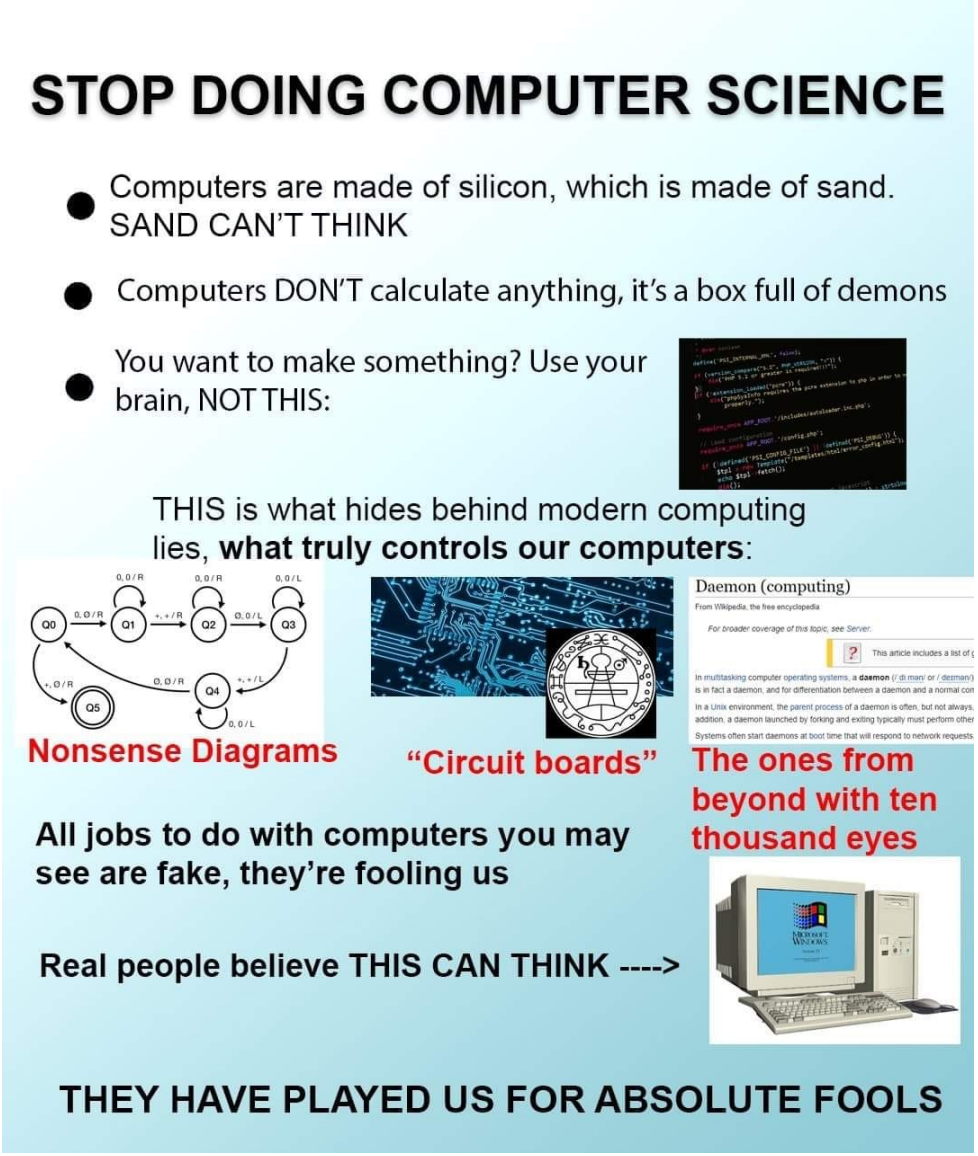
# Virtual Memory I

CSE 351 Winter 2022

**Instructor:**
Sam Wolfson

**Teaching Assistants:**
Angela Xu
Anirudh Kumar
Catherine Guevara
Dara Stotland
Harrison Bay
Ian Hsiao
Kevin Wang
Mara Kirdani-Ryan
Nick Durand
Sanjana Sridhar



# Relevant Course Information

- Midterm grades posted
  - Regrade requests will be open starting this afternoon, and open for a week
  - Solutions posted on course website (Exams page)
    - Please check your answer against the solutions before submitting a regrade request!
- Lab 3 grading in progress
- Lab 4 due next Monday (2/28)

# Processes

- Processes and context switching
- Creating new processes
  - `fork()` and `exec*()`
- **Ending a process**
  - **`exit()`, `wait()`, `waitpid()`**
  - **Zombies**

# `exit`: Ending a process

- **`void`** `exit(`**`int`** `status)`
  - Explicitly exits a process
    - Status code: 0 is used for a normal exit, nonzero for abnormal exit
- The `return` statement from `main()` also ends a process in C
  - The return value is the status code

equivalent

# Zombies! 

This is non-testable material

- A terminated process still consumes system resources
  - Various tables maintained by OS
  - Called a "zombie" (a living corpse, half alive and half dead)
- *Reaping* is performed by parent on terminated child
  - Parent is given exit status information and kernel then deletes zombie child process
  - In long-running processes (*e.g.*, shells, servers) we need *explicit* reaping (more on this in CSE 333)
- If parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (pid 1)
  - **Note:** on recent Linux systems, `init` has been renamed `systemd`

# `wait`: Synchronizing with Children

This is non-testable material

- **`int`** `wait(`**`int*`** `child_status)`
  - Suspends current process (*i.e.*, the parent) until one of its children terminates
  - Return value is the PID of the child process that terminated
    - *On successful return, the child process is reaped*
  - If `child_status != NULL`, then the `*child_status` value indicates why the child process terminated
    - Special macros for interpreting this status – see **`man wait(2)`**
- **Note:** If parent process has multiple children, `wait` will return when *any* of the children terminates
  - `waitpid` can be used to wait on a specific child process

# Process Management Summary

- `fork` makes two copies of the same process (parent & child)
  - Returns different values to the two processes
- `exec*` replaces current process from file (new program)
  - Two-process program:
    - First `fork()`
    - **if** (pid == 0) { */* child code */* } **else** { */* parent code */* }
  - Two different programs:
    - First `fork()`
    - **if** (pid == 0) { execv(…) } **else** { */* parent code */* }

- `exit` or `return` from `main` to end a process
- `wait` or `waitpid` used to synchronize parent/child execution and to reap child process

# Reading Review

- Terminology:
  - Paging: page size ($P$), page offset width ($p$) virtual page number (VPN), physical page numbers (PPN)
  - Page table (PT): page table entry (PTE), access rights (read, write, execute)

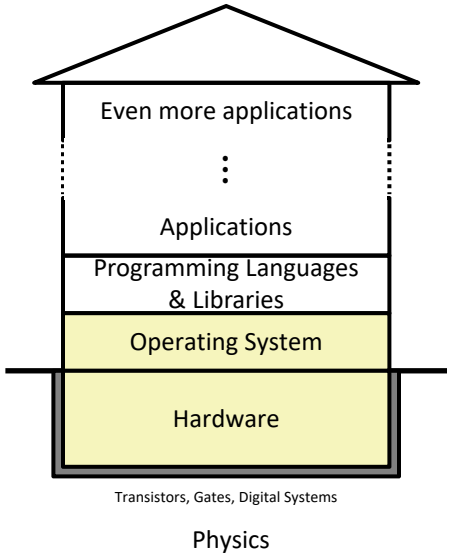
- Questions from the Reading?

# Review Questions

- Which terms from caching are most similar/analogous to the new virtual memory terms?
  - page size – block size
  - page offset width – block offset width
  - virtual page number – block number
  - physical page number – block number/cache set
  - page table entry – cache line (data + management bits)
  - access rights – management bits

# The Hardware/Software Interface

- Topic Group 3: **Scale & Coherence**
  - Caches, Processes, **Virtual Memory**, Memory Allocation




- How do we maintain logical consistency in the face of more data and more processes?
  - How do we support control flow both within many processes and things external to the computer?
  - How do we support data access, including dynamic requests, across multiple processes?

