Course Wrap-Up

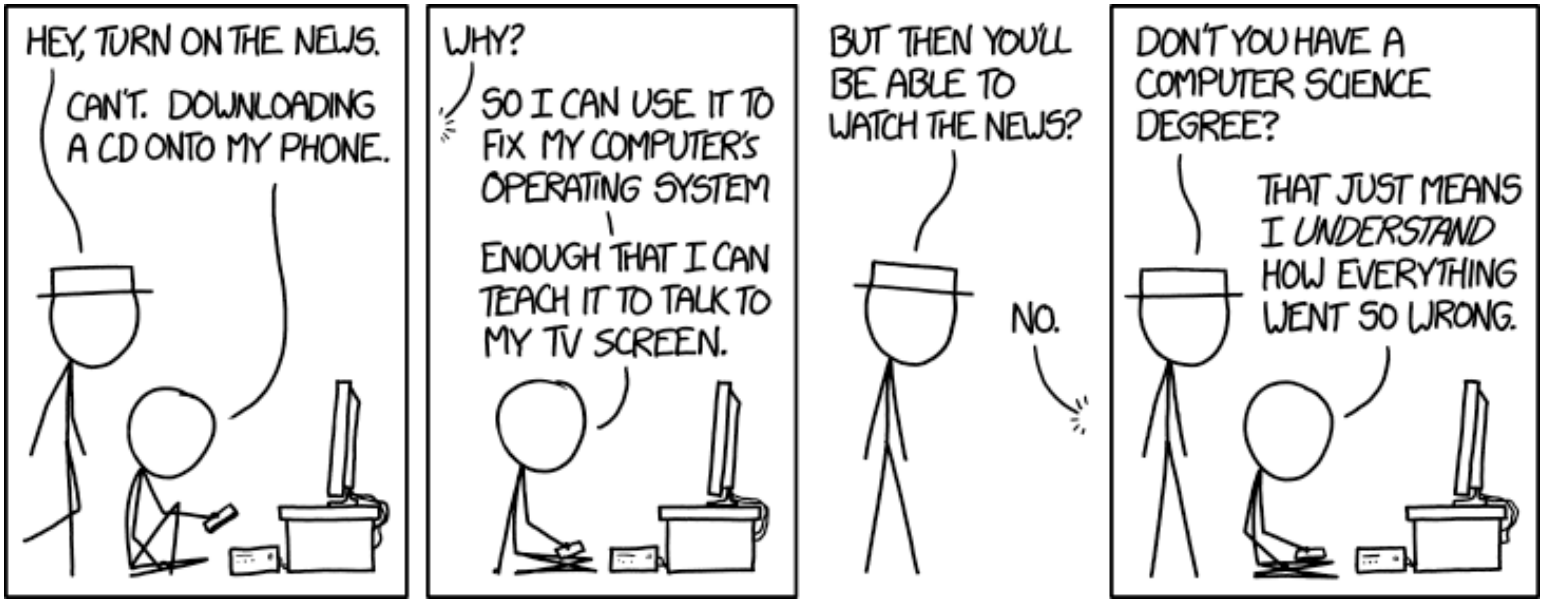
CSE 351 Spring 2020

Instructor:

Ruth Anderson

Teaching Assistants:

Alex Olshanskyy
Callum Walker
Chin Yeoh
Connie Wang
Diya Joy
Edan Sneh
Eddy (Tianyi) Zhou
Eric Fan
Jeffery Tian
Jonathan Chen
Joseph Schafer
Melissa Birchfield
Millicent Li
Porter Jones
Rehaan Bhimani



https://xkcd.com/1760/

Administrivia

- Lab 5 (on Mem Alloc) – NOW due Mon (6/08)
- Unit Summary #4 – due Wed (6/10)
- hw23 on Java and C – NOW due Thurs (6/11)
- Course evaluations now open
  - Please fill these out! Close Sunday (6/07)
  - Separate ones for Lecture and Section
- **You must log on with your @uw google account to access!!**
  - **Google doc** for 11:30 Lecture: https://tinyurl.com/351-06-05A
  - **Google doc** for 2:30 Lecture: https://tinyurl.com/351-06-05B

Today

- End-to-end Review
  - What happens after you write your source code?
    - How code becomes a program
    - How your computer executes your code
- Victory lap and high-level concepts (key points)
  - More useful for “5 years from now” than “next week’s final”

C: The Low-Level High-Level Language

- C is a "hands-off" language that "exposes" more of hardware (especially memory)
  - Weakly-typed language that stresses data as bits
    - Anything can be represented with a number!
  - Unconstrained pointers can hold address of *anything*
    - And no bounds checking – buffer overflow possible!
  - Efficient by leaving everything up to the programmer

