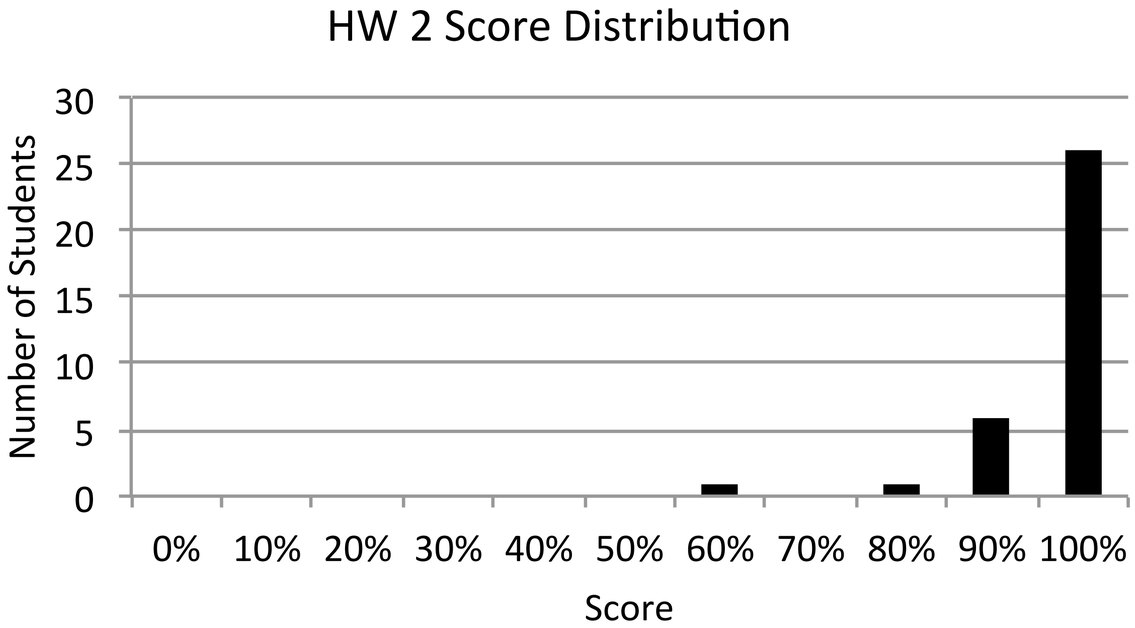
18-447: Computer Architecture
Lecture 16: Virtual Memory

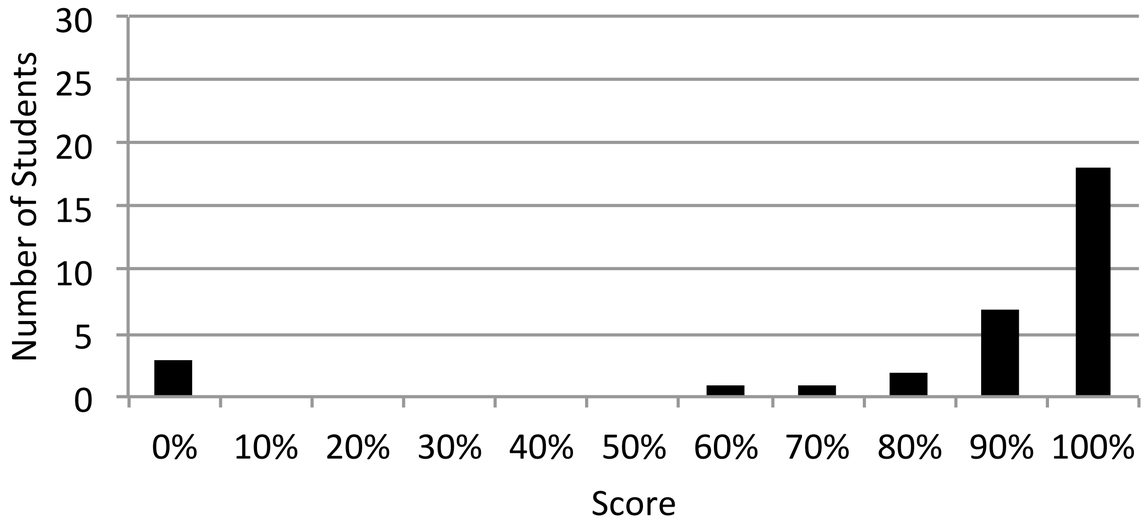
Justin Meza
Carnegie Mellon University
(with material from Onur Mutlu, Michael Papamichael, and Vivek Seshadri)

Notes

- HW 2 and Lab 2 grades will be posted soon
- If you have not filled out the feedback form, please do so!
- This week:
  - A couple of short lectures on virtual memory
  - A recitation session for HW 3 and last year's Midterm 1
- Midterm coming up soon (March 6)...

HW 2 and Lab 2 Distributions





Notes

- HW 2 and Lab 2 grades will be posted soon
- If you have not filled out the feedback form, please do so!
- This week:
  - A couple of short lectures on virtual memory
  - A recitation session for HW 3 and last year's Midterm 1
- Midterm coming up soon (March 6)...

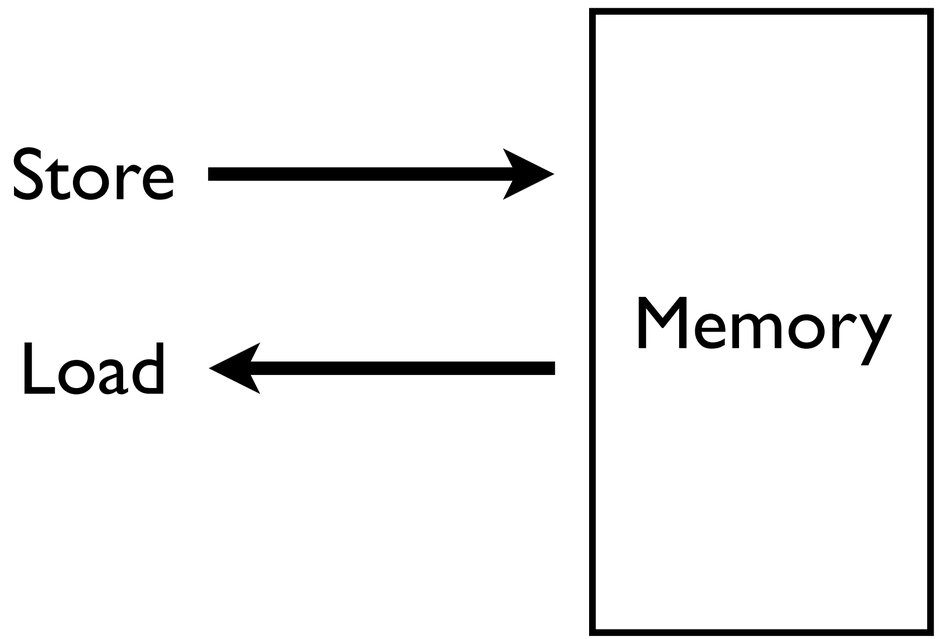
Midterm 1

- Wed. March 6 during class (12:30 to 2:20pm)
- Arrive early, sit with one seat in between you and the next person
- Closed book, closed notes, single 8.5 x 11 note sheet
- All topics covered (including this week) may be on exam
  - Lectures, homeworks, labs, required readings
- Look over the past midterm (http://www.ece.cmu.edu/~ece447/s12)

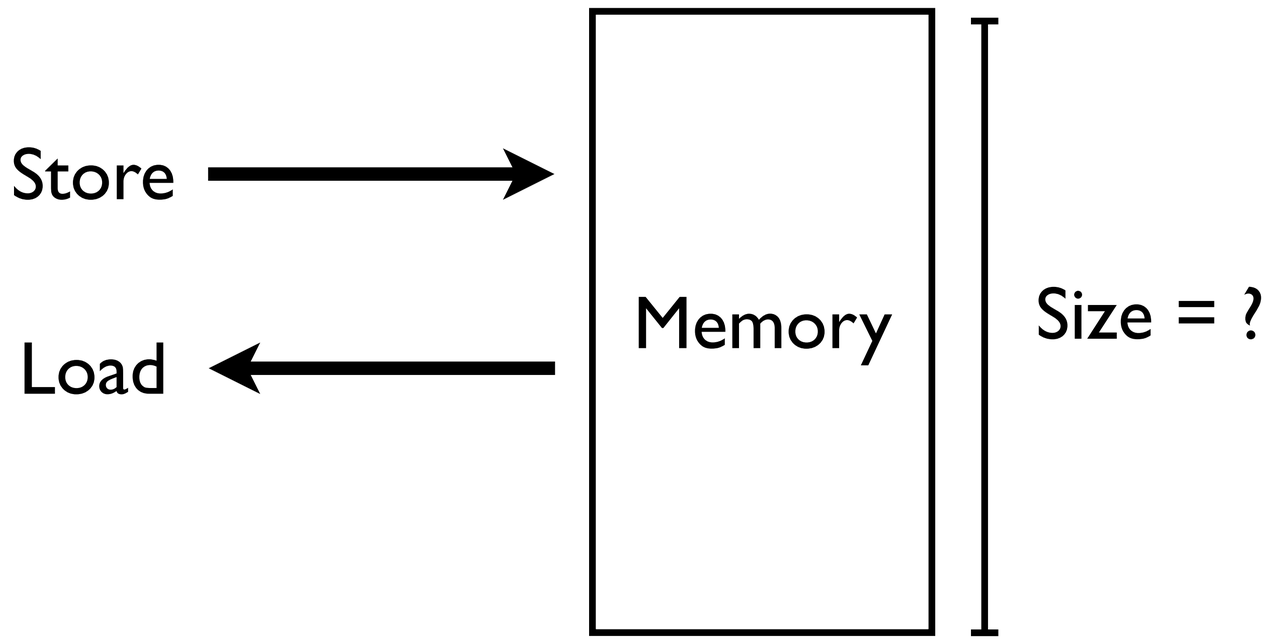
What We Learned Last Time

- Enabling more parallelism with dataflow and SIMD
- We'll cover a bit more of this material next week