# Virtual Memory (VM*)

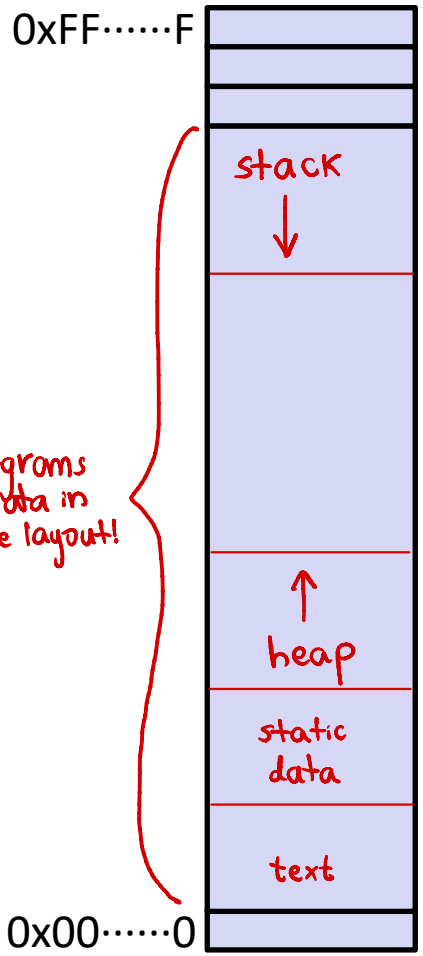
- **Overview and motivation**
- **VM as a tool for caching**
- Address translation
- VM as a tool for memory management
- VM as a tool for memory protection

**Not to be confused with "Virtual Machine" which is a whole nother thing.*

# Memory as we know it so far… is *virtual!*

- Programs refer to virtual memory addresses
  - `movq (%rdi),%rax` Virtual address in register
  - Conceptually memory is just a very large array of bytes
  - System provides private address space to each process
- Allocation: Compiler and run-time system
  - Where different program objects should be stored
  - All allocation within single virtual address space

all programs store data in the same layout!

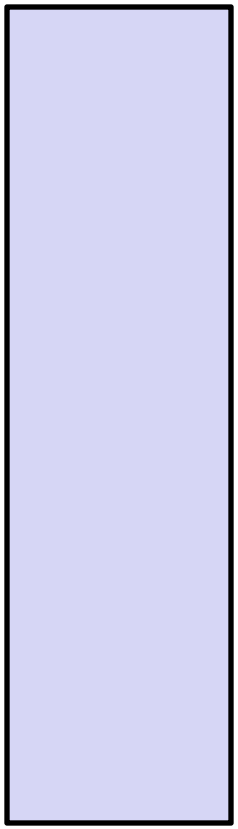



- But…
  - We *probably* don't have $2^w$ bytes of physical memory
  - We *certainly* don't have $2^w$ bytes of physical memory ***for every process*** :o
  - Processes should not interfere with one another
    - Except in certain cases where they want to share code or data

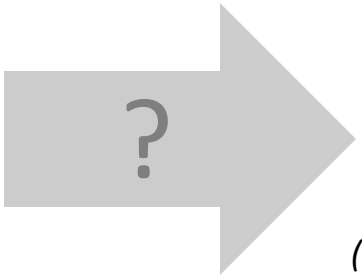
# Problem 1: How Does Everything Fit?

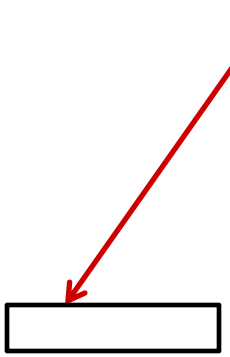
64-bit virtual addresses can address several exabytes (18,446,744,073,709,551,616 bytes)

Physical main memory offers a few gigabytes (*e.g.*, 8,589,934,592 bytes)



~ 18 exabytes!

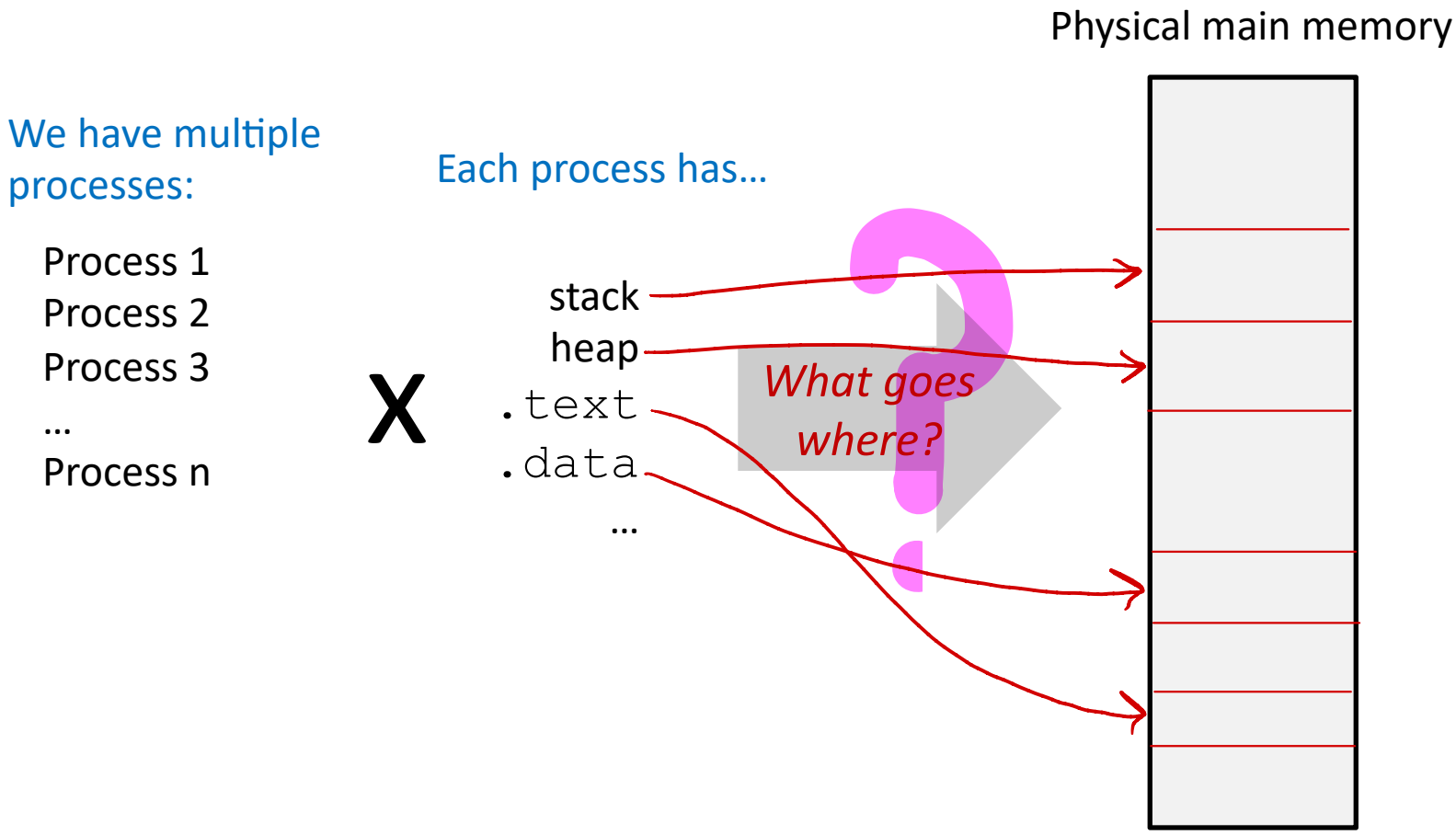


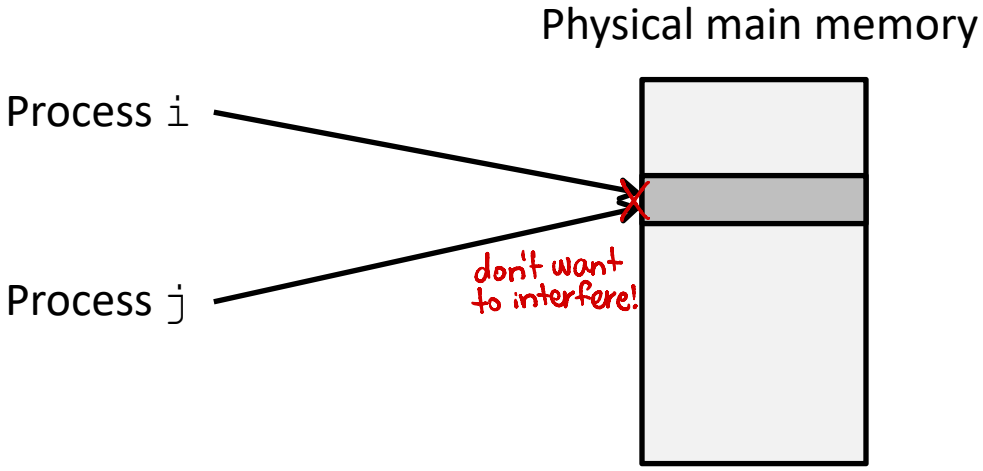
*(Not to scale; physical memory would be smaller than the period at the end of this sentence compared to the virtual address space)*