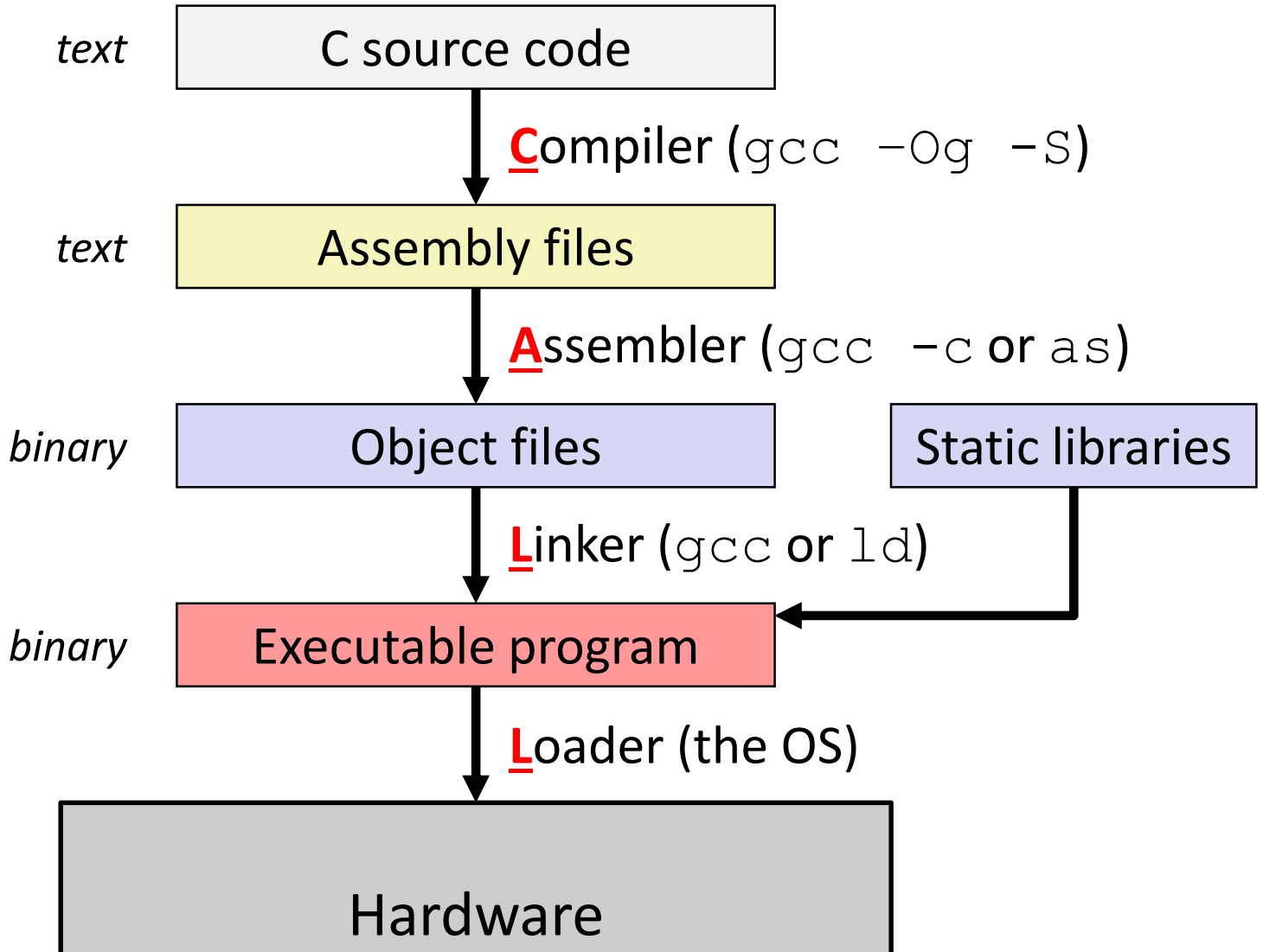
C Data Types

- C Primitive types
  - Fixed sizes and alignments
  - Characters (`char`), Integers (`short`, `int`, `long`), Floating Point (`float`, `double`)
- C Data Structures
  - Arrays – contiguous chunks of memory
    - Multidimensional arrays = still one continuous chunk, but row-major
    - Multi-level arrays = array of pointers to other arrays
  - Structs – structured group of variables
    - Struct fields are ordered according to declaration order
    - ***Internal* fragmentation:** space between members to satisfy member alignment requirements (aligned for each primitive element)
    - ***External* fragmentation:** space after last member to satisfy overall struct alignment requirement (largest primitive member)

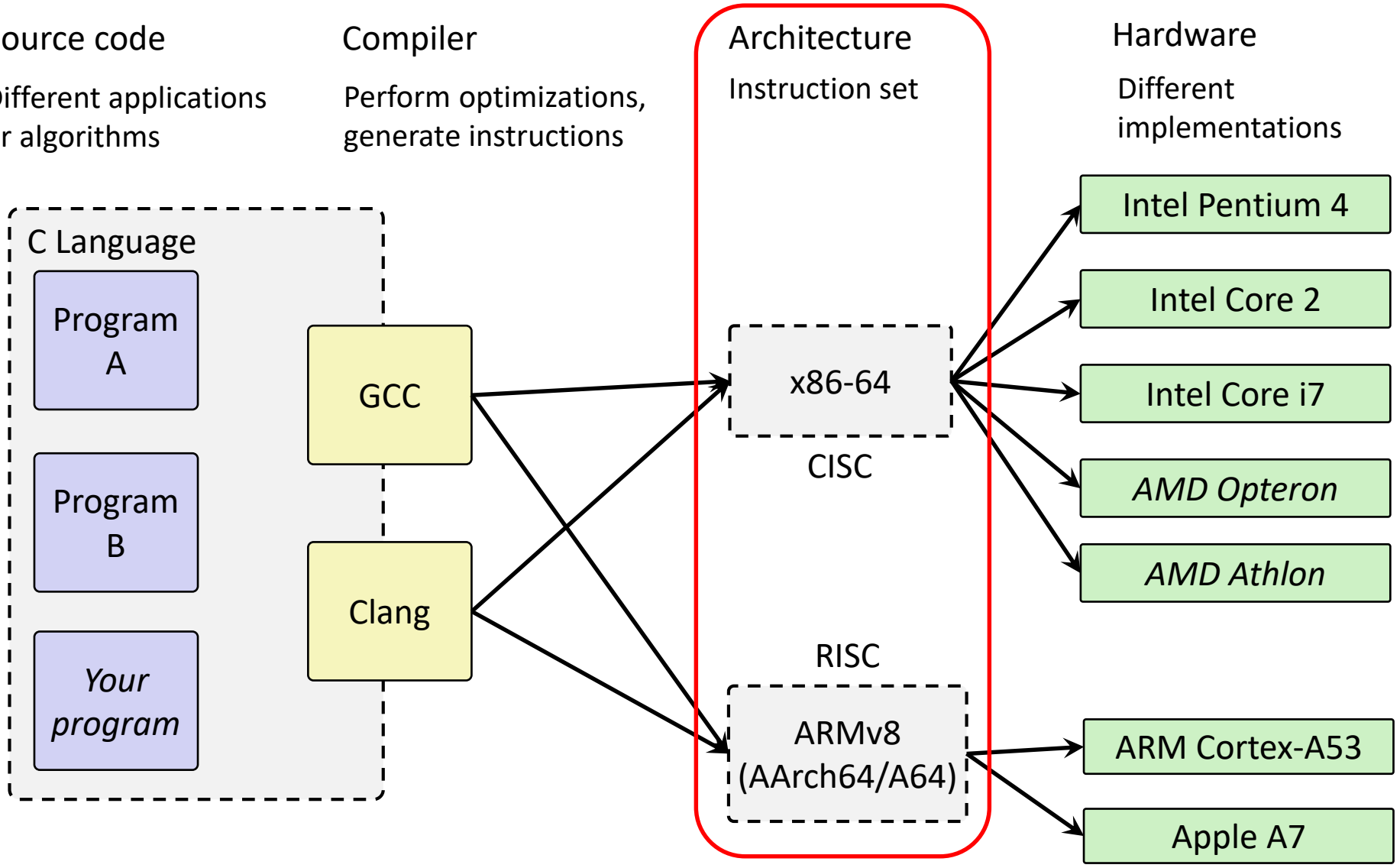
C and Memory

- Using C allowed us to examine how we store and access data in memory
  - Endianness (**only applies to memory**)
    - Is the first byte (lowest address) the least significant (little endian) or most significant (big endian) of your data?
  - Array indices and struct fields result in calculating proper addresses to access
- Consequences of your code:
  - Affects performance (locality)
  - Affects security
- But to understand these effects better, we had to dive deeper…

