Lecture 01
Introduction

CS213 – Intro to Computer Systems

Branden Ghena – Winter 2024

Slides adapted from:
St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Welcome to CS213!

- In brief: how *does* a computer work anyway?

- We will explore that question across four major sections:
    - **Representations** of information on a computer
    - How the **machine** executes software
    - How **memory** is organized
    - How the **operating system** manages this all for efficiency and security

Asking questions, four ways

1. You can always ask questions during lecture!
    - I’ll let you know if I need to move on for now and answer you after class

2. We’ll take breaks during lecture
    - I’ll pause after each break to see if any questions came up

3. I will hang out after class for questions
    - Plenty of time to answer everyone

4. You can always ask questions on Piazza too

    The class message board app

Branden Ghena (he/him)

- Assistant Faculty of Instruction
- Education
  - Undergrad: Michigan Tech
  - Master's: University of Michigan
  - PhD: University of California, Berkeley
- Research
  - Resource-constrained sensing systems
  - Low-energy wireless networks
  - Embedded operating systems
- Teaching
  - Computer Systems
    - CS211: Fundamentals of Programming II
    - CS213: Intro to Computer Systems
    - CS343: Operating Systems
    - CE346: Microprocessor System Design
    - CS397: Wireless Protocols for the IoT

Tock

Things I love

Cosmere

twitch

Critical Role

Michigan Football GO BLUE!

Today’s Goals

- Introduce the theme and goals of the course
- Describe how this class is going to function
- Discuss how a computer system works at a high level
- Begin exploring how computers represent information with bits and bytes

Outline

- **Course Themes**
- Logistics
- Running a program
- Representing numbers with binary

Convenient computing

- Computers operate on integers, reals, structs, arrays, etc.
- Computers operate on variables and functions
- Computers execute conditionals, loops, etc.
- Memory is an infinite bag of objects my program can allocate
- Memory doesn’t have to be shared with any other program
- Memory is always equivalently fast to access
- Etc.

Convenient **illusions** in computing

- Computers operate on integers, reals, structs, arrays, etc.
- Computers operate on variables and functions
- Computers execute conditionals, loops, etc.
- Memory is an infinite bag of objects my program can allocate
- Memory doesn’t have to be shared with any other program
- Memory is always equivalently fast to access
- Etc.

- None of these are actually true!
  - But we usually program as if they were, and we get away with it!
  - What’s going on?

The power of abstraction

- These illusions are called ***abstractions***
- They approximate reality, but leave out details
  - Instead, they provide an *interface* that we can work and think with
- We can forget about those details, and be more productive

- Abstractions we love
  - Abstract data types
  - Asymptotic analysis
  - High-level programming languages
  - Operating systems
  - Etc.

The Limits of Abstraction

- Sometimes, abstractions break down
  - Their implementation is buggy
  - Mismatch between expected interface and implementation
  - Their performance is inadequate
  - We need control over the details they hide
  - Security concerns make these details important

- At that point, details come rushing back
  - Can’t pretend they don’t exist anymore
  - We must know how to deal with them

- This class prepares you to be ready when that happens

When do abstractions break?

- Let’s talk about some real-world examples of “broken” software
  - That broke because of how the underlying system actually works

1. Dates and Times
2. Network Security
3. Simple Programming Styles

Complicated designs fail in unexpected ways

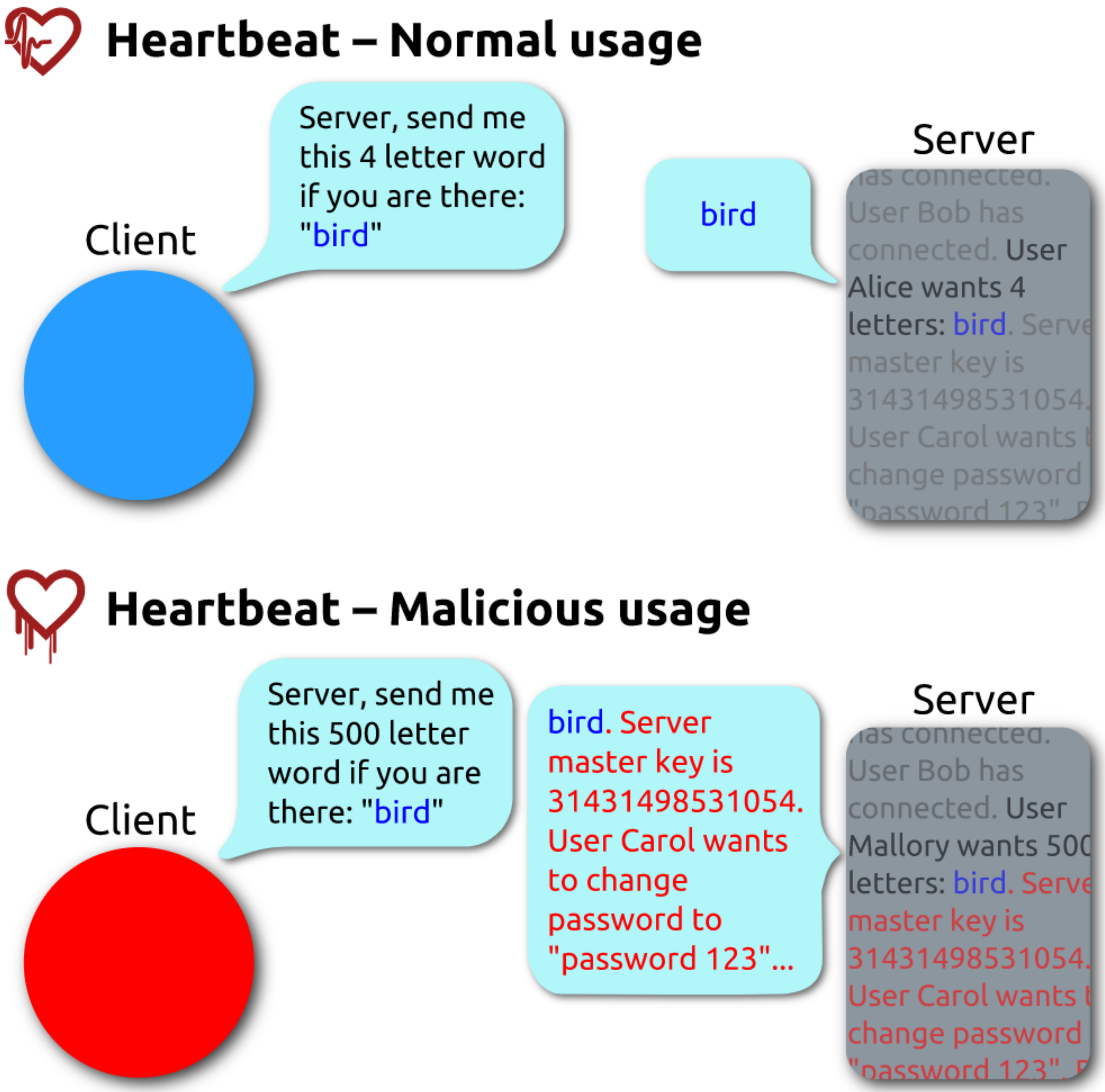
- Some software engineers at Microsoft came up with a cute way of storing dates
  - Two-digit year, month, date, hour, minute concatenated into a 10-digit number
  - Example: 2005230710 -> May 23, 2020 at 7:10 AM

