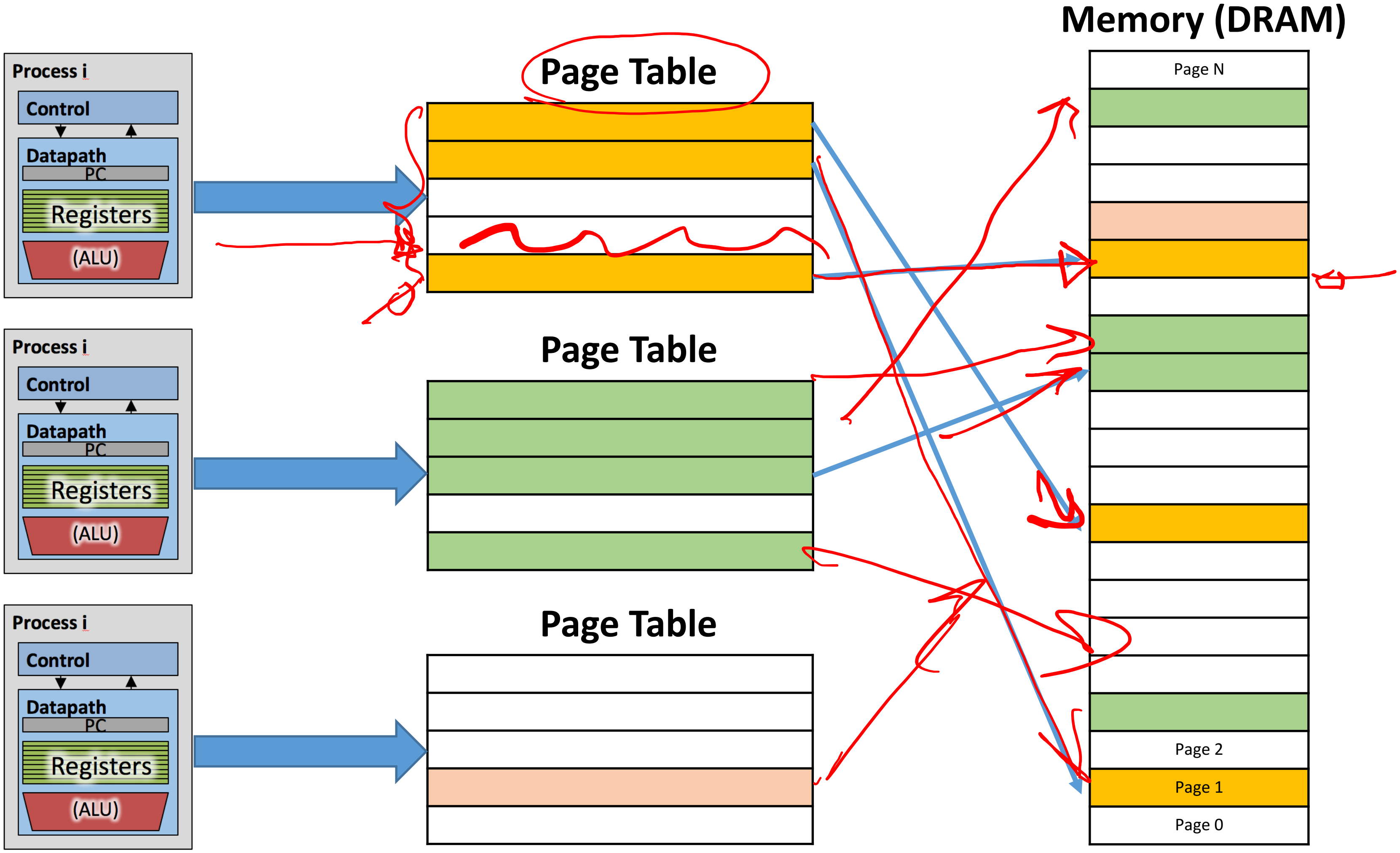


Virtual address (e.g. 32 Bits)

page number (e.g. 20 Bits)

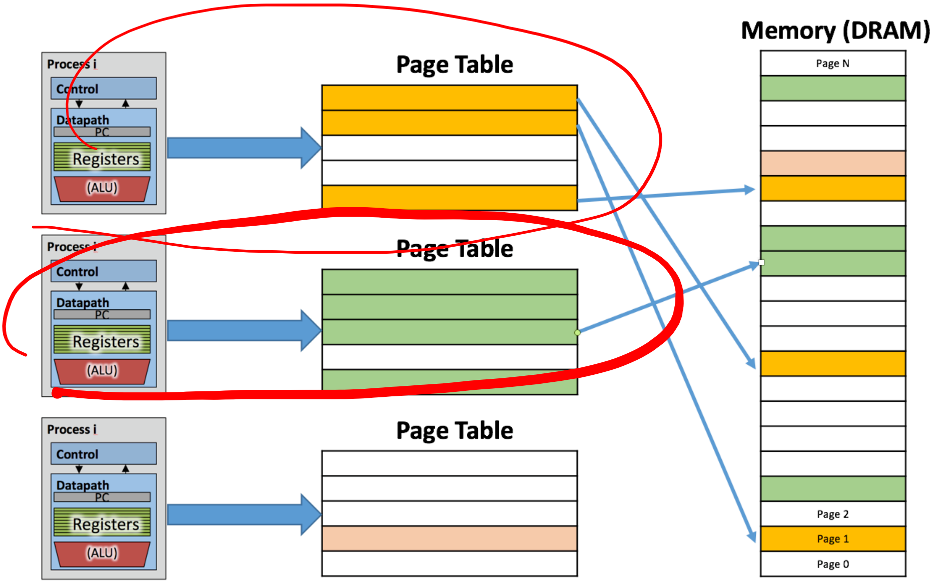
offset (e.g. 12 Bits)

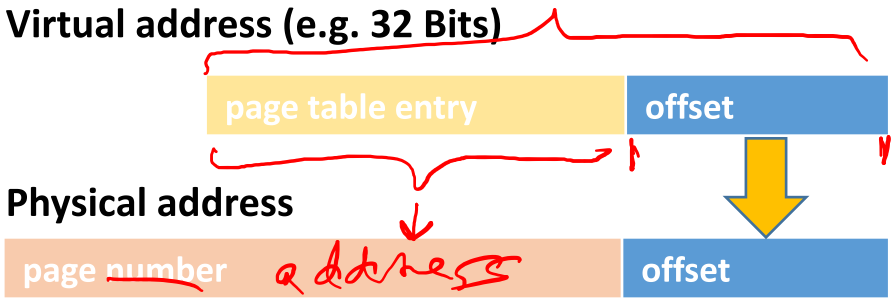
Paged Memory



Each process has a dedicated page table. Physical memory non-consecutive.

Paged Memory Address Translation

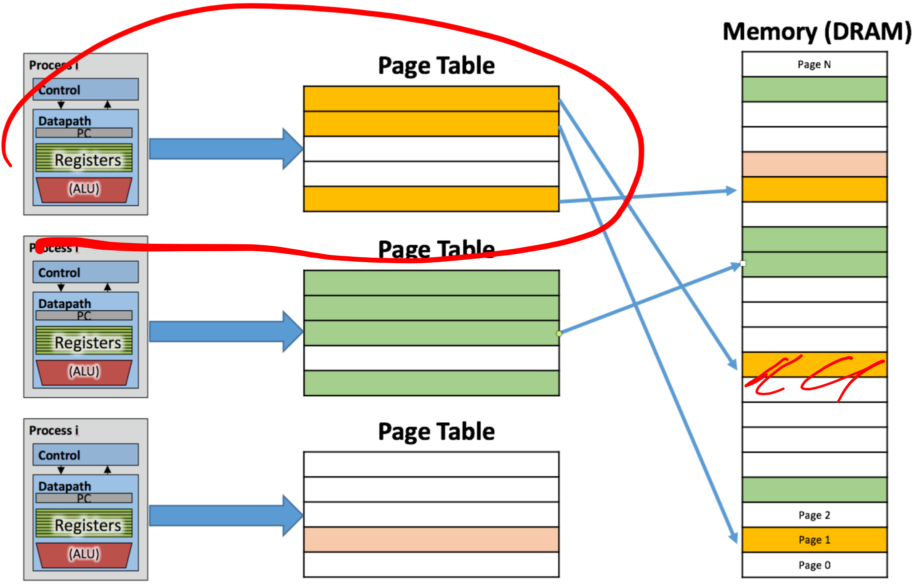




Physical addresses may (but do not have to) have more or fewer bits than virtual addresses