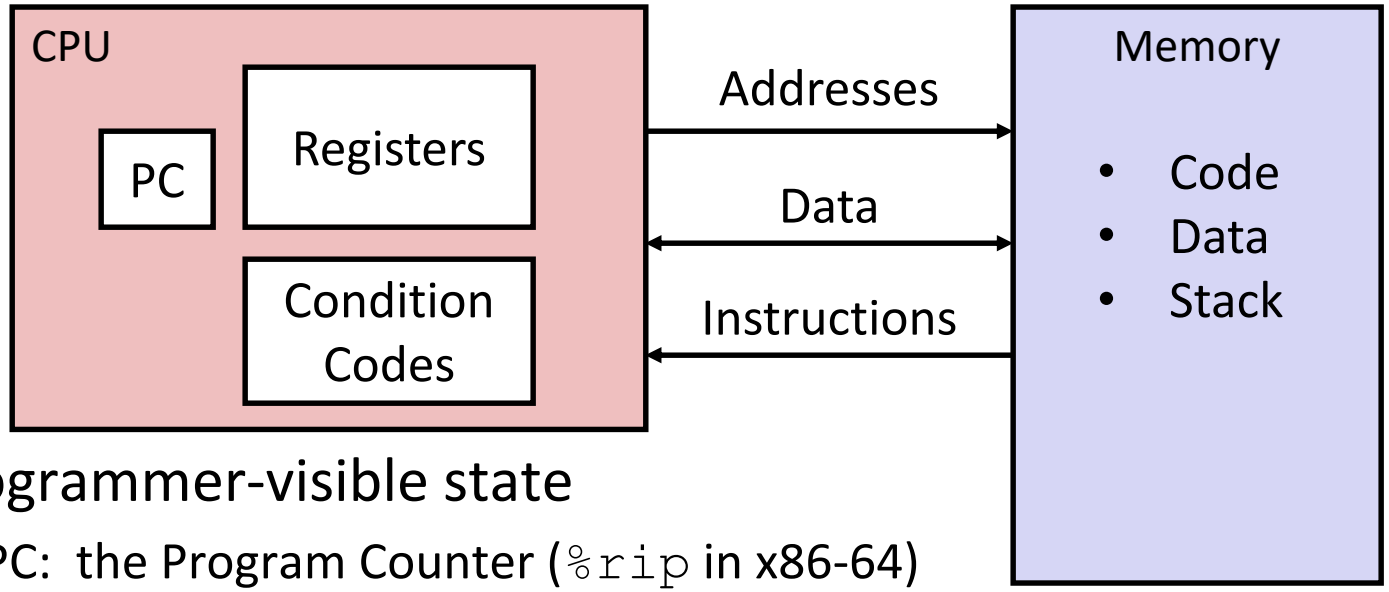
How Code Becomes a Program



Instruction Set Architecture

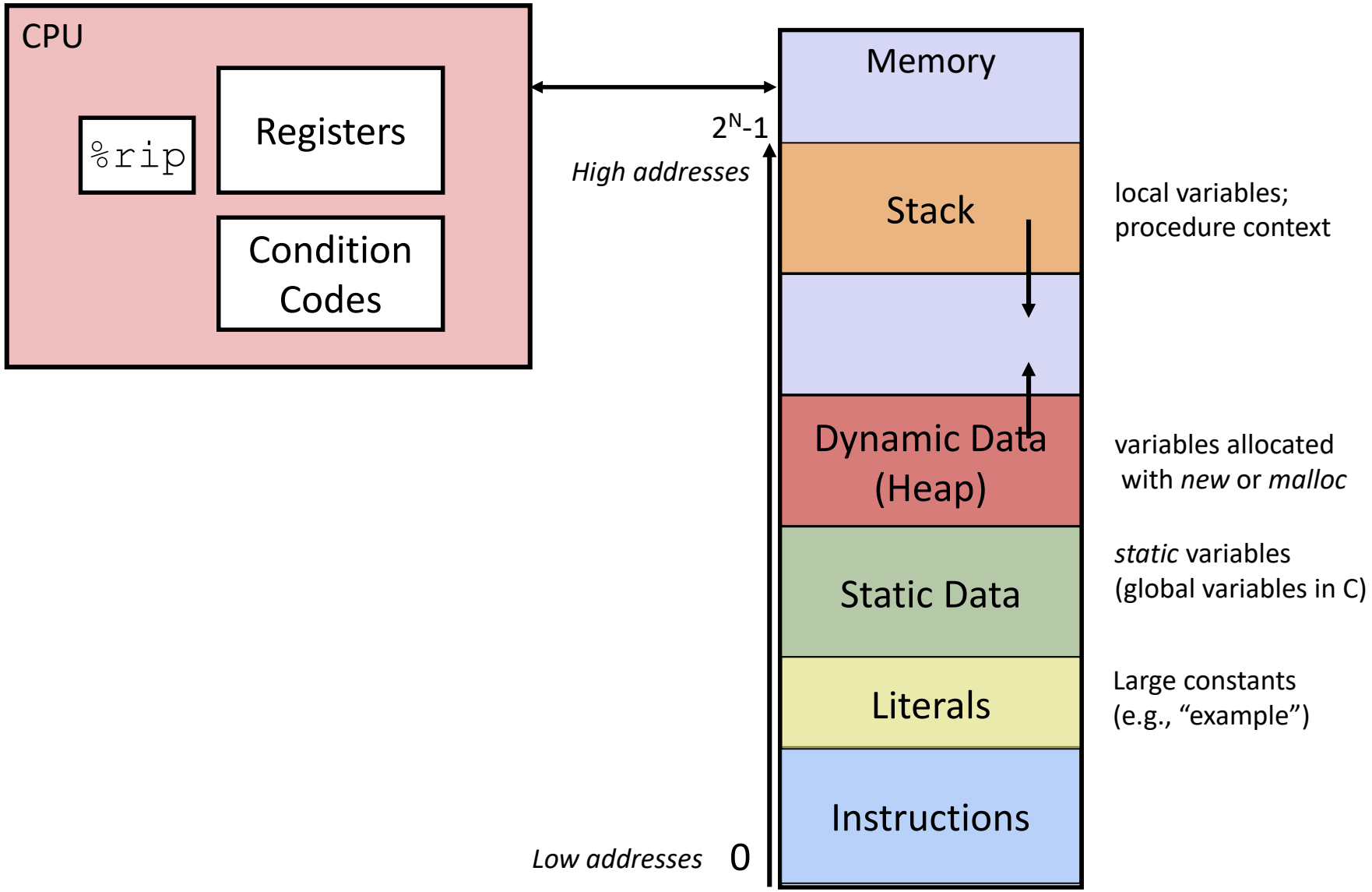


Assembly Programmer's View



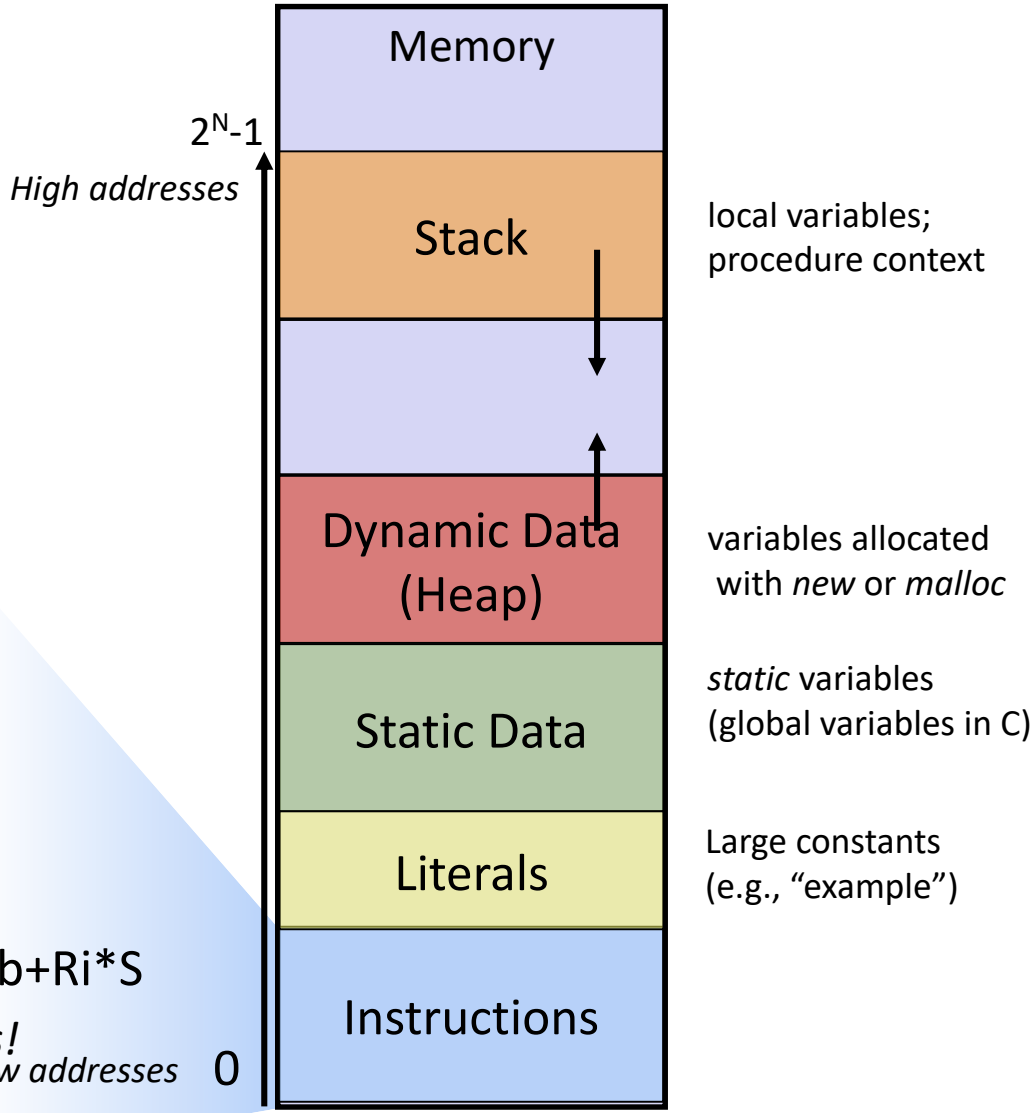
- Programmer-visible state
  - PC: the Program Counter (`%rip` in x86-64)
    - Address of next instruction
  - Named registers
    - Together in "register file"
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
- Memory
  - Byte-addressable array
  - Huge *virtual* address space
  - *Private, all to yourself…*