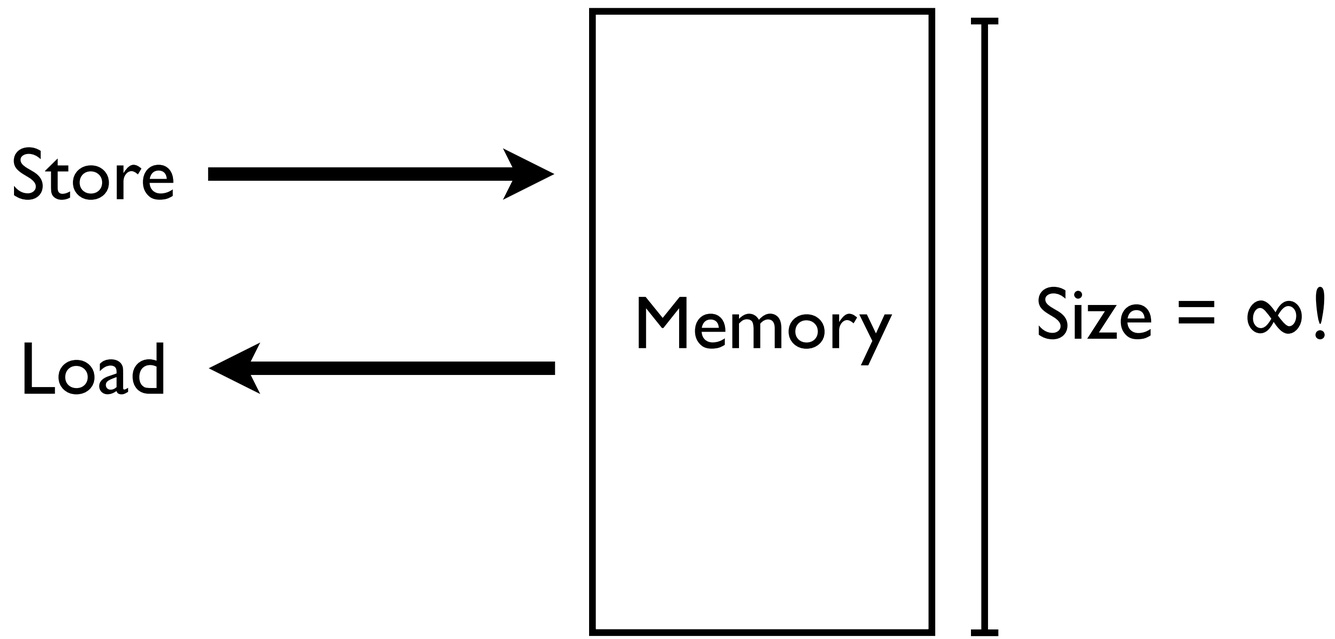
Memory: Programmer's View



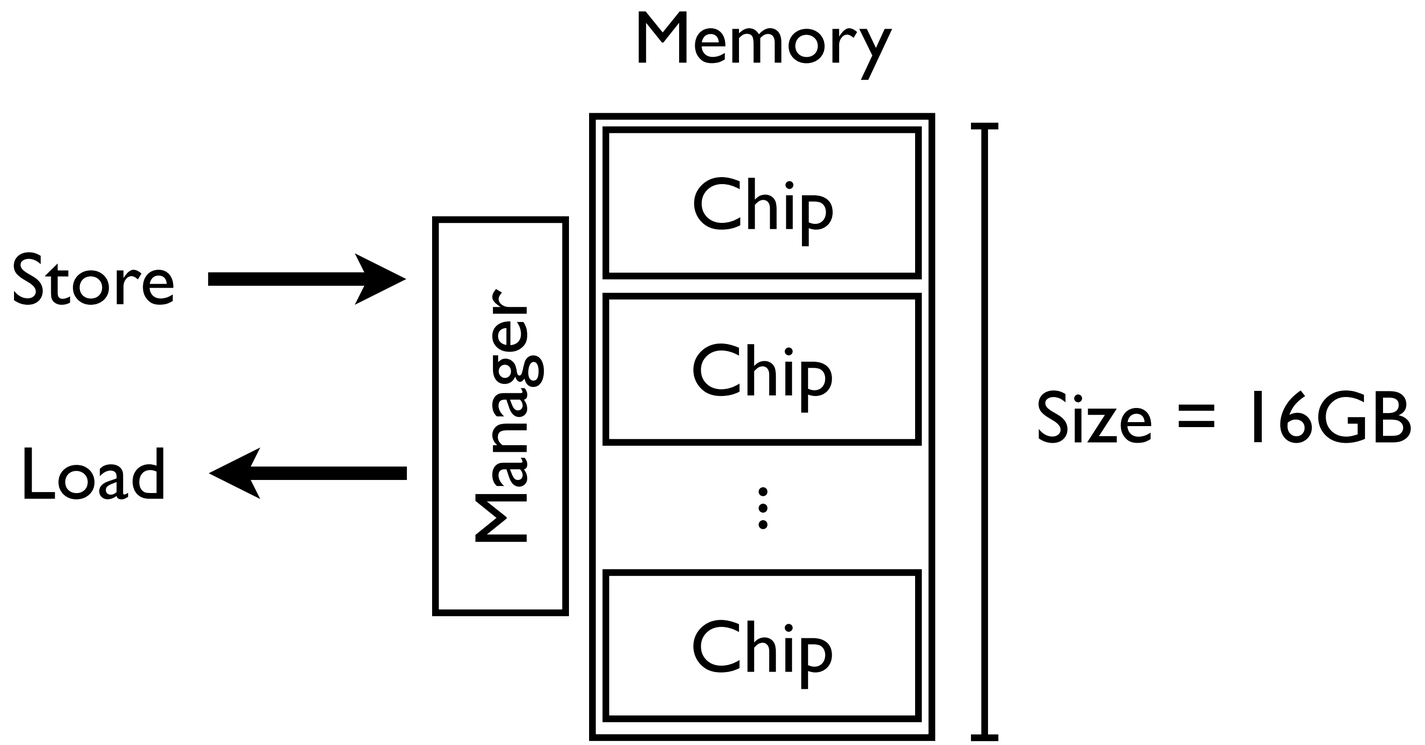
Memory: Programmer's View



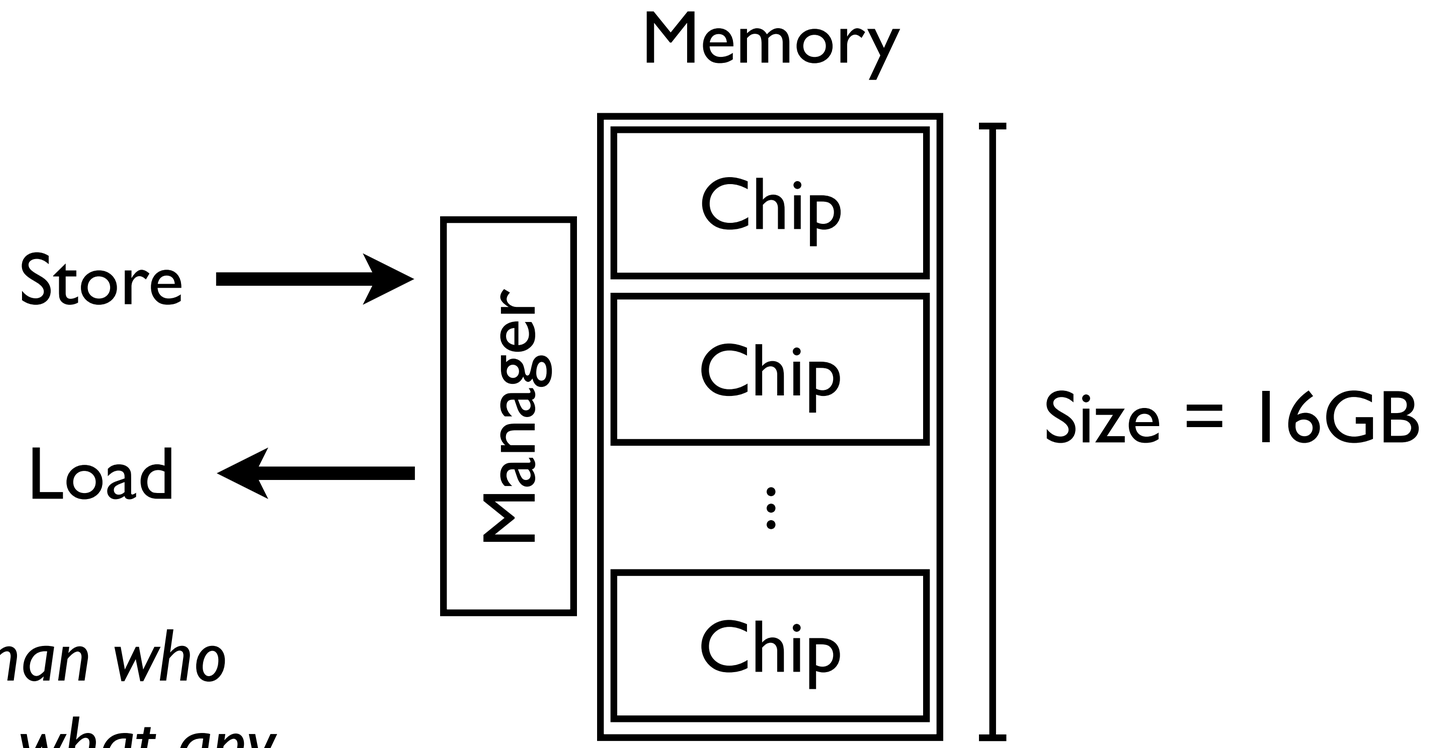
Memory: Programmer's View



Aside: How Memory Really Works



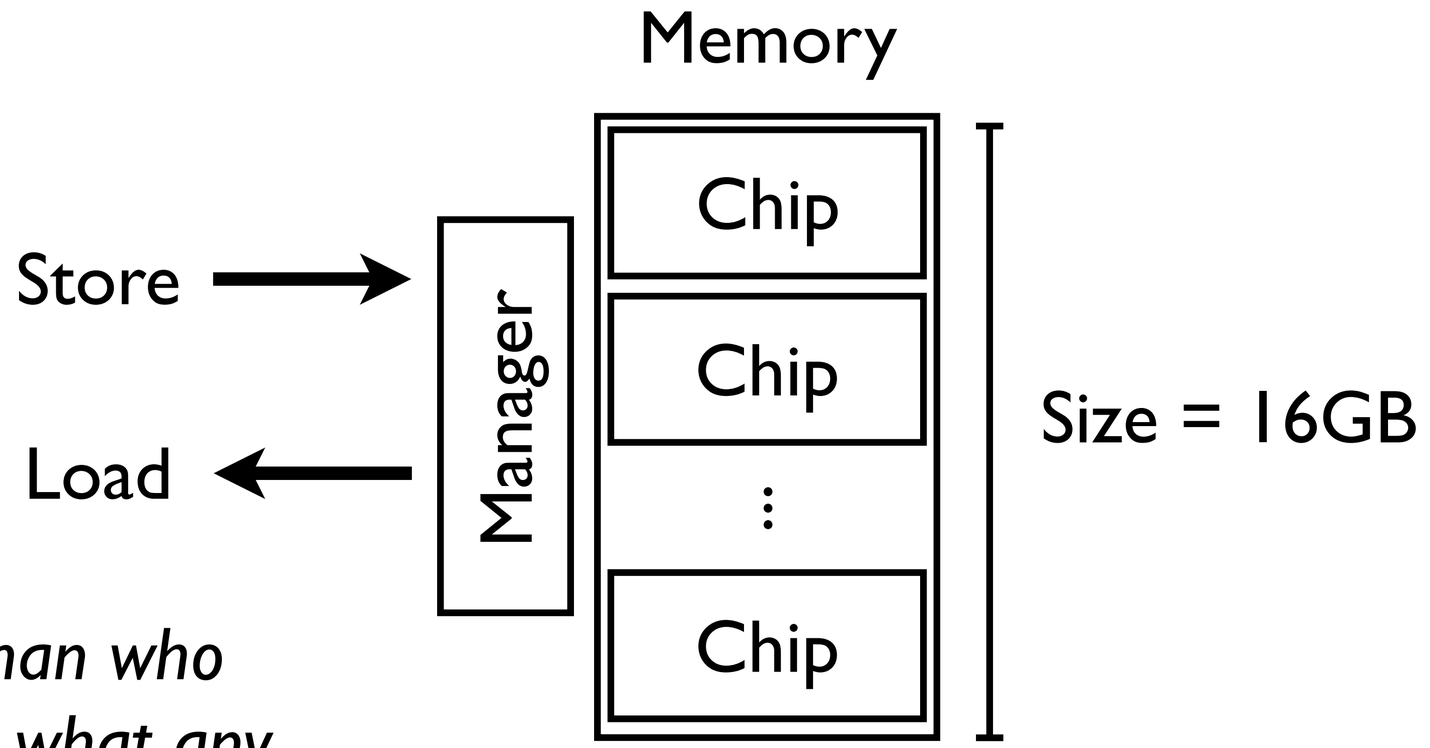
Aside: How Memory Really Works



"*An engineer is a man who can do **for a dime** what any fool can do **for a dollar***"

— Anonymous

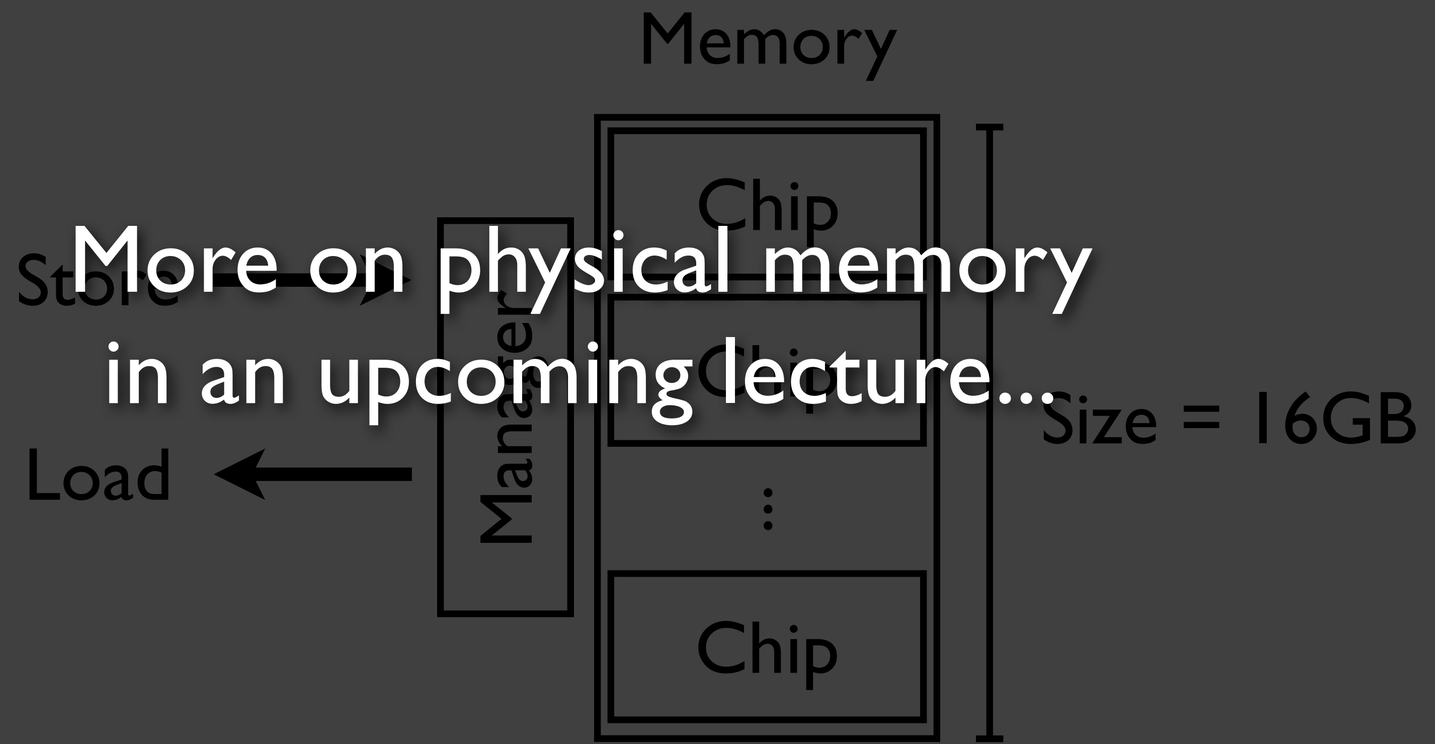
Aside: How Memory Really Works



"*An engineer is a man who can do **with 16GB** what any fool can do **with ∞**"*

— Justin

Aside: How Memory Really Works



More on physical memory in an upcoming lecture...

Today: Virtual Memory

- Why "virtual"?
  - If you think it's there, and it's there... it's *real*
  - If you think it's not there, and it's not there... it's *non-existent*
  - If you think it's not there, and it's there... it's *transparent*
  - If you think it's there, and it's not there... it's ***imaginary***
- Virtual memory is ***imaginary*** memory
  - It gives you the illusion of memory that's not physically there

Overview: Why Virtual Memory?

- Using physical memory *efficiently*
- Using physical memory *simply*
- Using physical memory *safely*

Using Physical Memory Efficiently

- Virtual memory uses gets the most out of physical memory