- Stored as a 32-bit signed number (`int`)
  - Maximum value: 2147483647

- Result: Starting January 1st, 2022, Microsoft Exchange email servers could no longer send email
  - 2201010001 is greater than the largest 32-bit number
  - Microsoft had to issue an emergency patch

Simple bugs can result in massive vulnerabilities

- 2014 vulnerability in OpenSSL
- Clients can check if server is active by sending a message and listening for echoed response
- C library forgot to check bounds of array and could be abused to return important memory



Hardware realities impact software performance

- Abstracted lower-level details can affect performance a lot!
- Question: does the order of iterating through an array matter?
  - Each column in a row OR each row in a column?
- Answer: right code is 10-32 times slower on Intel systems
  - Due to cache design and performance

```
void copyij(int src[4096][4096],
            int dst[4096][4096])
{
  for (int i=0; i<4096; i++){
    for (int j=0; j<4096; j++){
      dst[i][j] = src[i][j];
    }
  }
}
```

```
void copyji(int src[4096][4096],
            int dst[4096][4096])
{
  for (int j=0; j<4096; j++){
    for (int i=0; i<4096; i++){
      dst[i][j] = src[i][j];
    }
  }
}
```

CS213 goals

1. Break through abstractions to understand how computer processors and memories affect software design and performance

2. Introduce concepts of "computer systems" areas:
   - Architecture, Compilers, Security, Networks, Operating Systems, etc.

Course design goal

- Most systems courses are builder-centric
  - **Computer Architecture**: design a pipelined processor in Verilog
  - **Operating Systems**: implement portions of an operating system
  - **Compilers**: write a compiler for a simple language
  - **Networking**: implement and simulate network protocols
  - Fun, for sure
    - But ultimately, many more of you will ***build on*** systems
    - Rather than ***build systems*** directly

- This course is programmer-centric
  - Purpose is to show that by knowing more about the underlying system, one can be more effective as a programmer
  - Not just a course for dedicated hackers
    - **We want to bring out the hacker in everyone!**

Outline

- Course Themes
- **Logistics**
- Running a program
- Representing numbers with binary

Course Staff

- TA (1)
  - Mohammad Kavousi
    - PhD student in Computer Science

- PMs (15):
  - Alex Kang
  - Ethan Havemann
  - Jerry Han
  - Harita Duggirala
  - Joseph Grantham
  - Rachana Aluri
  - Ryan Wong
  - Vikram Achuthan
  - Elena Fabian
  - Evan Waite
  - Hannia Valera
  - Jay Park
  - Max Glass
  - Robert Pritchard
  - Daniel Lee

Their role: support student questions via office hours and piazza

Course details - how to learn stuff

- Lectures: here in class, Tuesdays and Thursdays
  - Please attend and ask questions!
  - Panopto tab on Canvas should have best-effort recordings (a few hours later) and I also post the slides right before class

- Textbook:
  - Computer Systems: A Programmer's Perspective **3rd Edition**
  - A *very* useful reference

- Office hours: (starting next week)
  - Likely a mix of mostly in-person and some online
  - More info will be posted to Piazza when schedule is ready
    - Can reach out on Piazza to schedule a meeting too

Asking questions

- Class and office hours are always an option!
  - We can do extra questions right after class too
- Piazza: (similar to Campuswire)
  - Post questions
  - Answer each other’s questions
  - Find lab partners
  - Find posts from the course staff
  - Post private info just to course staff
- Please do not email me! Post to Piazza instead!
  - I’ll be updating roster again a few times

Grades

- Grade breakdown
  - 50% Programming Labs (4 labs at 12.5% each)
  - 20% Homeworks (4 homeworks at 5% each)
  - 15% Midterm Exam 1
  - 15% Midterm Exam 2

- Exact number to letter mapping is a little flexible
  - But this course is NOT curved

Programming Labs

- Four labs
  1. Pack Lab – manipulate bits and bytes of a file
  2. Bomb Lab – deconstruct software to understand it
  3. Attack Lab – exploit security vulnerabilities in software
  4. SETI Lab – make software faster with concurrency

- Work on these preferably as a group of two
  - Work together and don’t split up assignments (otherwise you won’t learn)
  - Individual is acceptable but less good
  - We’ll do a pairing survey if you don’t already have a partner in mind

- Very different from CS211 style projects
  - Emphasis on the thinking rather than the programming

Lab difficult ranking (ranked by past PMs)

Lab	Difficulty (out of 10)	What is challenging about it?
1. Pack Lab	6	C programming
2. Bomb Lab	8	Interpreting assembly code
3. Attack Lab	5	Debugging what’s going wrong
4. SETI Lab	8	C programming AND big codebase

- Be warned: Bomb Lab and SETI Lab are during the busiest parts of the quarter
  - Midterm exam season and last week of quarter, respectively

Homeworks

- Worksheet-style practice problems to help you actually understand what’s going on and practice for exams
  - This class can feel a little like a math class sometimes
  - (But not all the time! I promise)

- Four homeworks that cover class topics
  - The first releases on Thursday!