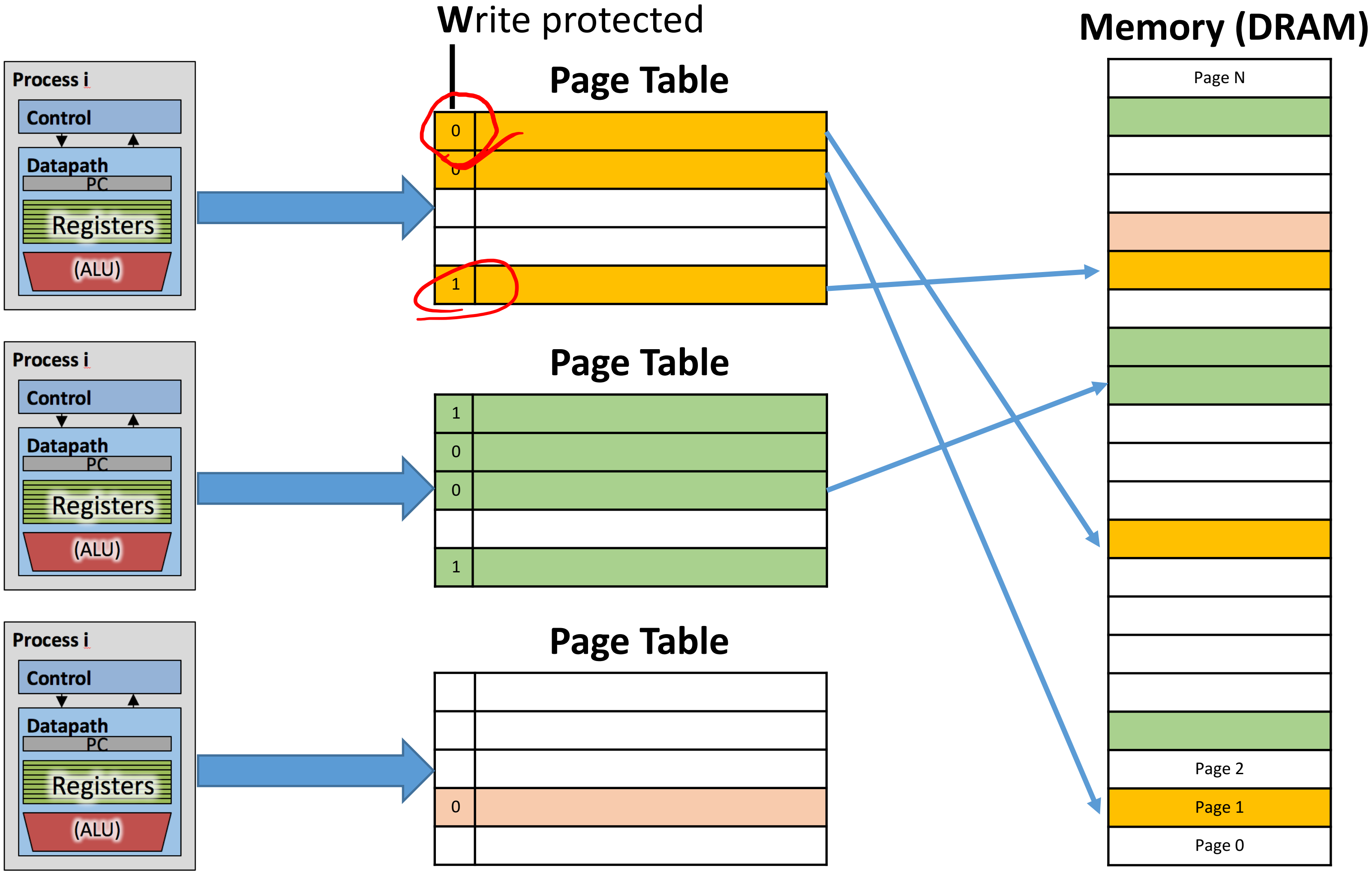
- OS keeps track of which process is active
  - Chooses correct page table
- Memory manager extracts page number from virtual address
- Looks up page address in page table
- Computes physical memory address from sum of
  - page address and
  - offset (from virtual address)

Protection



- Assigning different pages in DRAM to processes also keeps them from accessing each others memory
  - Isolation
  - Page tables handled by OS (in supervisory mode)
- Sharing is possible also
  - OS may assign same physical page to several processes

Write Protection



Exception when writing to protected page (e.g. program code).

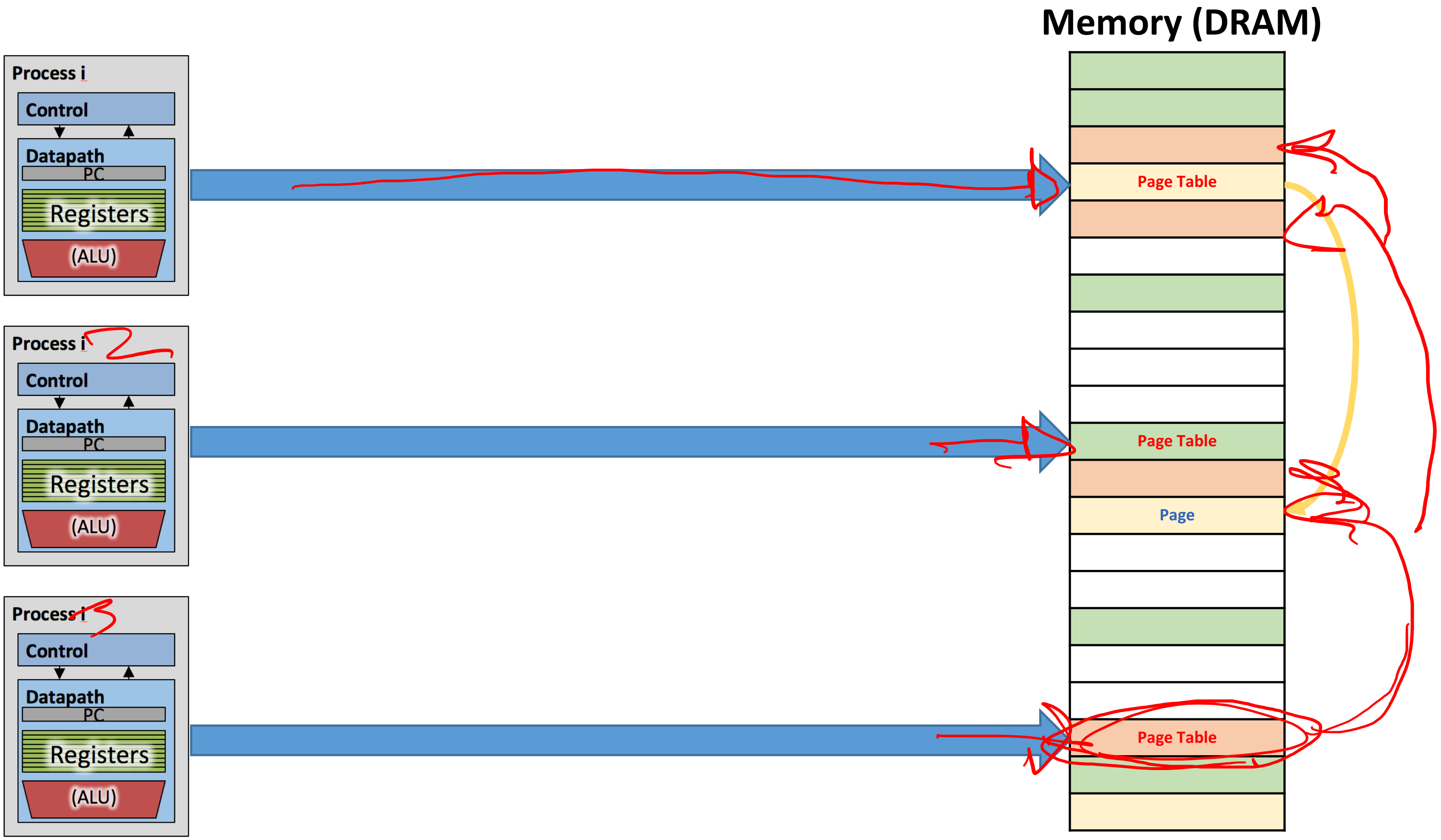
Where do Page Tables Reside?