1 virtual address space per process, with many processes…

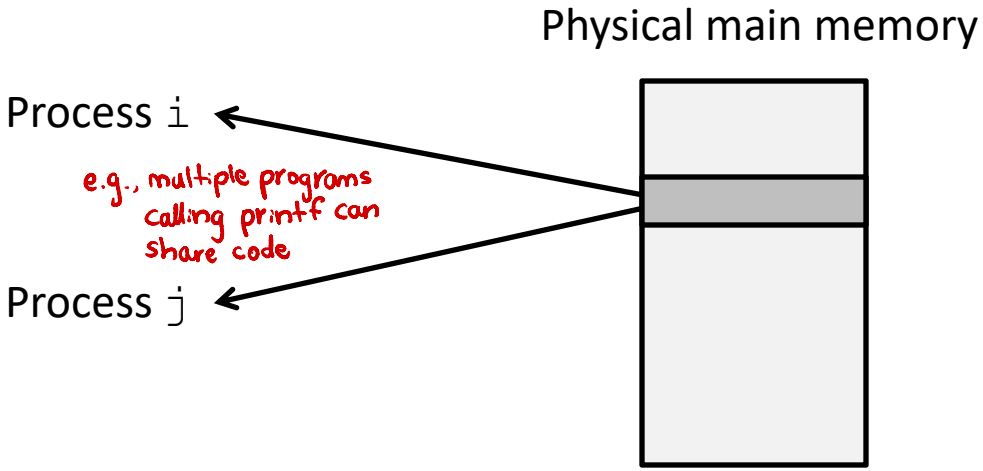
# Problem 2: Memory Management



# Problem 3: How To Protect



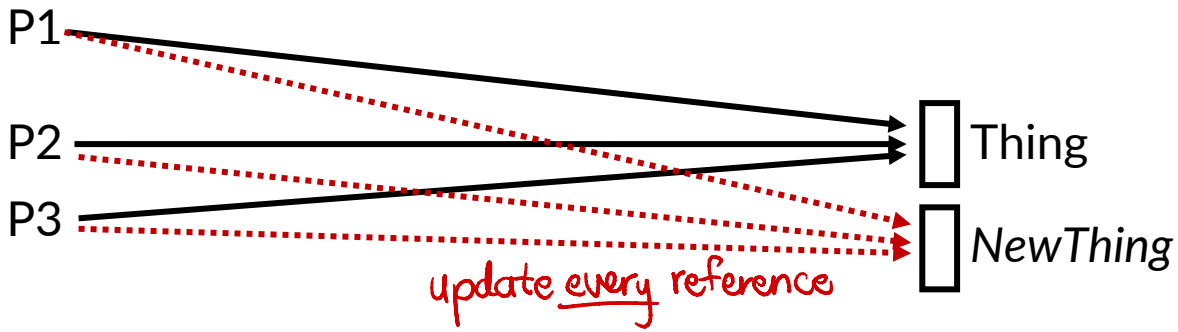

# Problem 4: How To Share?



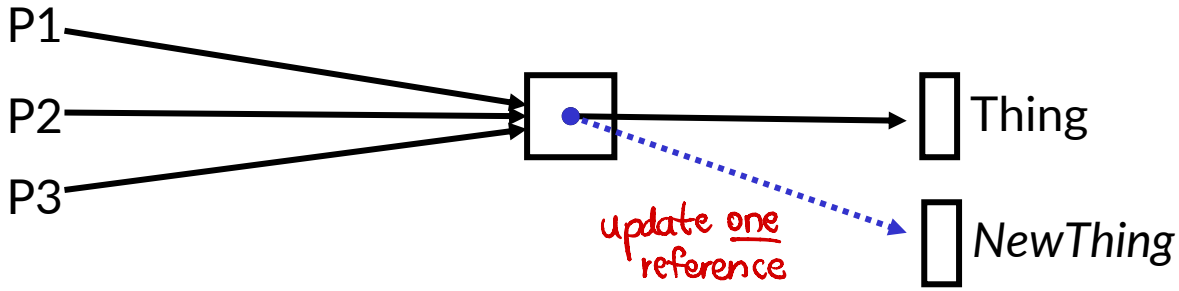
# How can we solve these problems?

- "Any problem in computer science can be solved by adding another level of **indirection**." – *David Wheeler, inventor of the subroutine*

- Without Indirection



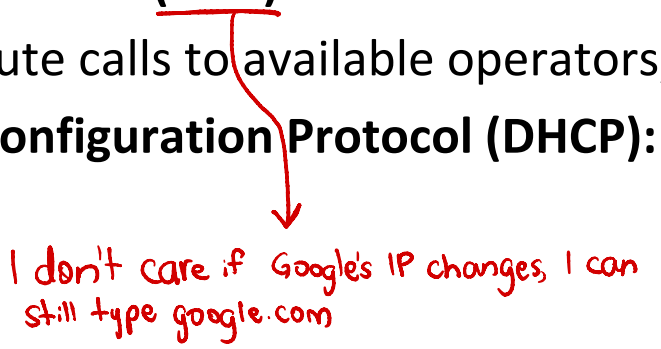

- With Indirection



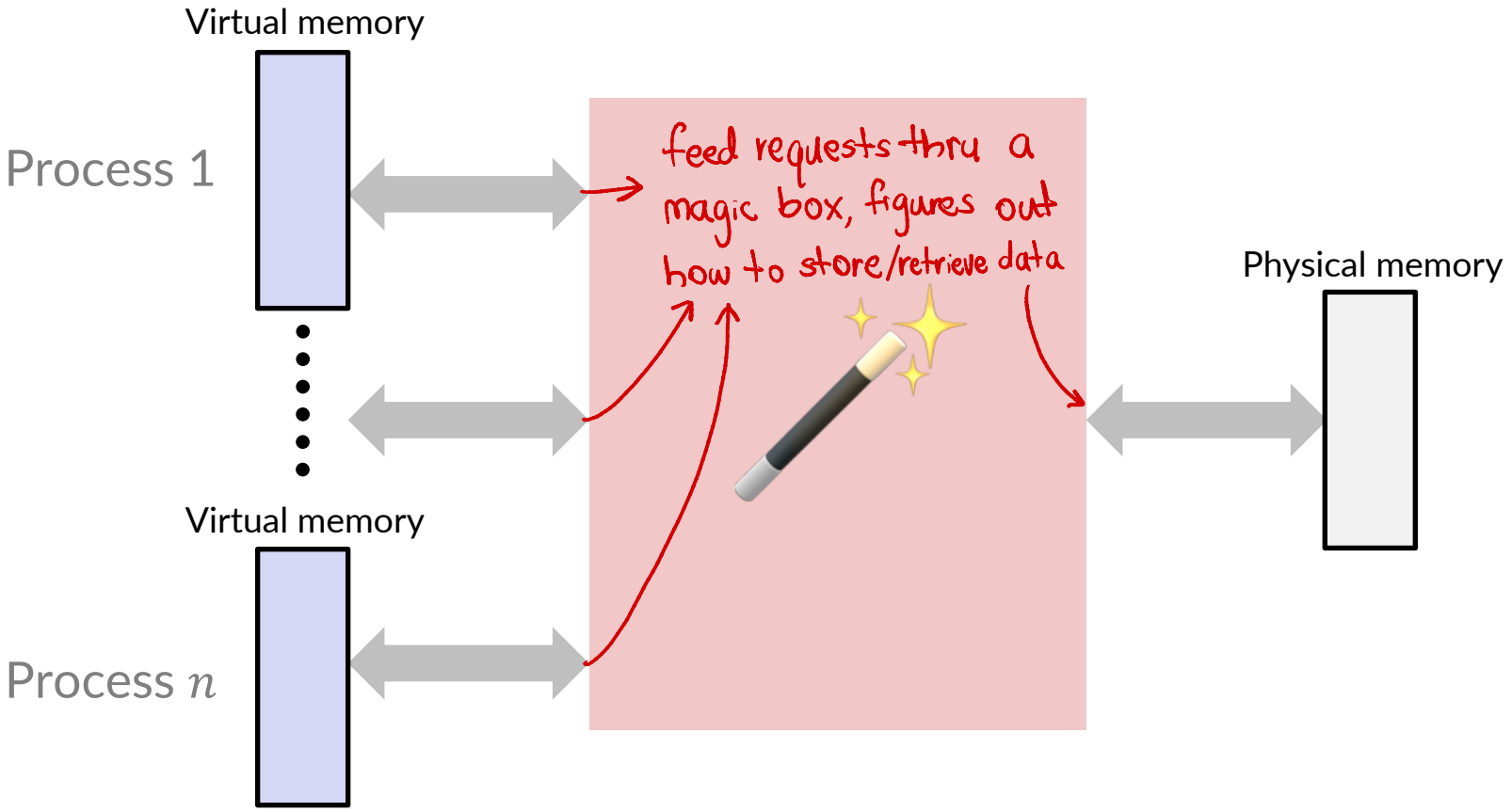

*What if I want to move Thing?*

# Indirection

- *Indirection*: The ability to reference something using a name, reference, or container instead of the value itself. A flexible mapping between a name and a thing allows changing the thing without notifying holders of the name.
  - Adds some work (now must look up 2 things instead of 1)
  - But don't have to track all uses of name/address (single source!)
- Examples:
  - **Phone system:** cell phone number portability
  - **Domain Name Service (DNS):** translation from name to IP address
  - **Call centers:** route calls to available operators, etc.
  - **Dynamic Host Configuration Protocol (DHCP):** local network address assignment



# Indirection in Virtual Memory




- Each process gets its own private virtual address space
- Solves the previous problems!

