

# **CSCI 4061**

## **Lecture 1**

### **Course Mechanics & Introduction**

Instructor: Jack Kolb

January 17, 2023

# Instructor Information

- Name: Jack Kolb
- Email: [jhkolb@umn.edu](mailto:jhkolb@umn.edu)
- Office Hours
  - 3pm -- 5pm on Wednesdays
  - 11:15am -- 12:15pm on Thursdays
  - Can always email to schedule an appointment (including on-line)
- Office Location: 300F Lind Hall



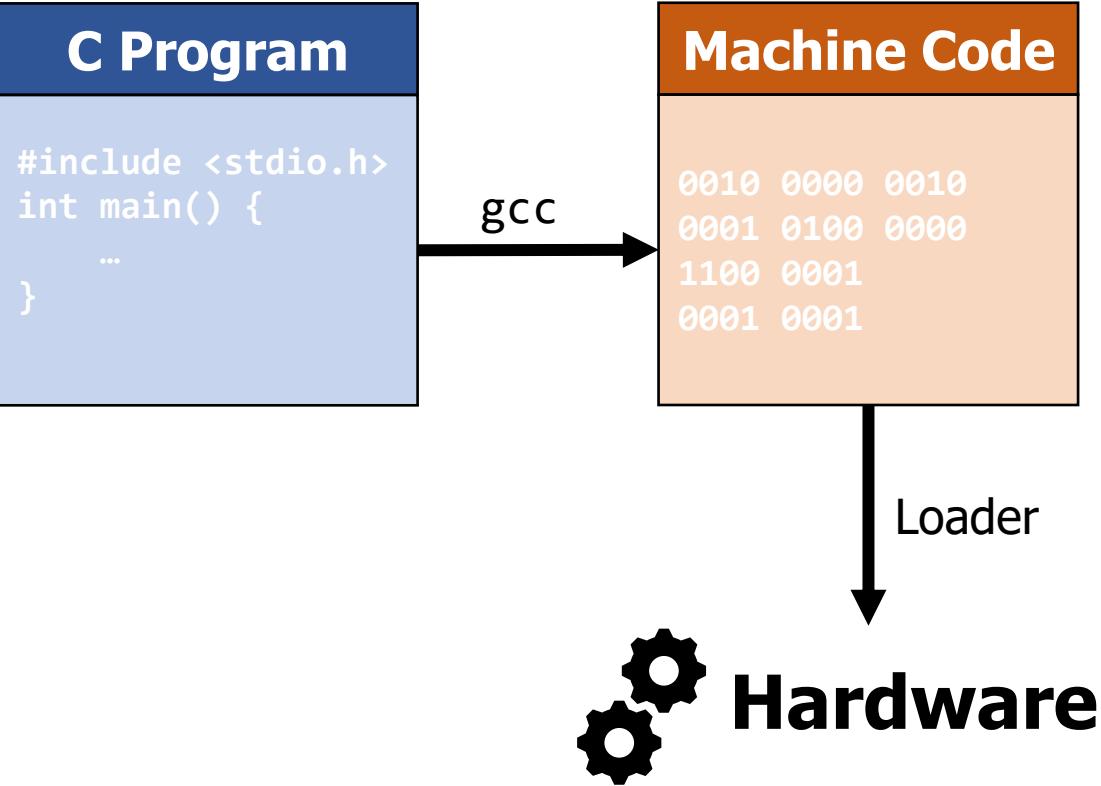
# What's the point of this class?

- Consider: What is computer science?
- The study of **solving problems** with **computers**
  - One aspect: designing **algorithms** and **data structures** to form your solution
  - But a solution only comes to life when it executes on **real hardware**
- 2021 is a good start to learning how execution of code really works
  - C -> assembly -> machine code -> electrical signals in hardware
  - Hardware ingredients: CPU, memory, buses, I/O, disk storage
- But it gives you a very incomplete picture!
  - One Exception: Virtual Memory