- E.g. 32-Bit virtual address, 4-KiB pages
  - Single page table size:
    - 4 x 2^{20} Bytes = 4-MiB
    - 0.1% of 4-GiB memory
    - But much too large for a cache!
- Store page tables in memory (DRAM)
  - Two (slow) memory accesses per **`lw/sw`** on cache miss
  - How could we minimize the performance penalty?
    - Transfer blocks (not words) between DRAM and processor cache
      - Exploit spatial locality
    - Cache for frequently used page table entries …

12

20 page

Paged Table Storage in DRAM



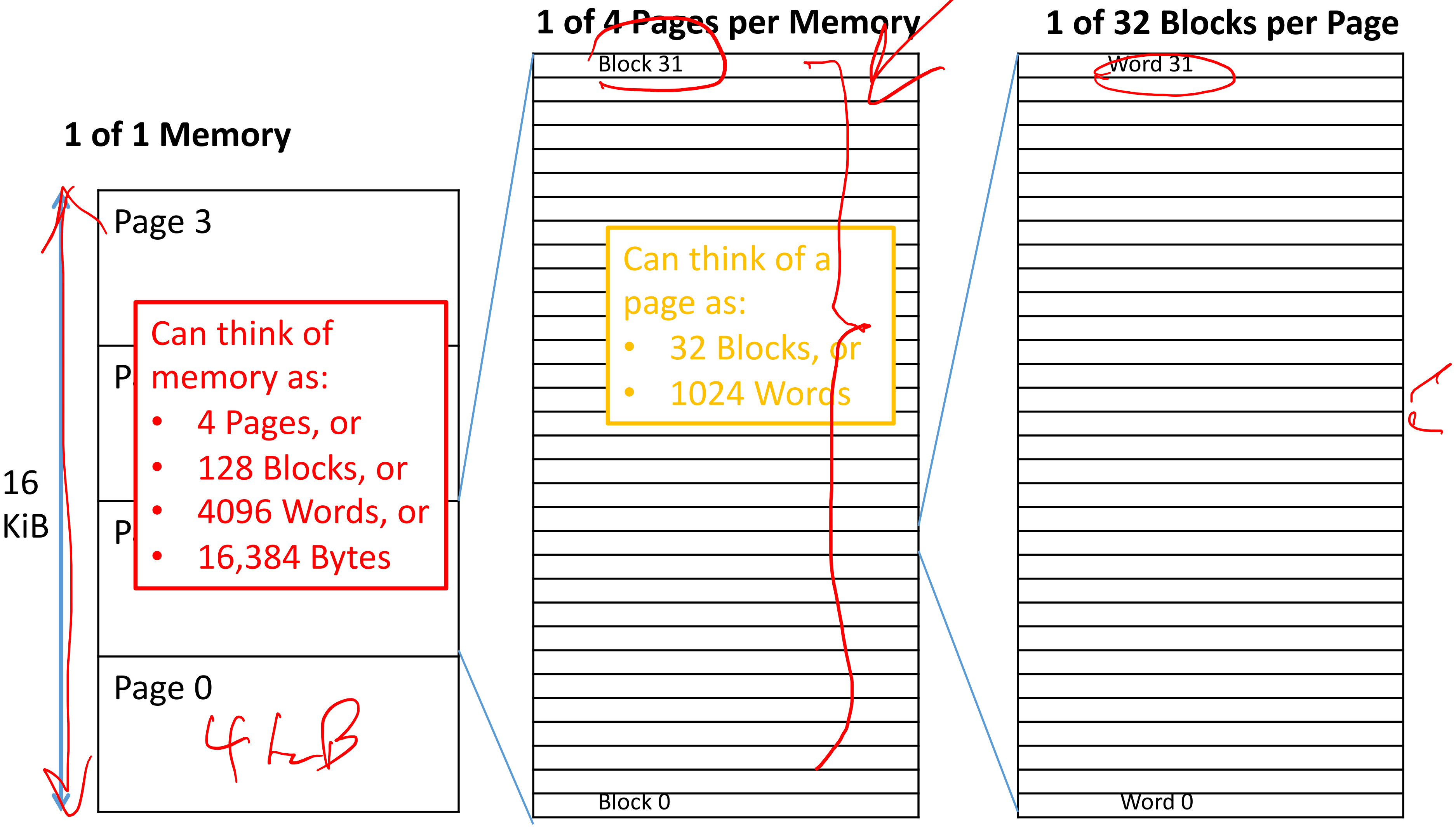
lw/sw take two memory references

Blocks vs. Pages

- In caches, we dealt with individual *blocks*
  - Usually ~64B on modern systems
- In VM, we deal with individual *pages*
  - Usually ~4 KB on modern systems
- Common point of confusion:
  - Bytes,
  - Words,
  - Blocks,
  - Pages
    - are all just different ways of looking at memory!

Bytes, Words, Blocks, Pages

Eg: 16 KiB DRAM, 4 KiB Pages (for VM), 128 B blocks (for caches), 4 B words (for **`lw/sw`**)



Can think of memory as:

- 4 Pages, or
- 128 Blocks, or
- 4096 Words, or
- 16,384 Bytes

Can think of a page as:

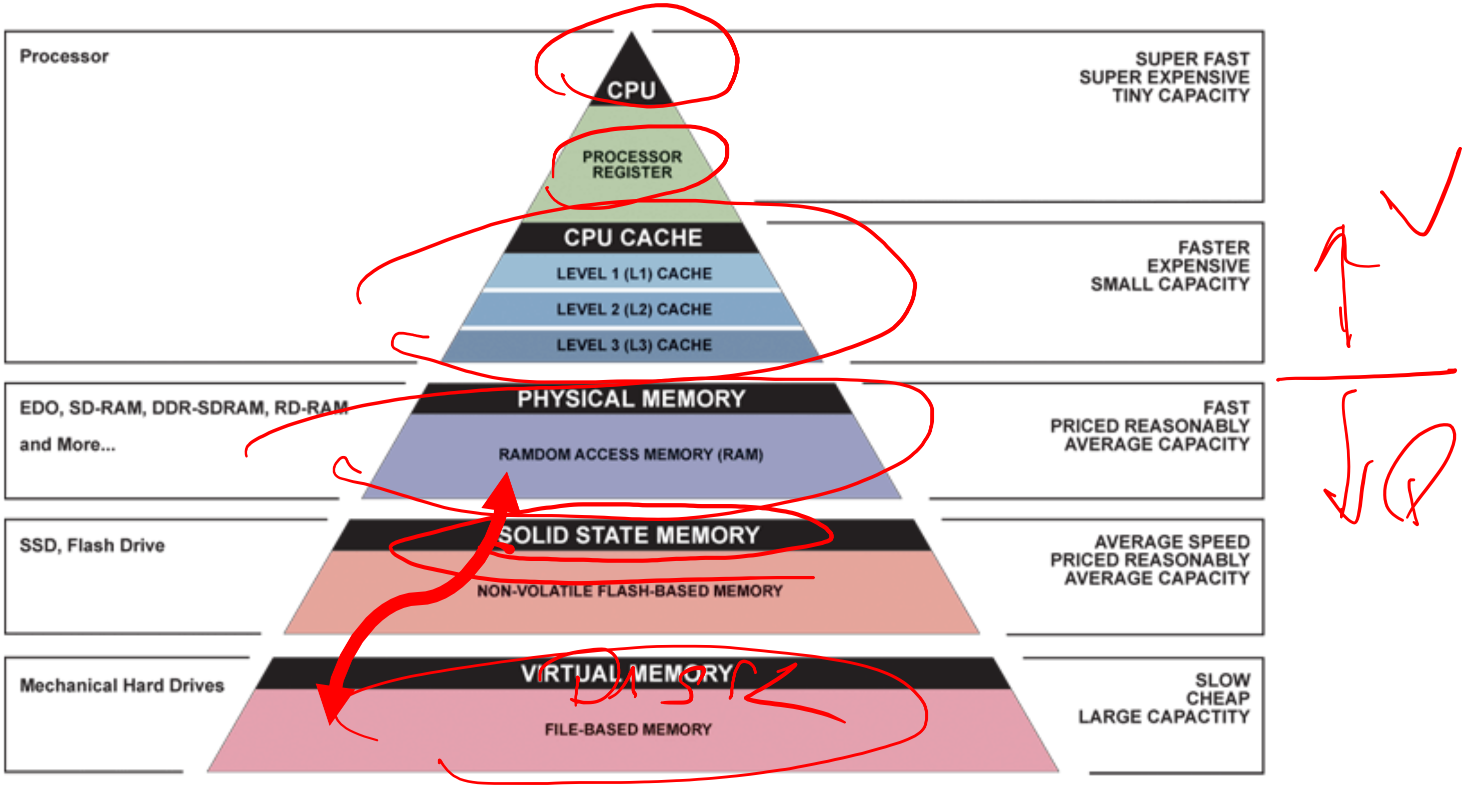
- 32 Blocks, or
- 1024 Words

Agenda

- Virtual Memory
- Paged Physical Memory
- **Swap Space**
- Hierarchical Page Tables
- Caching Page Table Entries (TLB)