# Address Spaces

- Virtual address space: Set of $N = 2^n$ virtual addr
  - {0, 1, 2, 3, …, N-1}

typically, N >> M

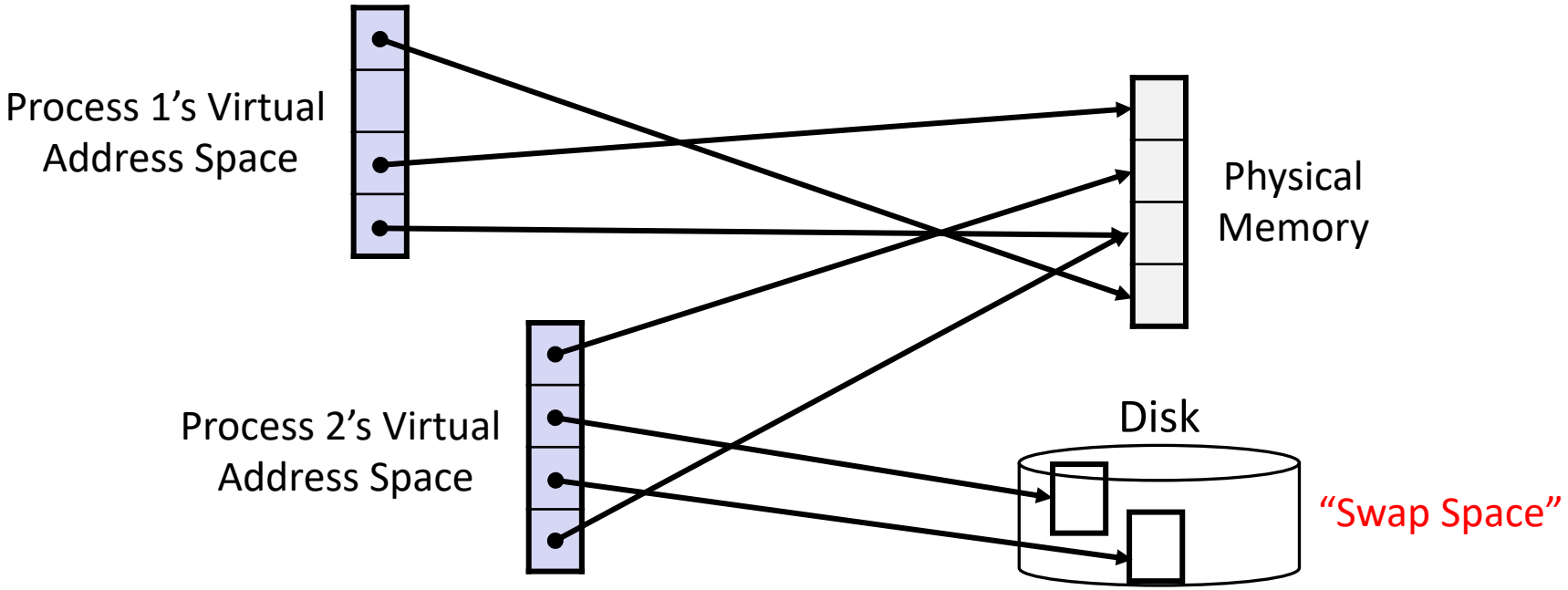
- Physical address space: Set of $M = 2^m$ physical addr
  - {0, 1, 2, 3, …, M-1}

- Every byte in main memory has:
  - one physical address (PA)
  - zero, one, *or more* virtual addresses (VAs)

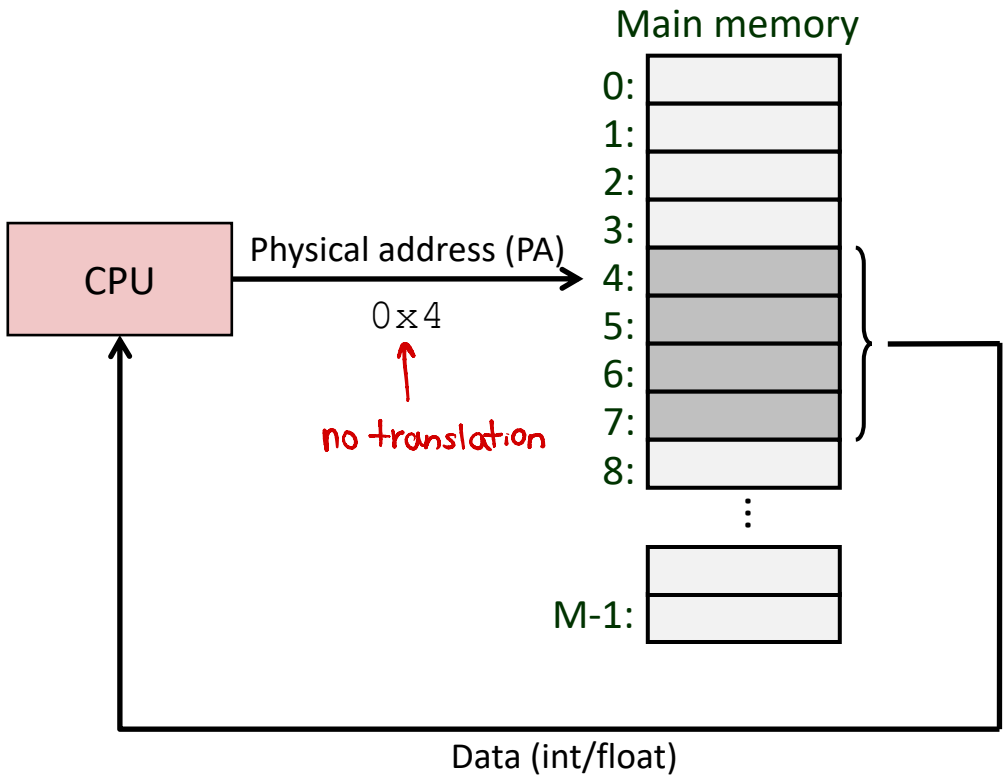
unused — one process — multiple processes

# Mapping

- A virtual address (VA) can be mapped to either physical memory or disk
  - store less-used data on disk!
  - ★ memory now a cache for disk
  - Unused VAs may not have a mapping
  - VAs from *different* processes may map to same location in memory/disk