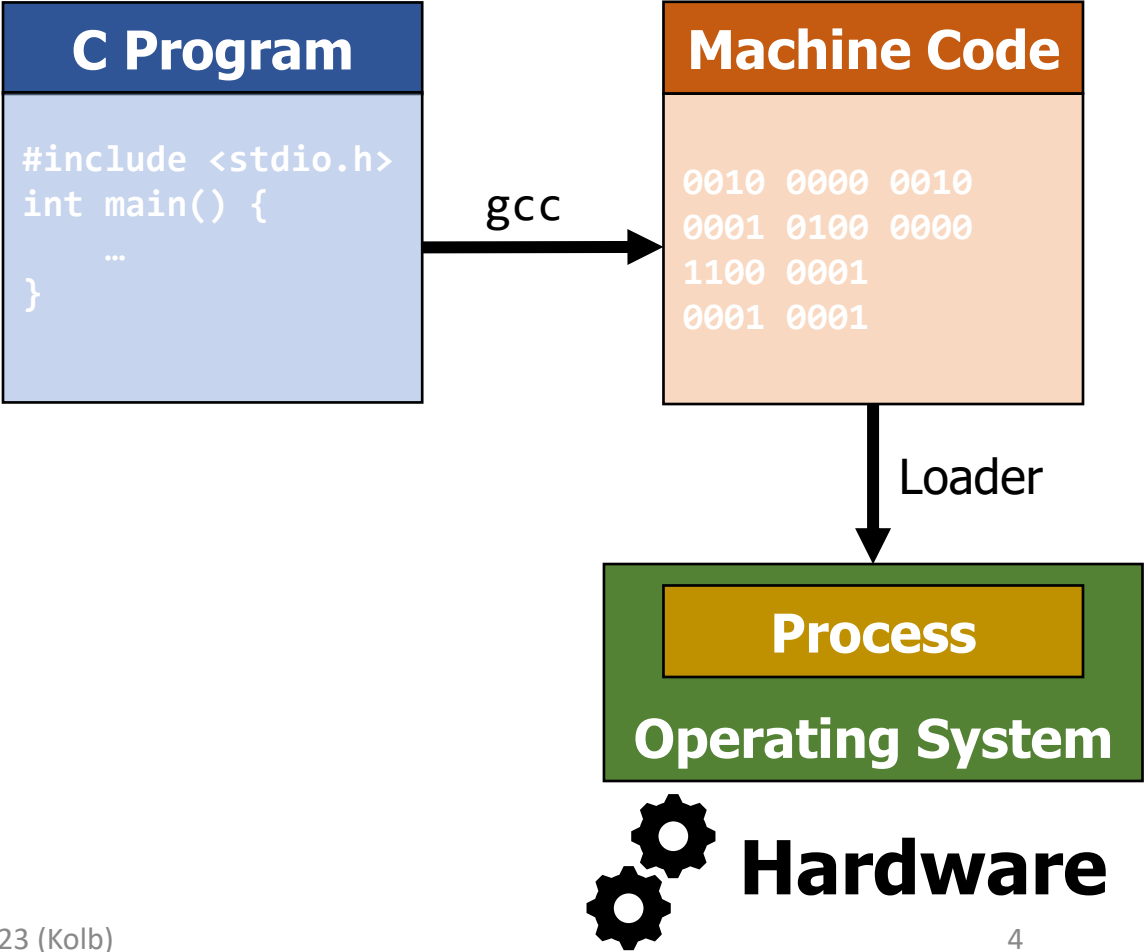


# Enter Operating Systems

## 2021 View of the World



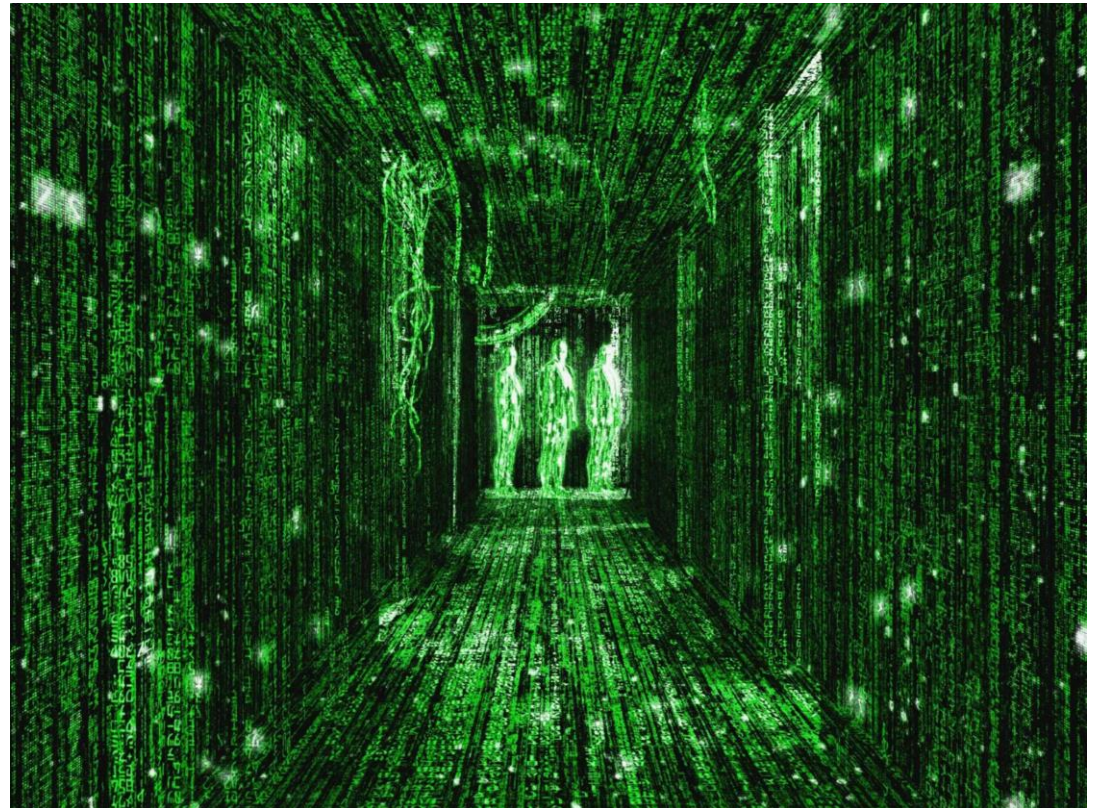
## 4061 (More Real) View of the World



# Process Sees an OS-Constructed “Reality” -- A Virtual Machine

- When a process is running, it sees a consistent set of abstractions:
  - Virtual Memory Addresses
  - Threads
  - Files
  - Signals (like a seg fault)
  - Sockets (for networking)
- Regardless of what the underlying hardware looks like

All illusions created by the OS and hardware to isolate and control your program!



# Why do this?

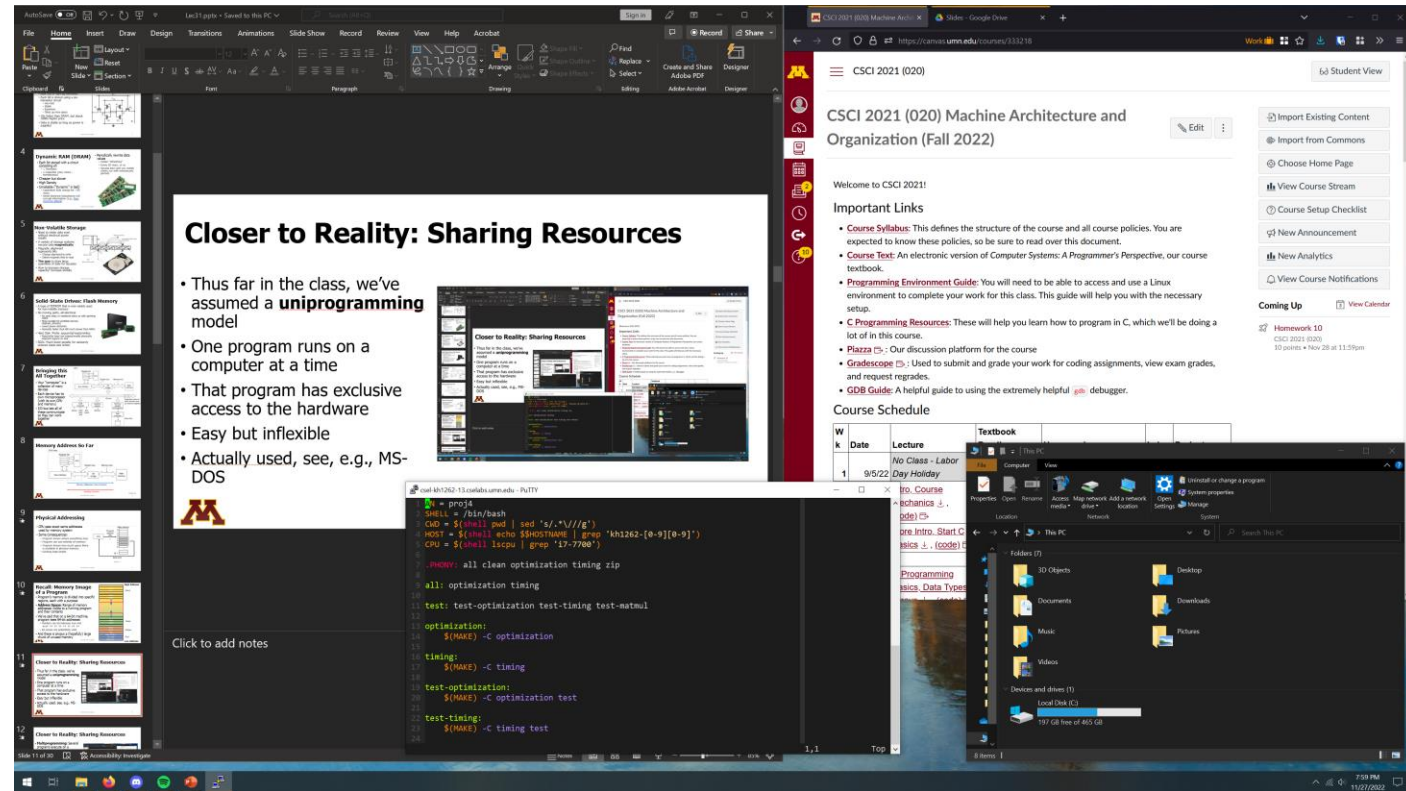
- Reason #1: **Hide diversity of hardware from programs**
- Example: Disk Storage
  - Spinning Metal or Solid-State
  - Made by Intel, Samsung, Toshiba, Micron, Western Digital, Seagate, ...
  - Different Capacities
- Programs just see files and same code “just works” for all disks
- Example: Networking
  - Wired: Ethernet and Cables
  - Wireless: Wi-Fi and Antennas
  - Made by Intel, Broadcom, Atheros, ...
- Programs just see sockets and same code “just works” for all devices





# Why do this?

- Reason #2: **Protect different processes from each other**
- **Multiprogramming:** Several programs execute on a computer at same time
- Share CPU (*Time*)
- Share Memory (*Space*)
- The OS is responsible for enforcing safe sharing of resources among all processes
- Processes interact with each other only in restricted ways



# Learning about Operating Systems

## CSCI 4061: Introduction to OS

- Looking at operating systems through lens of *systems programming*
- Understand what operating systems offer us and how to intelligently use these features
- Discuss OS internals when motivated by practical applications

## CSCI 5103: Operating Systems

- Looking at operating system internals for its own sake
- Understand theory and practice of designing/improving OS
- Actually implement some of the features that regular user programs depend on