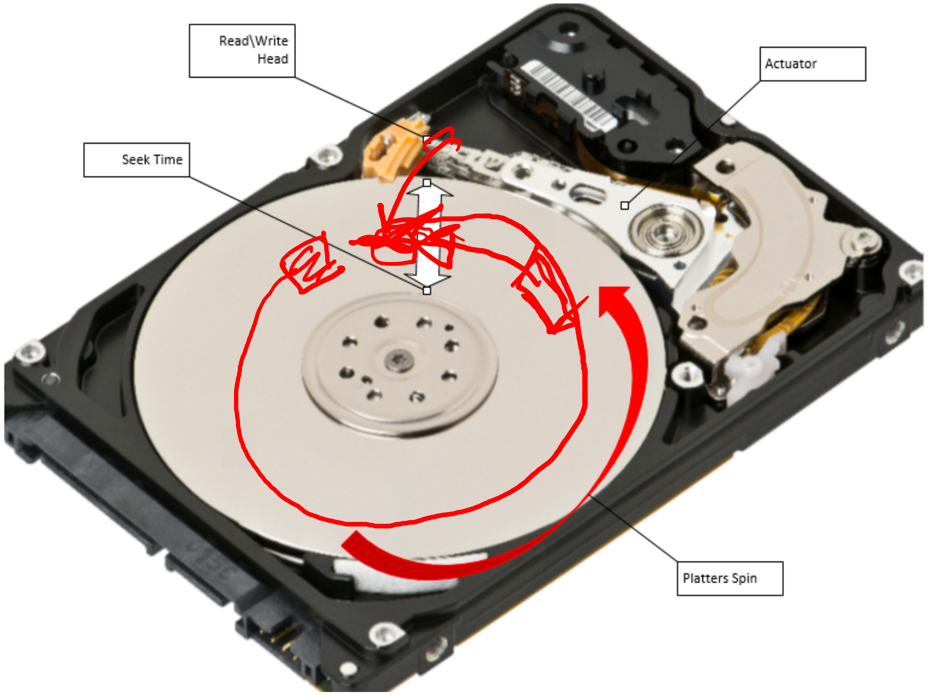
Memory Hierarchy

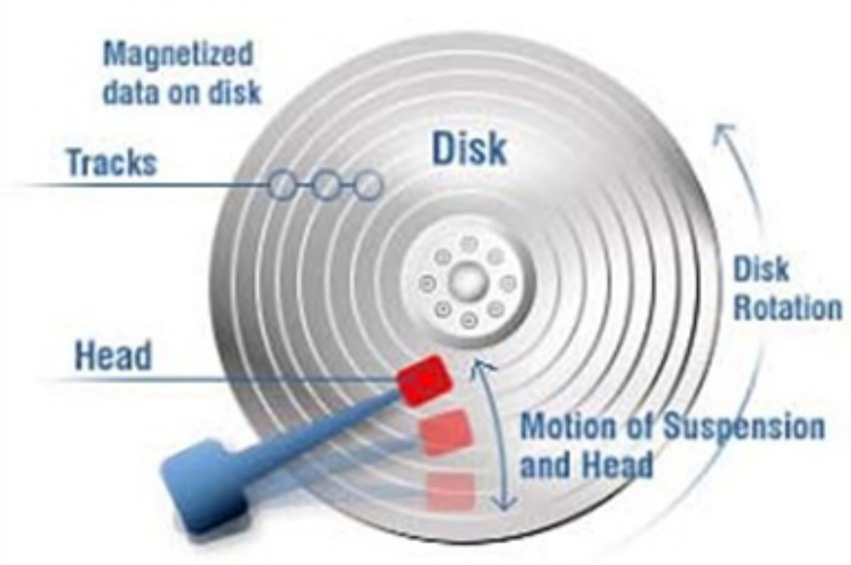
- Disk
  - Slow
  - But huge
  - How could we make use of its capacity (when running low on DRAM)?



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

Aside … why is disk so slow?



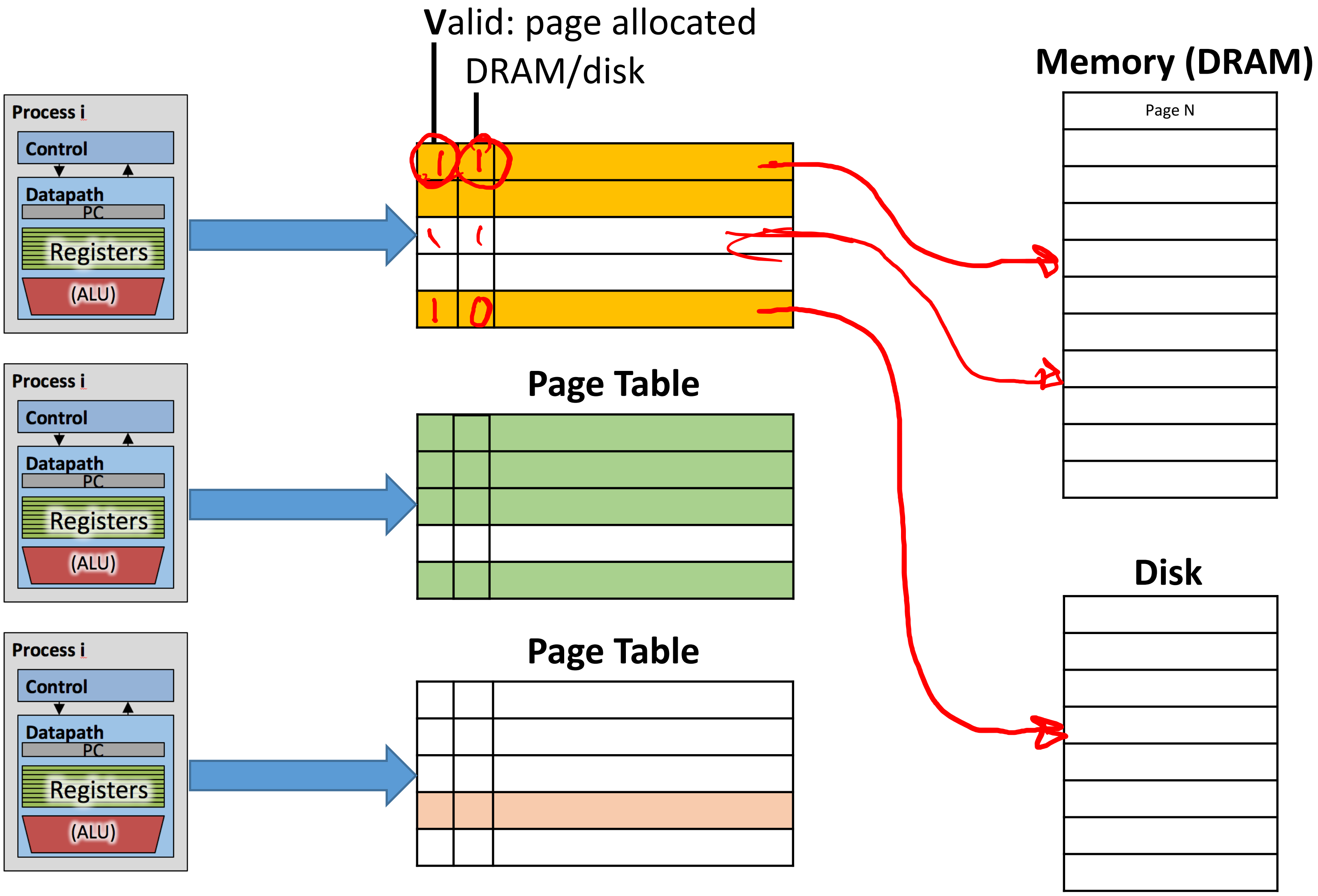


- 10,000 rpm (revolutions per minute)
- 6 ms per revolution
- Average random access time: 3 ms

What about SSD?

- Made with transistors
- Nothing mechanical that turns
- Like “Ginormous” register file
  - That does not ”forget” when power is off
- Fast access to all cells, regardless of address
- Still much slower than register, DRAM
  - Read/write blocks, not bytes
  - Potential reliability issues

Paged Memory



Each process has a dedicated page table. Physical memory non-consecutive.

