UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

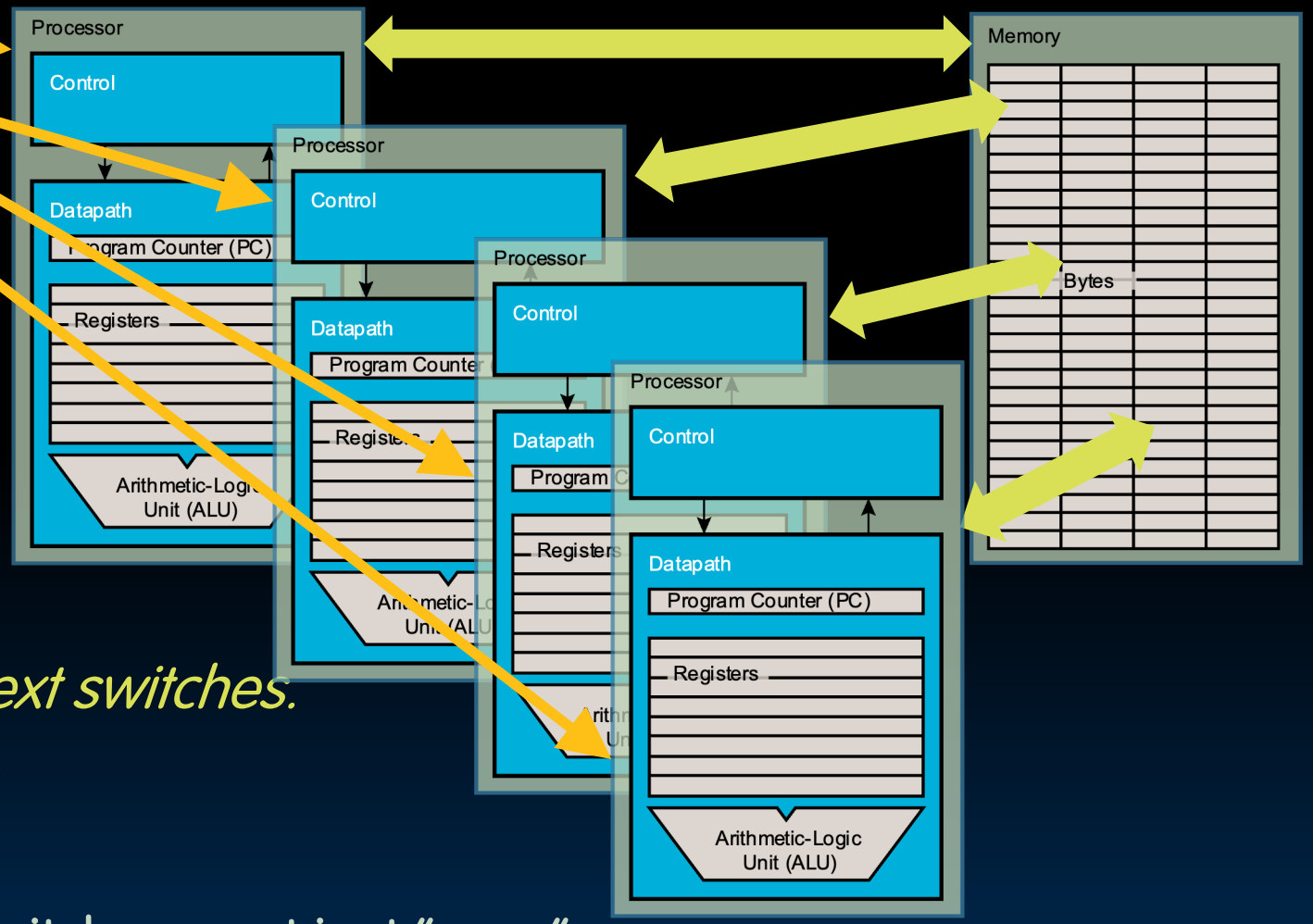
(a.k.a. Machine Structures)

UC Berkeley
Teaching Professor
Lisa Yan

Virtual Memory II: Page Faults, Multilevel, Interrupts/Exceptions

Memory Should Be Shared

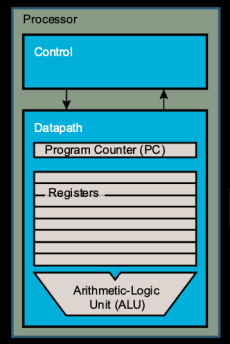

```
0:04.34 /usr/libexec/UserEventAgent (Aqua)
0:10.60 /usr/sbin/distnoted agent
0:09.11 /usr/sbin/cfprefsd agent
0:04.71 /usr/sbin/usernoted
0:02.35 /usr/libexec/nsurlsessiond
0:28.68 /System/Library/PrivateFrameworks/Calend
0:04.36 /System/Library/PrivateFrameworks/GameSe
0:01.90 /System/Library/CoreServices/cloudphotos
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
```



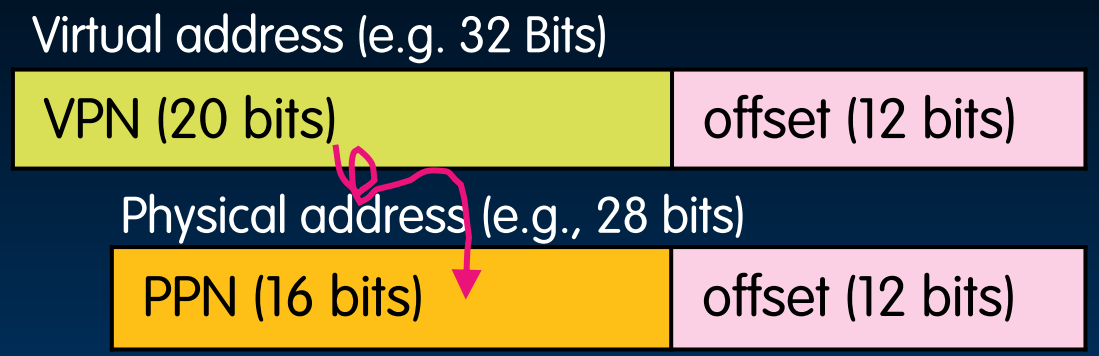
- **100's of processes**
  - OS multiplexes these over available cores.
  - If single-core, performs *context switches.*
- **But what about memory?**
  - There is only one! DRAM
  - Single-core: An OS context switch cannot just "save" memory's contents…too costly!

Virtual Memory: Page Tables

- **Each process has a dedicated page table.**
  - OS keeps track of which process is active.