- Important practice, but not meant to be too difficult
  - Last quarter 90% of the class had an A- or better on these

Midterm Exams

- First midterm exam will be during class time

- Second midterm exam will be during exam week
  - **Important:** Tuesday of exam week is our scheduled slot

- Not cumulative, second midterm is second half of class
  - But material in this class builds on itself…

- Exams are serious in CS213. They're how we judge your individual understanding

Three special policies in CS213

1. Minimum midterm average rule

2. Late policy

3. Slip days

Weighing midterm exams

- A concern in CS213: we allow lots of group work
  - But we need to individually assess you as well
  - Especially to make sure that you're ready for future systems courses
- Normal way to do this is make the exams a huge portion of your grade (like 50%+)
  - We really don't want to do that in CS213
  - Not fun to have your letter grade decided by a single test
- Compromise: require a minimum average exam grade to pass
  - But still keep exam weights low so most of your grade is the projects

Minimum Midterm Average Rule

- To pass, you need at least a 65% average across the two exams
  - Overall exam averages are usually in the high 70%s
  - Examples: 60% and 70% or 80% and 50% or 65% and 65%
- BUT, we do want to reward improvement
  - The average rule waived if your **second midterm is 85% or higher**
  - 30% and 85% (would be 57% average) has no penalty
  - Bottom line: either do well **or** show significant improvement
- By the numbers:
  - In Fall 2023, it affected 6 students out of 140
  - However: ~15 more dropped after first midterm

Late Policy

- You can submit homeworks and labs late

- 20% penalty to maximum grade per day late
  - Example: three days late means maximum grade is 40%

- There are exceptions to this:

1. We will be flexible with deadlines for problems outside of your control
   - Sick, family emergency, broken computer
   - Contact me (via Piazza)

Slip Days

2. Slip days let you turn in a homework late and receive no penalty

- Each student gets **3 slip days**
  - Apply to **homeworks and labs**
  - You don't need to tell us you're using them, we'll just automatically apply them at the end of the year
  - Be sure to coordinate about them on partner assignments

- Examples:
  - Turn in homework 1 three days late
  - Turn in homework 4 two days late and SETI lab one day late
  - Turn in homework 2 four days late with only a one-day penalty

Academic Integrity

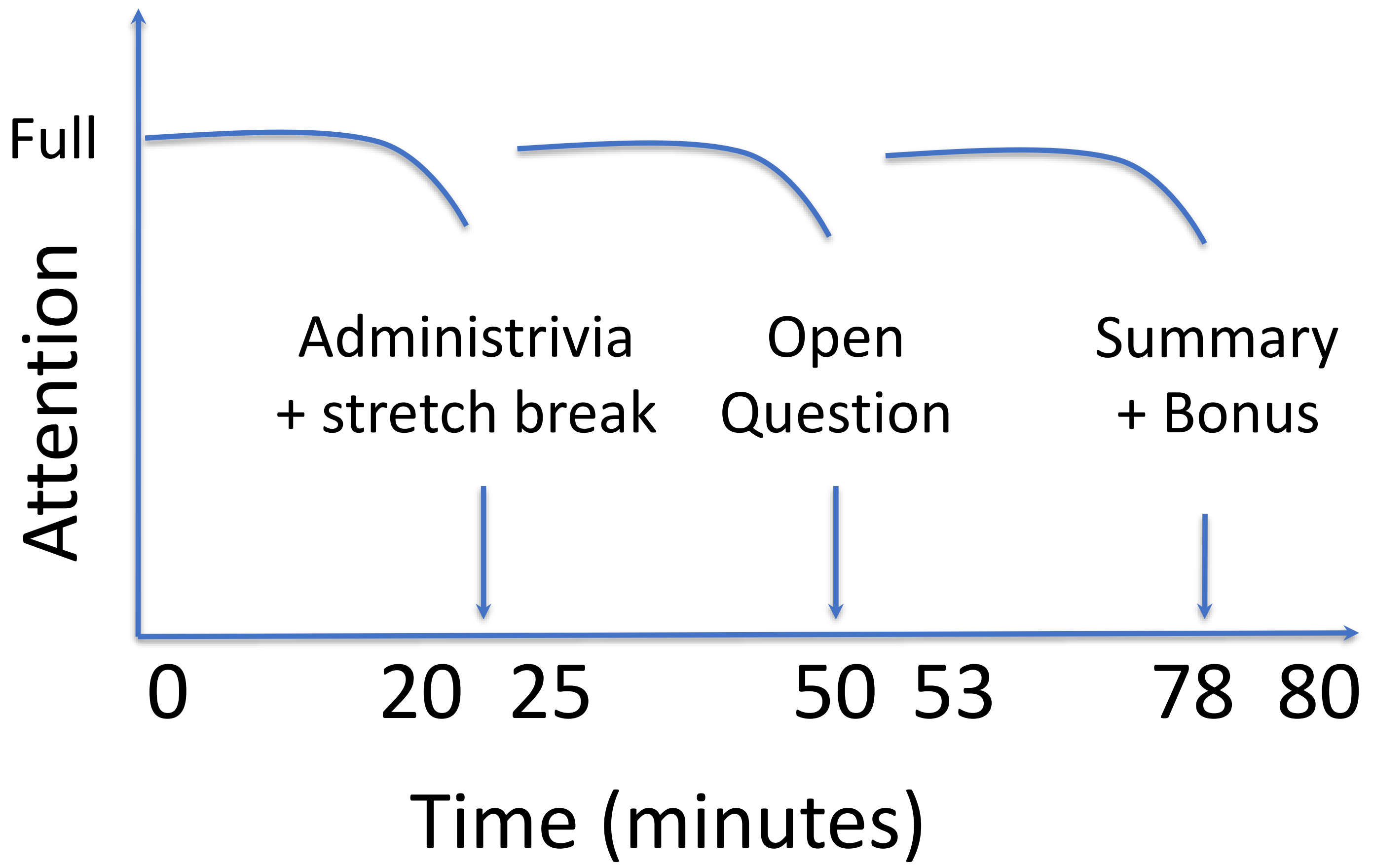
- This is something I take very seriously