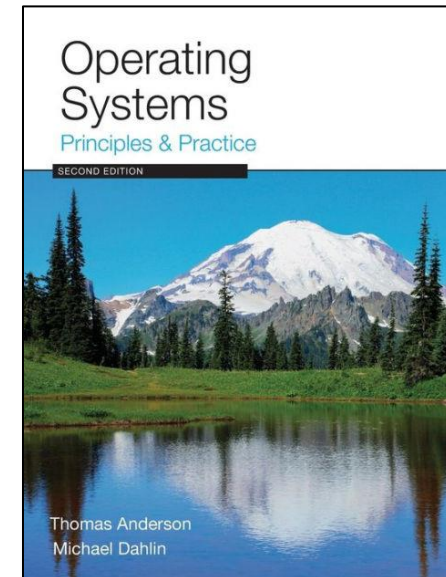
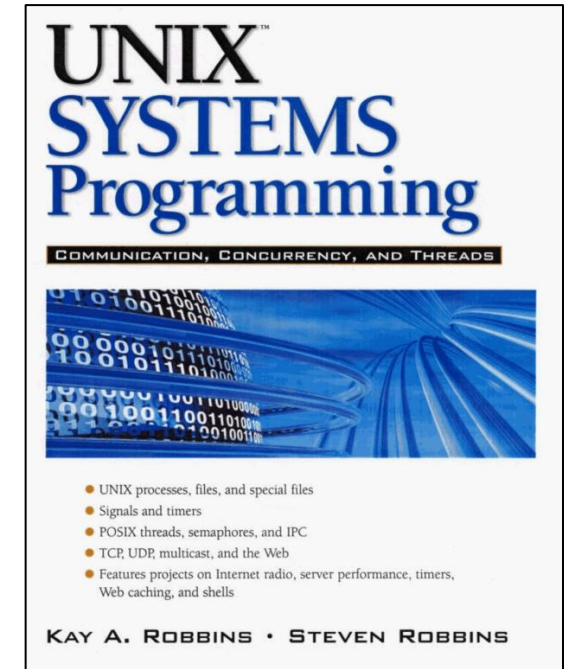
# Course Mechanics

- Lecture: Tuesday/Thursday, 75 minutes
  - Will feature in-class exercises to be done in small groups
  - Recordings will be posted on “Media Gallery” of our Canvas site
  - Usually at least one longer exercise/break
- Canvas Site: Main source of information for this class
  - Syllabus posted there has full details
  - Full Course Schedule, Lecture Slides, Assignment Documents
  - View grades
  - Office Hours Calendar: Bottom of Main Page
- Piazza: Q&A Forum
- Gradescope: Submit labs and projects, exam regrade requests



# Course Textbooks

- *UNIX Systems Programming* (Second Edition), Robbins & Robbins.
  - A decent text, may want to consult Stephens and Rago (listed on Syllabus) after this class
- *Operating Systems: Principles and Practice* (Second Edition), Anderson & Dahlin.
  - Optional but useful
  - Focuses on OS internals and implementation
  - Particularly relevant for virtual memory



# Labs

- Meets weekly on Monday, code due at end of Wednesday
- Lab assignments are 10% of course grade (drop two lowest)
- You must attend your assigned lab section for credit each week
- You will learn **a lot** by doing in this class!
  - Lecture will not be able to teach you all you need to know
- Encouraged to work in groups
- Submit to Gradescope



# Weekly Quizzes

- 10% of Course Grade (drop two lowest scores)
- Weekly: Released on Monday, due following Monday
  - Posted on Canvas site
- Meant to give you a chance to review and apply lecture concepts
  - And give you practice for exams!
- Open Resource
  - Collaboration encouraged (as long as you understand your answers)
  - Unlimited submissions
- First quiz is out now, Due January 23



# Projects

- 35% of Course Grade
- 4 Projects Planned: 8.75% of grade per project
- You will learn **a lot** by doing in this class
- To be completed individually or in pairs
- Submit to Gradescope
  - Grade determined by **both** test results and manual inspection
  - We will not provide exhaustive tests
  - Score on Gradescope will not match final score on Canvas



# Exams

- Midterm Exam 1: 12.5% of Course Grade
  - **Thursday, February 16**, During Normal Lecture Time
- Midterm Exam 2: 12.5% of Course Grade
  - **Thursday, March 30**, During Normal Lecture Time
- Final Exam: 15% of Course Grade
  - Cumulative
  - See syllabus for date and time -- clear your schedule now
- You must take all exams with your assigned section
  - If you take an exam in the wrong section, you will get a zero





# Participation

- 5% of Course Grade
- 3 Class Surveys
  - Entrance Survey (Out Now!)
  - Mid-Semester Survey
  - Exit Survey
- Engaging in Lecture Discussions
- Asking/Answering Questions on Piazza
- Don't stress this: expect to receive full credit if you submit all 3 surveys and participate reasonably in class/Piazza



# Academic Integrity

- **All work you submit (e.g., assignment answers, project code, exam answers) must be a result of your own effort and reflect your individual understanding of the material.**
- Collaboration is allowed in a few select cases (Quizzes, Labs)
- OK to talk about project ideas at a high level, but not to get into specifics about code
- Cheating cases will be referred to Office of Community Standards



# Online Behavior

- Some elements of the class will take place online
  - Example: Piazza Discussions
- Be respectful to others and courteous at all times
  - Full University policy available [online](#)
- Bottom Line: Don't be a troll



# Advice

- This can be a challenging class
- Mastering the material will take **practice** and **persistence**
- If you are struggling, ask for help!
  - No one knows all this stuff, and you are not alone!
  - Office Hours
  - Piazza
- Respect and learn from each other