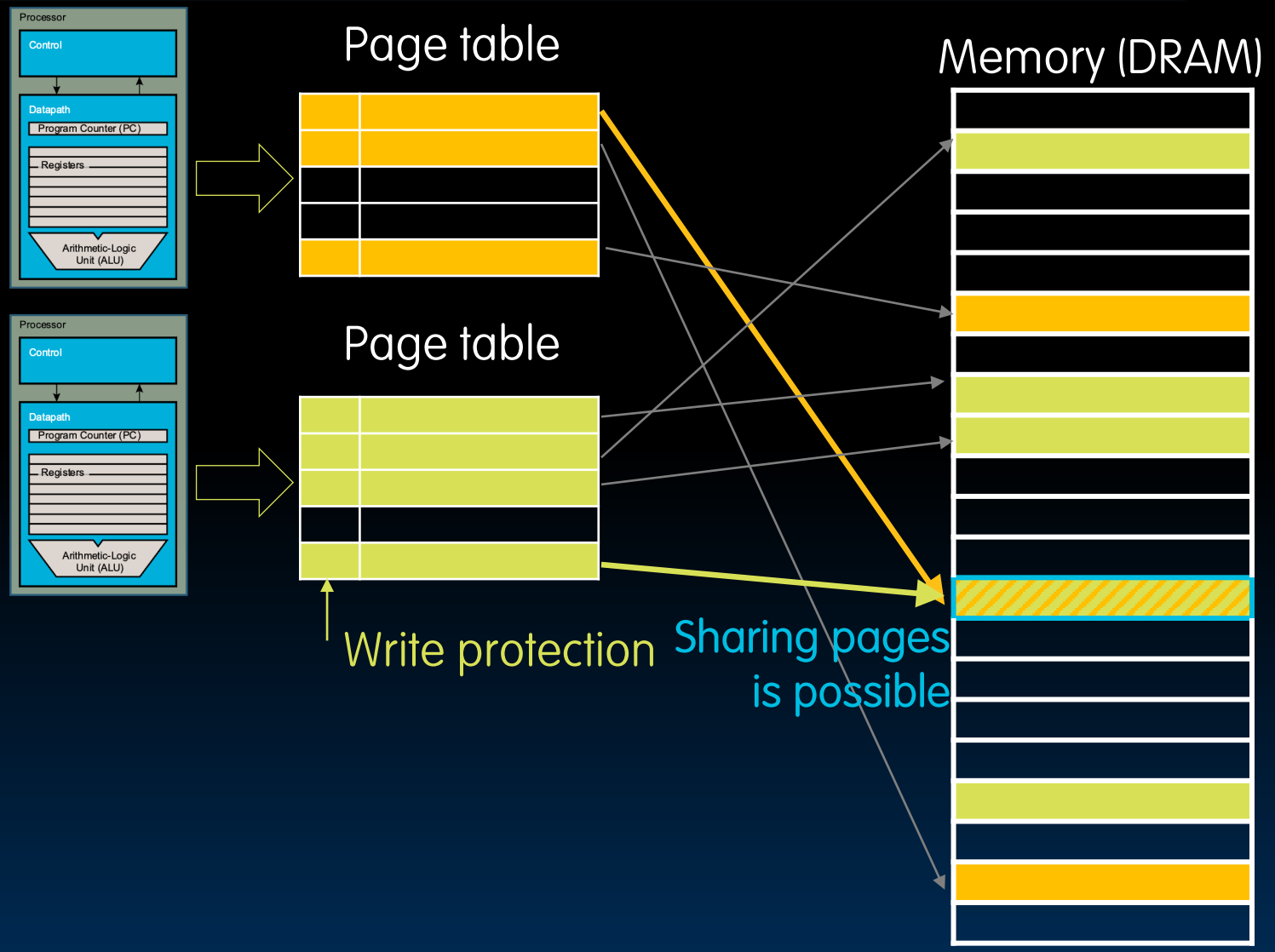
We now have **two address spaces**: virtual (per process) and physical (memory).



Virtual Memory: Page Tables

Review

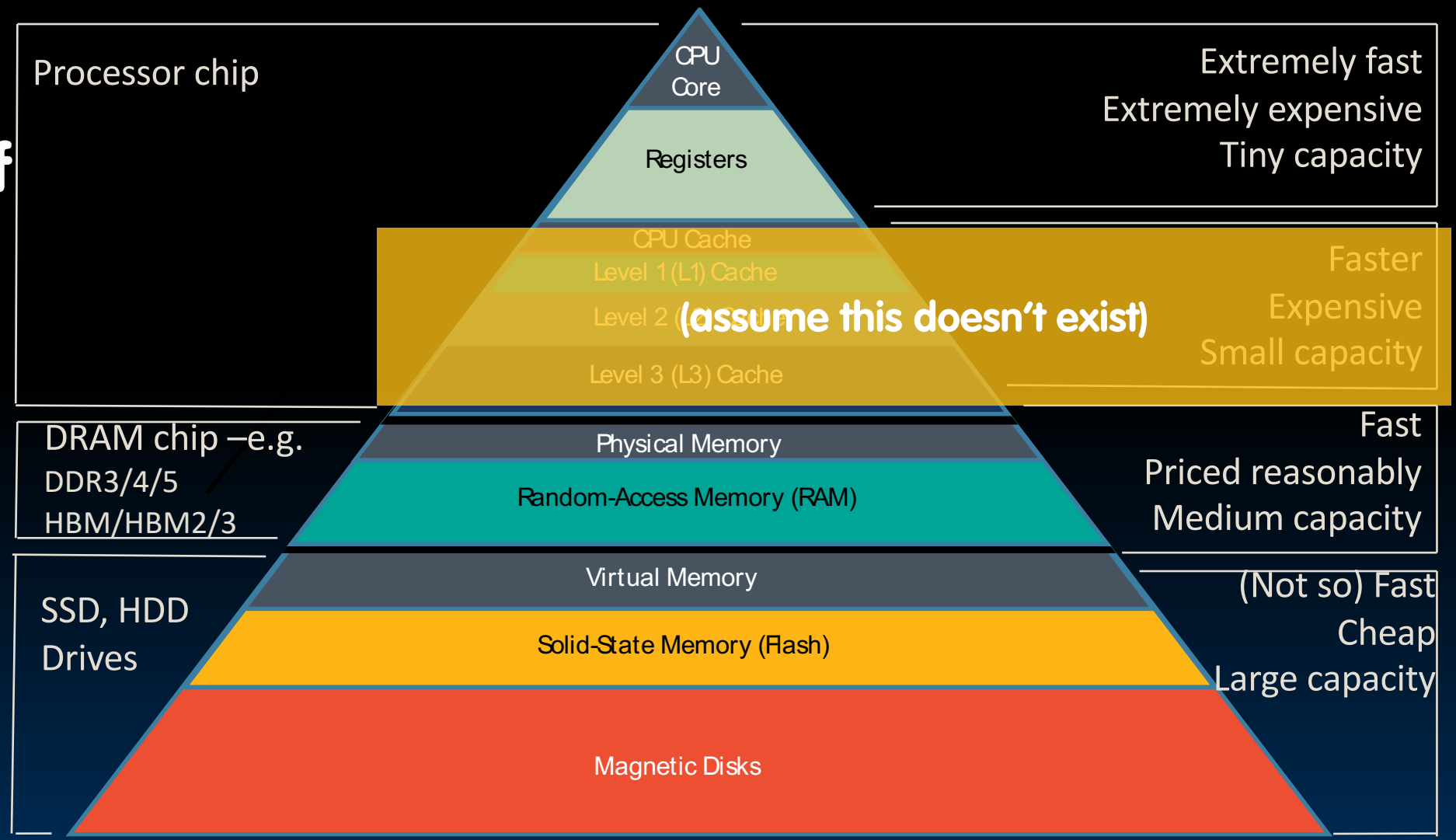
- **Each process has a dedicated page table.**
  - OS keeps track of which process is active.
- Isolation: **Assign processes different pages in DRAM**
  - Prevents (*protects*) a process from accessing other processes' data
  - Page tables managed by OS



⚠ Continue Assuming That Caches Don't Exist ⚠

Virtual Memory **is much easier to understand if we assume** no caches**.**

- We'll reintroduce caches along with *Translation Lookaside Buffers (TLBs)* soon.