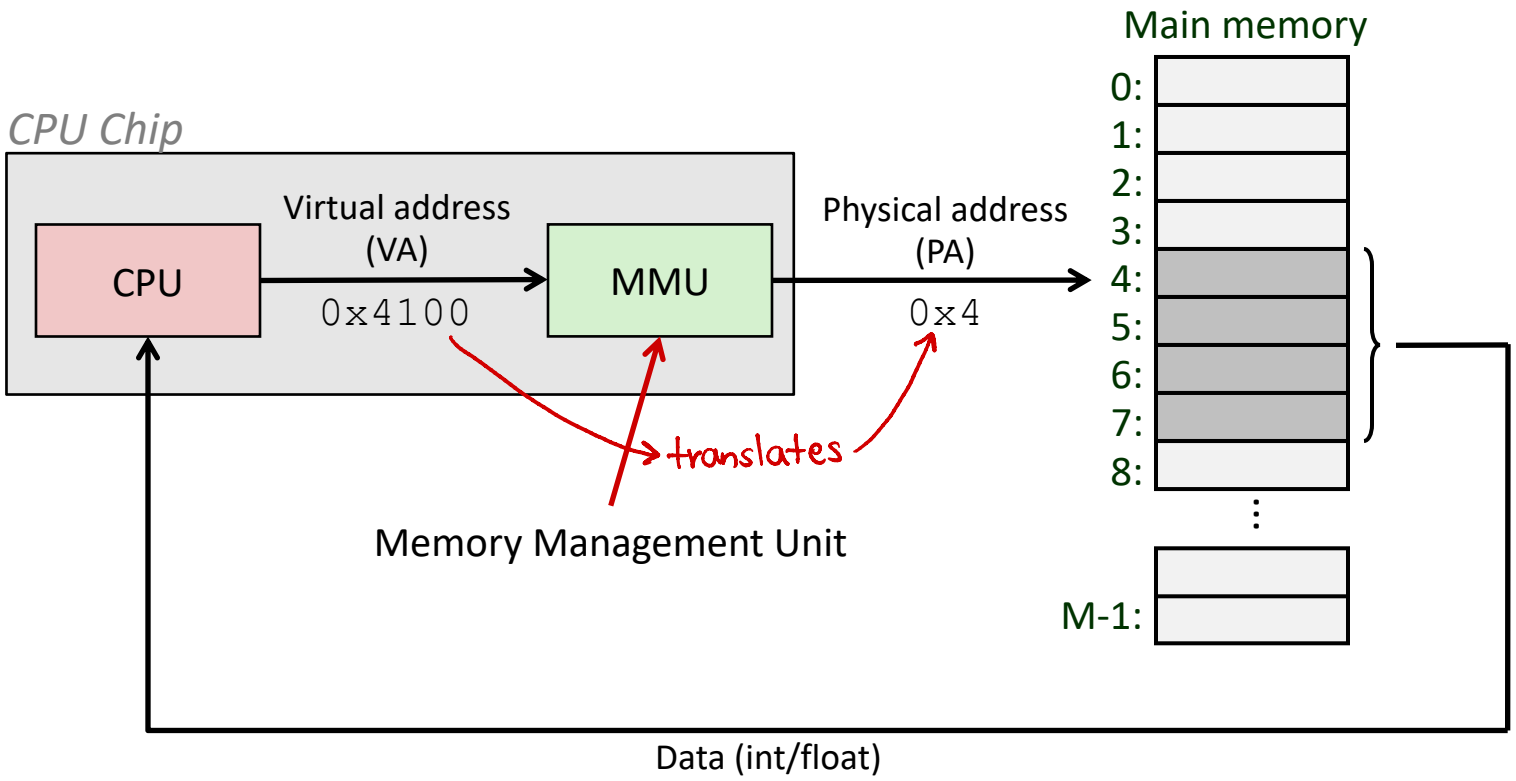


# A System Using Physical Addressing




- Used in "simple" systems with (usually) just one process:
  - Embedded microcontrollers in devices like cars, elevators, and digital picture frames

# A System Using Virtual Addressing



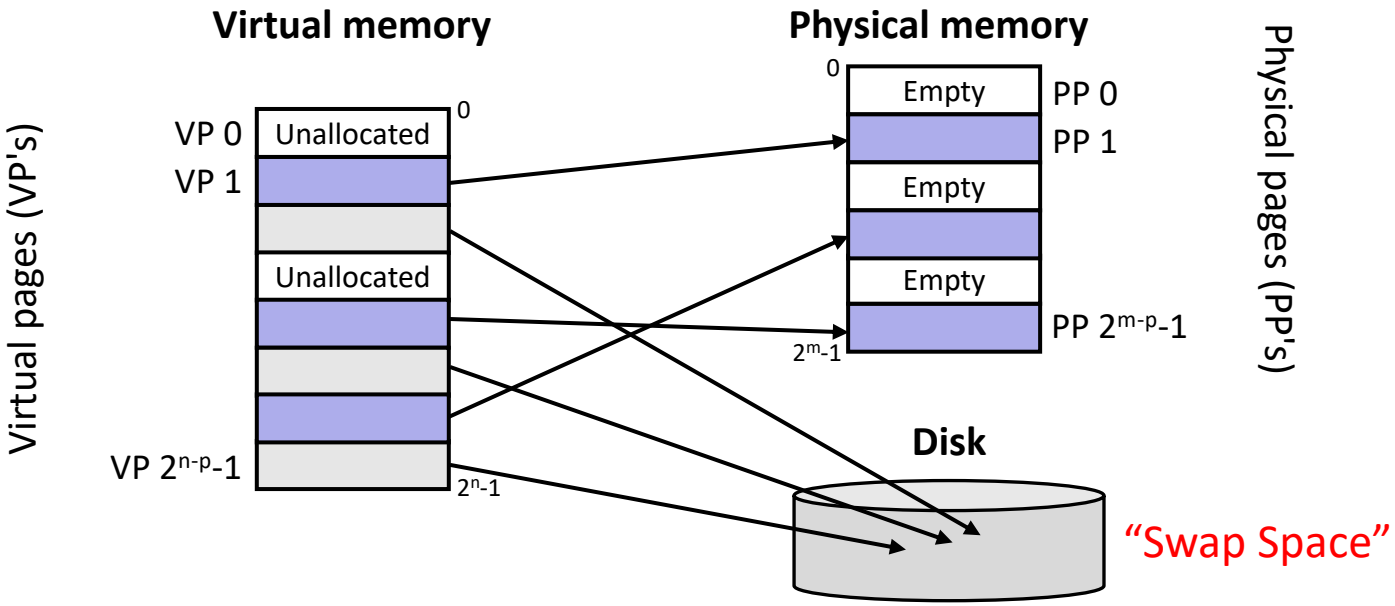

- Physical addresses are *completely invisible* *to programs*
  - Used in all modern desktops, laptops, servers, smartphones…

# Why Virtual Memory (VM)?

- Efficient use of limited main memory (RAM)
  - Use RAM as a cache for the parts of a virtual address space
    - Some non-cached parts stored on disk
    - Some (unallocated) non-cached parts stored nowhere
  - Keep only active areas of virtual address space in memory
    - Transfer data back and forth as needed
- Simplifies memory management for programmers
  - Each process "gets" the same full, private linear address space
- Isolates address spaces (protection)
  - One process can't interfere with another's memory
    - They operate in *different address spaces*
  - User process cannot access privileged information
    - Different sections of address spaces have different permissions

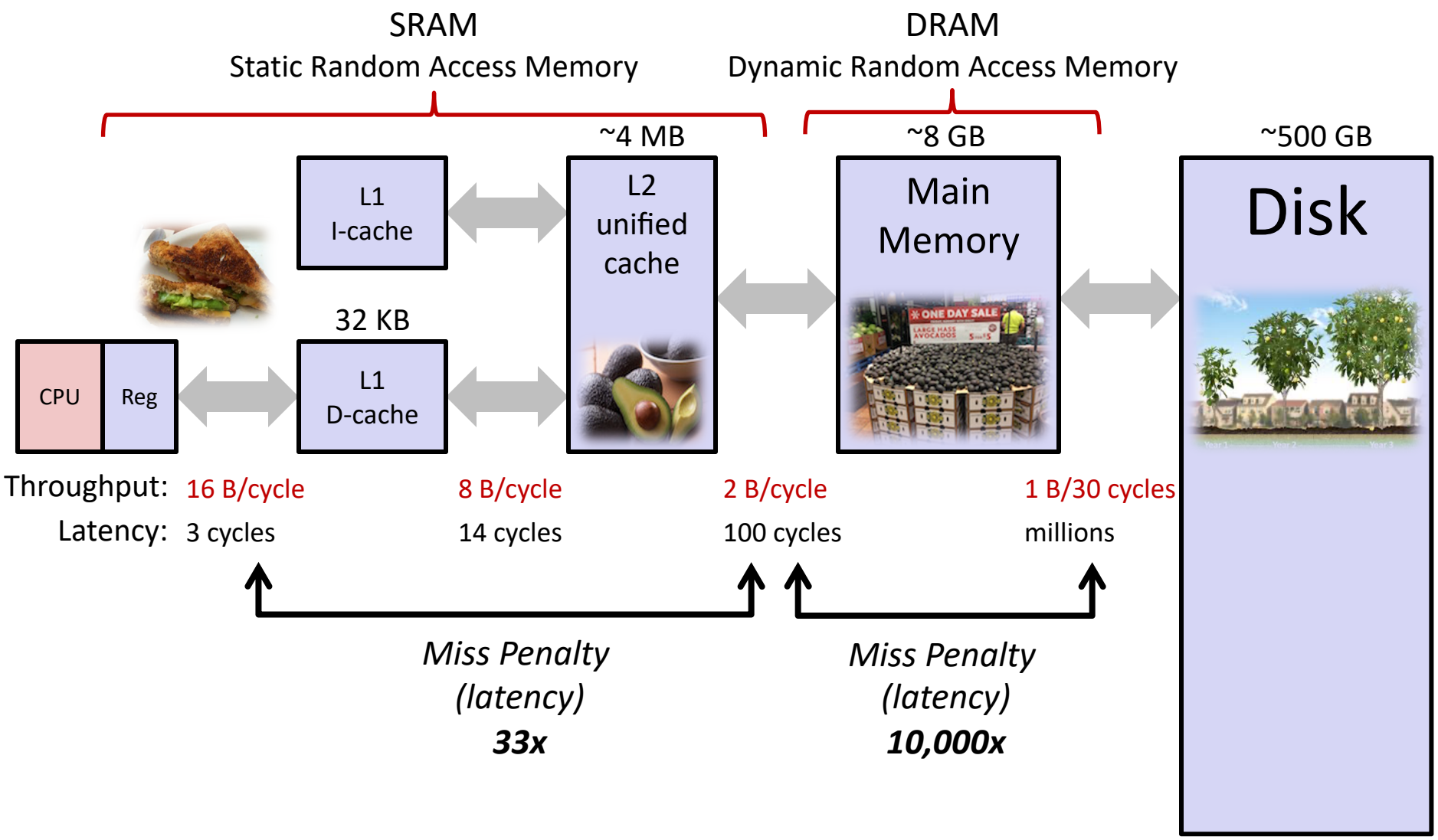
# VM and the Memory Hierarchy

- Think of memory (virtual or physical) as an array of bytes, now split into *pages*
  - Pages are another unit of aligned memory (size is $\mathrm{P} = 2^p$ bytes)
  - Each virtual page can be stored in *any* physical page (no fragmentation!)
- Pages of virtual memory are usually stored in physical memory, but sometimes spill to disk