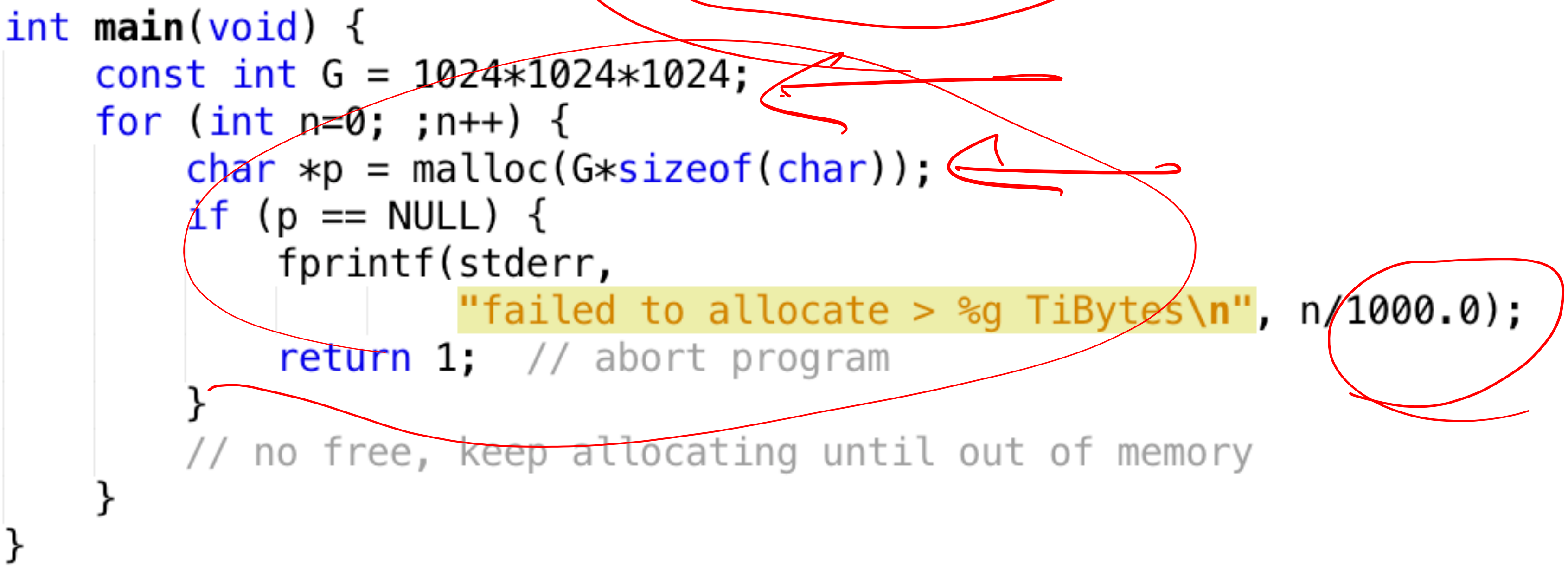
Memory Access

- Check page table entry:
  - Valid?
    - Yes, valid → In DRAM?
      - Yes, in DRAM: read/write data
      - No, on disk: allocate new page in DRAM
        - If out of memory, evict a page from DRAM
        - Store evicted page to disk
        - Read page from disk into memory
        - Read/write data
    - Not Valid
      - allocate new page in DRAM
        - If out of memory, evict a page
        - Read/write data

Page fault
OS intervention

Lecture 4: Out of Memory

- Insufficient free memory: **`malloc()`** returns **`NULL`**



```c
int main(void) {
    const int G = 1024*1024*1024;
    for (int n=0; ;n++) {
        char *p = malloc(G*sizeof(char));
        if (p == NULL) {
            fprintf(stderr,
                    "failed to allocate > %g TiBytes\n", n/1000.0);
            return 1;  // abort program
        }
        // no free, keep allocating until out of memory
    }
}
```

```
$ gcc OutOfMemory.c; ./a.out
failed to allocate > 131 TiBytes
```

What's going on?

Write-Through or Write-Back?

- DRAM acts like "cache" for disk
  - Should writes go directly to disk (write-through)?
  - Or only when page is evicted?

- Which option do you propose?
- Implementation?

→ Dirty Bit

Agenda

- Virtual Memory
- Paged Physical Memory
- Swap Space
- **Hierarchical Page Tables**
- Caching Page Table Entries (TLB)

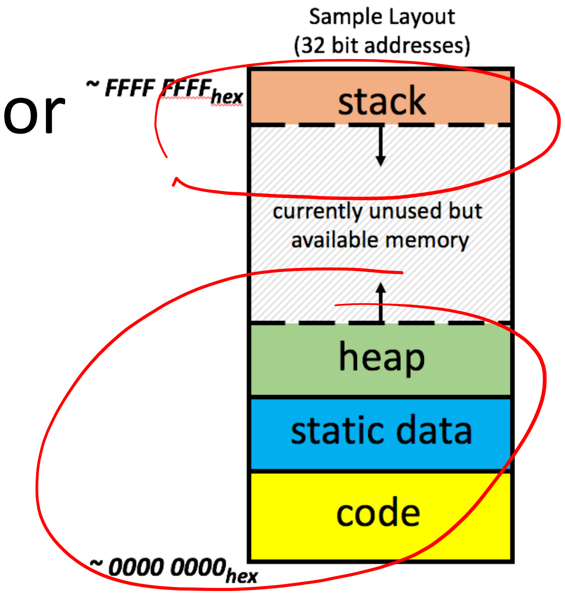
Size of Page Tables

- E.g. 32-Bit virtual address, 4-KiB pages
  - Single page table size:
    - 4 x 2^{20} Bytes = 4-MiB
    - 0.1% of 4-GiB memory
  - Total size for 256 processes (each needs a page table)
    - 256 x 4 x 2^{20} Bytes = 4-MiB
    - 1-GiB
    - 25% of 4-GiB memory!
- What about 64-bit addresses?

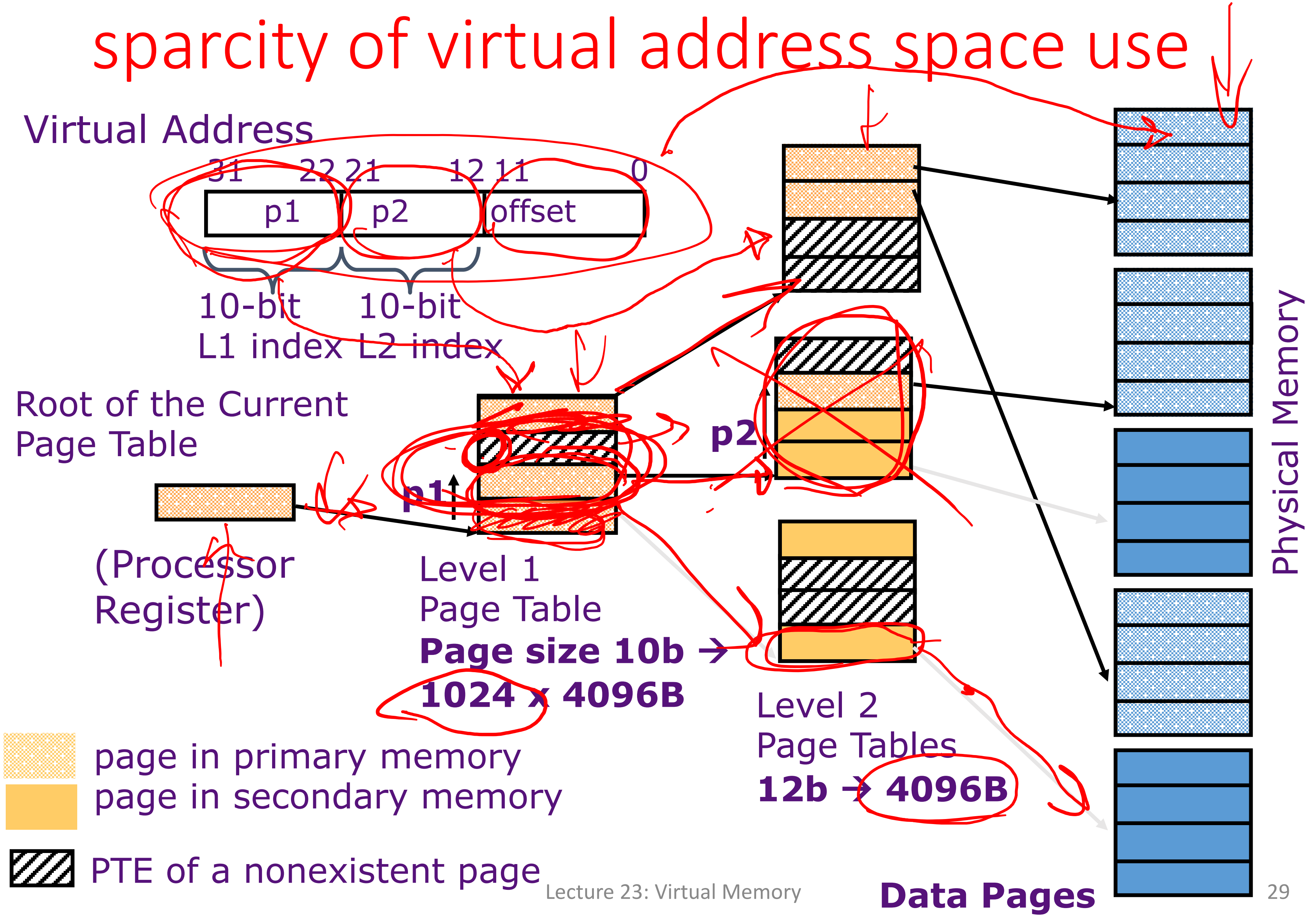
How can we keep the size of page tables "reasonable"?