Program's View: Parts of Memory



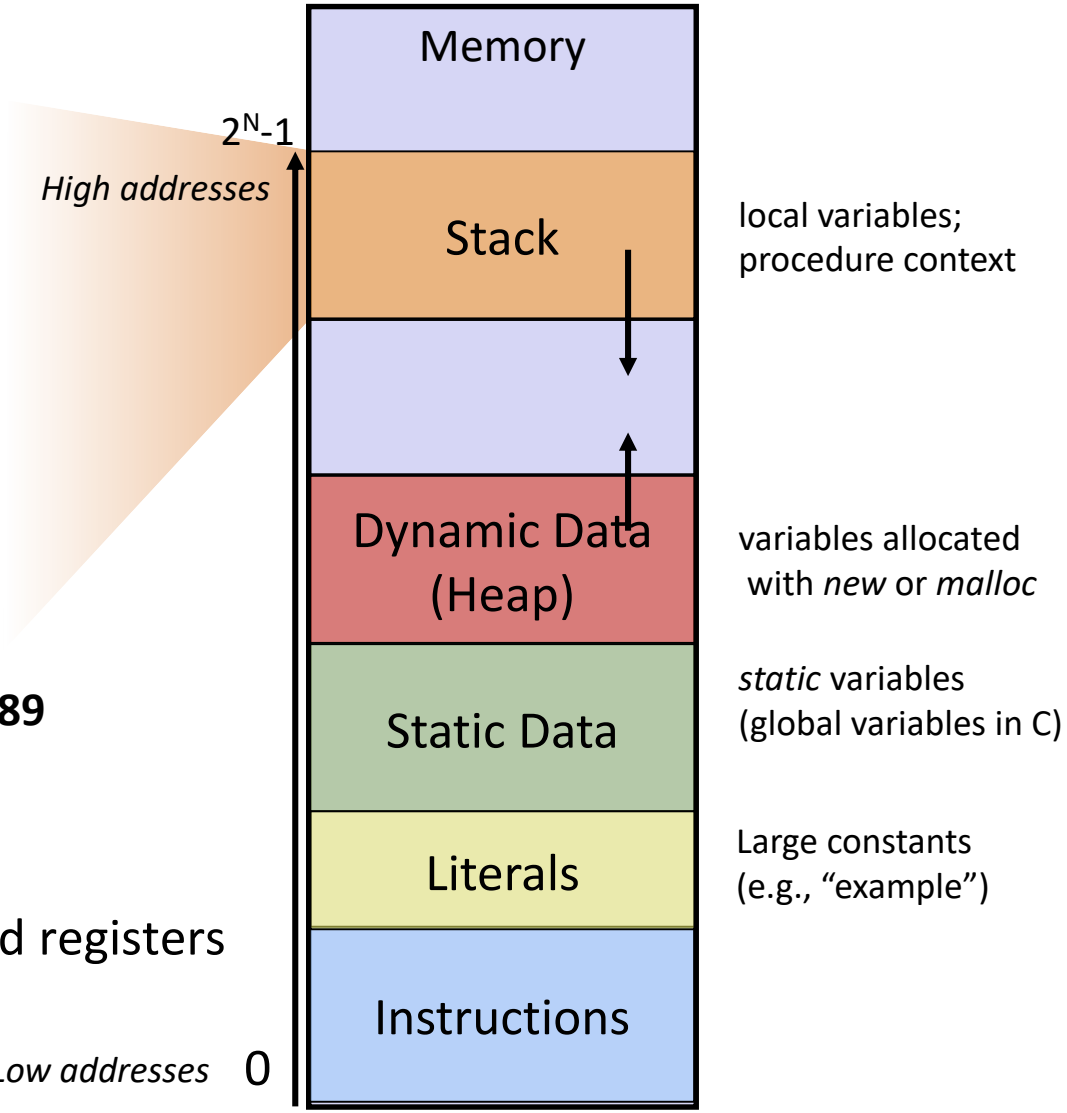
Program's View: Instructions

- Instructions
  - Data movement
    - `mov, movz, movz`
    - `push, pop`
  - Arithmetic
    - `add, sub, imul`
  - Control flow
    - `cmp, test`
    - `jmp, je, jgt, ...`
    - `call, ret`
- Operand types
  - Literal: `$8`
  - Register: `%rdi, %al`
  - Memory: D(Rb,Ri,S) = D+Rb+Ri*S
    - `lea`: *not a memory access!*



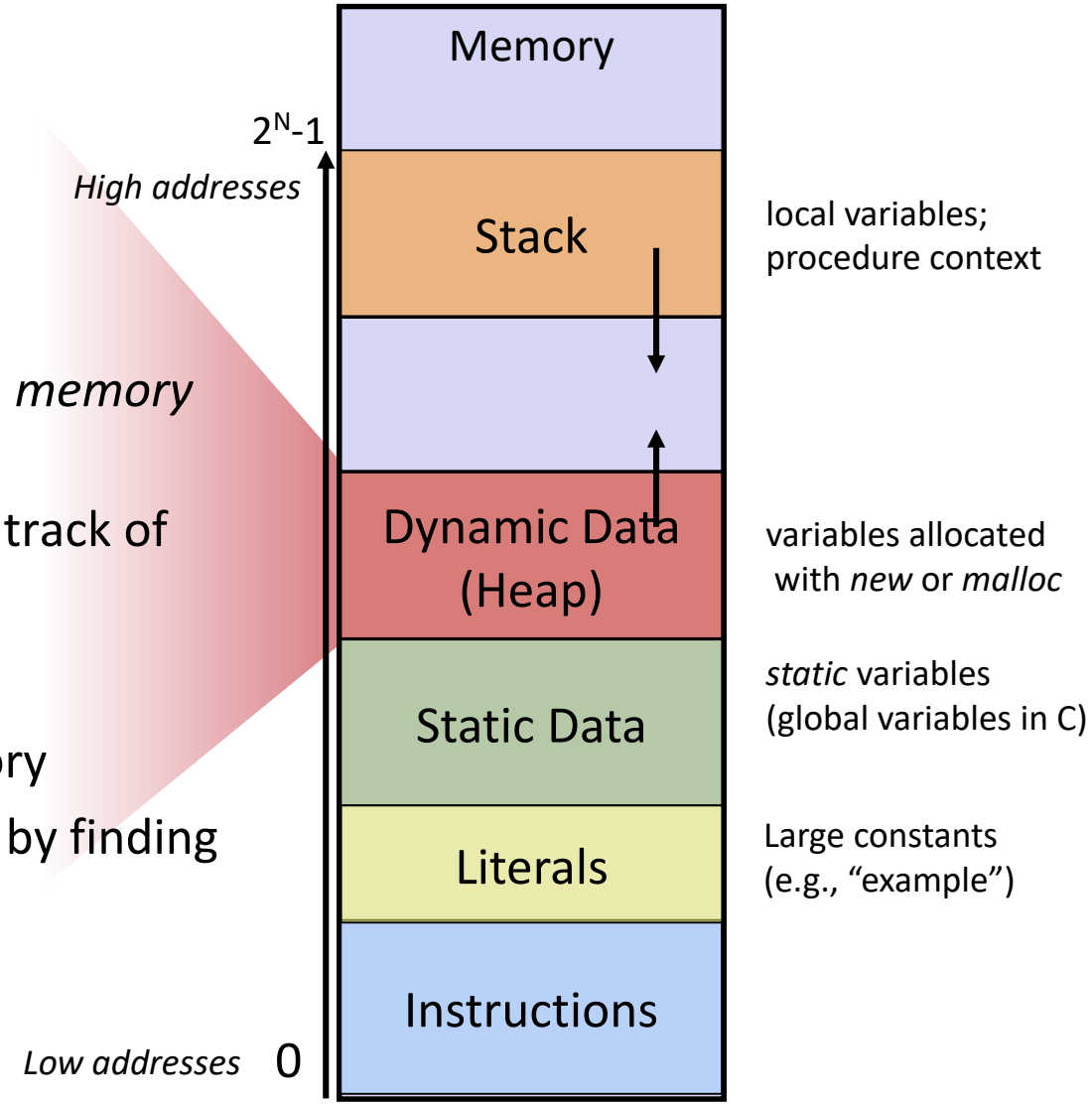
Program's View: Procedures & the Stack

- Procedures
  - Essential abstraction
  - Recursion…
- Stack discipline
  - Stack frame per call
  - Local variables
- Calling convention
  - How to pass arguments
    - **Di**ane's **Si**lk **D**ress **C**osts $**89**
  - How to return data
  - Return address
  - Caller-saved / callee-saved registers