- Collaboration good; plagiarism bad
  - You should know where that line is, and be nowhere near it
  - When in doubt, ask the instructor *before* you do something you're not sure about

- At no point should you see someone else's solutions
  - Not your colleagues', not your friends', not your cousin's, not something you found online

- I report everything suspicious to the dean

Break + Architecture of a lecture



Expectations

- This class is **hard**
  - And it’s hard in a different way. Lots of new material that builds on itself
  - You have an opportunity to learn a lot from it

- I’m confident that you can all succeed
  - Labs, Homeworks, Lecture, Office Hours are all designed to support you

- You’ll gain a much deeper understanding of how computers operate
  - Maybe it’s not for you, maybe you’ll love it

How to succeed in this class

- Come to lecture
- Ask questions
- Consult the textbook for clarity and practice
- Start assignments early
- Stay on top of the material

Outline

- Course Themes
- Logistics
- **Running a program**
- Representing numbers with binary

Hello World

- What happens when you run “hello” on your system?
  - And *why* does it happen?

```
/*
 * hello world
 */
#include <stdio.h>

int main()
{
   printf(“hello, world\n”);
}
```

- **Goal**: introduce key concepts, terminology, and components

Compiling `hello`

- Compiling **`hello`**

```
unix> gcc –o hello hello.c
```

- GCC is our compiler

1. It takes our source code (**`hello.c`**)
   - A text file containing characters
   - Text file = readable by humans
2. And translates (compiles) it into **assembly code**
   - A text representation of x86 instructions
   - Here, not explicitly stored in a file
   - We'll be working with assembly a lot this quarter
3. Then translates (assembles) that into an executable (**`hello`**)
   - A binary file containing x86 **machine code**
   - Binary file = not meant to be read by humans (but sometimes we have to)

Running **`hello`**

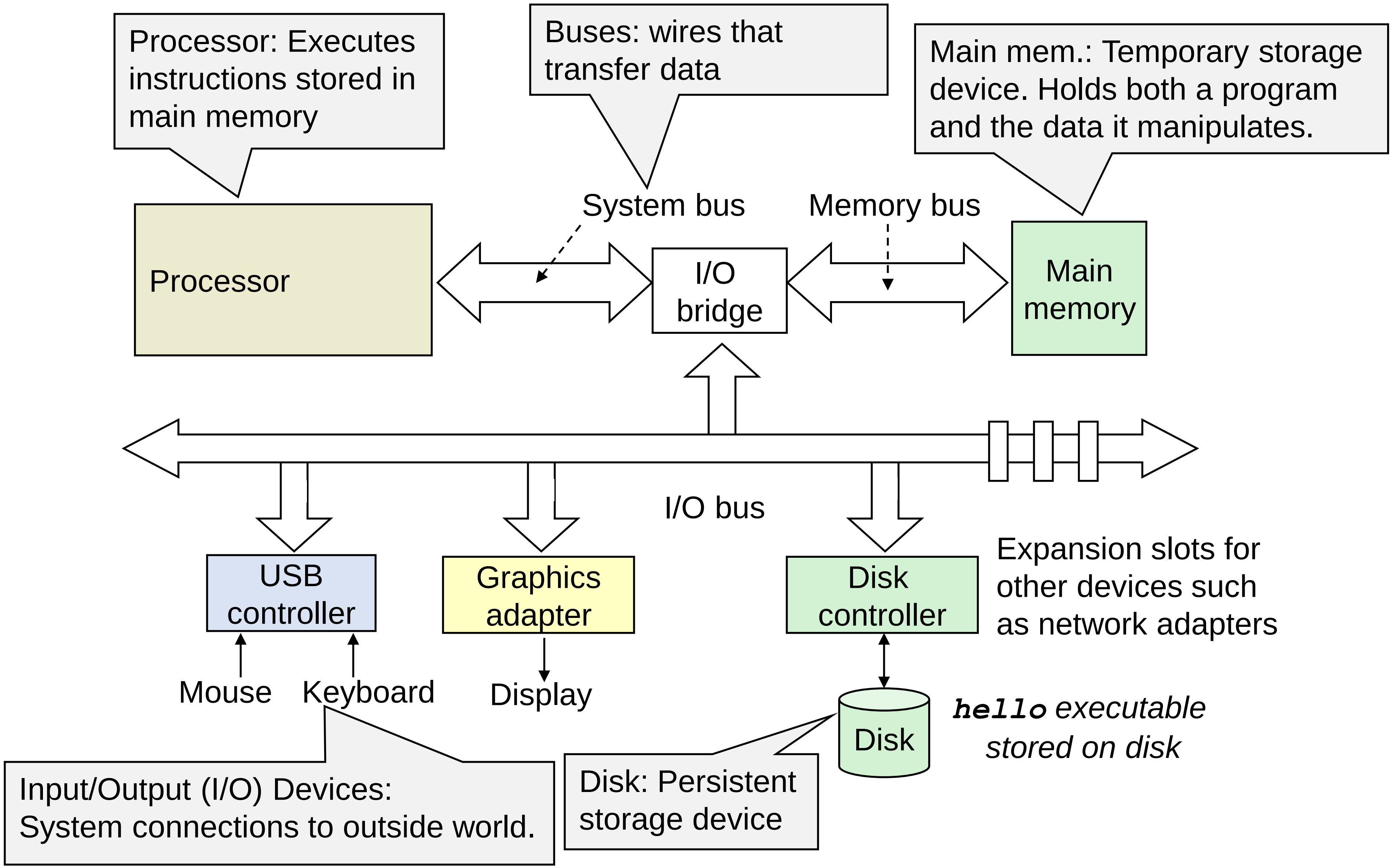
- Running **`hello`**

```
unix> ./hello
hello, world
unix>
```