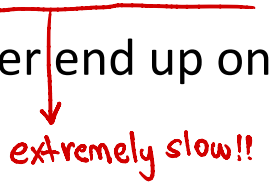
# Memory Hierarchy: Core 2 Duo

*Not drawn to scale*



# Virtual Memory Design Consequences

- Large page size: typically 4-8 KiB or 2-4 MiB
  - *Can* be up to 1 GiB (for "Big Data" apps on big computers)
  - Compared with 64-byte cache blocks
- Fully associative
  - Any virtual page can be placed in any physical page
  - Requires a "large" mapping function – different from CPU caches
- Highly sophisticated, expensive replacement algorithms in OS
  - Too complicated and open-ended to be implemented in hardware
- *Write-back* rather than *write-through*
  - *Really* don't want to write to disk every time we modify memory
  - Some things may never end up on disk (*e.g.*, stack for short-lived process)

# Why does VM work on RAM/disk?

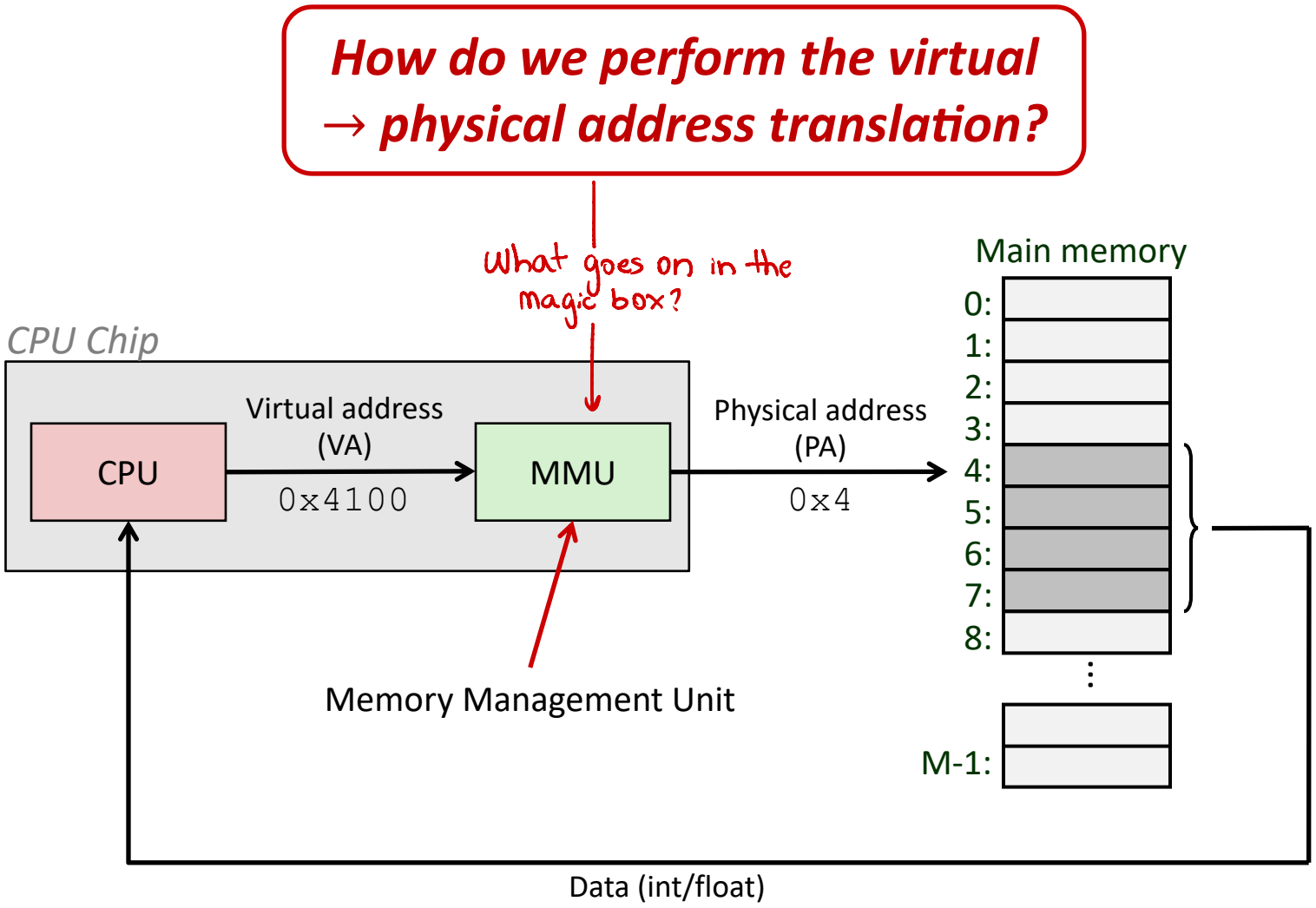
- Avoids disk accesses because of *locality*
  - Same reason that L1 / L2 / L3 caches work

    (pages much larger than blocks, so less need to explicitly optimize for page locality)
- The set of virtual pages that a program is "actively" accessing at any point in time is called its *working set*
  - If (*working set of one process* ≤ *physical memory*):
    - Good performance for one process (after compulsory misses)
  - If (*working sets of all processes* > *physical memory*):
    - ***Thrashing:*** Performance meltdown where pages are swapped between memory and disk continuously (CPU always waiting or paging)
    - This is why your computer can feel faster when you add RAM

# Virtual Memory (VM)

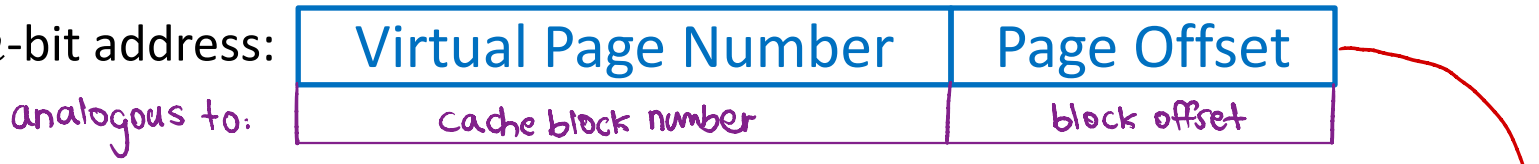
- Overview and motivation
- VM as a tool for caching
- **Address translation**
- VM as a tool for memory management
- VM as a tool for memory protection

# Address Translation

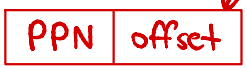


# Address Translation: Page Tables

- CPU-generated address (virtual address!) can be split into:

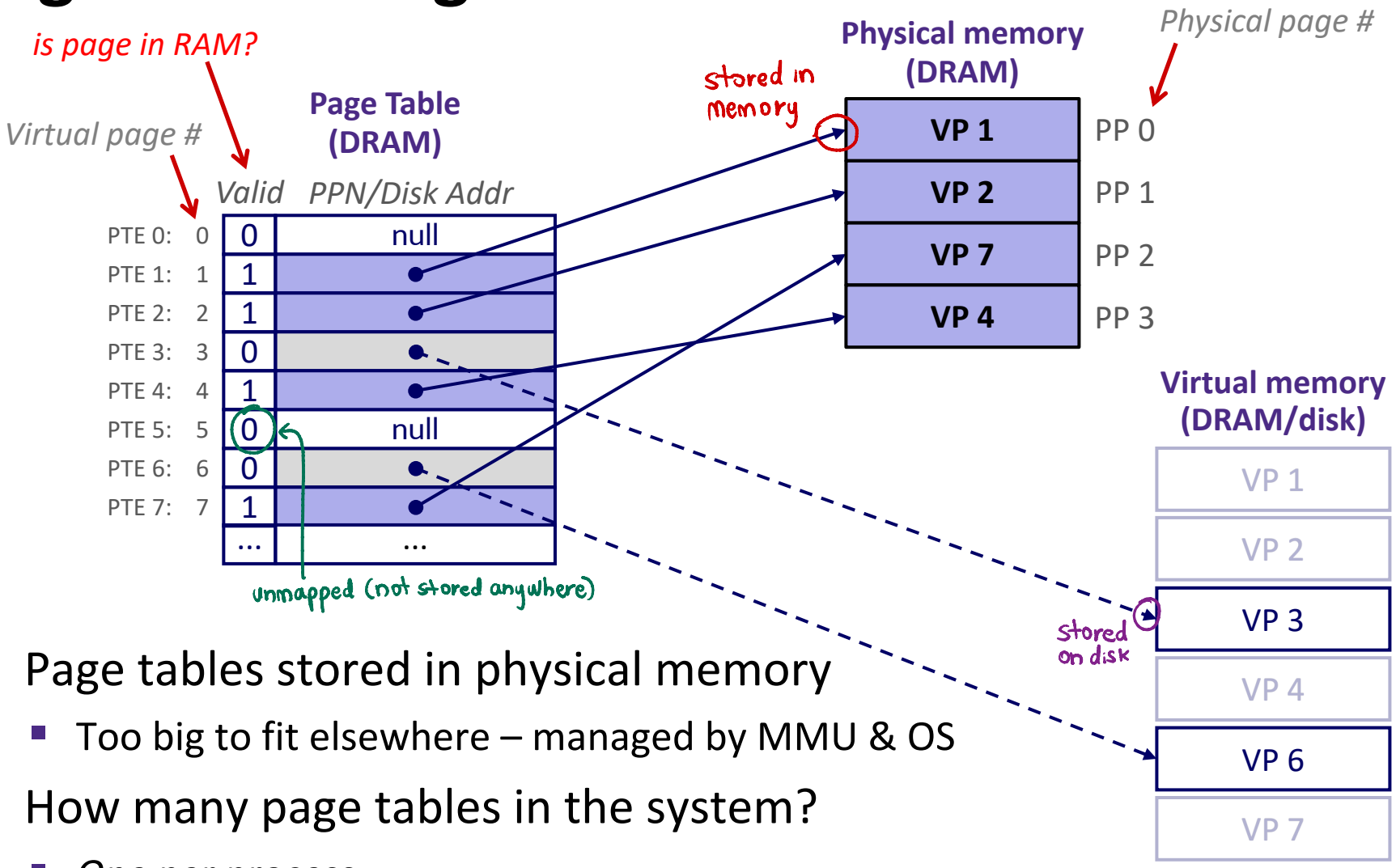



  - Request is Virtual Address (VA), want Physical Address (PA)
  - Note that Physical Offset (=) Virtual Offset (page-aligned)
    (just like with caches, don't need to translate offset!)
- Use lookup table that we call the *page table* (PT)
  - Replace Virtual Page Number (VPN) with Physical Page Number (PPN) to generate Physical Address



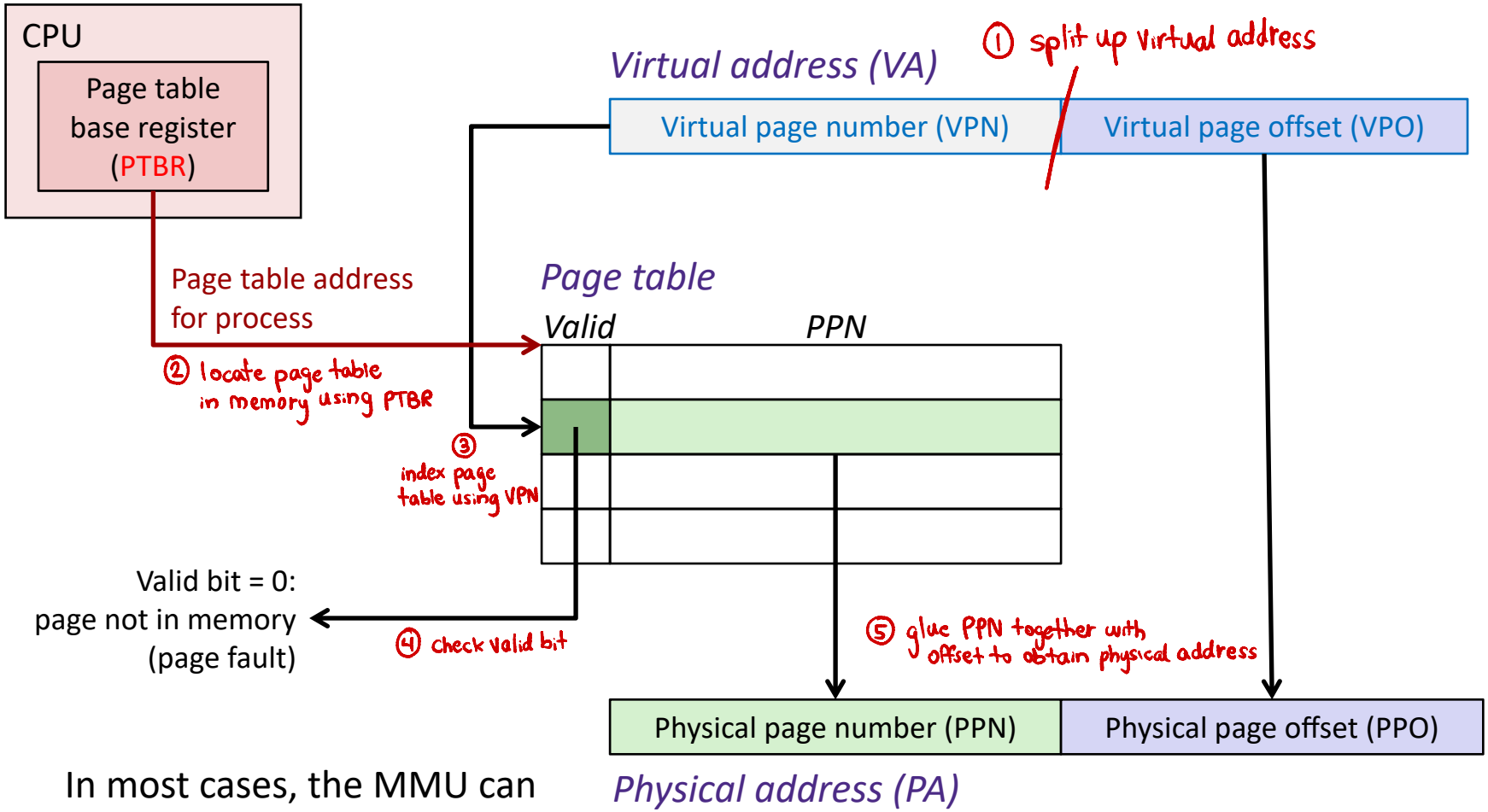

  - Index PT using VPN: page table entry (PTE) stores the PPN plus management bits (*e.g.*, Valid, Dirty, access rights)
  - Has an entry for *every* virtual page

# Page Table Diagram




- Page tables stored in physical memory
  - Too big to fit elsewhere – managed by MMU & OS
- How many page tables in the system?
  - *One per process*

# Page Table Address Translation




In most cases, the MMU can perform this translation without software assistance

# Polling Question

- How many bits wide are the following fields?
  - 16 KiB pages ($2^4$ $2^{10}$) $p = 14$ (bits to locate offset in page)
  - 48-bit virtual addresses $n = 48$
  - 16 GiB physical memory ($2^4$ $2^{30}$) $m = 34$ (bits to index physical memory)