Options

- Increase page size
  - E.g. doubling page size cuts PT size in half
  - At the expense of potentially wasted memory
- Hierarchical page tables
  - With decreasing page size
- Most programs use only fraction of memor
  - Split PT in two (or more) parts



Hierarchical Page Table – exploits sparcity of virtual address space use



Virtual Address

31 22 21 12 11 0

p1 p2 offset

10-bit L1 index

10-bit L2 index

Root of the Current Page Table

(Processor Register)

p1

p2

Level 1 Page Table

Page size 10b → 1024 x 4096B

Level 2 Page Tables

12b → 4096B

Physical Memory

Data Pages

▨ page in primary memory

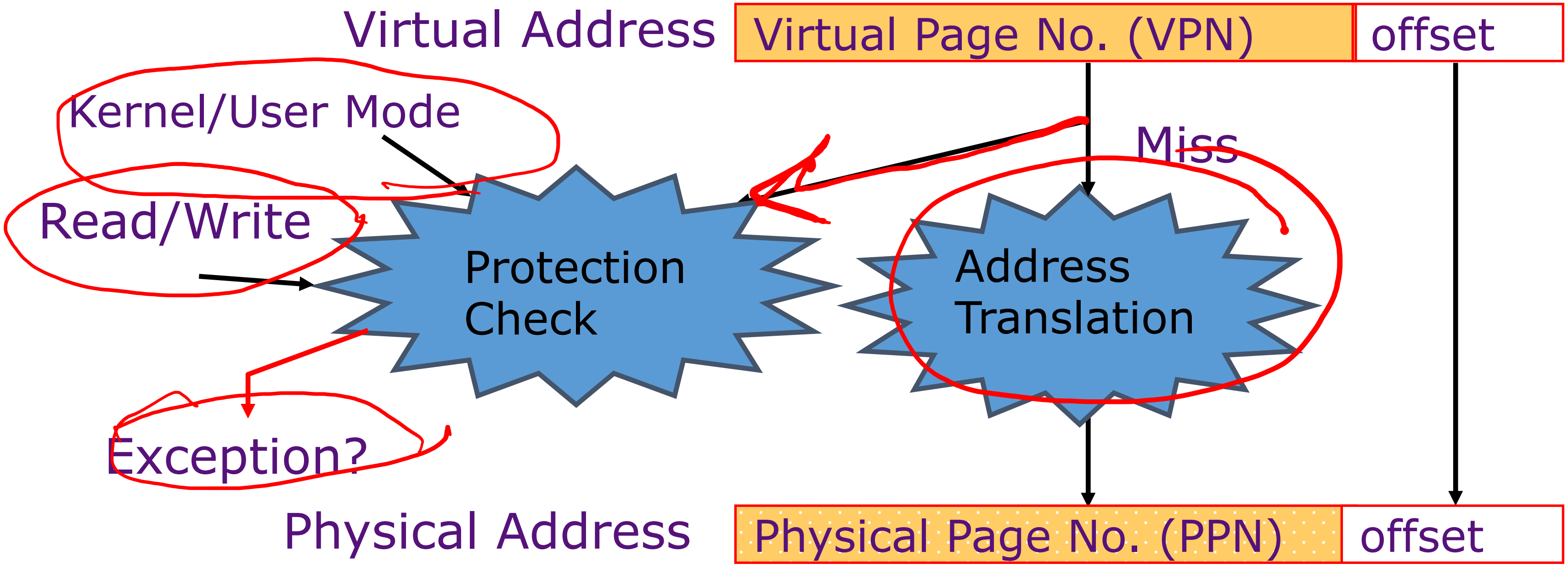
🟧 page in secondary memory

▨ PTE of a nonexistent page

Agenda

- Virtual Memory
- Paged Physical Memory
- Swap Space
- Hierarchical Page Tables
- **Caching Page Table Entries (TLB)**

Address Translation & Protection



- Every instruction and data access needs address translation and protection checks

A good VM design needs to be fast (~ one cycle) and space efficient

Translation Lookaside Buffers (TLB)

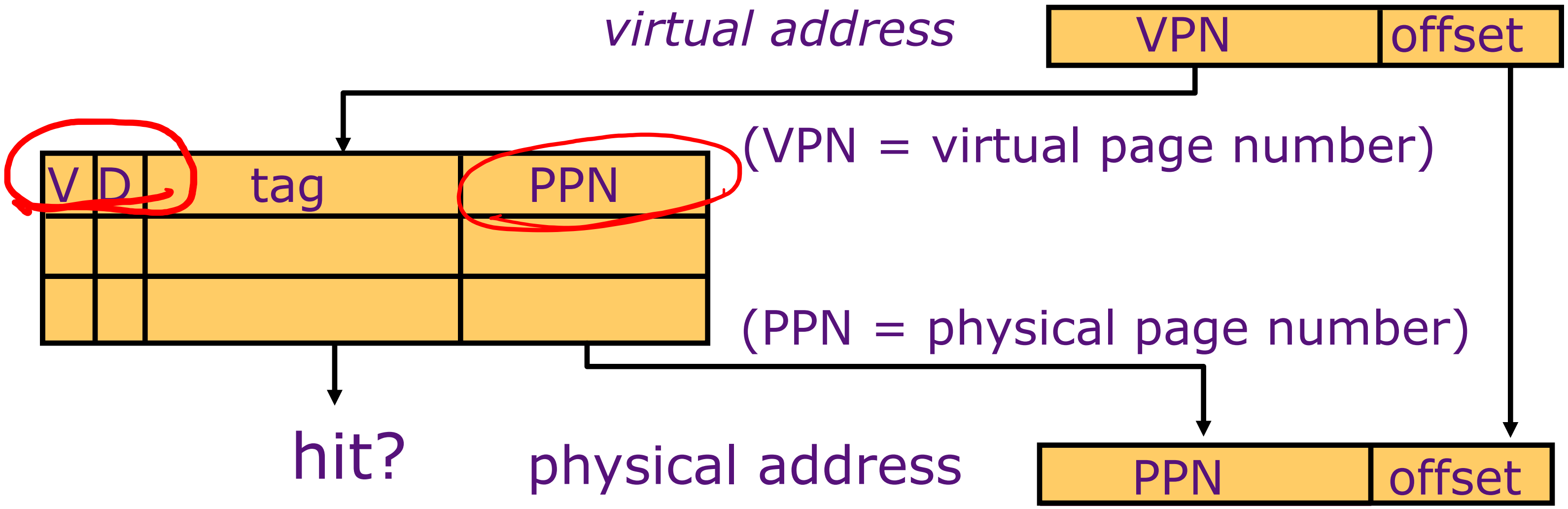
Address translation is very expensive!

In a two-level page table, each reference becomes three memory accesses

Solution: *Cache some translations in TLB*

TLB hit ⇒ *Single-Cycle Translation*

TLB miss ⇒ *Page-Table Walk to refill*



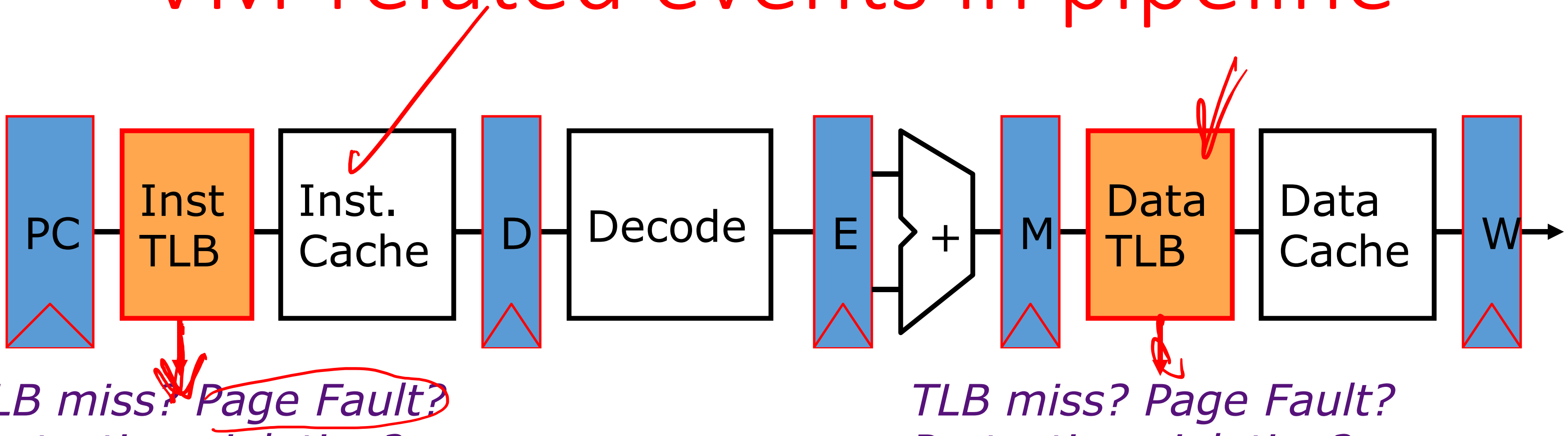
TLB Designs

- Typically 32-128 entries, usually fully associative
  - Each entry maps a large page, hence less spatial locality across pages => more likely that two entries conflict
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
  - Larger systems sometimes have multi-level (L1 and L2) TLBs
- Random or FIFO replacement policy
- "TLB Reach": Size of largest virtual address space that can be simultaneously mapped by TLB

Example: 64 TLB entries, 4KB pages, one page per entry

TLB Reach = ____?