- Demand paging
  - Main memory is a cache for portions of virtual address space
    - The rest of the virtual address space is stored on disk
- Keep only active areas of virtual address space in fast memory
- Transfer data back and forth as needed

Using Physical Memory Simply

- Virtual memory simplifies memory management
- Programmer can think in terms of a large, linear address space
- Processes access same large, linear address space

Using Physical Memory Safely

- Virtual memory protests process' address spaces
- Processes cannot interfere with each other
  - Because they operate in different address space
- User processes cannot access priviledged information
  - Different sections of address space have different permissions
  - Think: read-only, read/write, execute, ...

Virtual Memory Benefits

- We'll talk in depth about each of these next
  - **Demand paging**: Using physical memory *efficiently*
  - **Memory management**: Using physical memory *simply*
  - **Protection**: Using physical memory *safely*

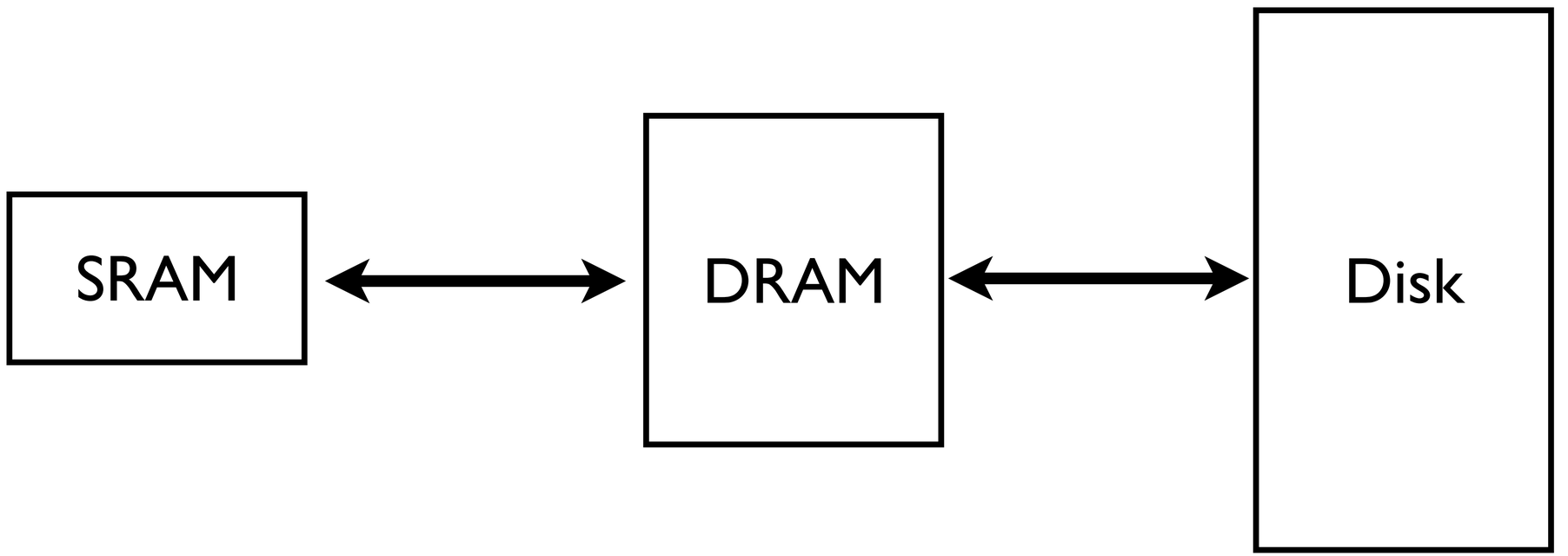
Virtual Memory Benefits

- We'll talk in depth about each of these next
  - **Demand paging**: Using physical memory *efficiently*
  - **Memory management**: Using physical memory *simply*
  - **Protection**: Using physical memory *safely*