Page Table Details II: Page Faults, Write Policy

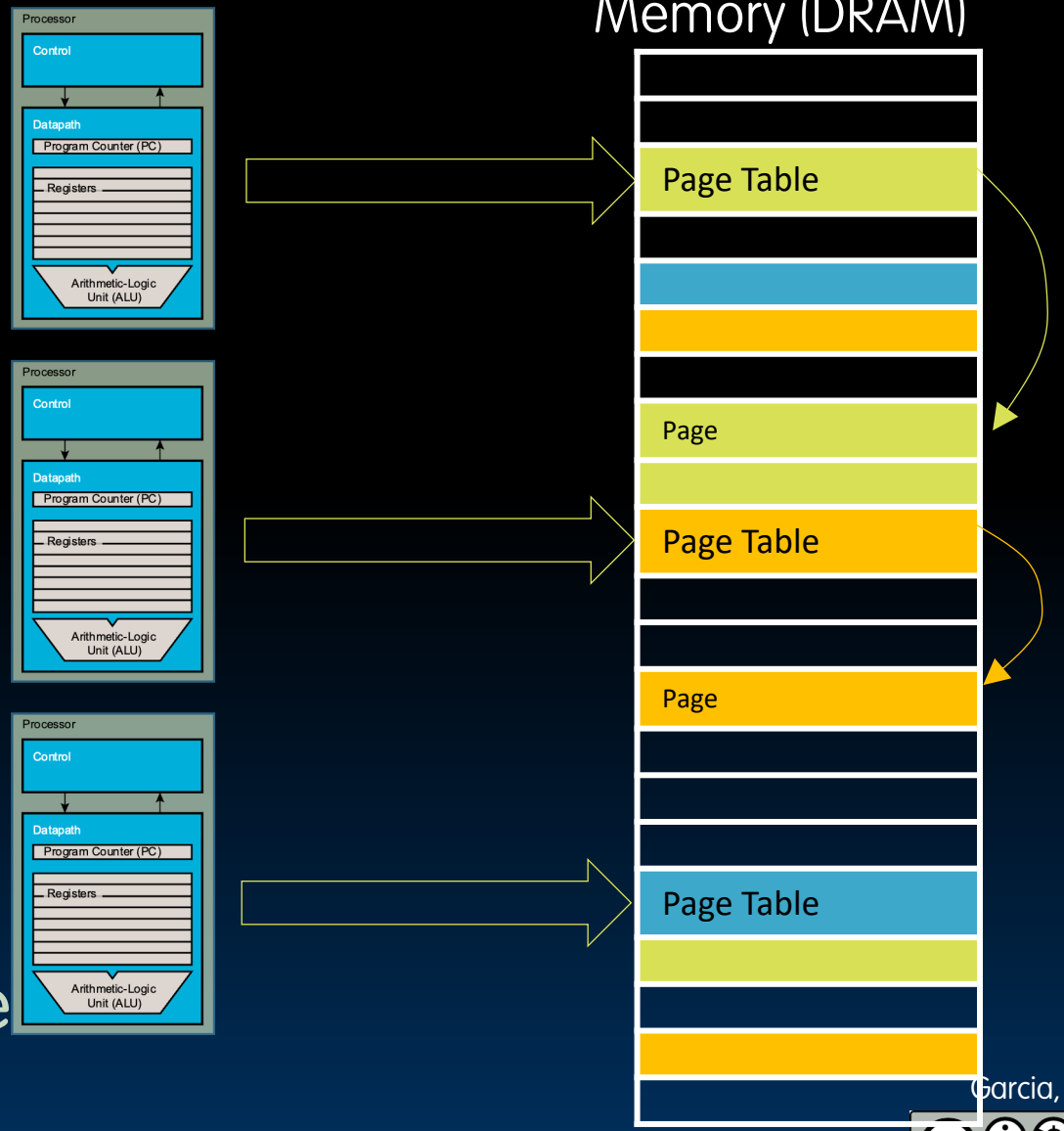
- Page Table Details II: Page Faults, Write Policy
- Hierarchical/Multilevel Page Tables
- OS: Supervisor Mode, Exceptions
- OS: Boot and System Calls

Page Tables Are Stored in Memory (1/2)

- **If 32-Bit virtual address space, 4 GiB DRAM, 4-KiB pages:**
  - # page table entries = # Virtual Page Numbers = 2^{32} / 2^{12} = 2^{20}
  - Suppose each page table entry = 4 B (PPN + status bits).
  - Page Table Size: 4 MiB = 0.1% DRAM. Not bad…
    - But *much* too large for a cache!
- **For now, store page tables in memory (DRAM).**
  - Caveat: *Two* (slow) memory accesses per **`lw/sw`** on cache miss!

Page Tables Are Stored in Memory (2/2)

- **Caveat: `lw/sw` then requires two memory accesses:**
  - Read page table (stored in main memory) to translate to physical address.
  - Read physical page, also in main memory.
- **To minimize the performance penalty:**
  - Transfer blocks (not words) between DRAM and processor cache.
  - Use a cache for frequently used page table entries… (more later, TLB)