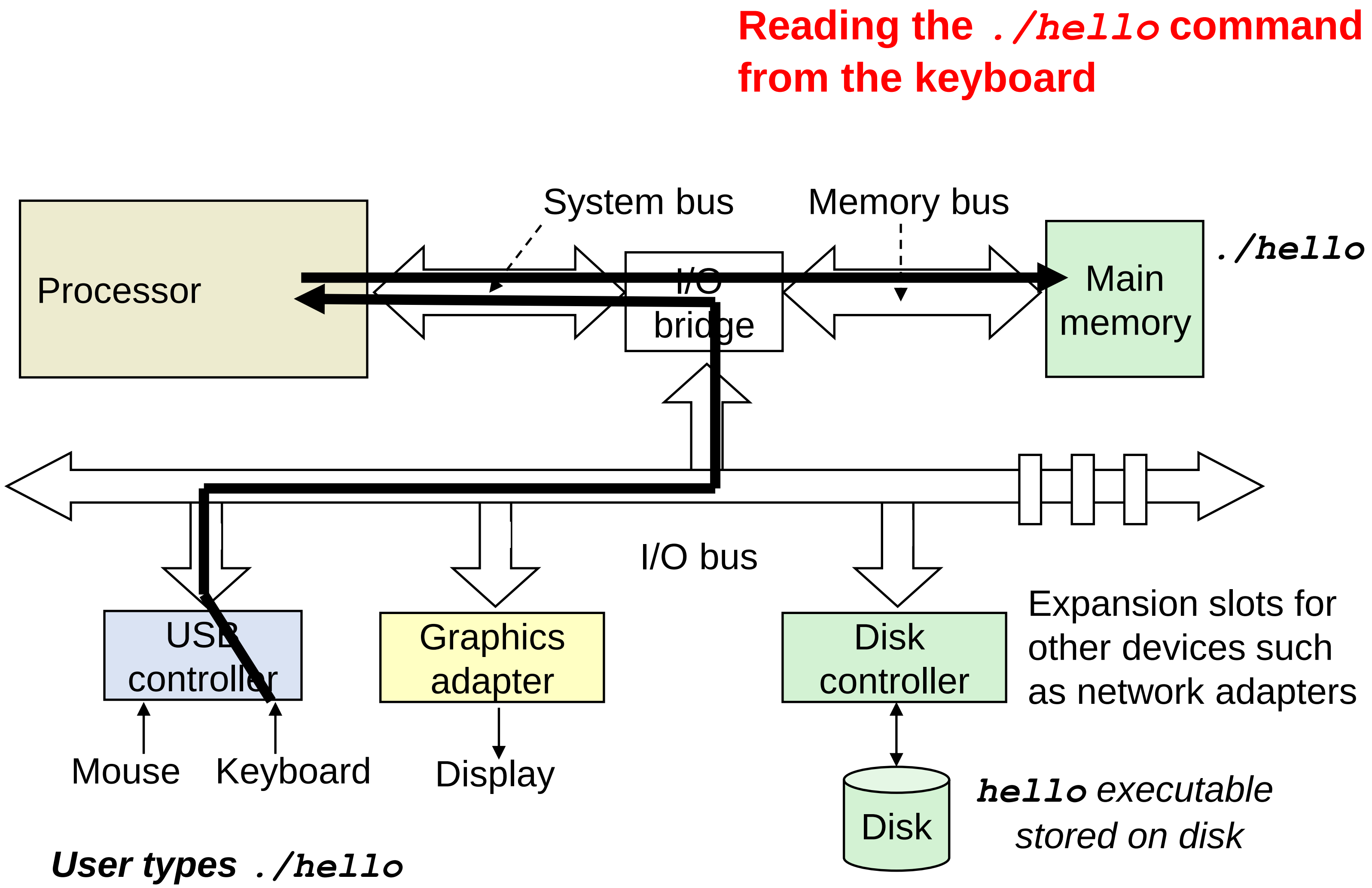
- What does the shell do?
  - Prints a prompt
  - Waits for you to type a command
  - Interpret the command
  - Then loads and runs the **`hello`** program

- What happens at the hardware level?