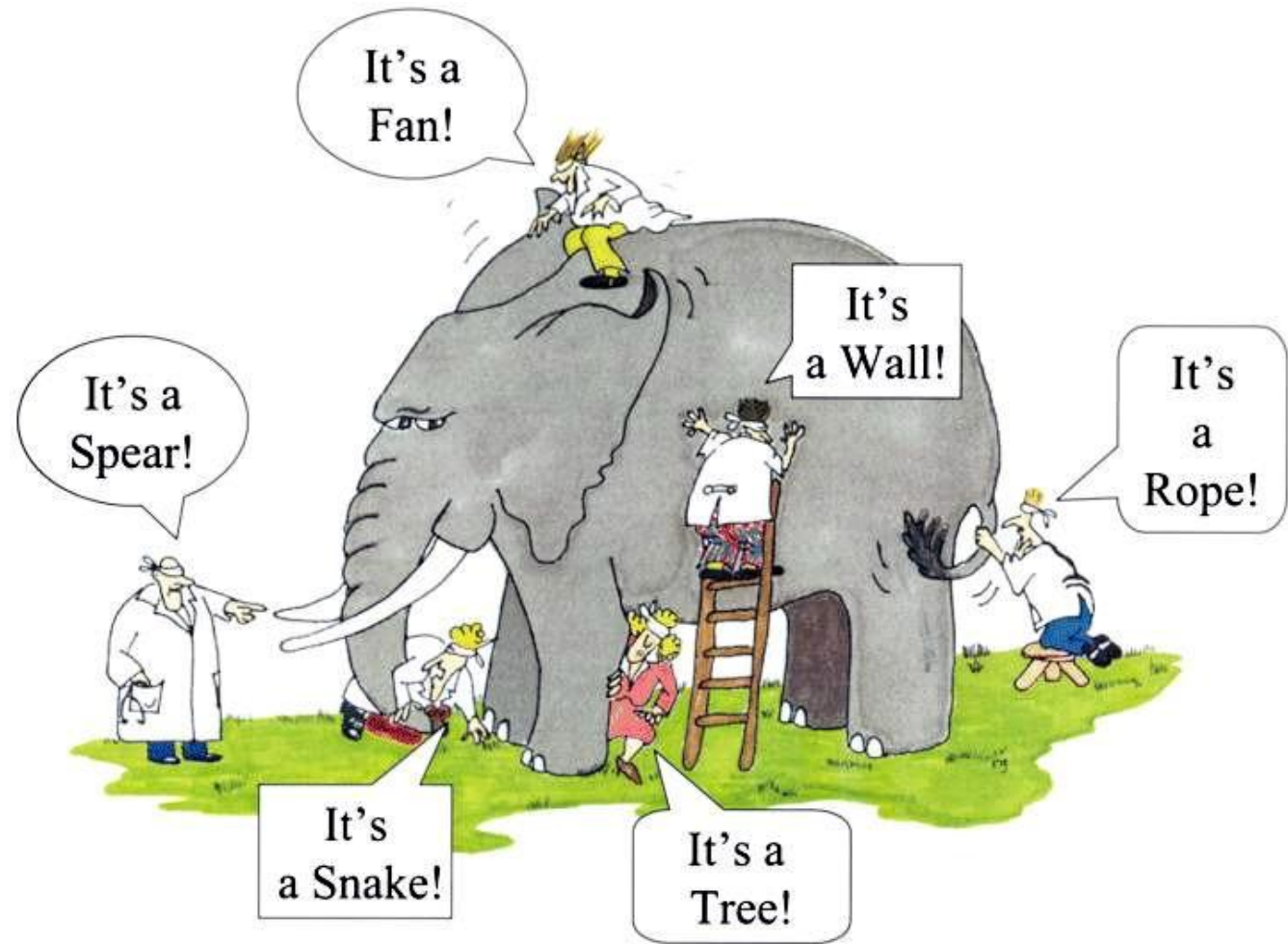
# Advice

- It is our job as course staff to help you get the most out of this class
  - But we can't force you to put in the necessary effort and focus
  - Invest your time and attention, consistent and disciplined effort is the way to succeed
- Keep up with Canvas website, Piazza announcements
- Keep up with weekly lectures and readings
  - It becomes much harder to learn material when you are short on time
- Start on assignments early
  - Projects are going to take you significant time to complete
  - You will run into bugs, problems, and questions



# What is an operating system?

- No universally accepted definition
- Depends on what you happen to care about at the moment
  - Computer User
  - Application Programmer
  - Systems Programmer
  - Hardware Engineer
- Example: Is the graphical user interface (window design and appearance, menus, etc.) part of the OS?





# Possible OS Definition

- The operating system is the **lowest, foundational layer of software** in a computer system
- Easily the most complex software we've ever created
- The only software that is allowed to talk directly to HW devices
- Manages all other software running on a machine
- Provides a consistent API that all other processes must go through to interact with the outside world: hardware, other processes
- Offers useful services to other processes (e.g., network management)



# Roles of the Operating System

## 1. **Referee:** Supervise all running processes

- Resource sharing (Time on CPU, space in RAM)
- Maintain isolation and protection between competing processes

## 2. **Illusionist:** Clean, easy to use abstractions

- As coders (and as a process), we think we have a dedicated machine
- “Infinite” memory, no one else using CPU
- Build higher-level objects from hardware: Files, users, messaging
- Mask limitations, evolution of hardware

## • **Glue:** Common Services

- Storage, graphical interface, networking, authorization



# Short Exercise: Three Programs

#1

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

#2

```
#include <string.h>
#include <unistd.h>

int main() {
    char *message = "Hello, World!\n";
    write(STDOUT_FILENO, message,
          strlen(message));
    return 0;
}
```

#3

```
.text
.global _start

_start:
    movq $1, %rax
    movq $1, %rdi
    movq $msg, %rsi
    movq $14, %rdx
    syscall

    movq $60, %rax
    movq $0, %rdi
    syscall

.data
msg:
    .ascii "Hello, world!\n"
```

1. What does each program do?
2. What are the similarities between these programs?
3. What are the differences between these programs?



# System Calls

- User programs (e.g., our `hello_print` program) never interact directly with hardware like the screen
- Instead, a program makes a request to the OS to do something on its behalf by making a **system call**
- Like a function call - control transfers to some specified destination
- Not like a function call - OS takes over and runs code inside kernel
  - CPU operates in *supervisor mode* rather than *user mode*, kind of like `sudo`
- Return control back to where program left off when done
  - But with requested work all completed, or an error indicated



# What do we mean by system call?

- Strictly speaking, system calls can *only* be done in assembly
  - That's the whole point of the `syscall` instruction in the x86\_64 ISA
- But we won't be writing much assembly in this class
  - So you'll never write a system call for all of 4061?
  - In a way, yes!
- A function like `write` is a C library function that does the work of setting up and making a system call for us
- But, almost all programmers (and textbooks) refer to a call to `write` as a system call
  - And we'll follow this convention as well



# System Call Design Consideration

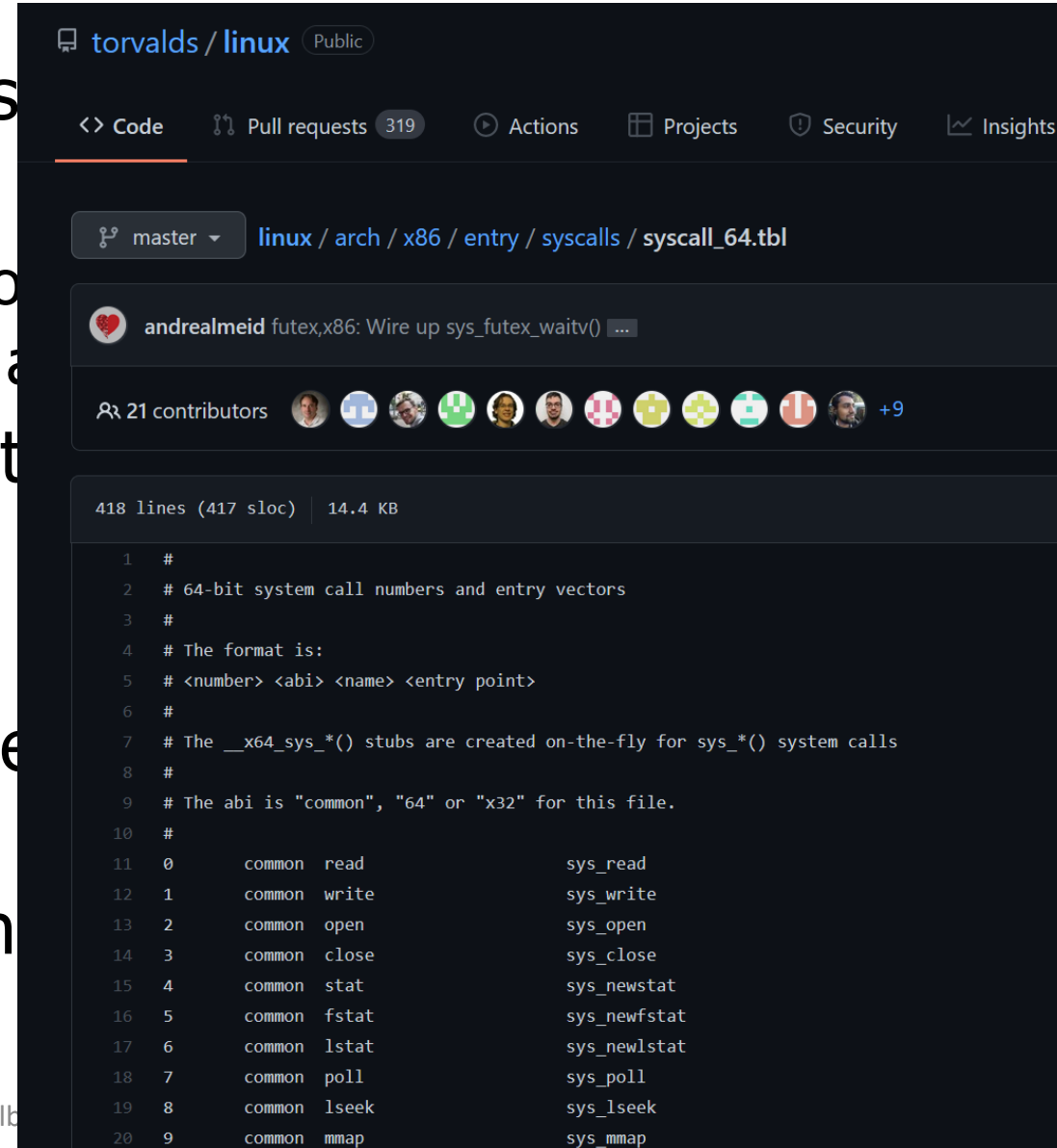
- Why is the series of steps to make a sys call so rigid?
- Put a specific number in a register (%rax in x86-64)
  - Mapping from number to operation to perform is part of OS standard
  - 64-bit Linux has ~300 different operations allowed, [each with a unique number](#)
- Transfer control to OS (syscall instruction in x86-64)
- Wouldn't it be easier to make this more like a regular function call?
  - Jump to address of first instruction in code want to execute
- Operating systems **do not trust** the user programs they supervise
  - That's part of their job
- Can only enter, start running OS code in very specific, controlled ways that are guaranteed to be safe