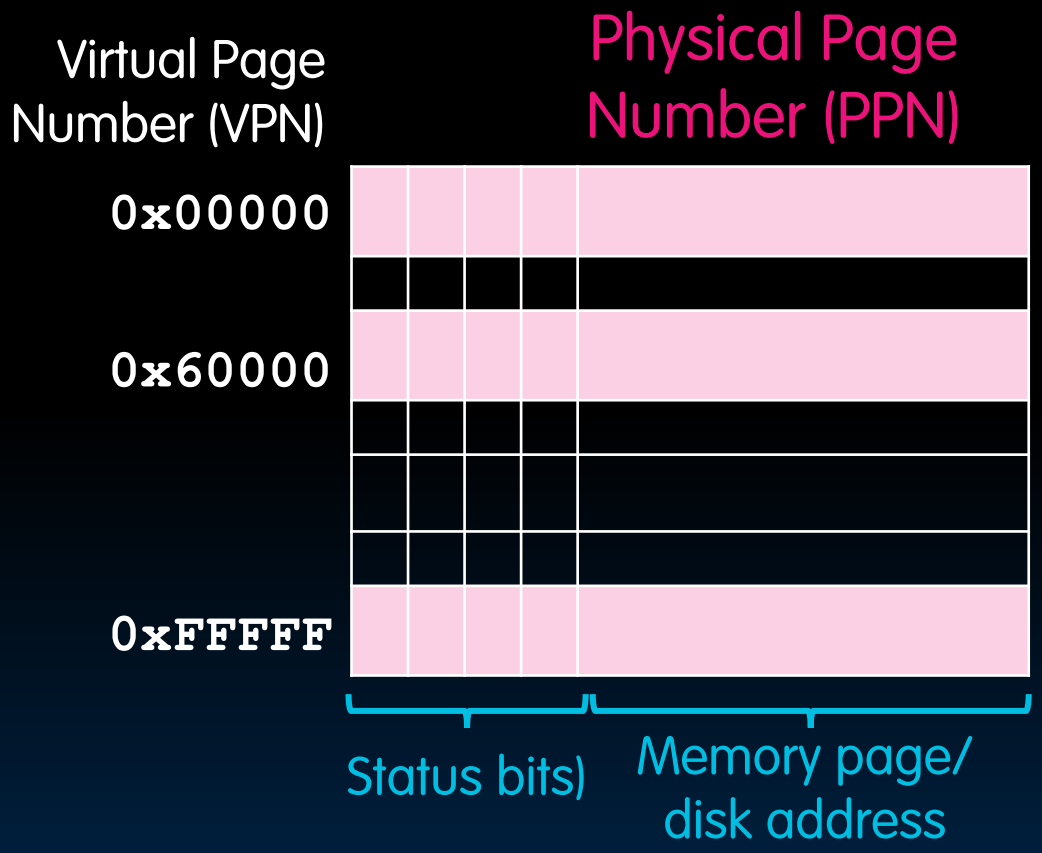
Page Faults

- **Page table entries store status to indicate if the page is in memory (DRAM) or only on disk.**
  - On each memory access, check the page table entry "*valid*" status bit.
- **Valid → In DRAM**
  - Read/write data in DRAM
- **Not Valid → On disk**
  - Triggers a *Page Fault*; OS intervenes to allocate the page into DRAM.
  - If out of memory, first evict a page from DRAM.
  - Store evicted page to disk.
  - Read requested page from disk into DRAM.
  - Finally, read/write data in DRAM.

The *page replacement policy* (e.g., LRU/FIFO/random) is usually done in OS/software; this overheard << disk access time.

Page Table Metadata: Status Bits

- **Write Protection Bit**
  - On: If process writes to page, trigger exception
- **Valid Bit**
  - On: Page is in RAM
- **Dirty Bit**
  - On: Page on RAM is more up-to-date than page on disk



Memory's Write Policy?

- **DRAM acts like a "cache" for disk.**
  - Should writes always go directly to disk (write-through), or
  - Should writes only go to disk when page is evicted (write-back)?
- **Answer: All virtual memory systems use** write-back**.**
  - Disk accesses take too long!

Hierarchical/ Multilevel Page Tables

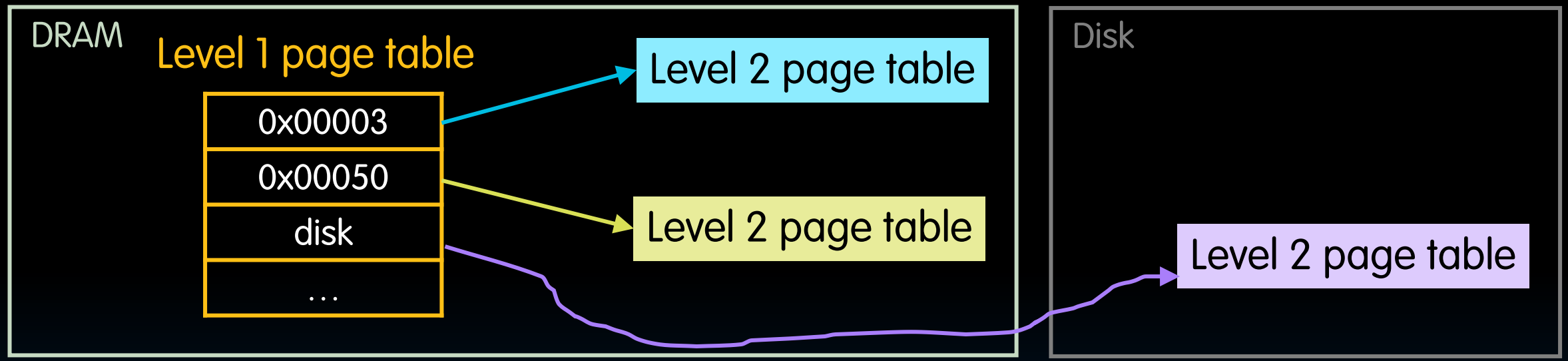
- Page Table Details II: Page Faults, Write Policy
- Hierarchical/Multilevel Page Tables
- OS: Supervisor Mode, Exceptions
- OS: Boot and System Calls

Page Tables Are Stored in Memory!

- **If 32-bit virtual address space, 4 GiB RAM, 4-KiB pages:**
  - # page table entries = # Virtual Page Numbers = $2^{32} / 2^{12} = 2^{20}$
  - Suppose each page table entry = 4 B (PPN + status bits).
  - Page Table Size: 4 MiB → 0.1% RAM. Not bad…
- **…except *each program needs its own page table.***
- **If we have 256 processes:**
  - 256 x 4 MiB = $2^8 \cdot 2^2 \cdot 2^{20}$ = 1 GiB → 25% RAM just for page tables!
- **Complication:** page tables must be in RAM **to be accessed.**
  - Can't swap out *entire* page table to disk…

Enter the Page Table Hierarchy

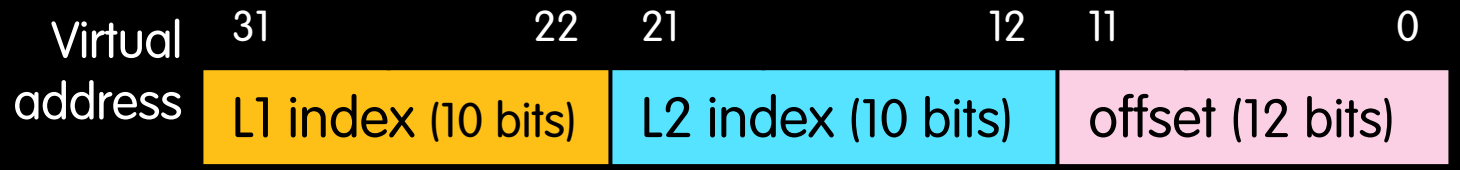
- **What if we page tabled our page tables?**