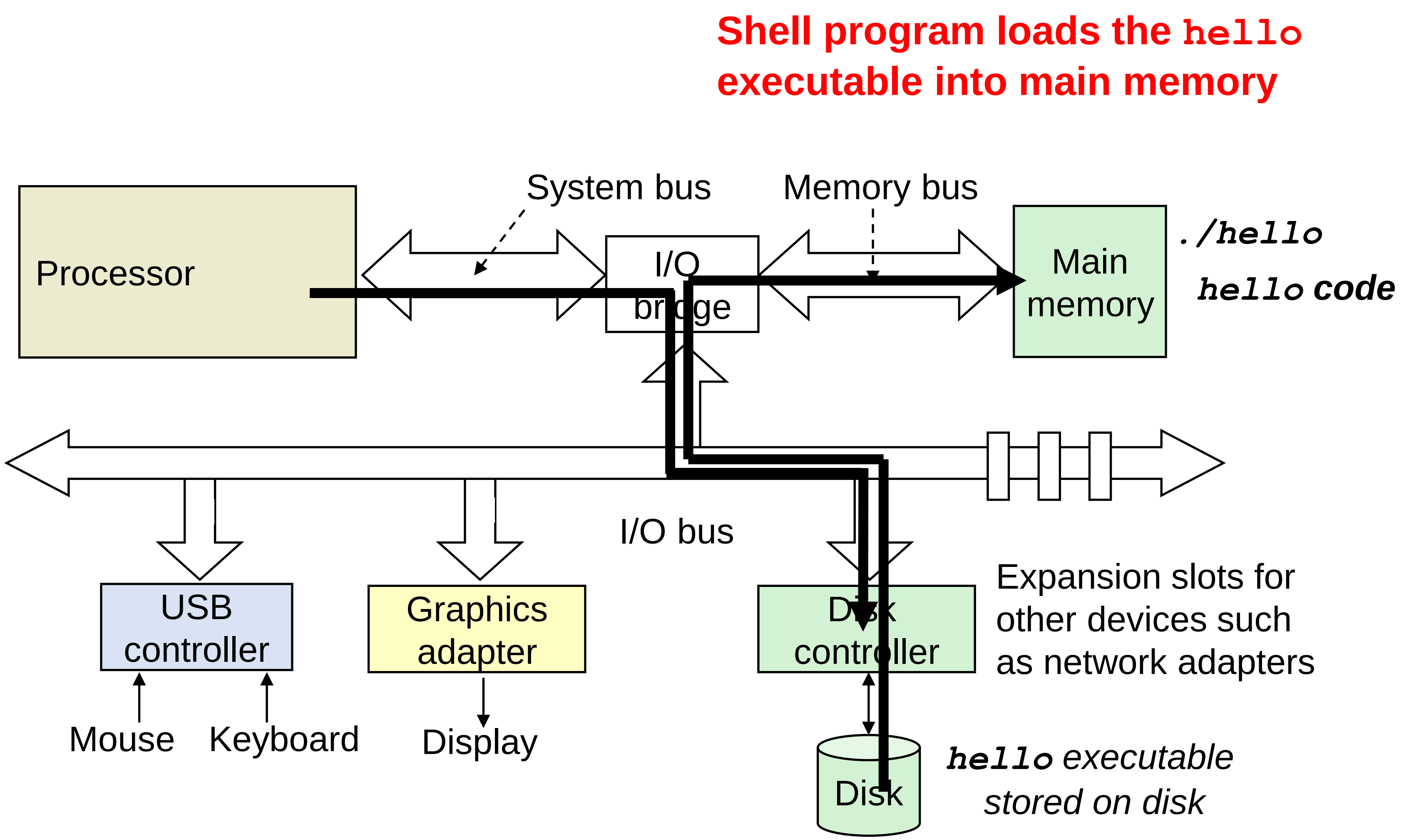
Hardware organization



Running `hello`

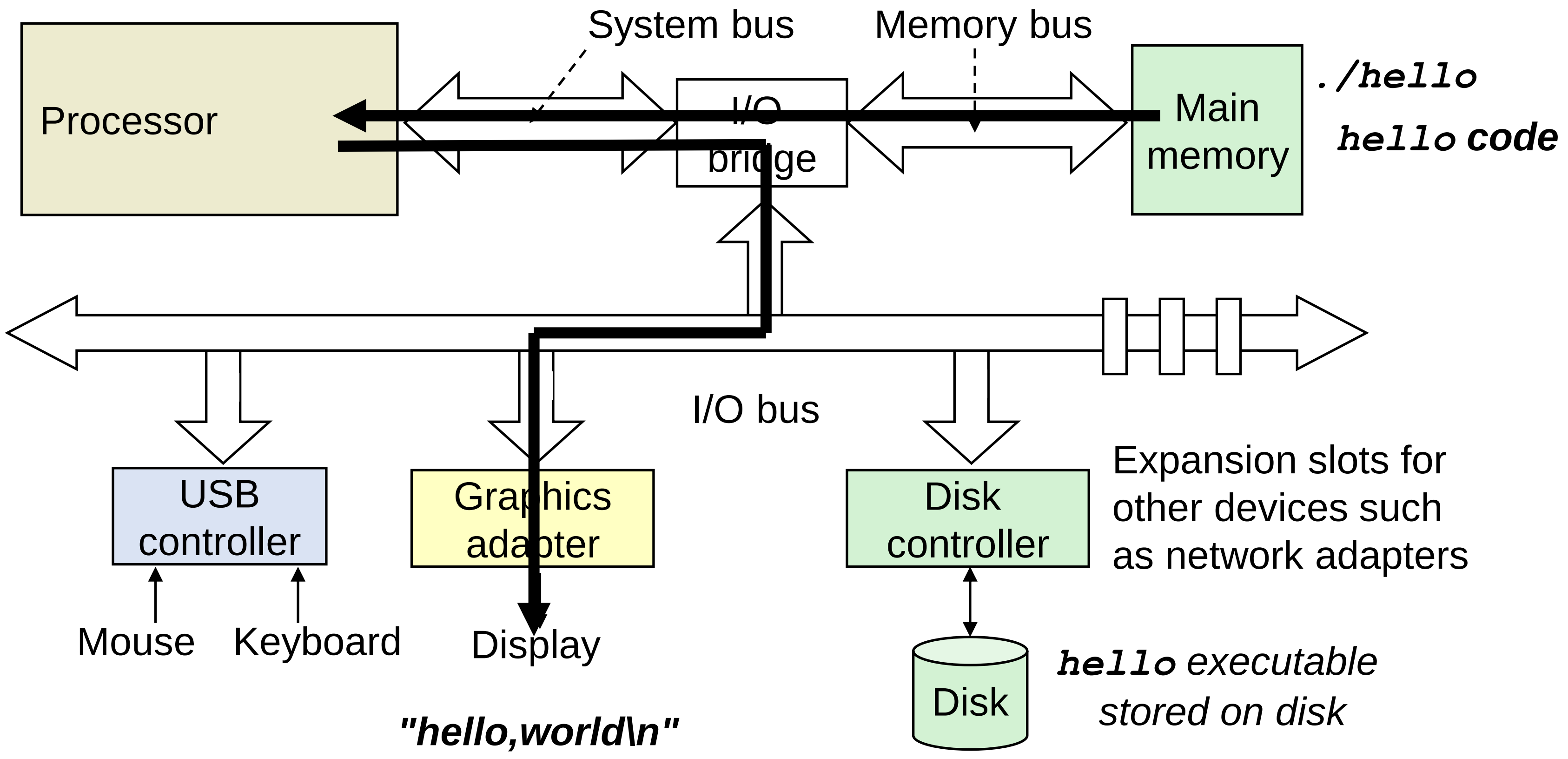


Running `hello`



Running `hello`

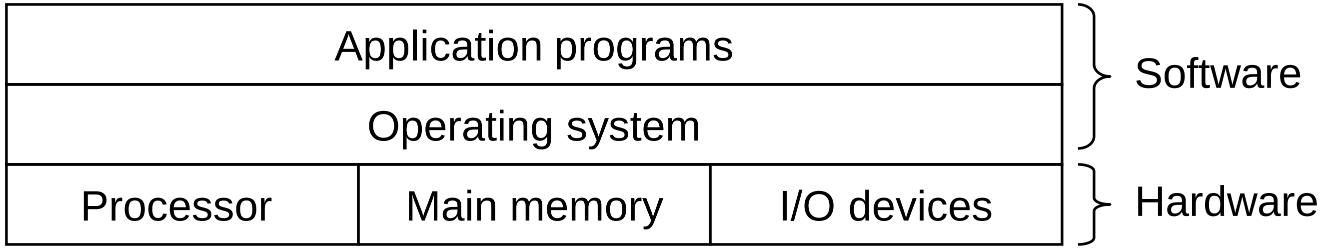
The processor reads the `hello` code, executes instructions, and displays “hello…”