# System Call Design Consideration

- Why is the series of steps to make a sys
- Put a specific number in a register
  - Mapping from number to operation to perform
  - 64-bit Linux has ~300 different operations
- Transfer control to OS (syscall instruction)
- Wouldn't it be easier to make this more
  - Jump to address of first instruction in code
- Operating systems **do not trust** the user
  - That's part of their job
- Can only enter, start running OS code in  
that are guaranteed to be safe



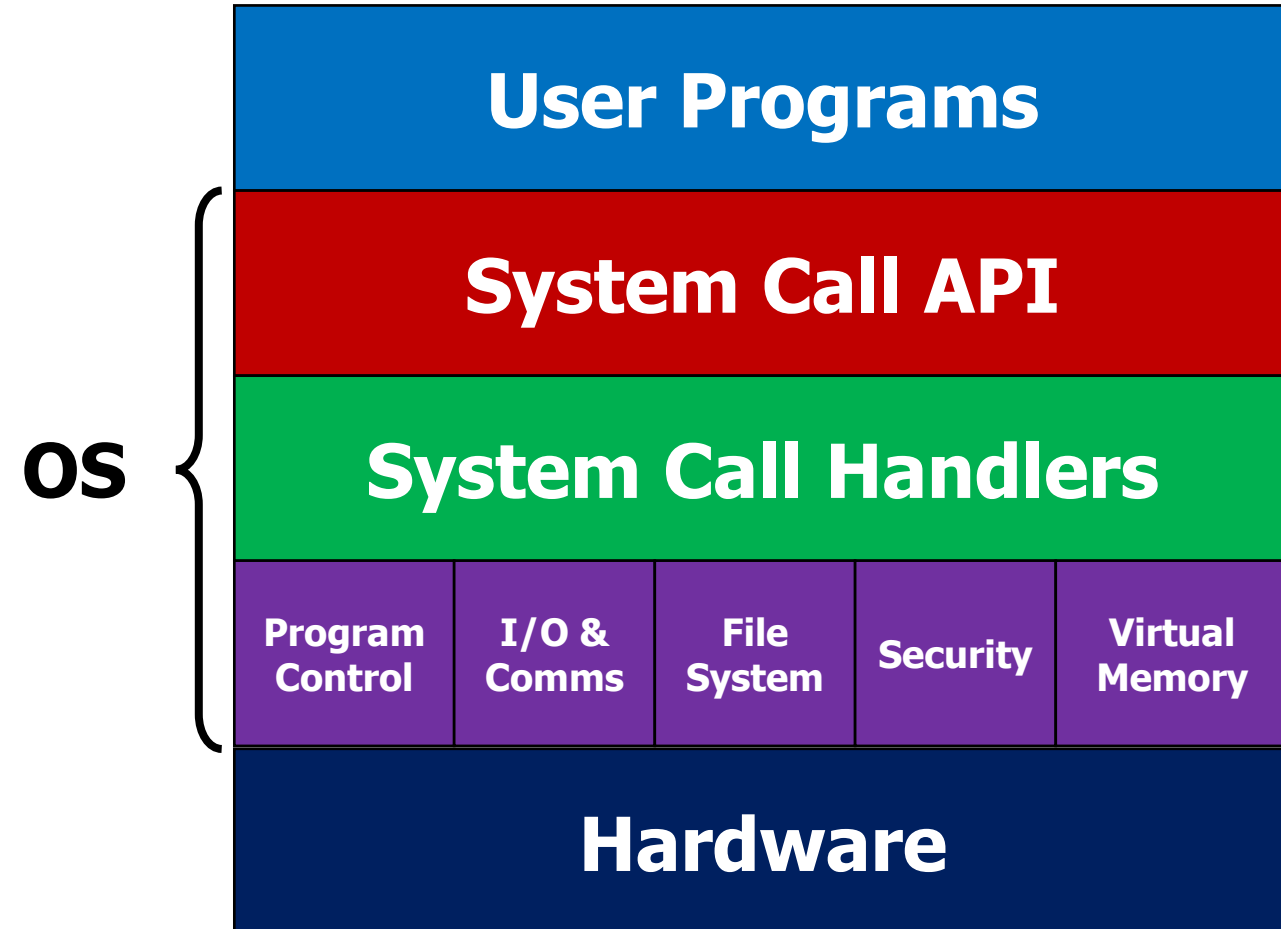
The screenshot shows the GitHub repository for the Linux kernel, specifically the file `linux/arch/x86/entry/syscalls/syscall_64.tbl`. The file is 418 lines long (417 sloc) and 14.4 KB in size. It contains a table of system call numbers and their corresponding entry points. The table is organized by ABI (common, 64, or x32) and lists the system call number, the ABI, the operation name, and the entry point function.

```
1 #
2 # 64-bit system call numbers and entry vectors
3 #
4 # The format is:
5 # <number> <abi> <name> <entry point>
6 #
7 # The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
8 #
9 # The abi is "common", "64" or "x32" for this file.
10 #
11 0      common  read          sys_read
12 1      common  write         sys_write
13 2      common  open          sys_open
14 3      common  close         sys_close
15 4      common  stat          sys_newstat
16 5      common  fstat         sys_newstat
17 6      common  lstat         sys_newstat
18 7      common  poll          sys_poll
19 8      common  lseek         sys_lseek
20 9      common  mmap          sys_mmap
```



# OS Structure

- Operating systems are built out of layers, like many things in CS
- **4061**: Outer layers
- **5103**: Inner layers
- Hardware? EE and architecture



# OS Structure

- Operating systems are built out of layers, like many things in CS
- **4061**: Outer layers
- **5103**: Inner layers
- Hardware? EE and architecture
- Reminder
  - OS unifies huge range of hardware under one API
  - And this API supports a huge range of software

OS



# Systems Programming

- Like operating systems, has a loose definition
- **An area of study/expertise:** Writing and designing programs that make heavy use of the operating system's API
- **A point of contrast**
  - *Applications programming* is about software that provides services to the end user (e.g., word processor)
  - *Systems programming* is about software that acts a foundation for other software (e.g., Python interpreter, Web Server)



# Systems Programming

- Also an **ethos/state of mind**
- Systems programmers have knowledge/awareness of hardware
  - It affects efficiency/correctness of your code!
  - Writing cache-aware code in 2021 is a good start
- Systems programmers are paranoid and meticulous
  - Almost all system calls can result in an error
  - What if you try to write a file to a full storage device?
  - True for library functions too: `malloc` can return `NULL` if error
  - Code must check for *any* possible error and address it
- Because they have to be: if your code fails, so does everything that depends on it



# Systems Programming

A problem has been detected and Windows has been shut down to prevent damage to your computer.

DRIVER\_IRQL\_NOT\_LESS\_THAN\_OR\_EQUAL\_TO

If this is the first time you've seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need.