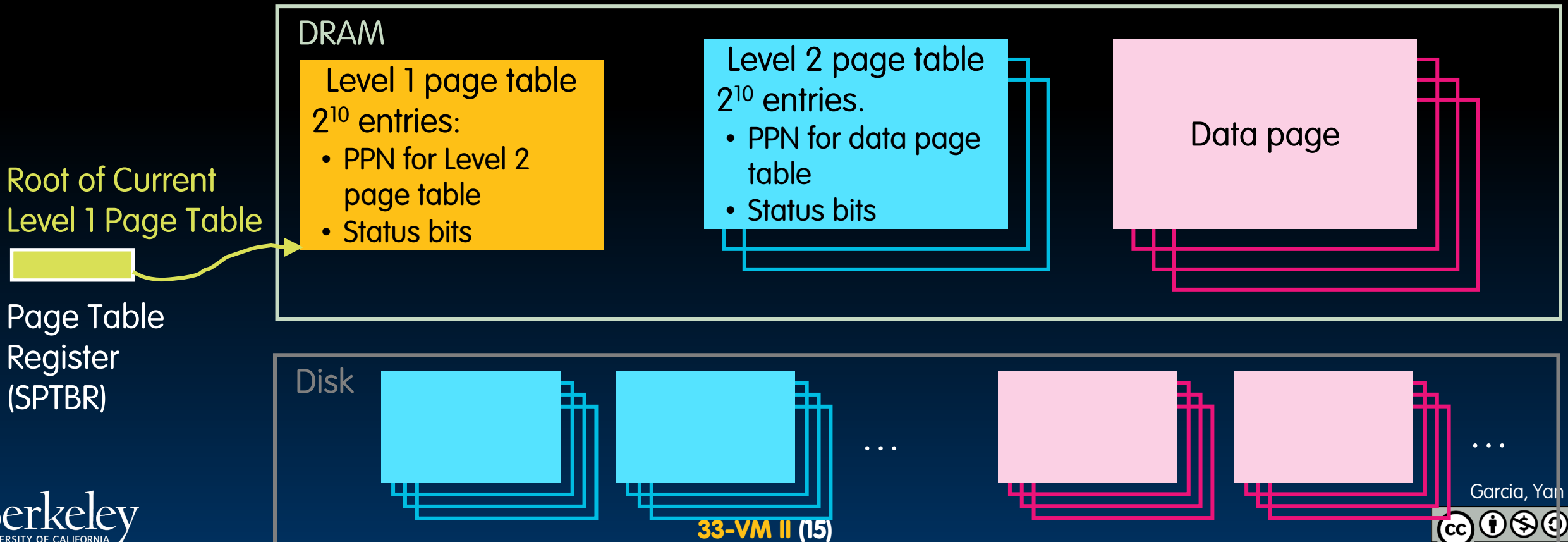


- **Multilevel page tables with decreasing page size:**
  - Key insight: *Sparsity of Virtual Address Space use.*
    - Most program use a fraction of virtual memory, so many page table entries are not accessed.
  - Level 1 page table *always* in DRAM.
  - Level 2 page tables can be in disk; loaded into DRAM via Level 1 access.

Multilevel Page Table Translation

- **32-bit virtual address space, 4 GiB DRAM, 4-KiB pages:**
  - Page table entry size is 4 B for all levels of page tables.
  - RV32I 2-level mapping:





1-Level vs. 2-Level Page Tables

- **32-bit computer (virtual address space), 4 GiB DRAM, 4-KiB pages.**
  - Page table entry size is 4 B for all levels of page tables.
  - Suppose we run 16 processes.

1. **How much RAM is consumed by page tables if we have only one level of page table?**

2. **How much RAM is consumed by Level 1 if we use the two-level hierarchy from the previous slide?**

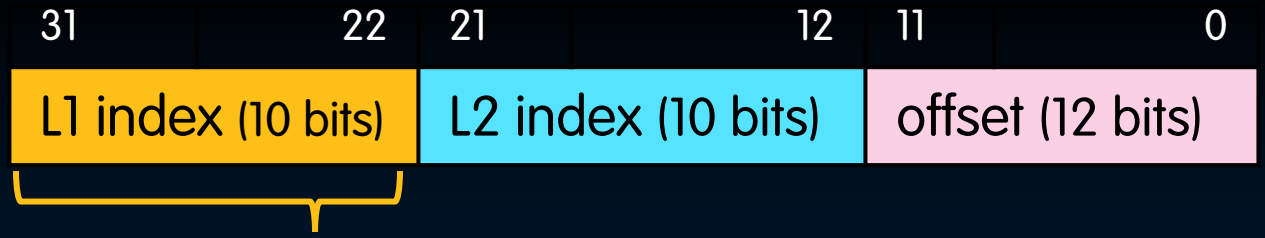
1-Level vs. 2-Level Page Tables

- **32-bit computer (virtual address space), 4 GiB DRAM, 4-KiB pages.**
  - Page table entry size is 4 B for all levels of page tables.
  - Suppose we run 16 processes.

1. How much RAM is consumed by page tables if we have only one level of page table?

- Page offset: log(page size) = log(4KiB) = log(2^{12}) = 12
- # entries in page table = # VPNs = 2^{32} / 2^{12} = 2^{20}
- Page table size = 2^{20} x (4 B) = 2^{22} B
- Total RAM consumed = (16 processes) x 2^{22} B = 64 MiB

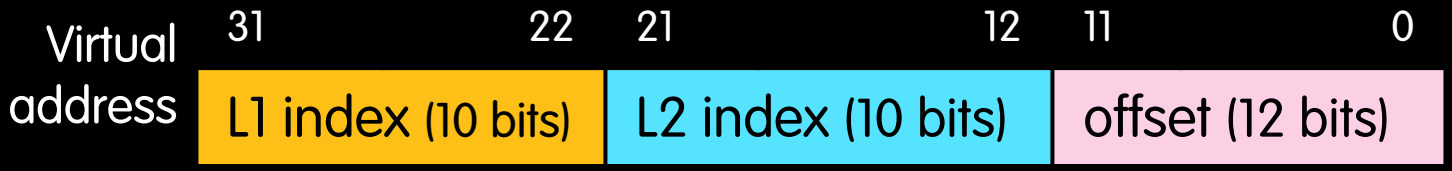
2. How much RAM is consumed by Level 1 if we use the two-level hierarchy from the previous slide?



