Computer Systems
CSE 410 Spring 2018

Instructor:
John Zahorjan

Teaching Assistants:
William Ceriale, Clara Sutton, Ying-Chao Tung, Logan Weber
Today’s Agenda

■ Administration
  ▪ Course overview
  ▪ Staff
  ▪ General organization
  ▪ Requirements, assignments, grading
  ▪ Texts and references
  ▪ Policies

■ The course
  ▪ What it’s about, our perspective
Organization and Administration

Everything is on the course web page:
http://www.cs.washington.edu/410

Including

- General information, policies, syllabus
- Staff information, office hours (still working on that)
- Links to discussion board
- Calendar(s) with lecture slides, links to assignments, etc.
- Information and links to computing resources and reference info
- Etc
Us

Instructor

John Zahorjan, CSE 442, zahorjan@cs

Tas

William Ceriale

Clara Sutton

Tony Tung

Logan Webber

Use the discussion board for most general interest communications.
Use cse410-staff@cs to contact (all) the course staff.
You

CSE 142

CSE 143

CSE 373

CSE 410

Computer Programming 1 (4)

Data Structures and Algorithms (4)

Computer Programming 2 (5)

Computer Systems (3, maybe 4)

Computer Programming 1 (4)
You and CSE 410

- Our goal is to maximize useful things learned per minute of your time spent
- There will be some programming in C
  - We will not even attempt to give you enough experience to be a skilled C programmer
  - You can work on Linux machine klaatu.cs.washington.edu
  - You can work on the “CSE Home VM” (see course home page)
- There will be reading
- There will be “book questions”
Optional Section

- If you want *more* programming in C, there is an optional section
  - CSE 490X: Thursdays, 4:30-5:20, MOR 220

- **This** Thursday: review, in case it’s been a while since you did any programming. How to connect to `klaatu.cs.washington.edu`

- **Most** Thursdays: there will be a short programming exercise for you to complete. The TAs will introduce the exercise and then go around the room helping groups complete it before the end of the section. Bring something to compute on.
Textbooks

  - Randal E. Bryant and David R. O’Hallaron
  - [http://csapp.cs.cmu.edu](http://csapp.cs.cmu.edu)

- A good C book (not required) may be useful
  - C: A Reference Manual (Harbison and Steele)
  - The C Programming Language (Kernighan and Ritchie)
  - Web queries mostly work...

- Linux (if you want a book)
  - Linux Pocket Guide (Barrett)
Course Components

- 3 lectures per week (~30 total)
- Reading, mostly from text
- Written assignments (about weekly)
  - Problems from text to solidify understanding
- Programming assignments (“a few”)
  - C is closely related to much of the course material
- Exams (midterm + final)
  - Test your understanding of concepts and principles
- 2 in-class quizzes
  - Being worked out. Dates will be on course calendar.
Policies: Grading

- Exams: midterm 15%, final 30% of total grade
- Written assignments: weighted according to effort required
  - We’ll try to make these about the same
- Programming assignments: weighted according to effort
  - These will likely increase in weight as the quarter progresses
- Grading (aprox.):
  - 25% written assignments
  - 25% lab assignments
  - 45% exams
  - 5% other

- Late policy: 4 total “late days”; at most 2 per assignment
  - Save ‘em for later!

- Academic integrity: policy on course web
  - I trust you completely
  - I have no sympathy for trust violations – nor should you
End of Part 1

- Questions?
What is this class about?

- You’ve done extensive Java programming
- You understand computers at the level of the Java language
- How is that language supported? What is required to execute your program?
  - What does computer hardware do?
  - How is it built?
  - What is the role of the compiler?
  - What is the role of the Java runtime system?
  - What is the role of the operating system?
  - How do these components support building and running applications?
Rough course overview

Application Process \[\rightarrow\] Operating System

Software in Execution

Instruction Set Architecture (e.g., x86-64)

Machine Organization (e.g., Core i7-8550)

Logic Implementation

Hardware
The Course in a Nutshell

- It’s about interfaces and the implementation of those interfaces.

- We intend to go broad rather than deep
  - Maximize useful information per minute of effort
  - Limit workload to what’s appropriate for a 3 hour course
  - (Offer optional 4th hour via sections)

- When done, you should have a big picture understanding of how computer systems work

- You’ll end up knowing things most CSE majors do not. (But they’ll know many things you don’t as well.)
Back to Interfaces

App.java

App.class

Java Virtual Machine

Java Language Interface

Operating System Interface

Java VM Interface

C Executable

Operating System

ISA Interface

HW
Interfaces

- **Interfaces** provide abstraction
  - They separate how to use a component from how the component is implemented
  - Here’s an interface:
  - The interface stays the same even if what’s behind it changes (hydro vs. coal vs nuclear vs wind ...)
  - The interface makes few requirements on what uses it (toasters, USB chargers, lamps, ...)
  - ⇒ The interface promotes innovation
    - Both above it and below
Interfaces

- Backward compatible changes to interfaces are good

- Incompatible changes are bad
Why **Layer** Interfaces?

Java Source

Java App

Java Virtual Machine

ISA Interface

HW

Java Language Interface

Java VM Interface

versus

ISA Interface

HW

Java Source
How About This?