The Cost of Maintaining the Illusion of ∞ Memory

- Address space is large:
  - 32-bits: ~4,000,000,000 (four billion) bytes
  - 64-bits: ~16,000,000,000,000,000,000 (sixteen quintillion) bytes
- Memory (DRAM) is expensive (1 TB of DRAM ~$10,000)
- But disk storage is relatively cheap (1 TB of disk < $100)
- Store most data on disk to maintain the illusion of ∞ memory in a cost-effective way

The Cost of Maintaining the Illusion of ∞ Memory

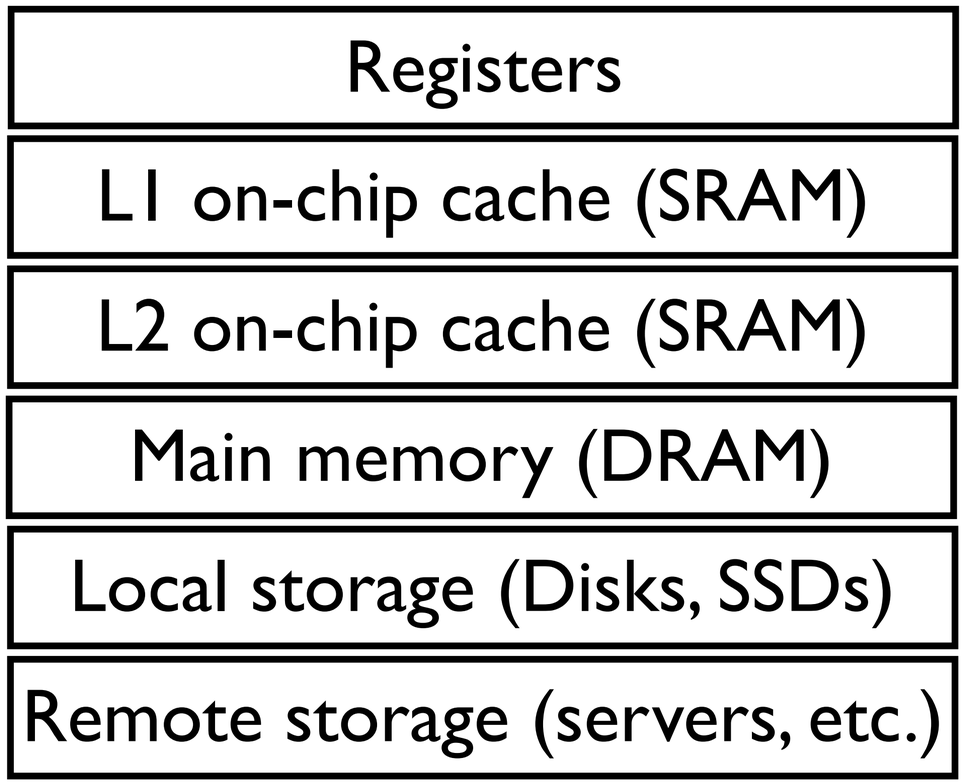


4 MB ~ $100

8 GB ~ $100

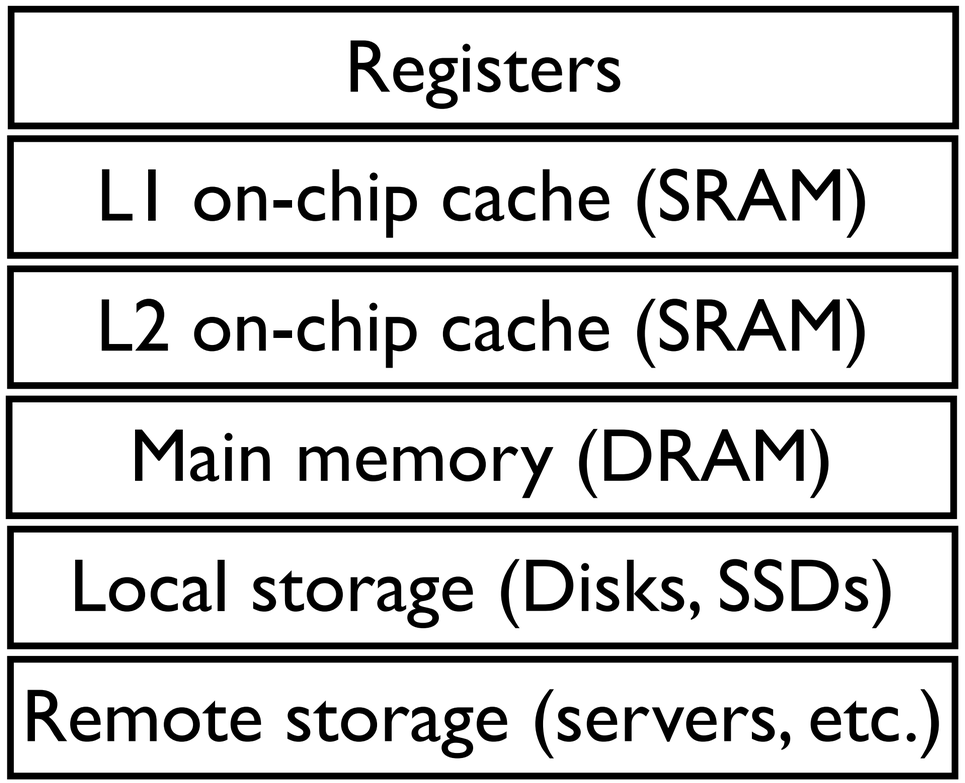
1 TB ~ $100

Aside: The Memory Hierarchy

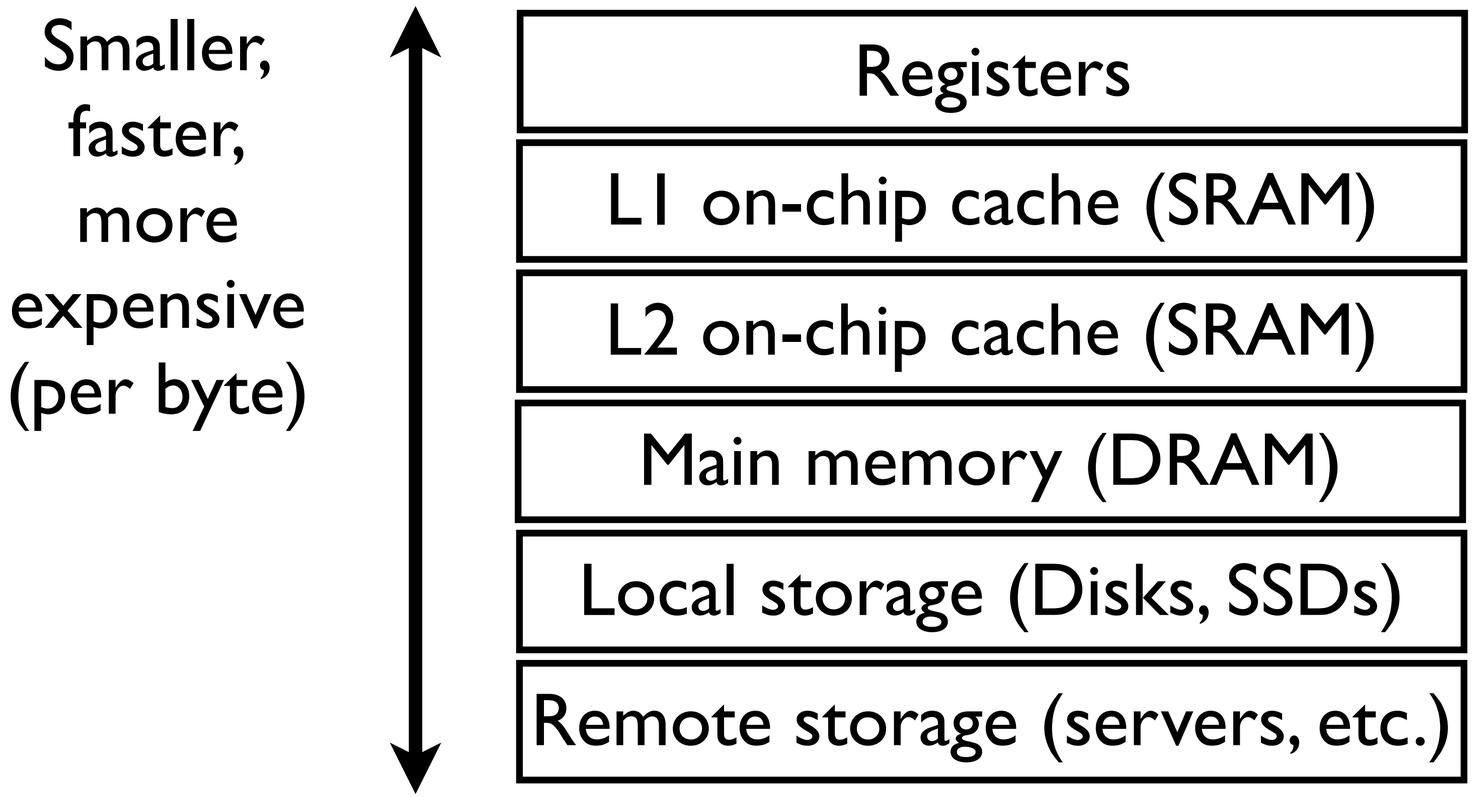


Aside: The Memory Hierarchy

Each level holds (or *caches*) data retrieved from level below it, and data displaced from level above it



Aside: The Memory Hierarchy



The Cost of Maintaining the Illusion of ∞ Memory

- So, DRAM *caches* disk data and SRAM *caches* DRAM data
- Should these caches be built in the same way?