- # entries in Level 1 PT (= # Level 2 PTs) = 2^{10}
- Page table size = 2^{10} x (4 B) = 2^{12} B
- Total RAM consumed = (16 processes) x 2^{12} B = 64 KiB

Multilevel Page Table Walk

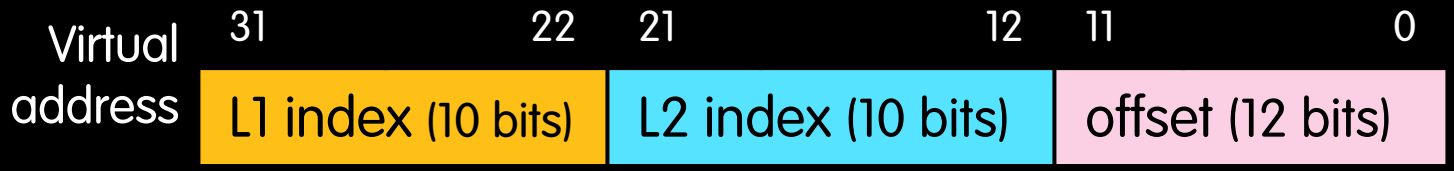
- **32-bit virtual address space, 4 GiB DRAM, 4-KiB pages:**
  - RV32I 2-level mapping:



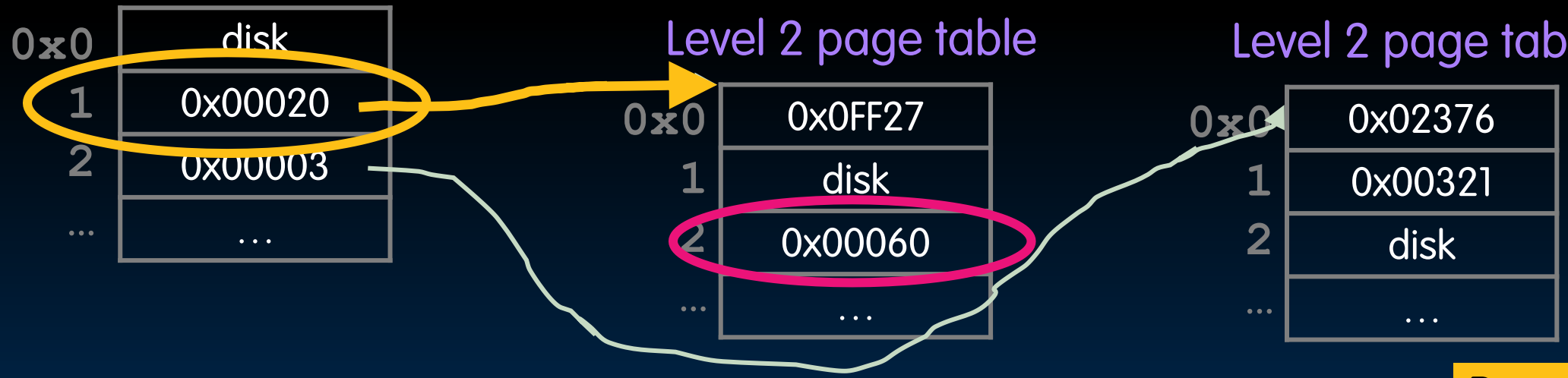
Given the page tables below, how to translate virtual address `0x00402450` to a physical address?

Multilevel Page Table Walk

- **32-bit virtual address space, 4 GiB DRAM, 4-KiB pages:**
  - RV32I 2-level mapping:



Given the page tables below, how to translate virtual address `0x00402450` to a physical address?



Page offset does not change!

(dramatic pause)

Q: How does the OS *manage* Virtual Memory (e.g., page faults)?

A: Exceptions!

OS: Supervisor Mode, Exceptions

- Page Table Details II: Page Faults, Write Policy
- Hierarchical/Multilevel Page Tables
- OS: Supervisor Mode, Exceptions
- OS: Boot and System Calls

Supervisor Mode vs. User Mode

- **If an application goes wrong (or rogue, e.g., malware), it could crash the entire machine!**
- **CPUs have a hardware supervisor mode (i.e., kernel mode).**
  - Set by a status bit in a special register.
  - An OS process in supervisor mode helps enforce constraints to other processes, e.g., access to memory, devices, etc.
  - Supervisor mode is a bit like "superuser"…
    - Errors in supervisory mode are often catastrophic (blue "screen of death", or "I just corrupted your disk").
- **By contrast, in user mode, a process can only access a subset of instructions and (physical) memory.**
  - Can change *out* of supervisor mode using a special instruction (e.g. `sret`).
  - Cannot change *into* supervisor mode directly; instead, HW interrupt/exception.
  - The OS mostly runs in user mode! Supervisor mode is used sparingly.

Exceptions and Interrupts

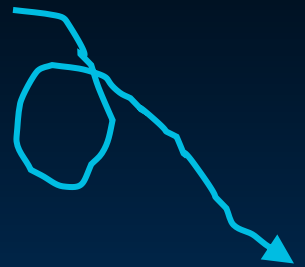
- **Exceptions**
  - Caused by an event *during* the execution of the current program.
  - *Synchronous*; must be handled immediately.
  - Examples:
    - Illegal instruction
    - Divide by zero
    - *Page fault*
    - Write protection violation

- **Interrupts** (more later)
  - Caused by an event *external* to the current running program.
  - *Asynchronous* to current program; does not need to be handled immediately (but should be soon).
  - Examples:
    - Key press
    - Disk I/O

Traps Handle Exceptions/Interrupts

- **The trap handler is code that services interrupts/exceptions.**

1. **Complete all instructions before the faulting instruction.**
2. **Flush all instructions after the faulting instruction.**
   - Like pipeline hazard: convert to noops/"bubbles."
   - Also flush faulting instruction.
3. **Transfer execution to trap handler (runs in supervisor mode).**
   - Optionally return to original program and re-execute instruction.