The Operating System (OS)

- Neither **`hello`** nor our shell interfaced with the hardware directly
  - All interactions were mediated by the ***operating system***

- ***Operating system:*** a layer of software interposed between the application program and the hardware



- Primary goals
  - Protect resources from misuse by applications
  - Provide simple and uniform mechanisms for manipulating hardware devices
  - Manage sharing of resources between applications

Key idea: a computer system is more than just hardware

- A collection of intertwined hardware and software that must cooperate to achieve the end goal – running applications
  - **Hardware**: expensive, fast, immutable
  - **Software**: cheap (comparatively), slow, flexible
  - Different tradeoffs
    - So we'll use them for different roles!

- The rest of the course will expand on this

Open Question + Break

- **What part of the `hello` example takes the longest to run on a computer?**

Open Question + Break

- **What part of the `hello` example takes the longest to run on a computer?**
  - The user typing (seconds)
    - Maybe that’s cheating and we should start after they hit enter

Open Question + Break

- **What part of the `hello` example takes the longest to run on a computer?**

  - The user typing (seconds)
    - Maybe that's cheating and we should start after they hit enter

  - Almost certainly loading the program from disk (milliseconds)
    - Possibly sending text to graphics (microseconds – milliseconds)
    - Definitely not executing the code (nanoseconds – microseconds)

Outline

- Course Themes
- Logistics
- Running a program
- **Representing numbers with binary**

Learning binary

- To understand how a computer really works we need to understand that data it operates on

- Computers hold data in memory as individual ones and zeros
  - These ones and zeros make up binary values

- So, we’re going to need to understand binary
  - Binary will ***definitely*** come up again in this and other classes

Positional Numbering Systems

- The position of a *numeral* (e.g., digit) determines its contribution to the overall number
  - Makes arithmetic simple (compared to, say, roman numerals)
  - Any number has one canonical representation

- Example: base 10
  - $10456_{10} = 1*10^4 + 0*10^3 + 4*10^2 + 5*10^1 + 6*10^0$
  - Usually, we leave out the zeros:
    - $1*10^4 + 4*10^2 + 5*10^1 + 6*10^0$

Other bases are also possible

- Base 60, used by the Babylonians
  - The source of 60 seconds in a minute, 60 minutes in an hour
  - And 360 degrees in a circle
- Base 20, used by the Maya and Gauls
  - Parts of this remain in French today
- Base 2, used by computers
  - Example: 10010010_2
  - Same idea as before: $1*2^7 + 1*2^4 + 1*2^1 = 128_{10} + 16_{10} + 2_{10} = 146_{10}$

Base 2 Example

- Computer Scientists use base 2 a ***LOT*** (especially in computer systems)
- Let's convert 138_{10} to base 2
- We need to decompose 138_{10} into a sum of powers of 2
  - Start with the largest power of 2 that is smaller or equal to 138_{10}
  - Subtract it, then repeat the process

$$138_{10} - \boxed{128_{10}} = 10_{10}$$

$$10_{10} - \boxed{8_{10}} = 2_{10}$$

$$2_{10} - \boxed{2_{10}} = 0_{10}$$

$$138_{10} = \underline{\mathbf{1}}\times128 + 0\times64 + 0\times32 + 0\times16 + \underline{\mathbf{1}}\times8 + 0\times4 + \underline{\mathbf{1}}\times2 + 0\times1$$

$$138_{10} = \underline{\mathbf{1}}\times2^7 + 0\times2^6 + 0\times2^5 + 0\times2^4 + \underline{\mathbf{1}}\times2^3 + 0\times2^2 + \underline{\mathbf{1}}\times2^1 + 0\times2^0$$

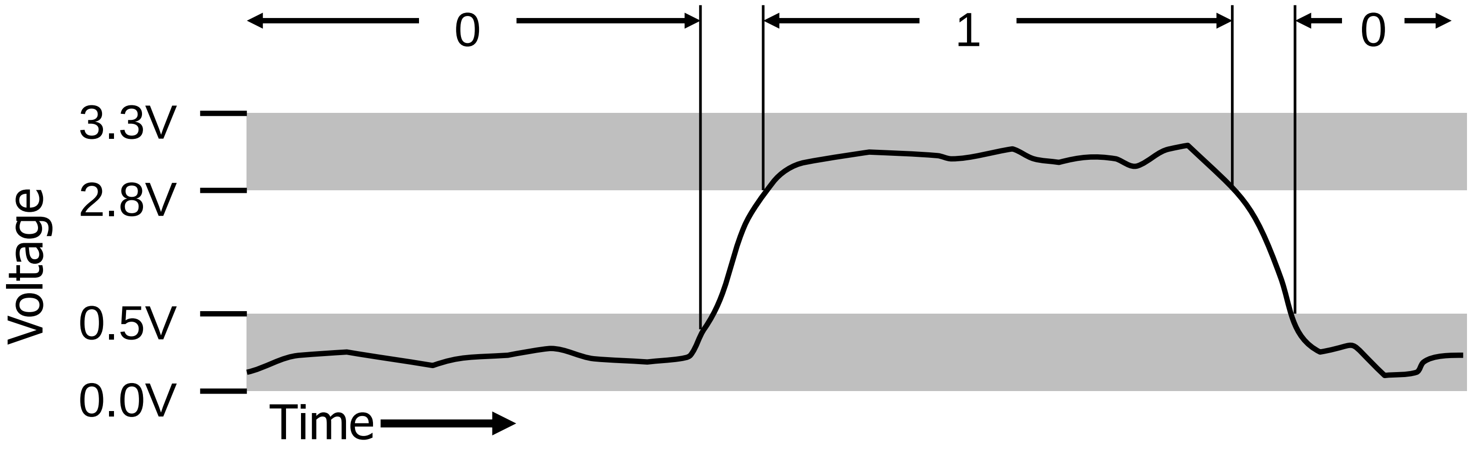
$$138_{10} = 10001010_2$$

Binary practice

- Convert 101_2 to decimal
  - $= 1\times2^2 + 0\times2^1 + 1\times2^0$
  - $= \quad 4 \ + \quad 0 \quad + \quad 1$
  - $= \quad 5_{10}$
- Convert 4_{10} to binary: 100_2 (one less than 5)