The Cost of Maintaining the Illusion of ∞ Memory

- So, DRAM *caches* disk data and SRAM *caches* DRAM data
- Should these caches be built in the same way?
  - Big difference: DRAM ~10X slower than SRAM but disk ~100,000X slower than DRAM
  - Another big difference: When accessing sequential data on disk the first byte is ~100,000X slower than successive bytes (DRAM is much smaller, ~4X)
- Let's learn a bit more about caches before we build virtual memory

Caching Primer

- More detail on this later (just enough for virtual memory for now)
- Some basic terminology
  - Block (or line): a chunk of data
  - Set: a group of blocks
  - Miss: when a requested block is not present in the cache
  - Hit: opposite of a miss
  - Insertion: triggered by a miss, fills a block into a set
  - Eviction: possibly triggered by an insertion, displaces a block from a set

Caching Primer

- Important cache design parameters
  - Block size: how much data to transfer to and from the cache?
  - Associativity: how many blocks per set?
  - Write through: whether stores bypass (and invalidate) all cached versions, and are written through to a lower level in the hierarchy
  - ... versus write back: whether stores are written back to the next level of the hierarchy

Caching Primer

- Let's apply what we've learned so far:
- For a disk cache in DRAM, how would you set (and why):
  - Line size (among of data to move to/from): Large or Small ?
  - Associativity (blocks per set): High or Low ?
  - Write through or write back ?

Caching Primer

- Let's apply what we've learned so far
- For a disk cache in DRAM, how would you set (and why):
  - Line size (among of data to move to/from): Large or Small ?
  - Associativity (blocks per set): High or Low ?
  - Write through or write back ?

Disks are better at transferring large blocks

Caching Primer

- Let's apply what we've learned so far
- For a disk cache in DRAM, how would you set (and why):
  - Line size (among of data to move to/from): Large or Small ?
  - Associativity (blocks per set): High or Low ?
  - Write through or write back ?

Want more diversity in which blocks to evict to reduce number of misses

Caching Primer

- Let's apply what we've learned so far
- For a disk cache in DRAM, how would you set (and why):
  - Line size (among of data to move to/from): Large or Small ?
  - Associativity (blocks per set): High or Low ?
  - Write through or write back ?

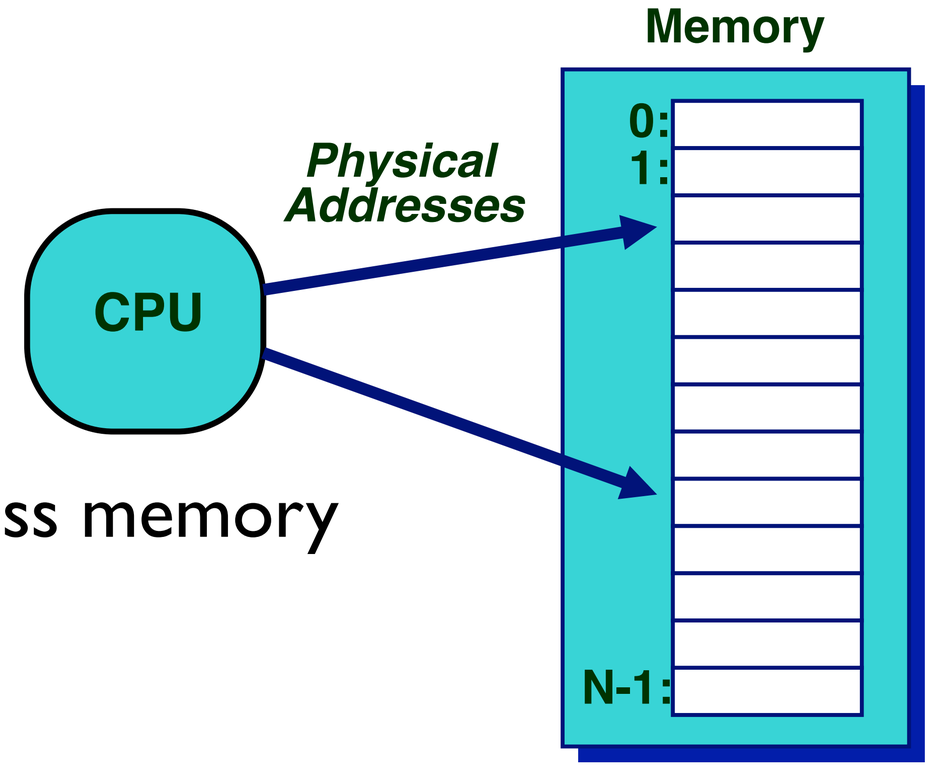
Want to coalesce writes and perform large writes back to disk

Caching Primer

- Let's apply what we've learned so far
- For a disk cache in DRAM, how would you set (and why):
  - Line size (among of data to move to/from): Large or Small ?
  - Associativity (blocks per set): High or Low ?
  - Write through or write back ?
- Let's get back to memory

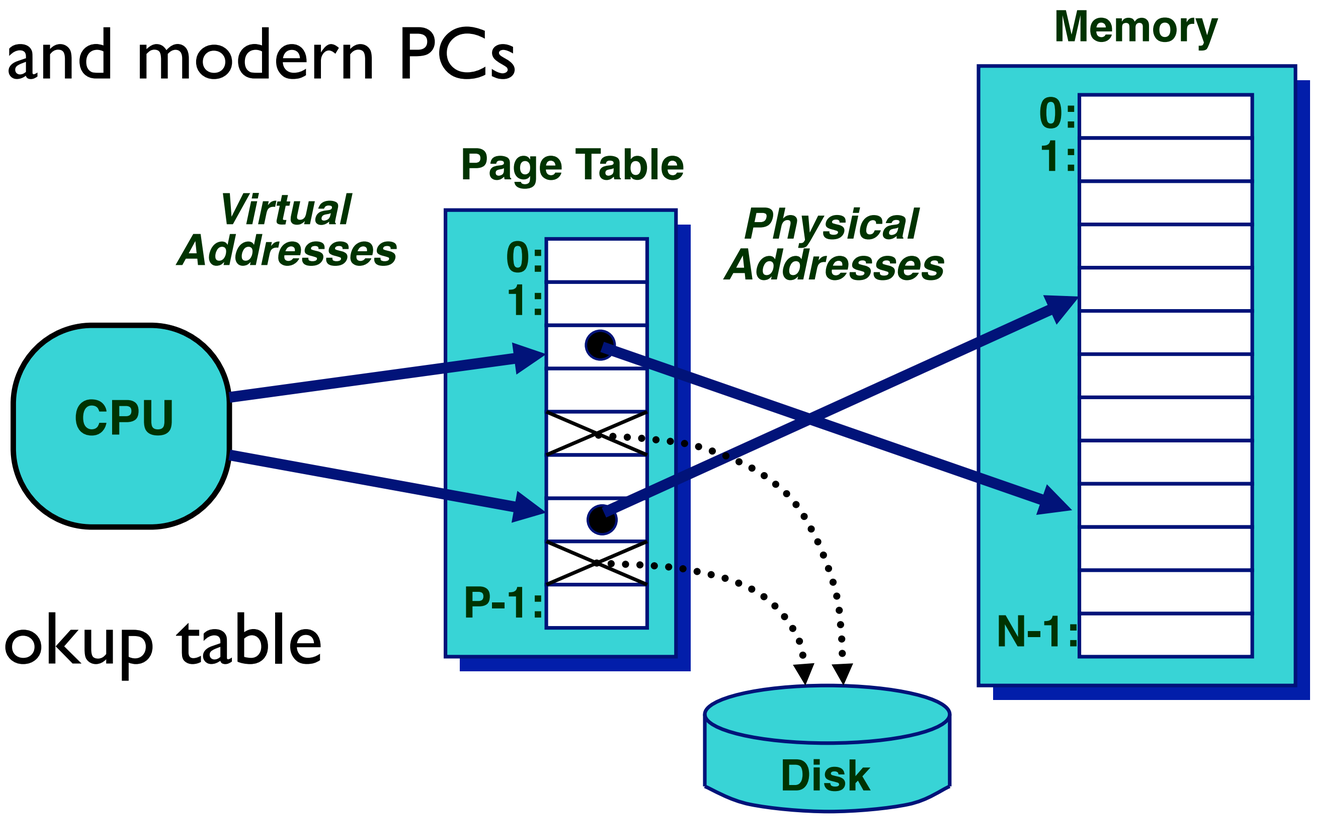
A System with Only Physical Memory

- Examples:
  - Most Cray machines
  - Early PCs
  - Nearly all embedded systems
- Loads and stores uses directly to access memory



A System with Virtual Memory

- Examples:
  - Most laptops, server, and modern PCs
- *Page* (i.e., a block)
- Address translation: Hardware converts virtual addresses into physical addresses using an OS-managed lookup table (the *page table*)