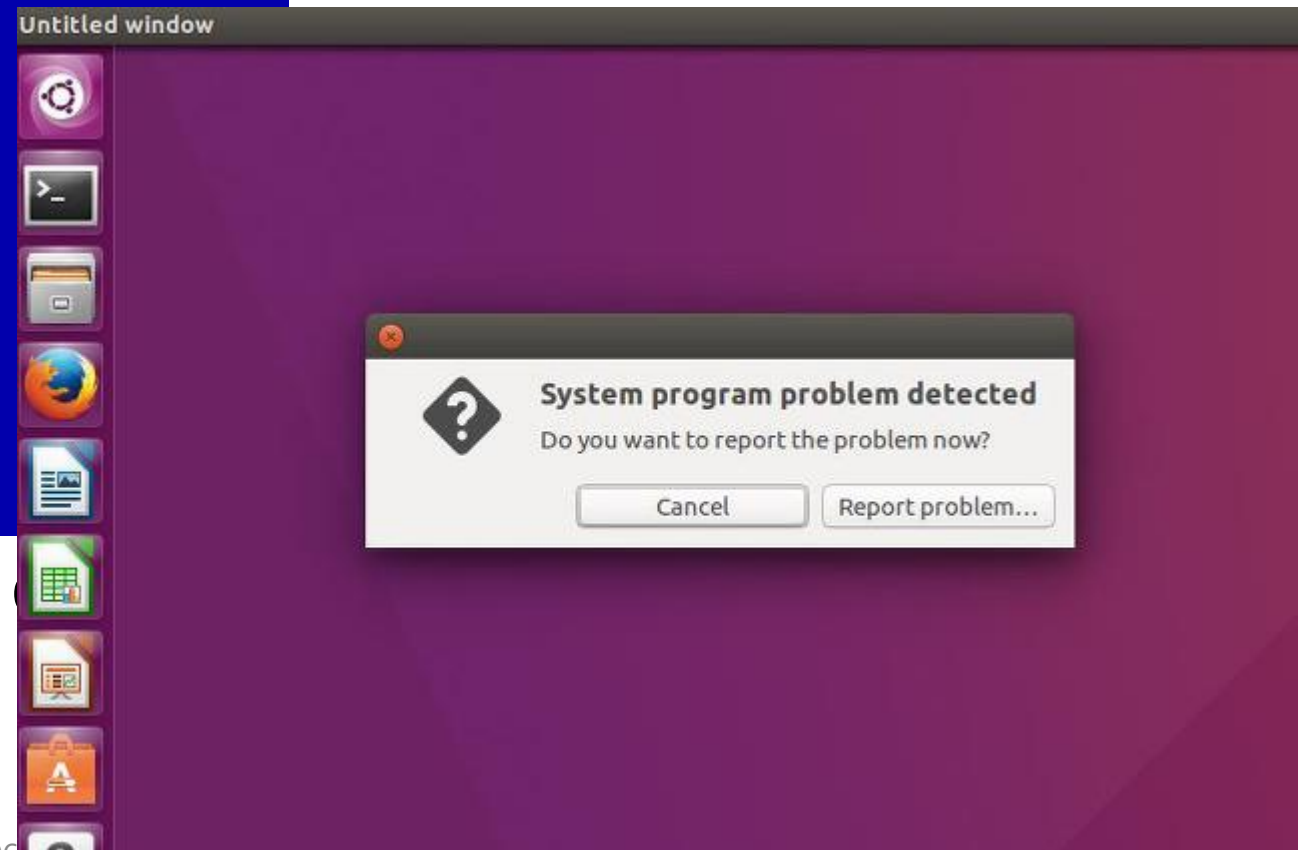
If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup options, and then select Safe Mode.

Technical information:

\*\*\* STOP: 0x000000D1 (0x00000000, 0x00000000)

awareness of hardware

- Because they have to be: if your code depends on it





# You Will be Graded on Error Handling

- Learning respect for errors is an important part of 4061
- We will be manually reviewing your project code for:
  1. Checks with all operations for possible errors
  2. Performing necessary cleanup (freeing memory, closing files) on all possible paths through your code (e.g., error cases vs. successful case)
- -1 point per missed error check, -1 point per missed cleanup step
- Be ready for this on Project 1



# Core Concepts in Systems Programming

<b>Concurrency</b>	Multiple things are in progress at once, their order is unpredictable
<b>Asynchrony</b>	Events can happen at any point
<b>Coordination</b>	Competing processes/threads must avoid interfering with each other
<b>Communication</b>	Share info within a machine (processes/threads) or across the world (networking)
<b>Security</b>	Access to info, operations can be restricted
<b>File Storage</b>	Layout of data on disks, algorithms for efficient reading/writing
<b>Memory</b>	Illusion of independent address space for each process, safe and controlled sharing
<b>Robustness</b>	Handle unexpected events gracefully (unless you want to crash)
<b>Efficiency</b>	Use CPU, Memory, Disk to fullest potential, minimize overhead for other programs



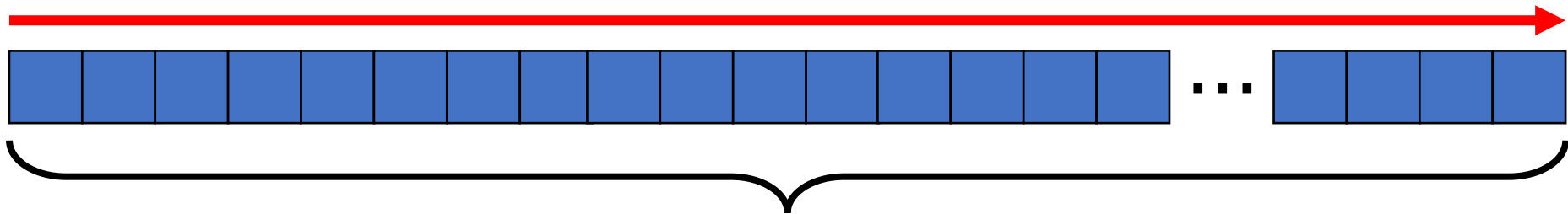
# Major Course Topics

1. Quick C Review
2. Processes, Process Creation and Management
3. Low-Level I/O
4. File Systems
5. Virtual Memory
6. Signals and Event Handling
7. Interprocess Communication
8. Threads
9. Thread Synchronization
10. Network Communication and Programming
11. Networked Systems (Time Permitting)



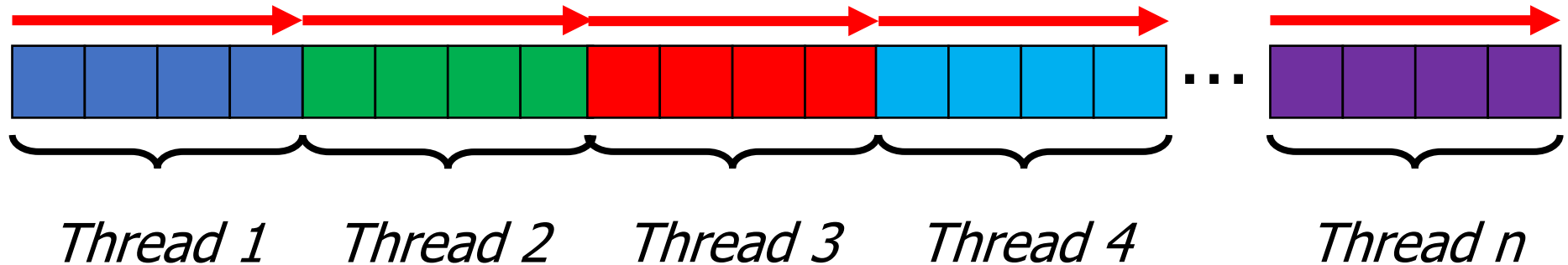
# Preview: Multithreading

- Arguably the most important concept in this class
- One process can contain multiple **threads**, each executing its own sequence of instructions (code)
  - Lots of reasons we may want to do this
  - One obvious one: do work *in parallel* on a CPU with multiple cores
- Example: Summing all elements in an array
- Traditional Sequential Approach:



# Preview: Multithreading

- Arguably the most important concept in this class
- One process can contain multiple **threads**, each executing its own sequence of instructions (code)
  - Lots of reasons we may want to do this
  - One obvious one: do work *in parallel* on a CPU with multiple cores
- Example: Summing all elements in an array
- Multithreaded Approach:



# Exercise: Sketching Thread Code

- Assume we have some way to create threads, each executing a call to the `sum_elements` function in parallel with other threads
- We have two versions of `sum_elements`:

```
void sum_elements_v1(long *sum, int
    *elements, int start, int stop) {
    for (int i = start; i < stop; i++) {
        *sum += elements[i];
    }
}
```

```
void sum_elements_v2(long *sum, int
    *elements, int start, int stop) {
    long temp_sum = 0;
    for (int i = start; i < stop; i++) {
        temp_sum += elements[i];
    }
    *sum += temp_sum;
}
```

- If we have 8 threads and an array with 100 elements, what should the function arguments be for each thread?
- Generalize this to a case with  $n$  threads and an array with  $m$  elements
- Is one version of `sum_elements` better than the other? Why?

# Exercise: Sketching Thread Code



# Exercise: Sketching Thread Code

- Threads give you a lot of power as a programmer, but they can be very difficult to use
- You'll have to adjust your whole mental model of how code executes
- Understanding these concepts is a major advantage
  - Academic standpoint: Many advanced topics/courses rely on this knowledge
  - Career standpoint: Distinguishes you as a developer/engineer
- Point 1: Understanding hardware (CPU and memory) is even more crucial to writing efficient code when using multiple threads
- Point 2: Need to take extra care with multithreaded code to ensure it behaves correctly and consistently
  - We'll spend a lot of time on why this is and the tools we use to address it





# Assumption: You Know Some C

- CSCI 2021 (or equivalent) is prereq, covers some hardware, basic C programming, and some interactions between C and hardware
- We'll assume you know C syntax and its major features, such as:
  - Loops
  - Conditionals
  - Functions
  - Data types, including arrays and structs
  - Pointers and memory addresses
  - Memory management with `malloc()` and `free()`
  - Stack vs. heap storage
  - How to edit, compile and execute code
  - How to debug code, particularly with `gdb`



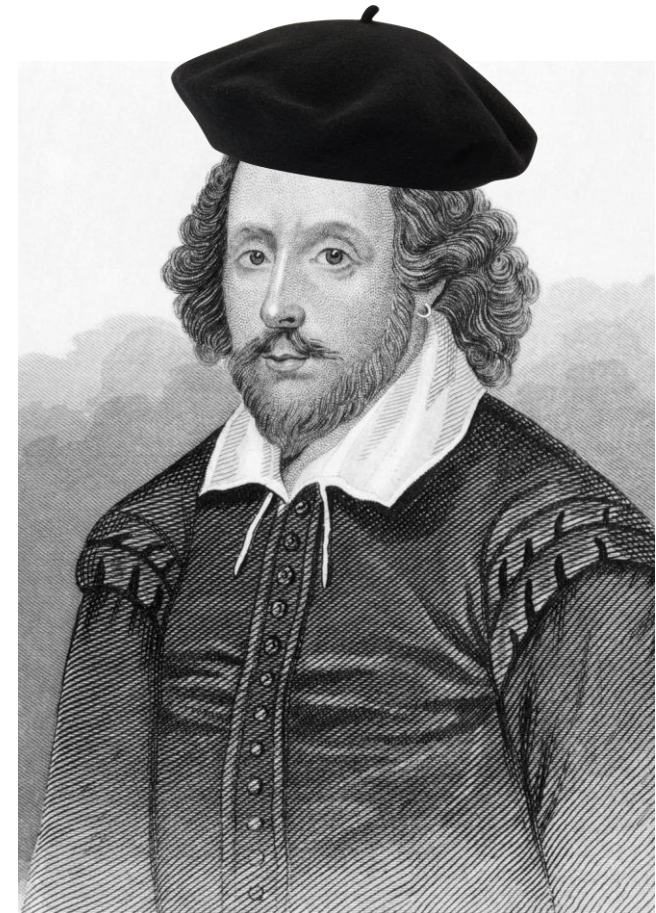
# Getting up to Speed on C

- We'll do some review in our next lecture
  - Meant as a reminder, not a complete tutorial
- May want to locate a good C reference
  - *The C Programming Language* by Kernighan and Ritchie is a good one
- "C Programming Resources" [page](#) on our Canvas site
- For suggestions on useful tools, see "Programming Environment Guide" page on our Canvas site
- GDB [guide](#) linked to from our Canvas site
  - Originally written for CSCI 2021 but highly relevant
- **Make sure you invest the necessary effort here, otherwise this course will be very difficult for you**



# Why use C in 4061?

- C can certainly be challenging
  - Memory management
  - Pointers
  - Strange bugs
- But it is **the** language for operating systems and systems programming
- Gives you most direct access to reality of real hardware
- Gives you direct and “unfiltered” access to the real OS API