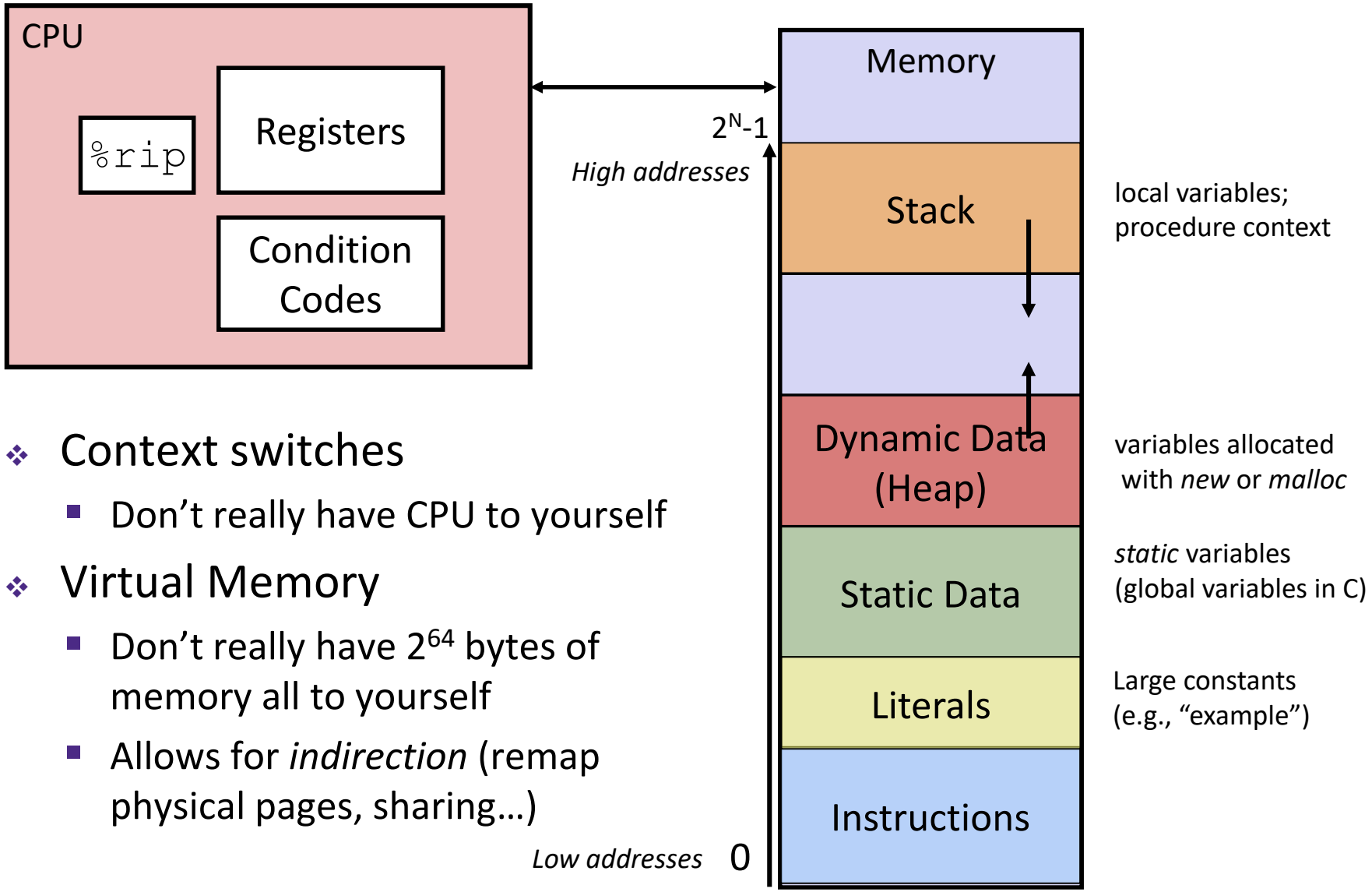


Program's View: The Heap

- Heap data
  - Variable size
  - Variable lifetime
- Allocator
  - Balance *throughput* and *memory utilization*
  - Data structures to keep track of free blocks
- Garbage collection
  - Must always free memory
  - Garbage collectors help by finding anything *reachable*
  - Failing to free results in *memory leaks*