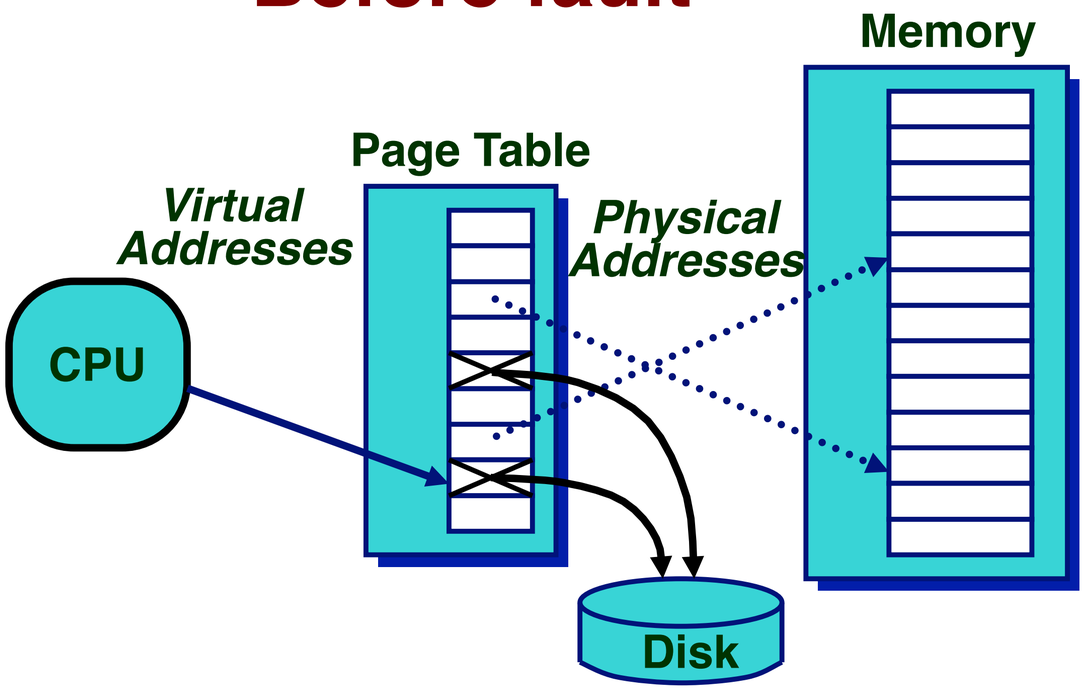


Page Faults (i.e., Cache Misses)

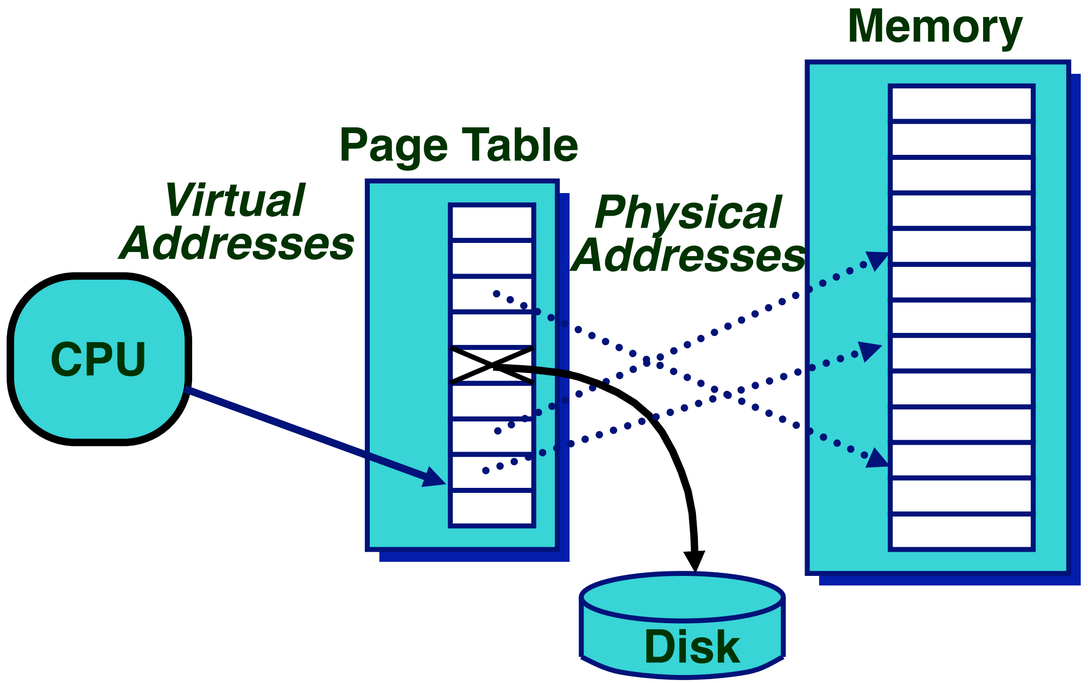
- Problem: A page is on disk and not in memory
  - Page table entry indicates virtual address is not in memory
- Solution: An OS routine is called to load data from disk to memory
  - Current process suspends execution, others may resume
  - OS has full control over placement

Page Faults (i.e., Cache Misses)

Before fault

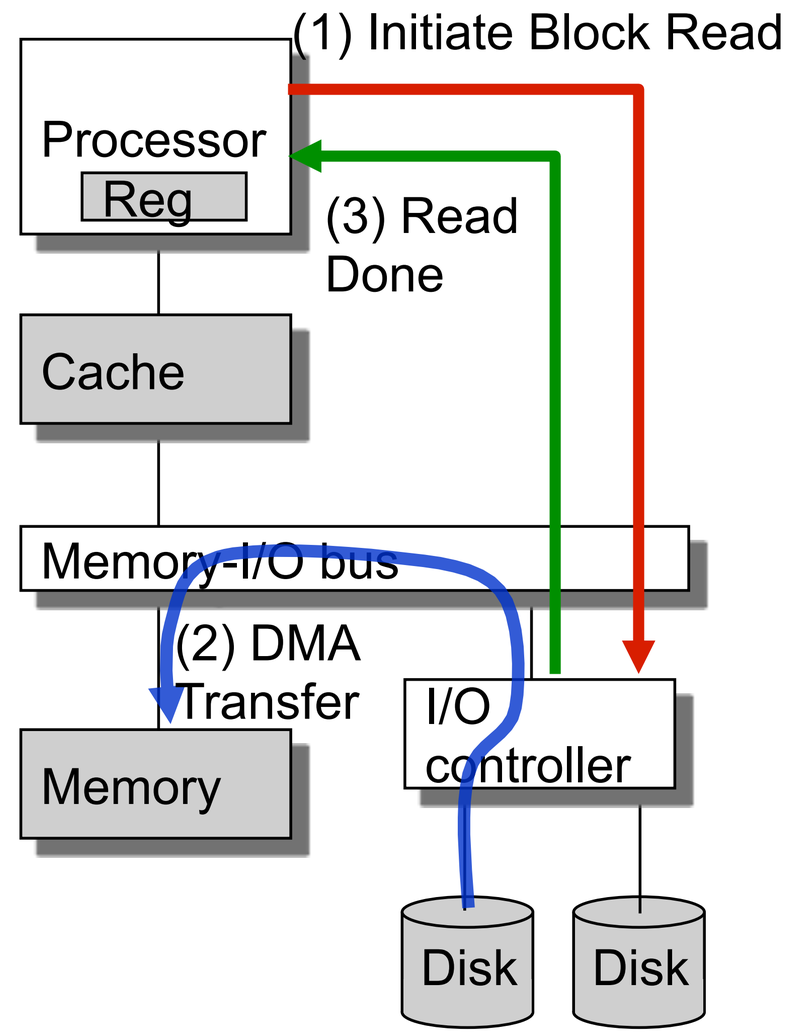


After fault



Servicing a Page Fault

- Processor communicates with controller
  - Read block of length *P* starting at disk address *X* and store starting at memory address *Y*
- Read occurs
  - Direct Memory Access (DMA)
  - Done by I/O controller
- Controller signals completion
  - Interrupt processor invokes OS
  - OS resumes suspended process



Why Does Virtual Memory Work?

Why Does Virtual Memory Work?

- Locality!
  - Temporal locality: Accessing the same data within a short amount of time
  - Spatial locality: Accessing data within a short amount of space
- Working set: The set of active virtual pages
  - Programs with higher temporal locality have smaller working sets
  - If working set < memory size: good performance after initial misses
  - If working set > memory size: thrashing, pages are copied in and out

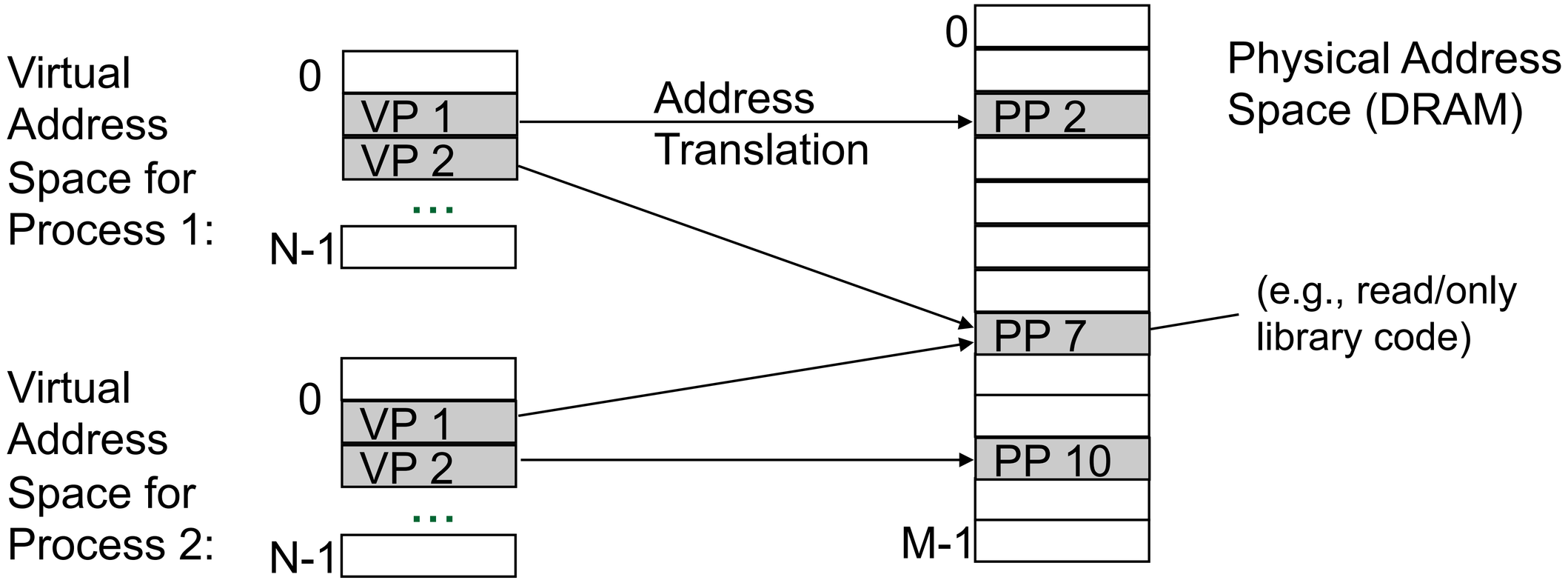
Virtual Memory Benefits

- We'll talk in depth about each of these next
  - **Demand paging**: Using physical memory *efficiently*
  - **Memory management**: Using physical memory *simply*
  - **Protection**: Using physical memory *safely*