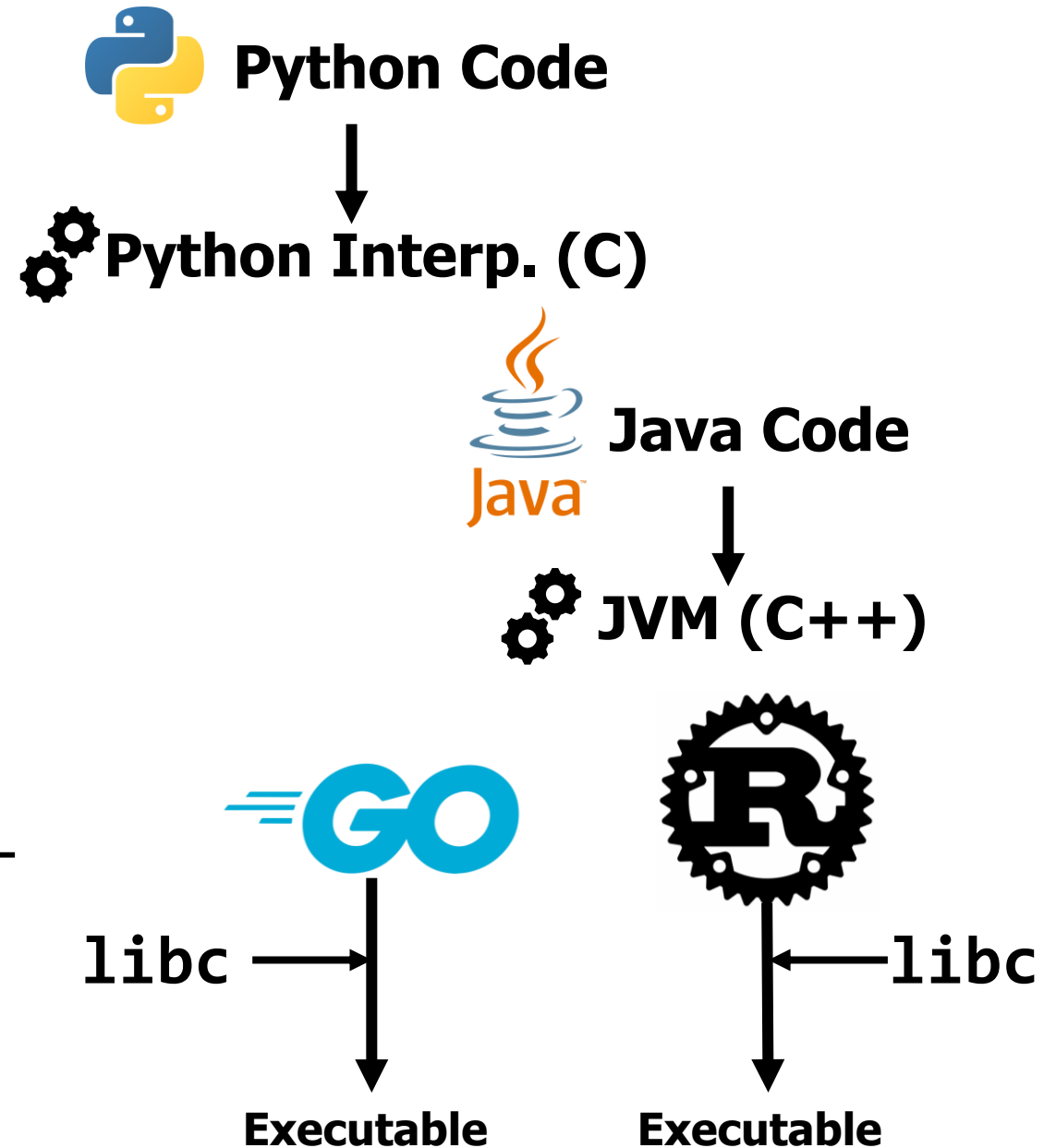


Same Reason We Don't  
Study Shakespeare as  
Translated to French



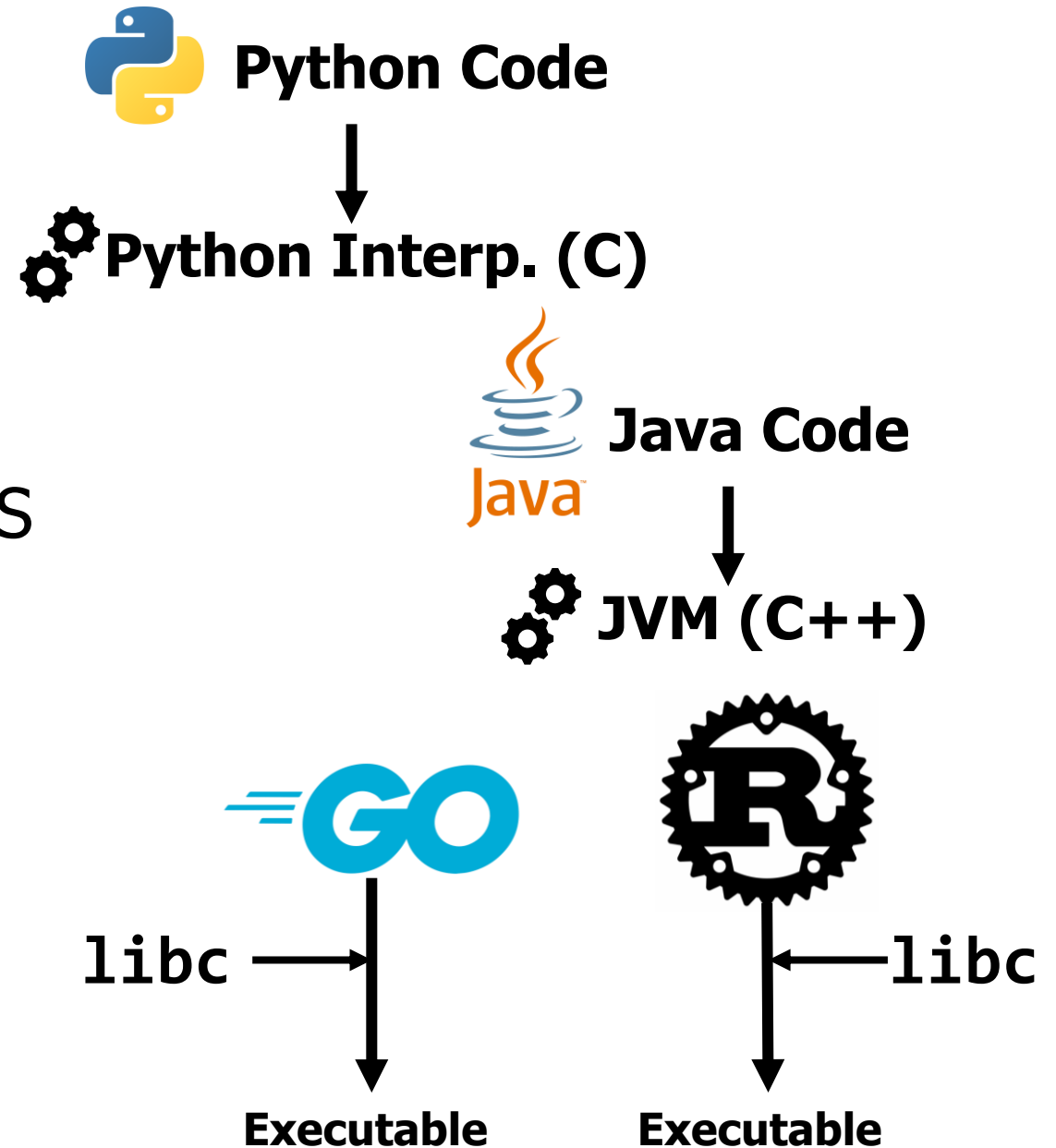
# Why use C in 4061?

- C is **the** language for operating systems and systems programming
- Almost everything else is built on top of C and its low-level libraries (libc)
- Python, Java have interpreters written in C and C++
- JavaScript interpreted by your browser (probably) written in C++
- Even trendy languages like Go and Rust rely on libc



# Why use C in 4061?

- C is **the** language for operating systems and systems programming
- These other languages have libraries that give you access to OS API
- They usually just wrap C code
- Sacrifice flexibility/expressiveness for convenience
  - Assumptions about common cases
  - Hard to work outside of these



# Some (Hopefully) Clarifying History

- In the “beginning” there was **Multics**
  - Multiplexed Information and Computing Service
- This is back in the days of mainframes shared by many users
  - Time sharing among their programs was a key concern
- Started in 1964, heavily developed into the 1970s
- Introduced a lot of ideas we now take for granted
  - Hierarchical File System (Directories and Subdirectories)
  - Dynamic Linking (map code into memory at run time)
- Very large and complex, leading to problems



# Huge Milestone in OS History: Unix

- Development began in 1969 by researchers who split from Multics
- Key Idea: Keep OS internals as simple as possible
  - Build small, sharply focused tools that combine flexibly
  - Keep Kernel implementation as small as possible while still useful
- Defined an API still in wide use today and has been hugely influential
  - The “language” in which we think about organizing computation on real systems
  - Processes, communication, files, etc.
  - Old can be a **good** thing: well-tested and has remained useful and effective over decades of technological advances
- Also often **open** to inspection and community development
- Led to many “Unix-style” OS implementations that retain this API



# Unix-Derived Operating Systems



**Notable Exception:**



Different API and history,  
same concepts



**QNX SOFTWARE SYSTEMS**





# POSIX: Modern, Unifying OS Standard

- By the 1990s and 2000s, many Unix-derived operating systems were being maintained and widely used
- All incorporated design and API ideas from original Unix
- Remember, “API” here means set of available system calls, their names, and their precise behavior (what are they guaranteed to do?)
- Different Unix descendants were largely the same but still had API differences, especially over time as new features/ideas added in
  - Makes it impossible to write code that would work on multiple systems
- POSIX defines a standard API that everyone adheres to
  - So, if you know POSIX, you can write code for many OS variants
  - Still not perfect, individual operating systems at times add own functionality



# Summarizing All of This

- Our textbook would more accurately be titled *Posix Systems Programming*
- And in this class, we'll be focusing on **Linux** systems programming
- *Confused yet?*
- We aren't writing code for Unix itself, but we are using a lot of the API features it originally defined
- Linux generally conforms to the Posix standard, and Posix is based on the ideas/conventions established in Unix
- Unfortunately, you'll see people say or write "Unix" when they really mean "Posix"
  - Not precise, but common enough that you'll have to get used to it