VM-related events in pipeline

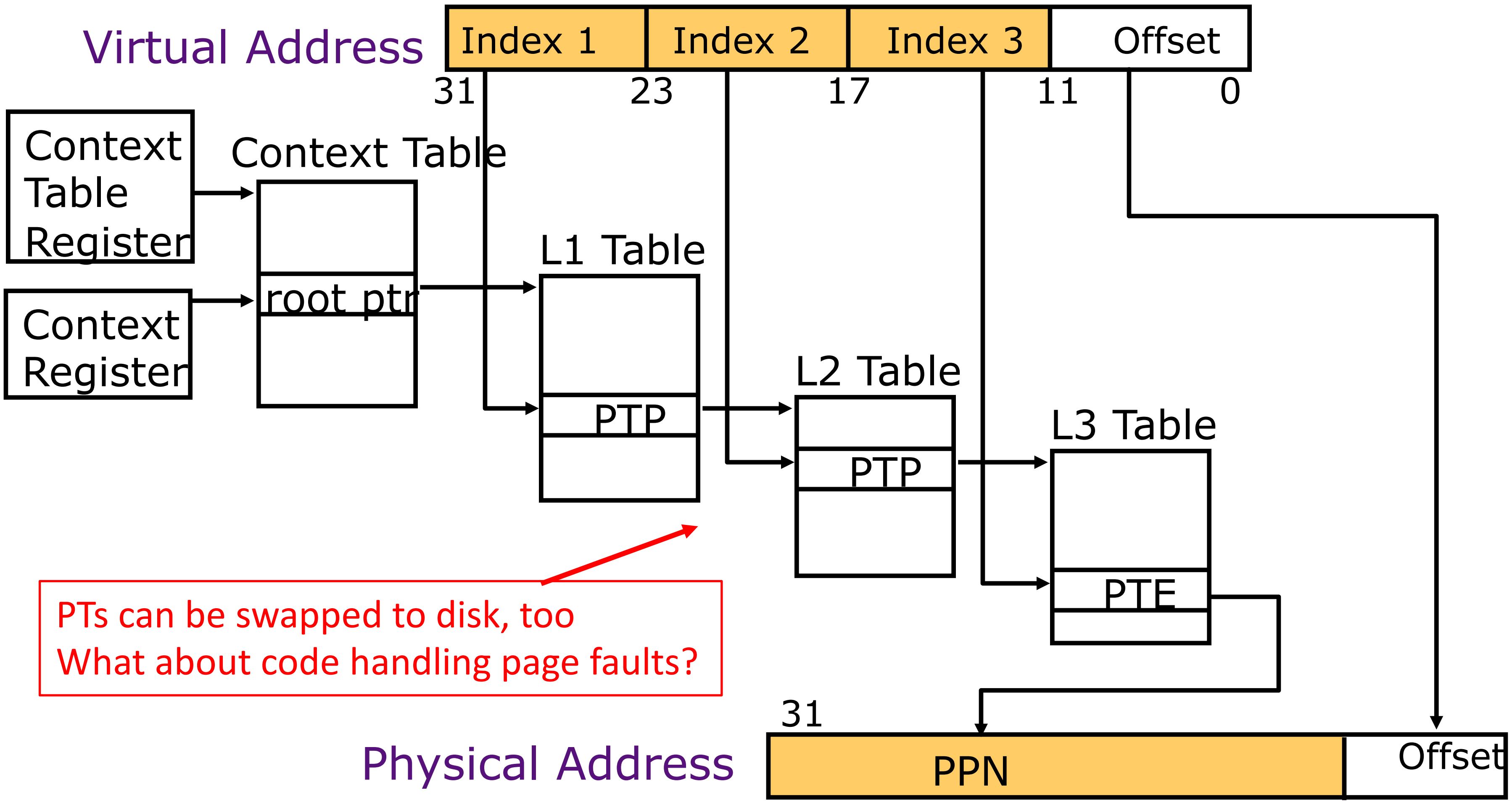


TLB miss? Page Fault?
Protection violation?

TLB miss? Page Fault?
Protection violation?

- Handling a TLB miss needs a hardware or software mechanism to refill TLB
  - usually done in hardware
- Handling a page fault (e.g., page is on disk) needs a ***precise*** trap so software handler can easily resume after retrieving page
- Protection violation may abort process

Hierarchical Page Table Walk: SPARC v8

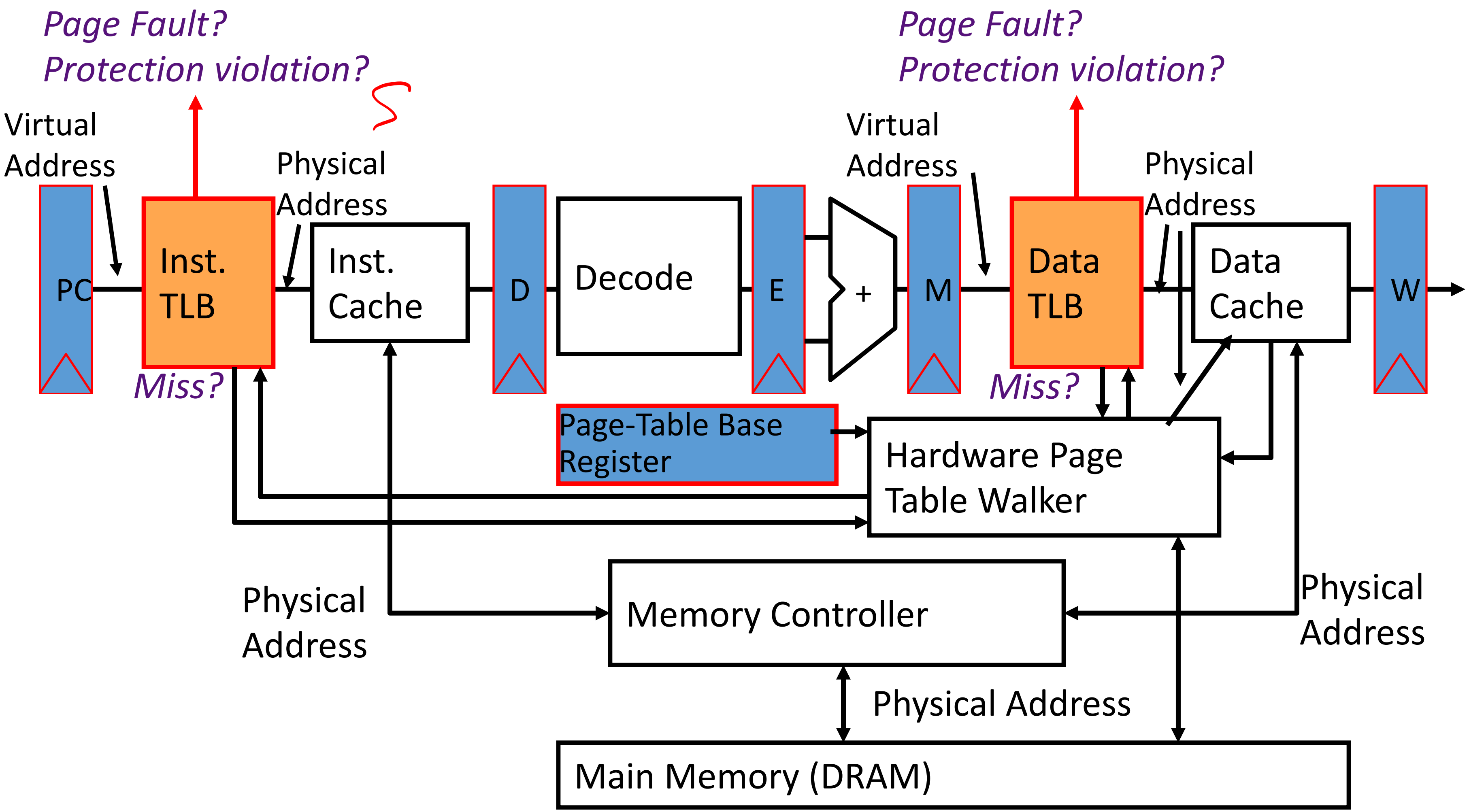


PTs can be swapped to disk, too
What about code handling page faults?

- MMU does this table walk in hardware on a TLB miss
- FSM?