Memory Management

- Virtual / physical address spaces divided into equal-sized blocks
  - "Virtual pages" in virtual memory
  - "Physical pages" or "frames" in physical memory
- Key idea: Each process has its own virtual address space
  - Simplifies memory allocation
    - A virtual page can be mapped to any physical page
  - Simplifies sharing code and data among processes
    - The OS can map virtual pages to same shared physical page

Memory Management

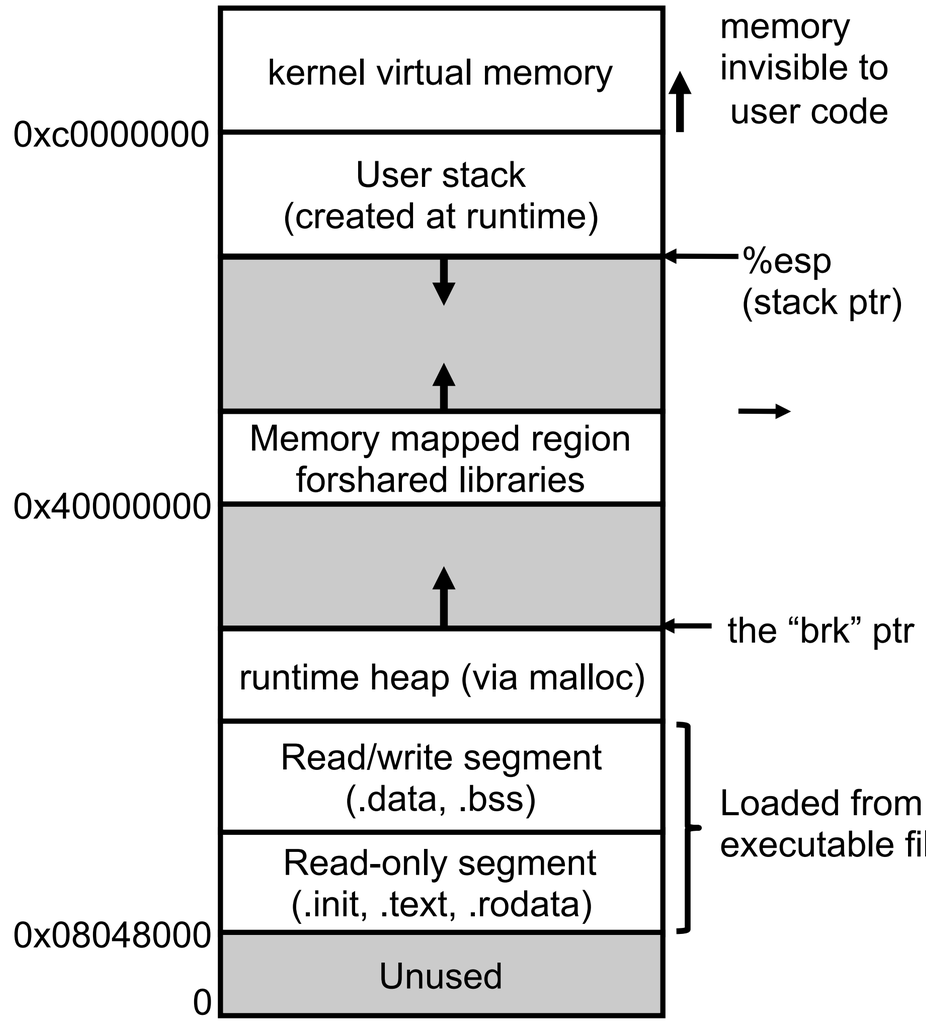


Memory Management

- Key idea: Each process has its own virtual address space
  - Simplifies memory allocation
    - A virtual page can be mapped to any physical page
  - Simplifies sharing code and data among processes
    - The OS can map virtual pages to same shared physical page
- Simplifies linking and loading

Memory Management

- Linking
  - Each program has similar virtual address space
  - Code, stack, and shared libraries always start at the same address
- Loading
  - Virtual pages can be loaded on demand (on first access)



Memory Management

- Key idea: Each process has its own virtual address space
  - Simplifies memory allocation
    - A virtual page can be mapped to any physical page
  - Simplifies sharing code and data among processes
    - The OS can map virtual pages to same shared physical page
  - Simplifies linking and loading
- Great that it's simple, but what good is that if it's not secure?

Virtual Memory Benefits

- We'll talk in depth about each of these next
  - **Demand paging**: Using physical memory *efficiently*
  - **Memory management**: Using physical memory *simply*
  - **Protection**: Using physical memory *safely*

Protection with Virtual Memory

- A normal user process should not be able to:
  - Read/write another process' memory
  - Write into shared library data
- How does virtual memory help?
  - Address space isolation
  - Protection information in page table
  - Efficient clearing of data on newly allocated pages

Protection: Address Isolation

- Processes only access virtual addresses
- Cannot access physical addresses directly
- Go through per-process page table to perform translation
  - If physical page is not in page table, it is not accessible
- A normal user process should not be able to:
  - ~~Read/write another process' memory~~
  - Write into shared library data

Protection: Page Table Information

- Page table entry contains permission information
  - Hardware enforces this protection
  - OS is summoned if a violation occurs (send process SIGSEGV, segmentation fault)
  - The page table itself is in protected memory (only OS can update)
- A normal user process should not be able to:
  - ~~Read/write another process' memory~~
  - ~~Write into shared library data~~

Protection: Leaked Information

- Example (with the virtual memory we've discussed so far):
  - Process *A* writes "my password = ..." to virtual address 2
  - OS maps virtual address 2 to physical page 4 in page table
  - Process *A* no longer needs virtual address 2
  - OS unmaps virtual address 2 from physical page 4 in page table
- Attack vector:
  - Sneaky Process *B* continually allocates pages and searches for "my password = <string>"

Protection: Leaked Information

- Programmer shouldn't have to worry about their data being leaked
- OS can ensure that pages are initialized to all zeros when allocated
- Let's use what we've learned to do this quickly in virtual memory

Protection: Leaked Information

- Programmer shouldn't have to worry about their data being leaked
- OS can ensure that pages are initialized to all zeros when allocated
- Let's use what we've learned to do this quickly in virtual memory
  - Remember shared pages? New pages can share an all-zero page
    - Saves a lot of initial stores of the value zero to memory
  - The OS can *copy-on-write* when the all-zero page is stored to
    - Allocates a new virtual page on demand (what is this also useful for?)

Protection: Leaked Information

- Programmer shouldn't have to worry about their data being leaked
- OS can ensure that pages are initialized to all zeros when allocated
- Let's use what we've learned to do this quickly in virtual memory
  - Remember shared pages? New pages can share an all-zero page
    - Saves a lot of initial stores of the value zero to memory
  - The OS can *copy-on-write* when the all-zero page is stored to
    - Allocates a new virtual page on demand (what is this also useful for? ➔ forking / threading)

Aside: Evolution of Memory Management Mechanisms

- Single-user machines
- Base and bound registers
- Segmented address space

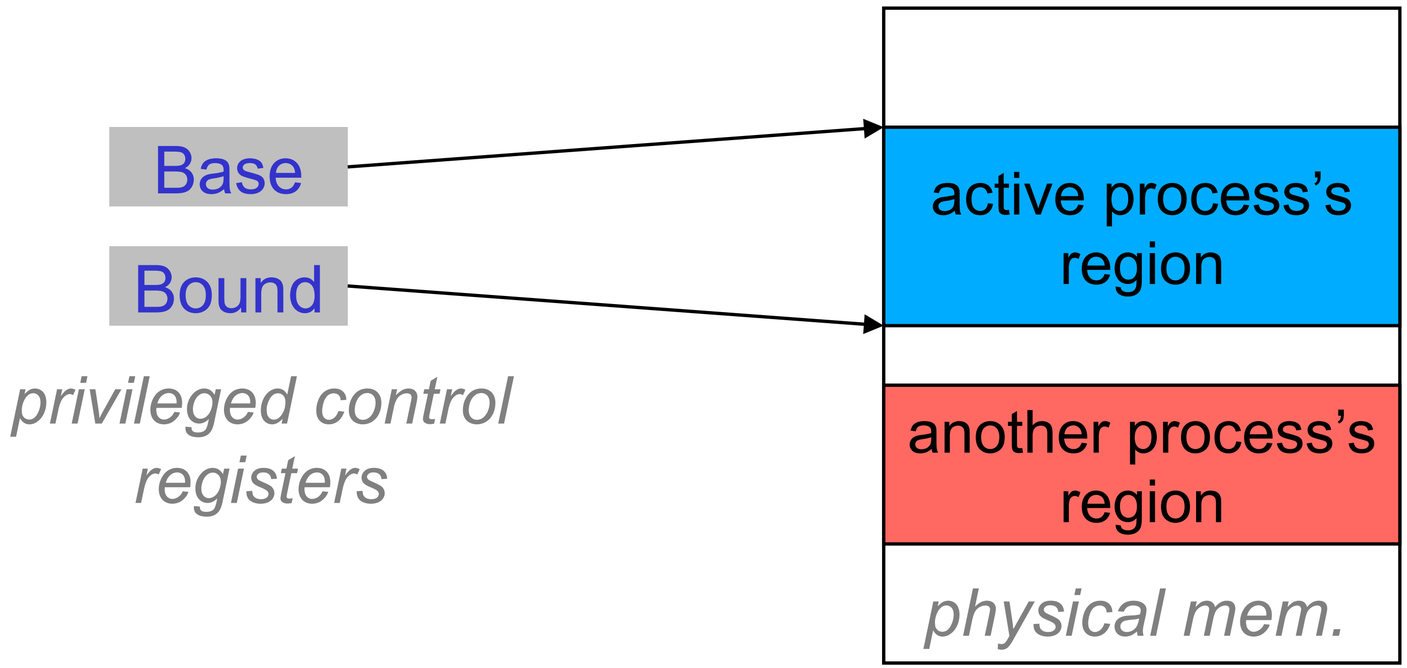
Single-User Machines

- Early machines had no concept of protection and address translation
  - No need with a single process and a single user
  - Such machines were automatically private and had a uniform view of the address space
  - Programs operated on physical addresses directly