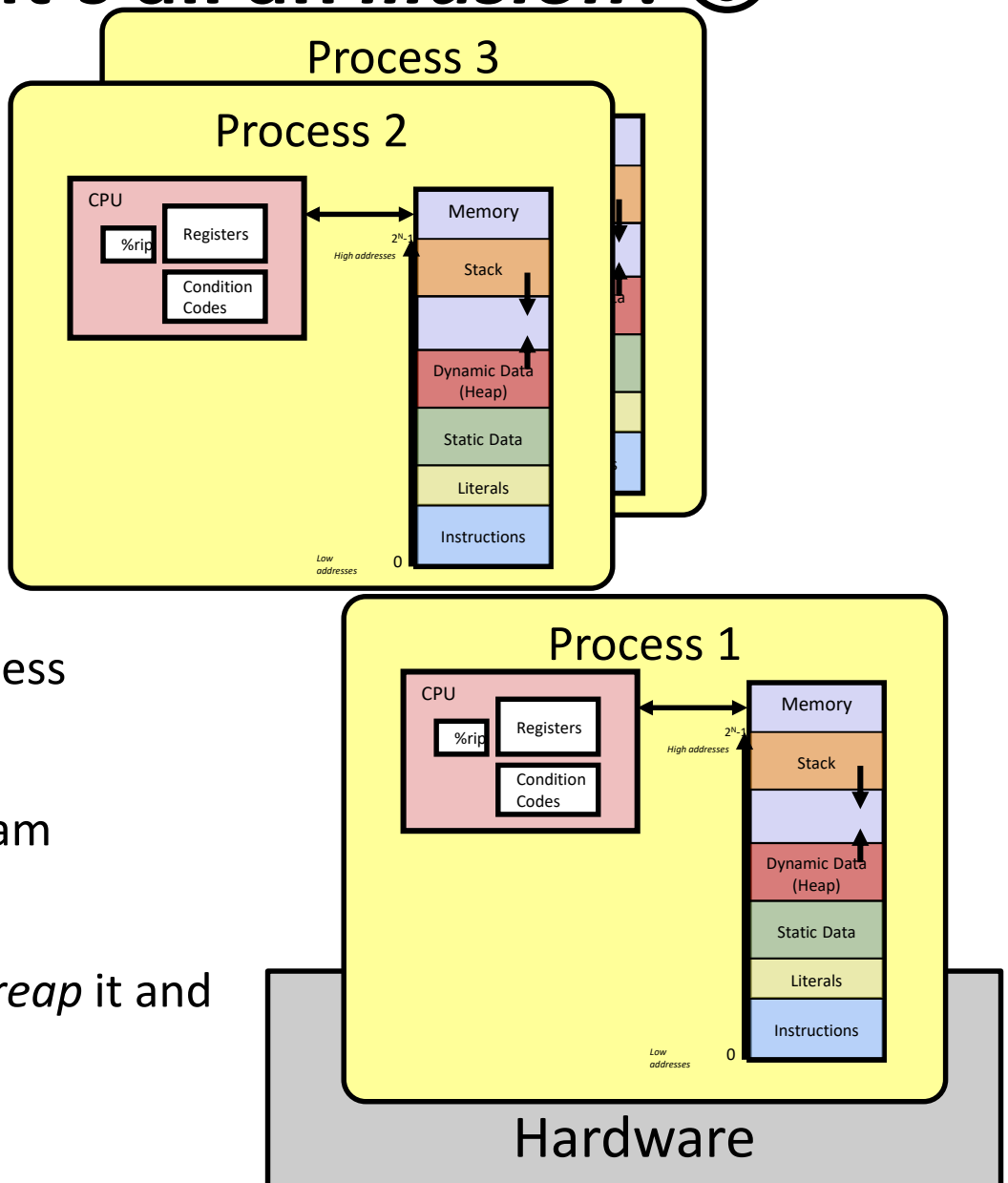


But remember… it's all an *illusion*! 😮



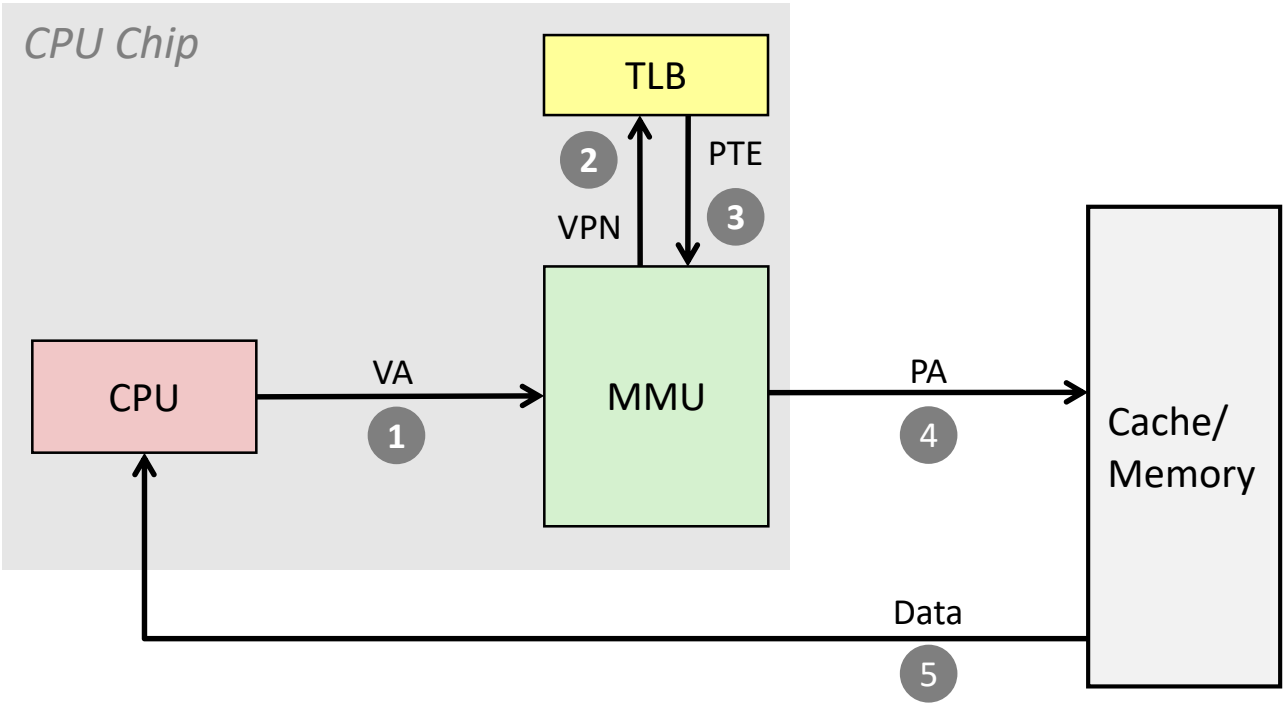
- Context switches
  - Don't really have CPU to yourself
- Virtual Memory
  - Don't really have 2^{64} bytes of memory all to yourself
  - Allows for *indirection* (remap physical pages, sharing…)

But remember… it's all an *illusion*! 😮



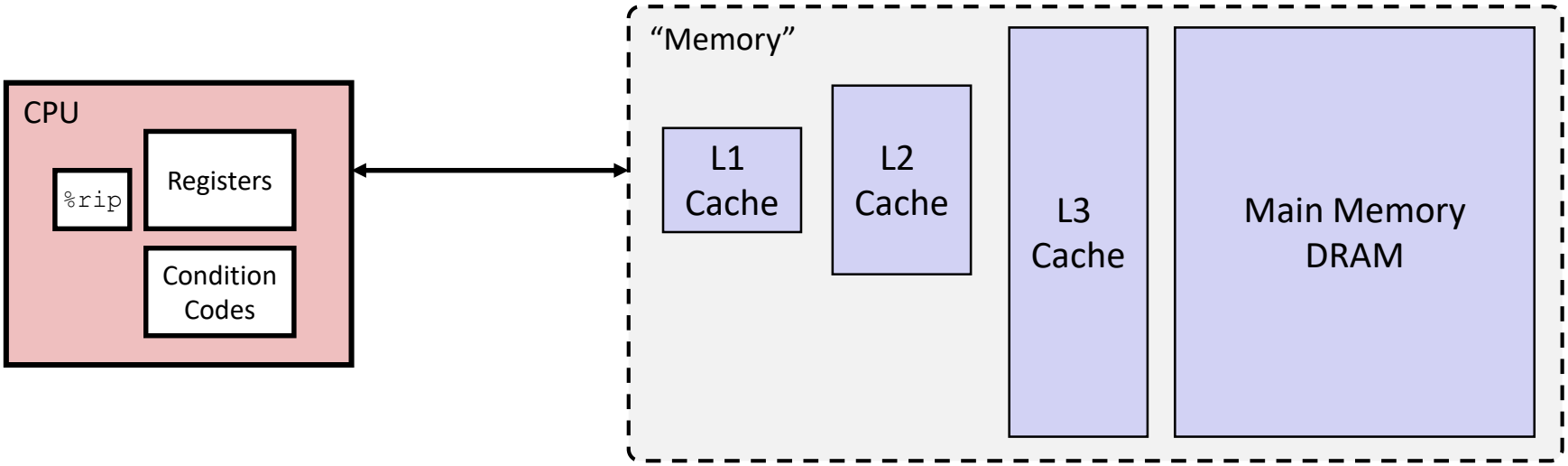
- `fork`
  - Creates copy of the process
- `execv`
  - Replace with new program
- `wait`
  - Wait for child to die (to *reap* it and prevent *zombies*)