Page-Based Virtual-Memory Machine

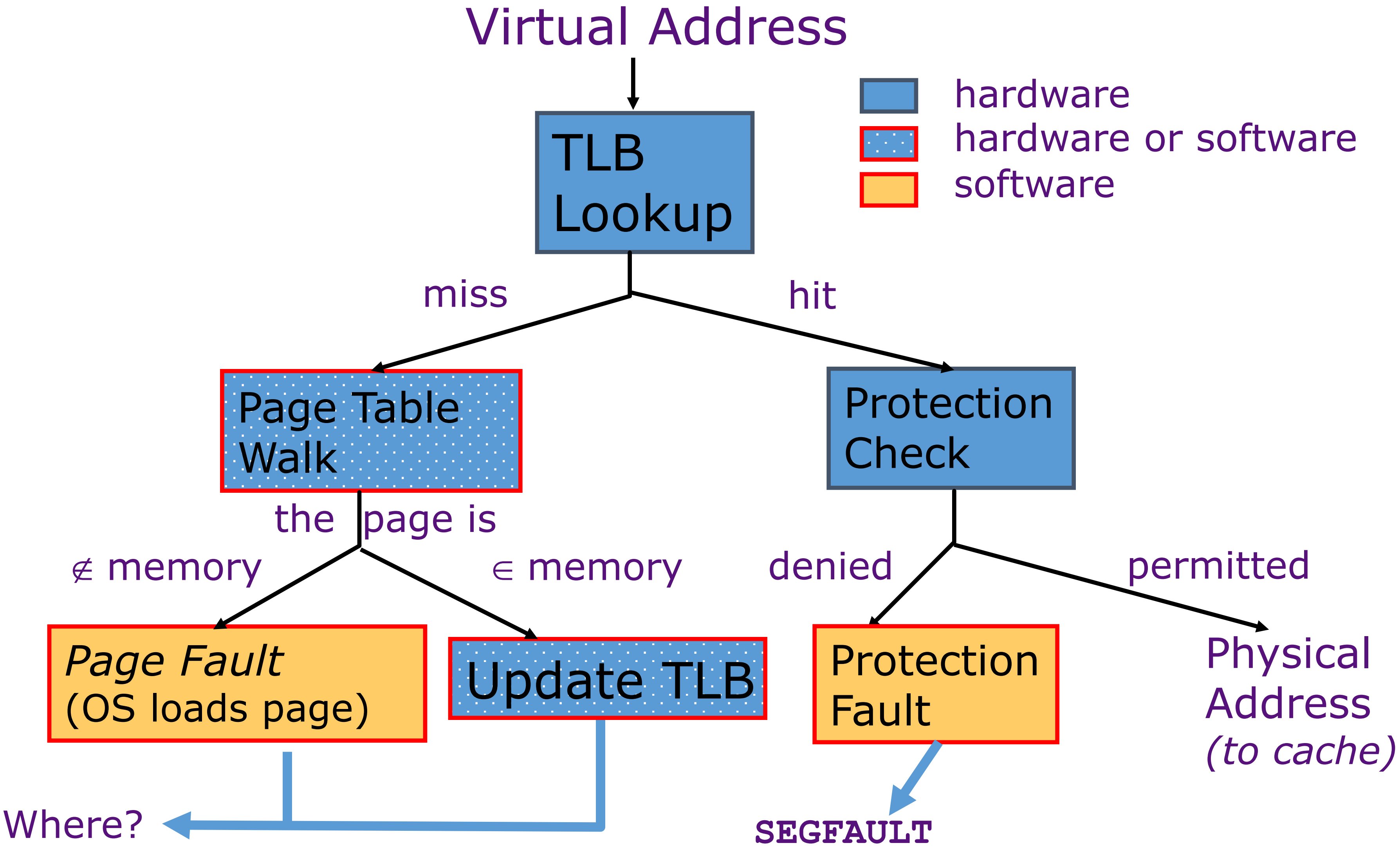
(Hardware Page-Table Walk)



- Assumes page tables held in untranslated physical memory

Address Translation:

putting it all together



Virtual Address

🟦 hardware
▨ hardware or software
🟧 software

TLB Lookup

miss

hit

Page Table Walk

the page is

∉ memory

∈ memory

Protection Check

denied

permitted

Page Fault
(OS loads page)

Update TLB

Protection Fault

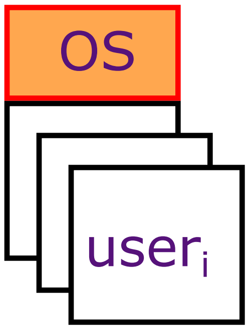
Physical Address
(to cache)

Where?

SEGFAULT

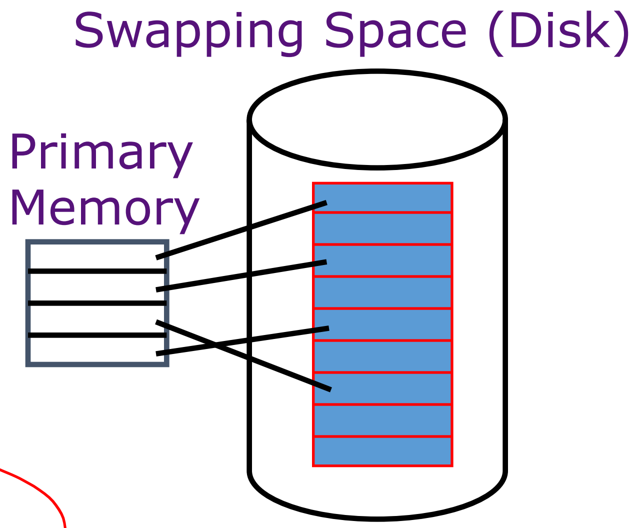
Modern Virtual Memory Systems

Illusion of a large, private, uniform store



Protection & Privacy

Several users/processes, each with their private address space

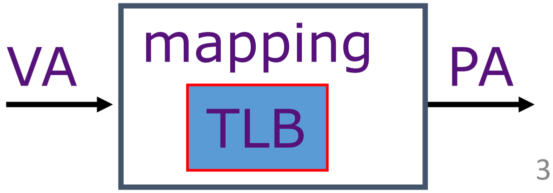


Demand Paging

Provides the ability to run programs larger than the primary memory

Hides differences in machine configurations

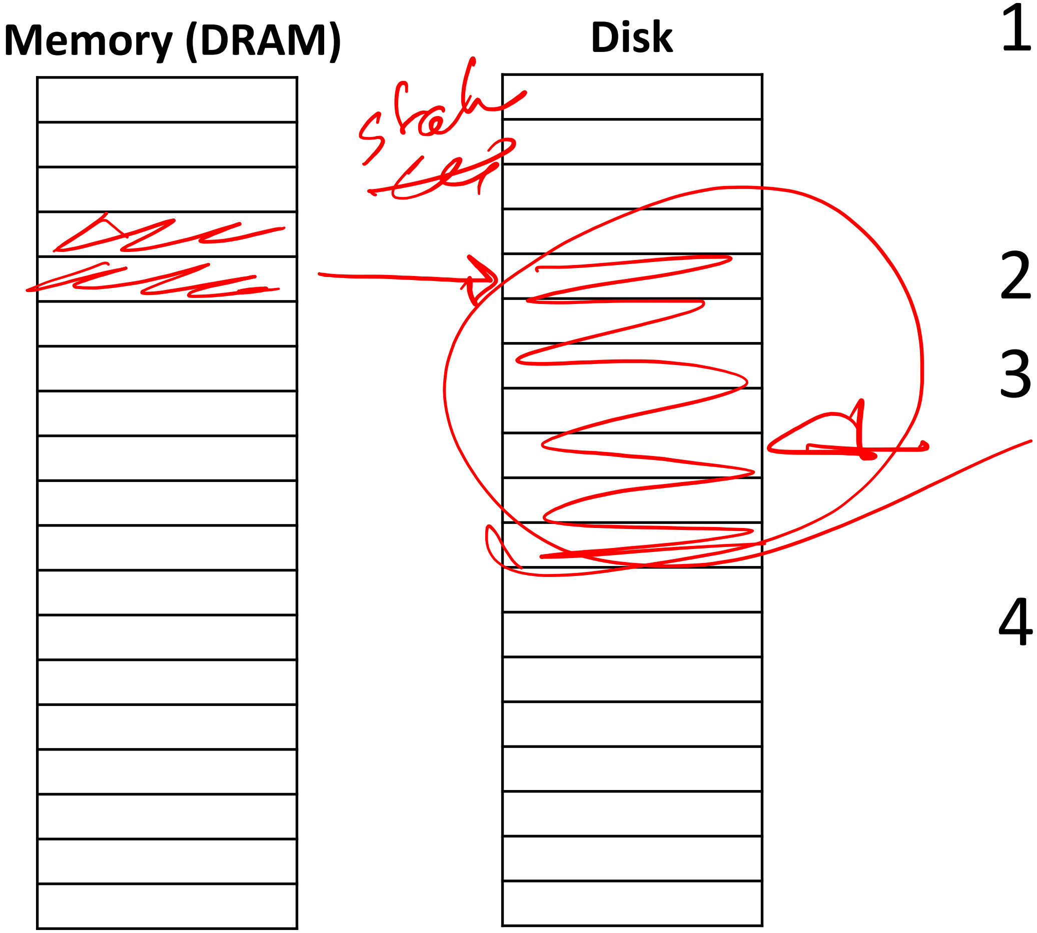
The price is address translation on each memory reference



It’s just the “OS” …

- Let’s write execve
  - Code the loads program into memory for execution
  - What’s the best way?

Execve



1. Set up PT
   - for program code
   - and stack
2. Init argv, argc
3. Call main
   - what happens?
   - page fault
4. Anything wrong with this?
   - linker?
   - fix?

Architecture matters!

And, in Conclusion …

- Virtual & physical addresses
  - Program → virtual address
  - DRAM → physical address
- Paged Memory
  1. Facilitates virtual → physical address translation
  2. Provides isolation & protection
  3. Extends available memory to include disk
- Implementation issues
  - Hierarchical page tables
  - Caching page table entries (TLB)