| | VPN | PPN |
|---|---|---|
| (A) | 34 | 24 |
| (B) | 32 | 18 |
| (C) | 30 | 20 |
| (D) | 34 | 20 |

VPN = $n - p$ = 34 bits

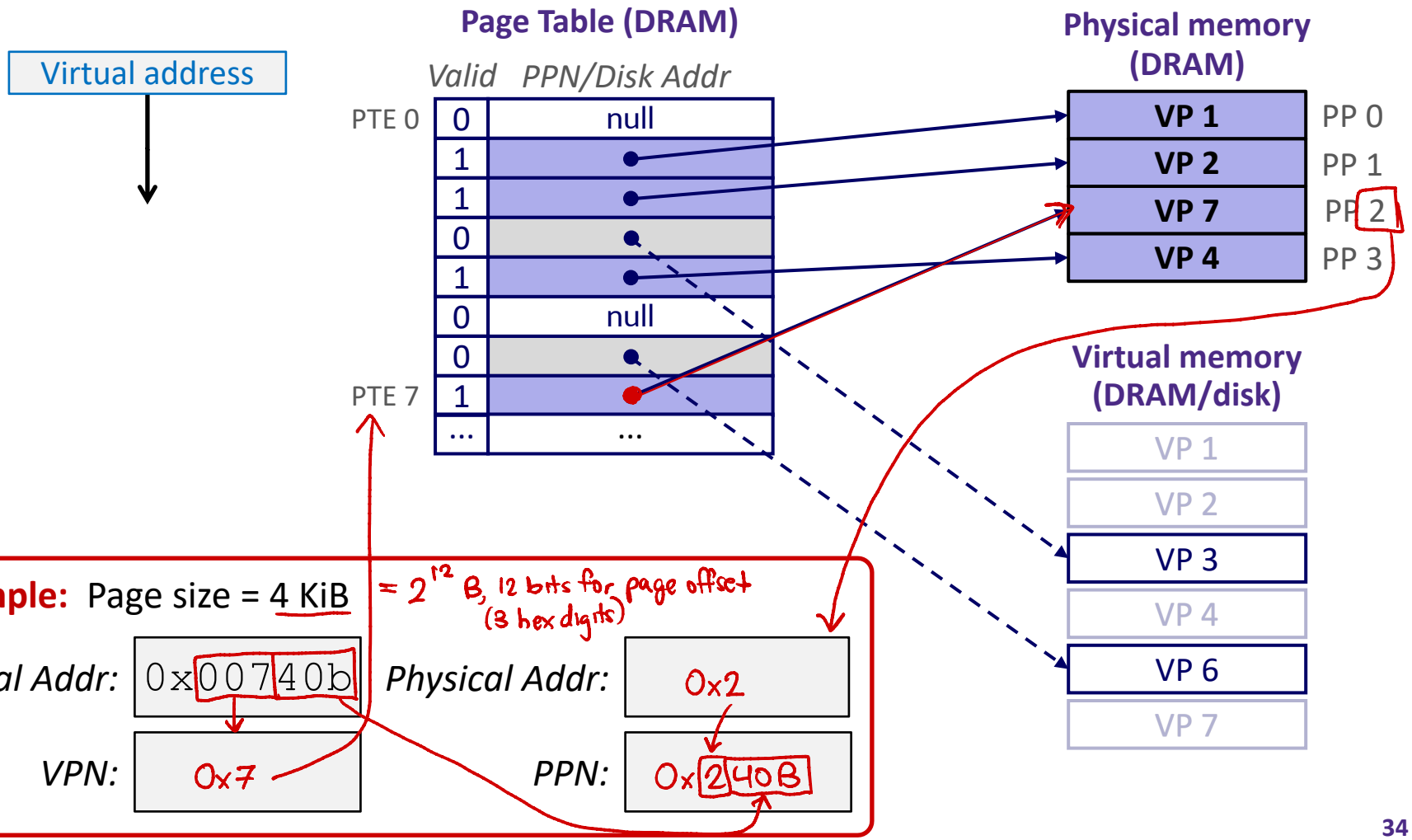
PPN = $m - p$ = 20 bits

address width

page offset bits

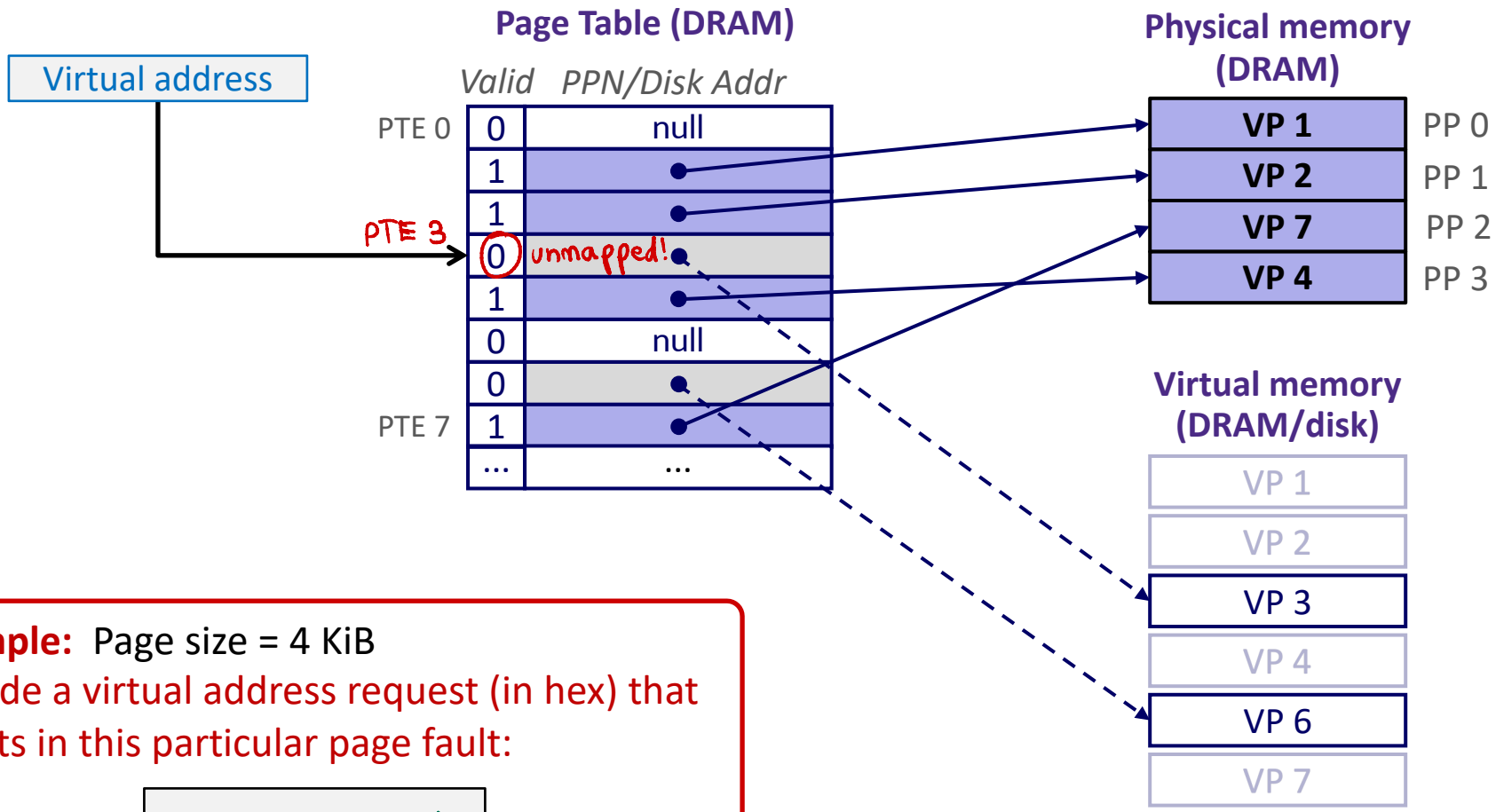
# Page Hit

- ***Page hit:*** VM reference is in physical memory



# Page Fault

- ***Page fault:*** VM reference is NOT in physical memory




**Example:** Page size = 4 KiB

Provide a virtual address request (in hex) that results in this particular page fault:

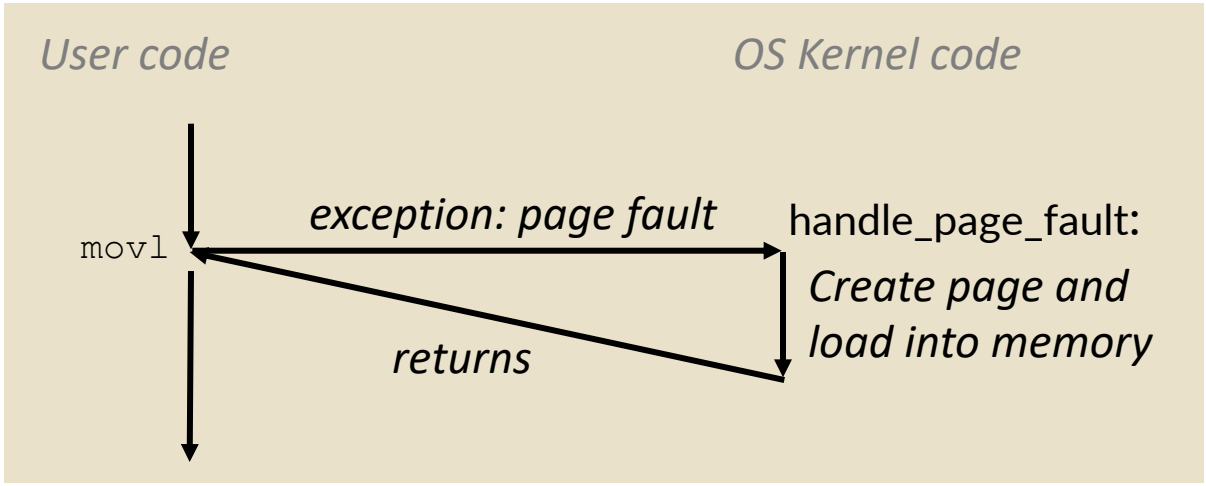
*Virtual Addr:* 0x3000 can be anything

# Reminder: Page Fault Exception

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
int a[1000];
int main () {
    a[500] = 13;
}
```

```
80483b7:    c7 05 10 9d 04 08 0d    movl   $0xd,0x8049d10
```




- Page fault handler must load page into physical memory
- Returns to faulting instruction: `mov` is executed again!
  - Successful on second try