Why computers use Base 2

- Simple electronic implementation
  - Easy to store with bi-stable elements
  - Reliably transmitted on noisy and inaccurate wires

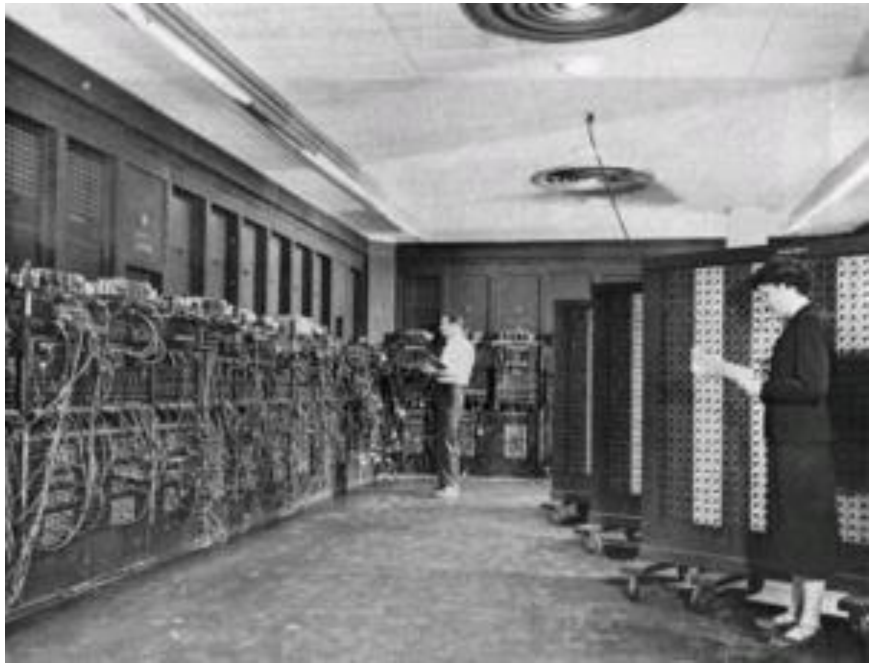


- Straightforward implementation of arithmetic functions

- (Pretty much) all computers use base 2

Why don't computers use Base 10?

- Because implementing it electronically is a pain
  - Hard to store
    - ENIAC (first general-purpose electronic computer) used 10 vacuum tubes / digit
  - Hard to transmit
    - Need high precision to encode 10 signal levels on single wire
  - Messy to implement digital logic functions
    - Addition, multiplication, etc.
    - (See CE203 for details)



Base 16: Hexadecimal

- Writing long sequences of 0s and 1s is tedious and error-prone
  - And takes up a lot of space on a page!
- So we'll often use base 16 (also called *hexadecimal*)
- Base 2 = 2 symbols (0, 1)
  Base 10 = 10 symbols (0-9)
  Base 16, need 16 symbols
  - Use letters A-F once we run out of decimal digits

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Base 16: Hexadecimal

- $16 = 2^4$, so every group of 4 bits becomes a hexadecimal digit (or *hexit*)
  - If we have a number of bits not divisible by 4, add 0s on the left (always ok, just like base 10)

0 0 1 0 ¦ 1 0 0 1 ¦ 0 1 1 1 ¦ 1 0 1 1 → 0x297B

"0x" prefix = it's in hex

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Bytes

- A single bit doesn't hold much information
  - Only two possible values: 0 and 1
  - So we'll typically work with larger groups of bits

- For convenience, we'll refer to groups of 8 bits as ***bytes***
  - And usually work with multiples of 8 bits at a time
  - Conveniently, 8 bits = 2 hexits

- Some examples
  - 1 byte: 0b01100111 = 0x67
  - 2 bytes: $11000100\ \ 00101111_2$ = 0xC42F

"0b" prefix = it's in binary

Practice problem

- **Convert 0x42 to decimal**

- Steps
  - Convert 0x42 to binary:
  - Convert binary to decimal:

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Practice problem

- **Convert 0x42 to decimal**

- Steps
  - Convert 0x42 to binary:
    - 0x4 -> 0b0100 0x2 -> 0b0010
    - 0x42 -> 0b 0100 0010
  - Convert binary to decimal:

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Practice problem

- **Convert 0x42 to decimal**

- Steps
  - Convert 0x42 to binary:
    - 0x4 -> 0b0100   0x2 -> 0b0010
    - 0x42 -> 0b 0100 0010
  - Convert binary to decimal:
    - $1*2^6 + 1*2^1 = 64 + 2 = 66$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Practice problem

- **Convert 0x42 to decimal**

- Alternate method:
  - 0x42
  - $= 4 \times 16^1 + 2 \times 16^0$
  - = 64 + 2
  - = 66

- But you're honestly better off converting hex to binary first
  - It's good practice!

Big idea: bits can be used to represent anything

- Depending on the context, the bits `11000011` could mean
  - The number 195
  - The number -61
  - The number -1.1875
  - The value `True`
  - The character '├'
  - The **`ret`** x86 instruction

- You have to know the **context** to make sense of any bits you have!
  - People and software they write determine what the bits actually mean

Outline

- Course Themes
- Logistics
- Running a program
- Representing numbers with binary