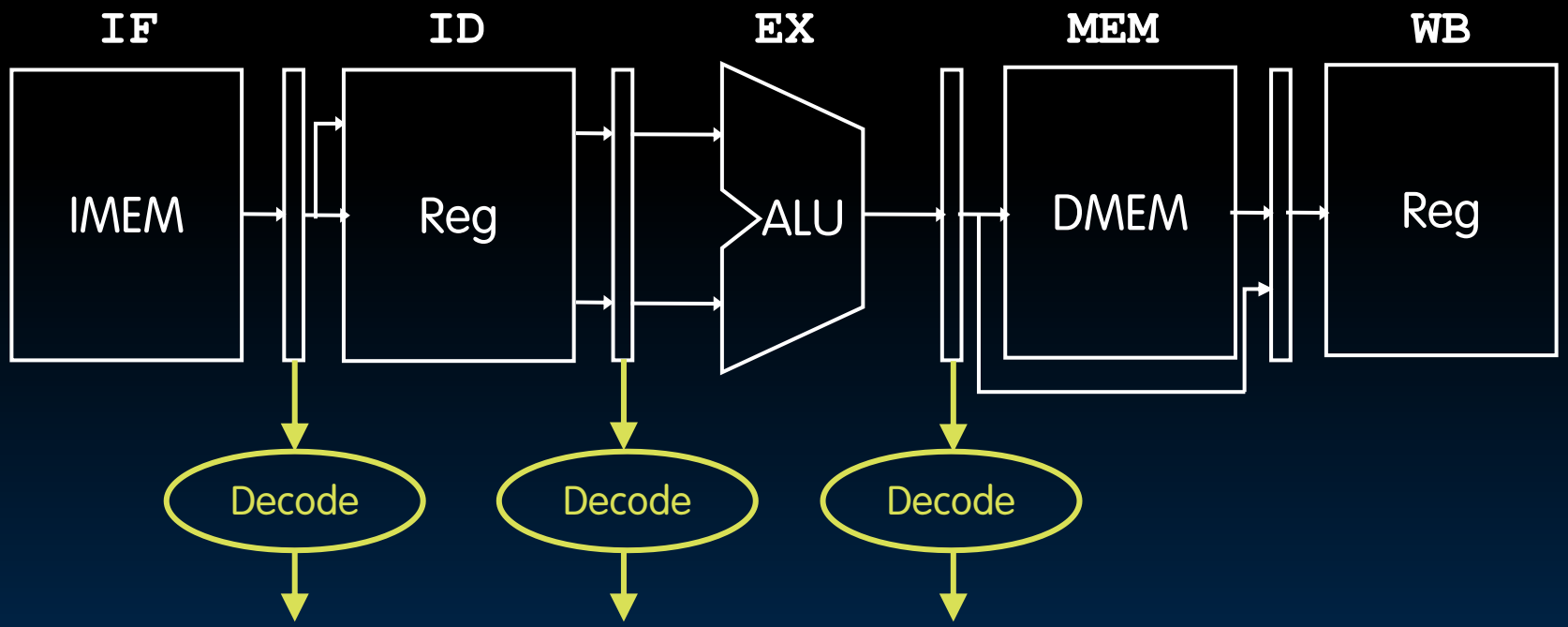


If the trap handler returns, then from the program's point of view it must look like nothing has happened!

Exceptions in a 5-Stage Pipeline

(for next time)

- Traps are handled similarly to pipeline hazards.
- In RISC-V, the exception cause can be inferred by the faulting instruction and its current pipeline stage.



(for example)

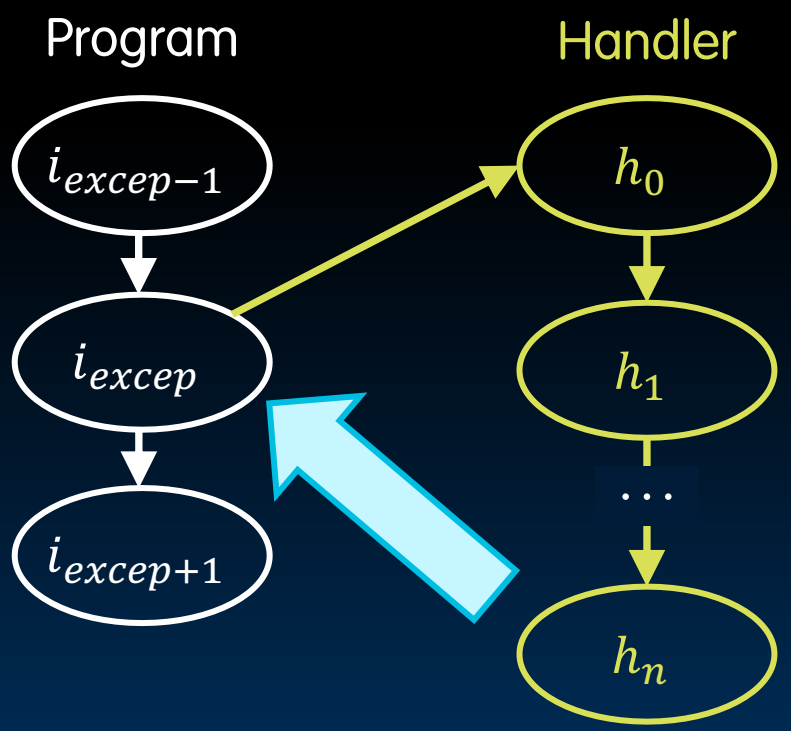
The Trap Handler

(for next time)

1. **Save the state of the current program.**
   - Save ALL of the registers!
2. **Determine what caused the exception/interrupt.**
3. **Handle exception/interrupt, then do one of two things:**

Continue execution of the program:

4. **Restore program state.**
5. **Return control to the program.**



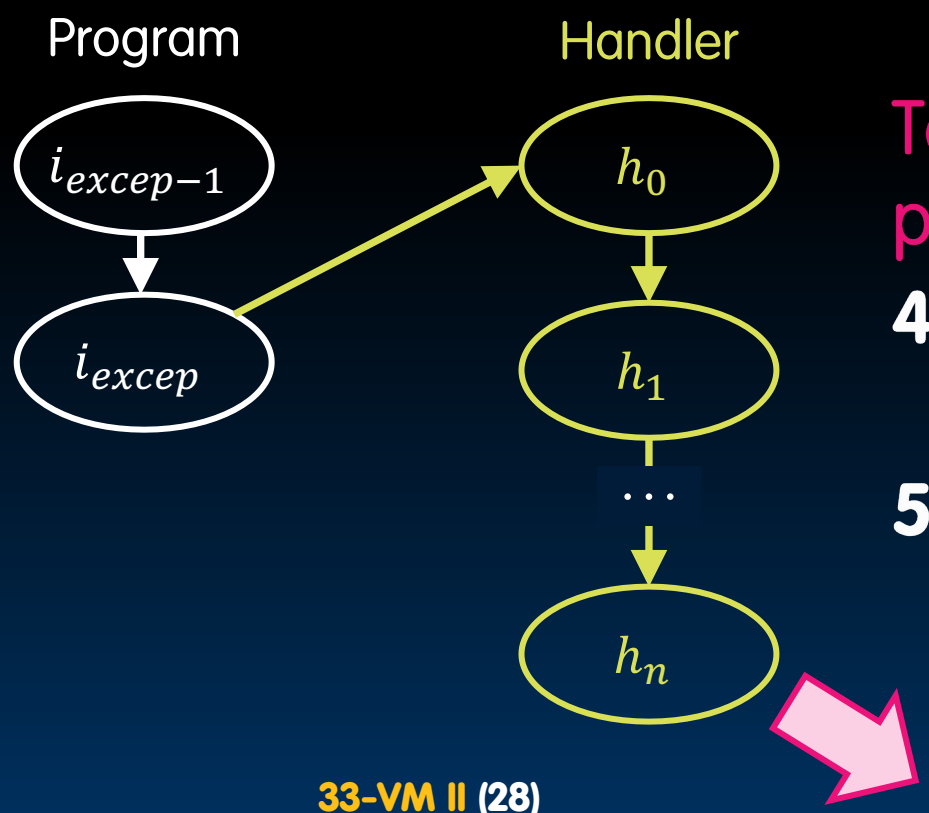
The Trap Handler

(for next time)

1. **Save the state of the current program.**
   - Save ALL of the registers!
2. **Determine what caused the exception/interrupt.**
3. **Handle exception/interrupt, then do one of two things:**

Continue execution of the program:

4. **Restore program state.**
5. **Return control to the program.**



Terminate the program:

4. **Free the program resources, etc.**
5. **Schedule a new program.**

Handling Context Switches

(for next time)

- **Recall the context switch:**
  - OS switches between processes by changing the internal state of the processor.
  - Allows a single processor to "simultaneously" run many programs.
- **At a high-level:**
  - The OS sets a timer. When it expires, perform a *hardware interrupt*.
  - Trap handler saves all register values, including:
    - Program Counter (PC)
    - *Page Table Register* (SPTBR in RV32I)
      - The memory *address* of the active process's page table.
  - Trap handler then loads in the next process's registers and returns to user mode.