Virtual Memory



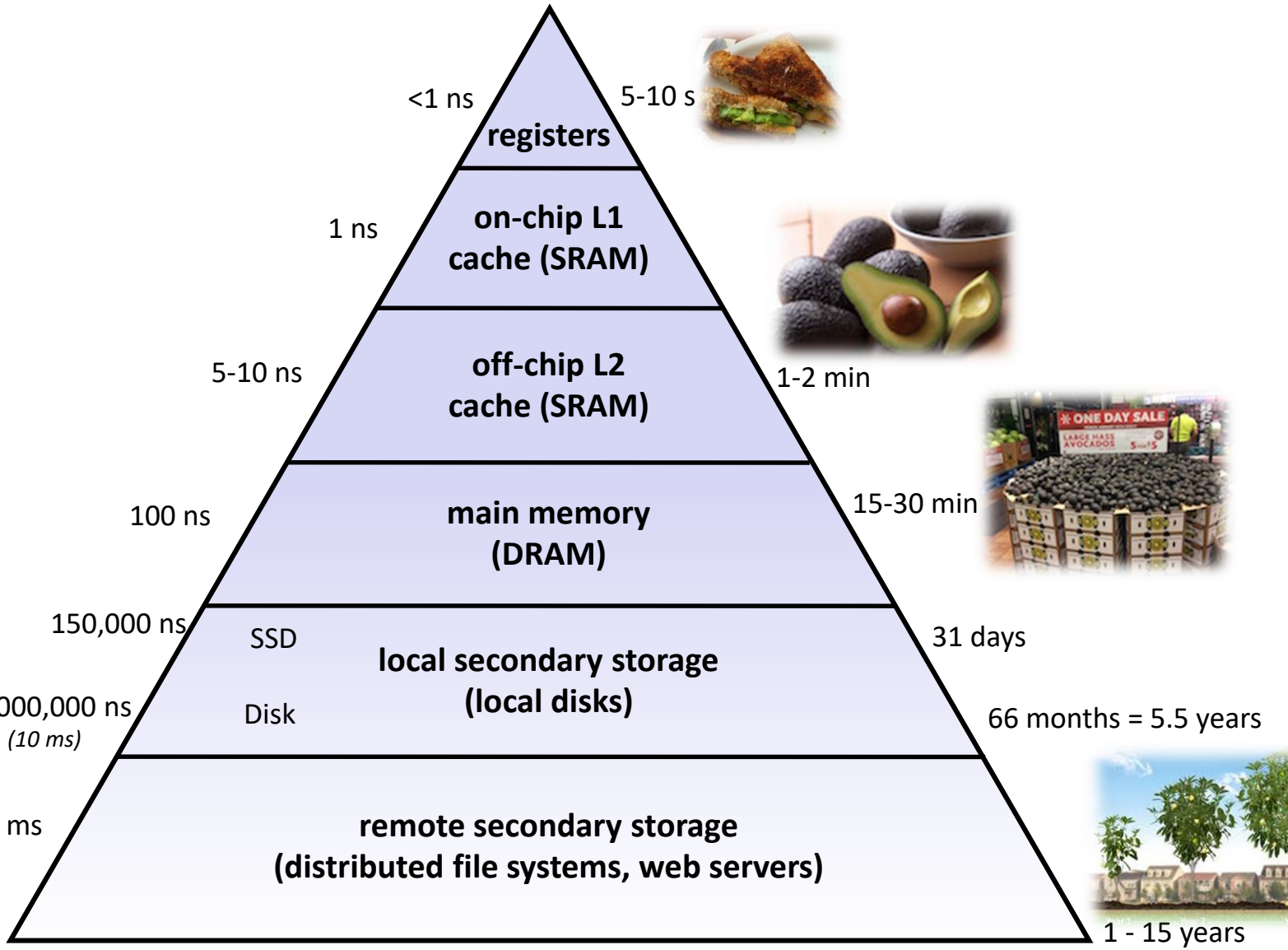
- Address Translation
  - Every memory access must first be converted from virtual to physical
  - *Indirection:* just change the address mapping when switching processes
  - Luckily, TLB (and page size) makes it pretty fast

But Memory is Also a Lie! 😮



- *Illusion* of one flat array of bytes
  - But *caches* invisibly make accesses to physical addresses faster!
- Caches
  - **Associativity** tradeoff with miss rate and access time
  - **Block size** tradeoff with spatial and temporal locality
  - **Cache size** tradeoff with miss rate and cost

Memory Hierarchy



Review of Course Themes

- Review course goals
  - They should make much more sense now!

Big Theme: Abstractions and Interfaces

- Computing is about abstractions
  - (but we can't forget reality)
- What are the abstractions that we use?
- What do you need to know about them?
  - When do they break down and you have to peek under the hood?
  - What bugs can they cause and how do you find them?
- How does the hardware relate to the software?
  - Become a better programmer and begin to understand the important concepts that have evolved in building ever more complex computer systems

Little Theme 1: Representation

- All digital systems represent everything as 0s and 1s
  - The 0 and 1 are really two different voltage ranges in the wires
  - Or magnetic positions on a disc, or hole depths on a DVD, or even *DNA*…
- "Everything" includes:
  - Numbers – integers and floating point
  - Characters – the building blocks of strings
  - Instructions – the directives to the CPU that make up a program
  - Pointers – addresses of data objects stored away in memory
- Encodings are stored throughout a computer system
  - In registers, caches, memories, disks, etc.
- They all need addresses (a way to locate)
  - Find a new place to put a new item
  - Reclaim the place in memory when data no longer needed

Little Theme 2: Translation

- There is a big gap between how we think about programs and data and the 0s and 1s of computers
  - Need languages to describe what we mean
  - These languages need to be translated one level at a time

- We know Java as a programming language
  - Have to work our way down to the 0s and 1s of computers
  - Try not to lose anything in translation!
  - We encountered C language, assembly language, and machine code (for the x86 family of CPU architectures)

Little Theme 3: Control Flow

- How do computers orchestrate everything they are doing?
- Within one program:
  - How do we implement if/else, loops, switches?
  - What do we have to keep track of when we call a procedure, and then another, and then another, and so on?
  - How do we know what to do upon "return"?
- Across programs and operating systems:
  - Multiple user programs
  - Operating system has to orchestrate them all
    - Each gets a share of computing cycles
    - They may need to share system resources (memory, I/O, disks)
  - Yielding and taking control of the processor
    - Voluntary or "by force"?

Course Perspective

- CSE351 will make you a better programmer
  - Purpose is to show how software really works
    - Understanding of some of the abstractions that exist between programs and the hardware they run on, why they exist, and how they build upon each other
  - Understanding the underlying system makes you more effective
    - Better debugging
    - Better basis for evaluating performance
    - How multiple activities work in concert (e.g. OS and user programs)
  - "Stuff everybody learns and uses and forgets not knowing"

- CSE351 presents a world-view that will empower you
  - The intellectual and software tools to understand the trillions+ of 1s and 0s that are "flying around" when your program runs

Courses: What's Next?

- Staying near the hardware/software interface:
  - **CSE369/EE271:** Digital Design – basic hardware design using FPGAs
  - **CSE474/EE474:** Embedded Systems – software design for microcontrollers
- Systems software (CSE majors/non-majors courses)
  - **CSE341/CSE413:** Programming Languages
  - **CSE332/CSE373:** Data Structures and Parallelism
  - **CSE333/CSE374:** Systems Programming – building well-structured systems in C/C++
- Looking ahead
  - **CSE401/CSE413:** Compilers (pre-reqs: 332)
  - **CSE451:** Operating Systems (pre-reqs: 332, 333)
  - **CSE461:** Networks (pre-reqs: 332, 333)

Thanks for a great quarter!

- Huge thanks to your awesome TAs!

- Don’t be a stranger!
  - I hope to see you in a course sometime in the future!