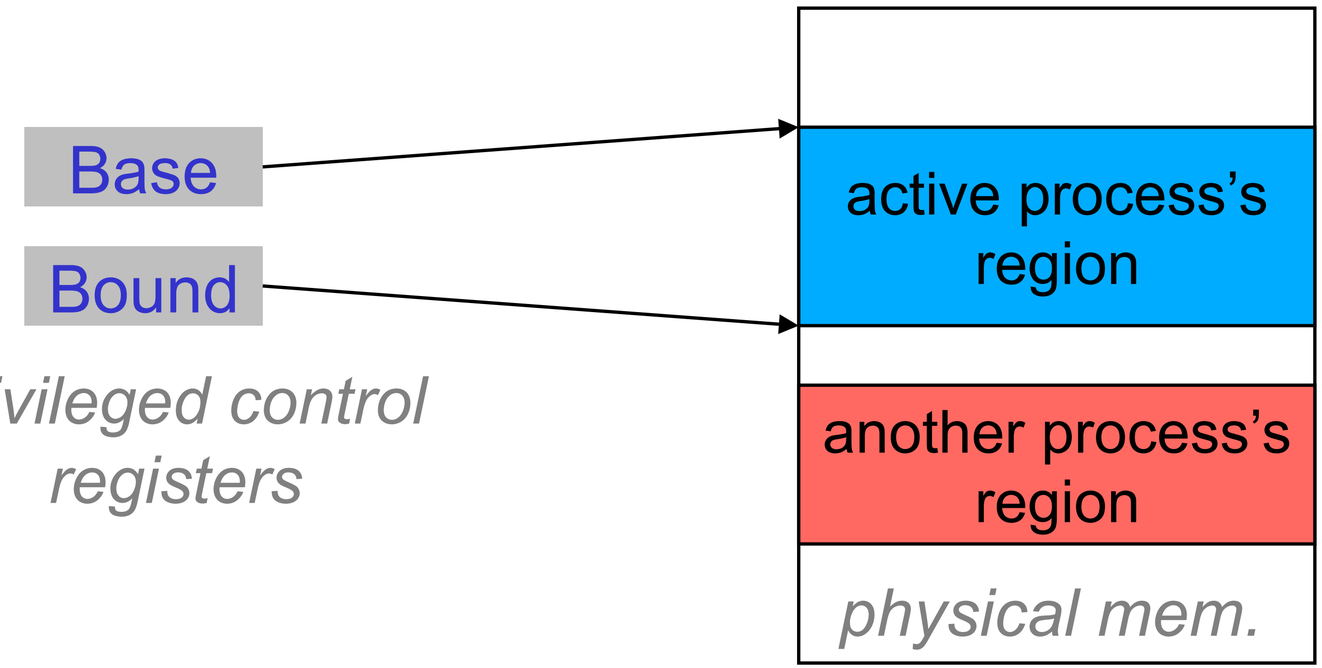
Base and Bound Registers

- *Multi-tasking* machines complicated things slightly
  - Each process is given a non-overlapping, contiguous physical memory region
  - *All data had to fit in that region!*
  - When a process is swapped in, the OS sets a register for the *base* of the process' memory region and the *bound* to the end
  - Hardware checked if addresses were in bound

Base and Bound Registers



Base and Bound Registers

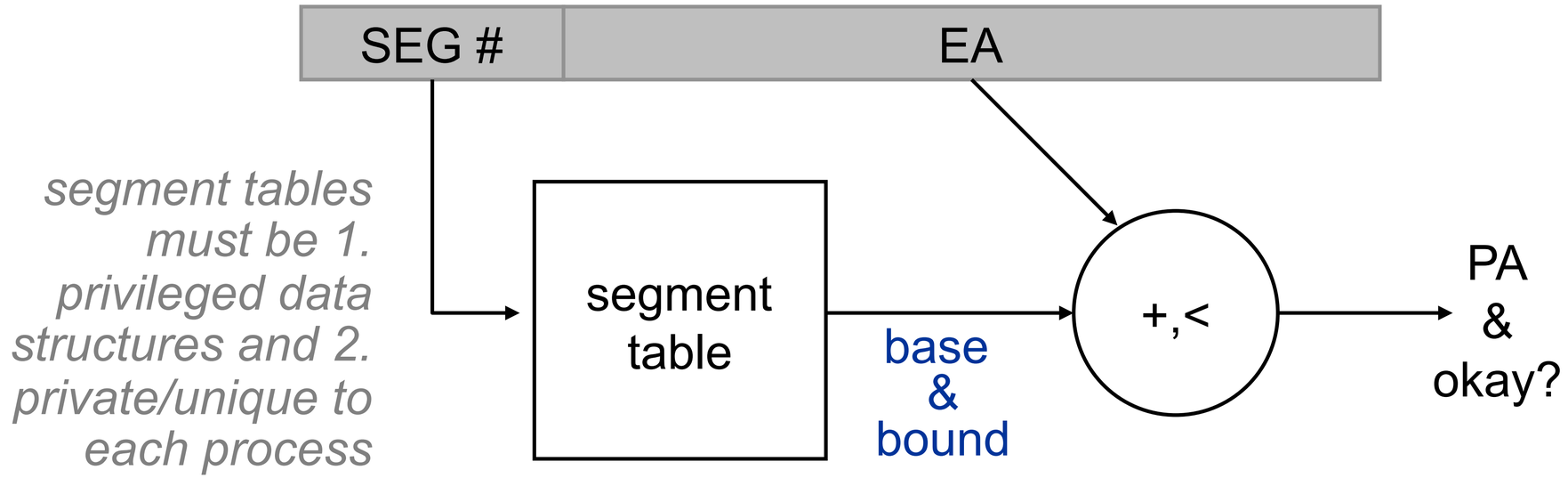


Why not use base and bound registers in today's machines?

Segmented Address Space

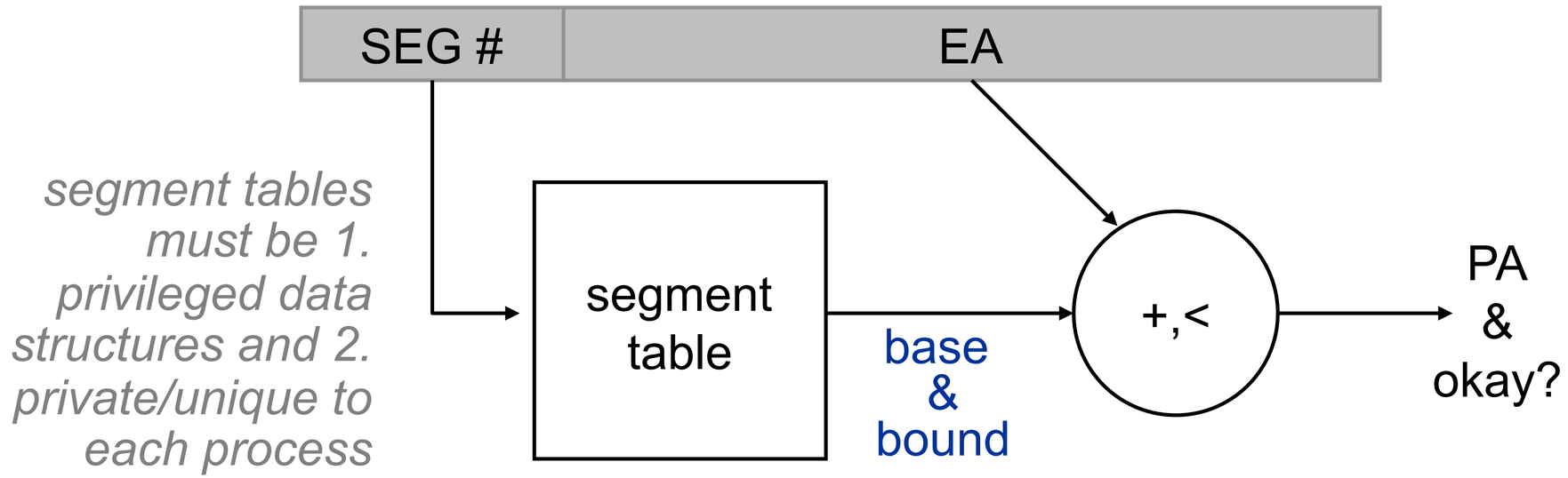
- More flexible than base and bound registers
- A segment is a single base and bound pair
- A process may possess multiple segments
  - Early machines used separate segments for code and data
  - What about stack? Add another segment! (Can get complex)

Segmented Address Space



segment tables must be 1. privileged data structures and 2. private/unique to each process

Segmented Address Space



Why not use a segmented address space in today's machines?

Adding Protection to Page-Based Virtual Memory

- Extend page table entries with permission bits
- OS page handling routine checks these before accessing / mapping
  - If violated, send process SIGSEGV (segmentation fault)

What We Learned Today

- Virtual memory allows processes to access memory *efficiently* (demand paging), *simply* (memory management), and *safely* (protection)
- Virtual memory uses DRAM like a cache for data on disk
  - But it's designed differently from on-chip caches
- How virtual memory is mapped to physical pages (page table)
- How virtual memory on disk is placed in physical memory (page fault)

What We'll Learn Next Time

- The details of translating a virtual address to a physical address
- How this data is stored and managed in on-chip caches