Handling Page Faults

(for next time)

- **Recall** page faults:
  - An accessed page table entry has valid bit off → data is not in DRAM.
- **Page faults are handled by the trap handler.**
  - The *page fault exception handler* initiates transfers to/from disk and performs any page table updates.
  - (If pages needs to be swapped from disk, perform *context switch* so that another process can use the CPU in the meantime.
    - (ideally need a "precise trap" so that resuming a process is easy.)
  - Following the page fault, *re-execute the instruction*.
- **Side note: Write protection violations also trigger exceptions.**

(for next time)

Recorded if we don't get to it:
https://youtu.be/1mZ-ztcwbZk

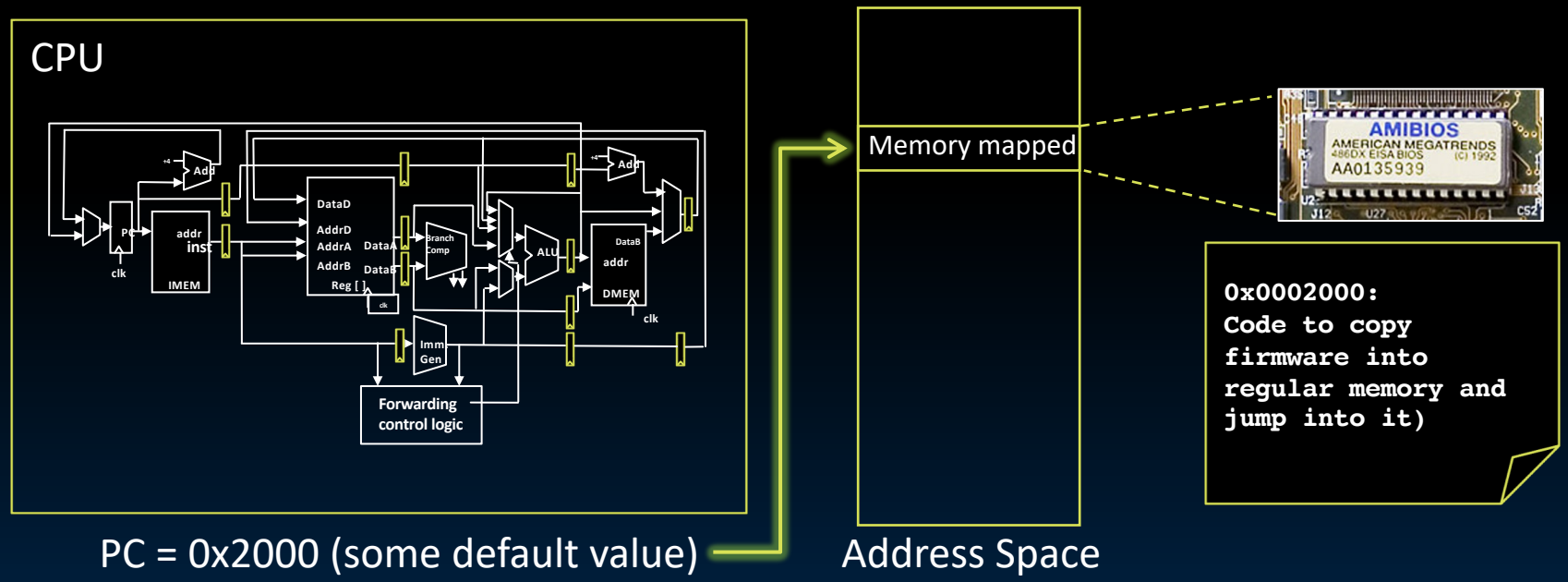
OS: Boot and System Calls

- Page Table Details II: Page Faults, Write Policy
- Hierarchical/Multilevel Page Tables
- OS: Supervisor Mode, Exceptions
- OS: Boot and System Calls

What Happens at Boot? (1/2)

(for next time)

- **When the computer switches on, it does the same as Venus:**
  - The CPU executes instructions from some start address stored in Flash ROM (Read-Only Memory).



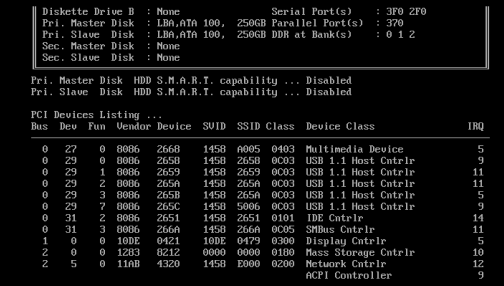
What Happens at Boot? (2/2)

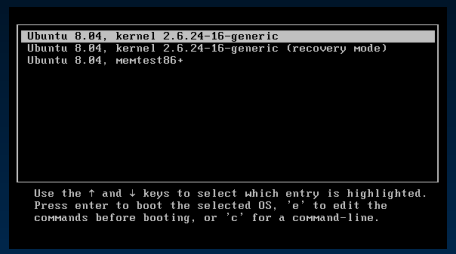
(for next time)

- **Then, the BIOS (Basic Input Output System) firmware loads the bootloader, which loads the OS kernel.**

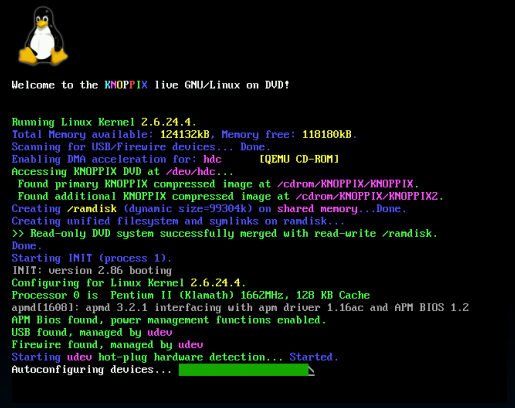
1. *BIOS*: Find a storage device and load the first sector (block of data).

2. *Bootloader*: (stored on, e.g., disk) Load the OS kernel from disk into a location in memory and jump into it.





3. *OS Boot*: Initialize services, drivers, etc.



4. *Init*: Launch an application (e.g., Terminal/Desktop/…) that waits for input in loop.

System Calls and Launching Applications (for next time)

- **A system call (syscall) is a "software interrupt" that allows a program to request a service from the operating system.**
  - Similar to a function call, except now executed by kernel.
  - Examples:
    - Creating and deleting files; reading/writing files;
    - Accessing external devices (e.g., scanner);
    - `printf`, `malloc`, etc. (`ecall`s in RISC-V); etc.
    - Launch a new process
- **Suppose shell (a user process) wants to launch a new app:**
  - Shell *forks* (in Linux): a *syscall* that traps into the OS kernel process
  - OS (supervisor mode): Load program (see CALL); jump to start of `main`. Return to user mode.
  - Shell: "wait" for `main` to return (*join*)