Java Source
→ Java App
→ Java Virtual Machine
→ HW

versus

Java Source
→ Java App
→ HW

Java Language Interface
Java VM Interface
ISA Interface
Translation

- A “program” is written against (using) some interface
  - Java program → Java language interface
    - plus Java library interfaces
  - C program → C language interface
  - Code running on HW → ISA interface
  - ISA interface → machine organization interface
  - machine organization interface → logic interface
  - logical interface → hw implementation

- In general, higher level interfaces are more expressive
  - We prefer them because it’s easier to say what we want

- Actual execution, though, relies on low level interfaces
  - For example, it’s faster for the hw to be primitive? Why?

- Main idea: write to a high level interface and use a program to automatically translate to an equivalent lower level interface for execution
  - A “compiler”
Example: C, assembly, and machine code languages (interfaces)

```latex
\noindent
\begin{center}
\begin{tabular}{ll}
\texttt{if (x \neq 0) y = (y+z)/x;}
\end{tabular}
\end{center}
```

```
\begin{verbatim}
cmpl $0, -4(%ebp)
je .L2
movl -12(%ebp), %eax
movl -8(%ebp), %edx
leal (%edx, %eax), %eax
movl %eax, %edx
sarl $31, %edx
idivl -4(%ebp)
movl %eax, -8(%ebp)
\end{verbatim}
```
Example of Translation

The three program fragments are equivalent
You'd would rather write C! – a more human-friendly language
Aside: “Decompiling”

if (x != 0) y = (y+z)/x;

Tricky (but needed for debuggers)

cmpl $0, -4(%ebp)
je .L2
movl -12(%ebp), %eax
movl -8(%ebp), %edx
leal (%edx, %eax), %eax
movl %eax, %edx
sarl $31, %edx
idivl -4(%ebp)
movl %eax, -8(%ebp)

Relatively easy
**HW/SW Interface: The Historical Perspective**

- Hardware started out quite primitive
  - The hardware was expensive to design/build/own
  - Machine time was worth more than people time
  - → There wasn’t much code in the world
  - Programmers wrote code for a particular machine
    - in binary!

![Diagram of HW/SW Interface](image-url)
An Advance: Architectures

- IBM separated the notions of the programming interface and the machine implementation
  - Could sell families of machines with different performance, and software ran on all of them

![Diagram showing ISA Specification and Hardware]
Another advance: Assemblers

- Writing binary was too hard (expensive)
- Humans preferred reading text (characters)
- **Assembler**: 1 assembly instruction → 1 machine instruction

```
cmpl $0, -4(%ebp)
je .L2
```
Higher-Level Languages

- 1 HLL instruction → many assembler instructions

```
if (i < j) i = j;
```
Making It Work: Code / Compile / Run Times

Code Time

Compile Time

C compiler

User program in C

100000110111110000 100100000111000000

.c file

.exe file
Making It Work: Code / Compile / Run Times

Run Time

Assumption: We’ll run much more than we’ll compile.
interpreted languages

- Sometimes a language wants to provide features that are not easy to compile into code that runs directly on the ISA.
- Sometimes those features would be more efficiently supported if we could "change the ISA".

- Compiled code (e.g., .class file)
- "Language ISA interface"
- Language virtual machine interpreter (e.g., JVM)
- HW ISA interface
- Hardware
Operating Systems – The Traditional Picture

“The OS is everything you don’t need to write in order to run your application”

The picture invites you to think of the OS as a layer between your code and the hardware

- In some ways, it is:
  - For example, all operations on I/O devices require OS calls (syscalls)
- In other ways, it isn't:
  - you use the CPU/memory without OS calls
  - the OS can take control of the hardware without having been explicitly called by your code
The Big Theme

- **Interfaces**
  - Especially The HardwareSoftware Interface

- How does the software you write in C/Java relate to the hardware?
- What happens during execution?
- What happens during application build (compile/link)?
### Roadmap

**C:**

```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

**Java:**

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

**Assembly language:**

```
get_mpg:
pushq   %rbp
movq    %rsp, %rbp
...
popq    %rbp
ret
```

**Machine code:**

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

**Computer system:**

**OS:**

- Windows 8
- Mac

**Memory & data**
- Binary integers

**Logic Circuits**
- Machine code
- Assembly code

**C**
- Procedures
- Arrays / structs

**Java vs. C**
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
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 electronics
- “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
  - Objects – an assembly of more primitive things
  - Classes – an assembly of instructions

- Does it matter that it’s 0 and 1, rather than 0, 1, 2, and 3?
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
  - Syntax – “y = 6;” vs. “y is 6;” vs. “6 -> y"
  - Semantics – “y has value 6”
    - Contrast with “ (y+4) = 10;” means “y=6”

- Limitation on languages: must be able to automatically translate into a lower level language (and eventually to machine code)
Little Theme 2: Translation

- Humans are good at using very high level languages with apparently no clear rules for either syntax or semantics
  - Of course, human languages often lead to many misunderstandings

- Computer languages are designed to be unambiguous
  - No misunderstanding, there is a clear (formal) syntax and semantics

- Given two unambiguous languages, we have a chance of writing an automated procedure to translate from one to the other
  - The semantics of the programs is the same
  - They compute the same values
Little Theme 3: State, Control Flow

- How do computers orchestrate the many things they are doing – seemingly in at once
- What do we have to keep track of when we call a method, and then another, and then another, and so on
- How do we know what to do upon “return”
- How do we run multiple user programs and let them share a single computer and memory
Medium Theme 4: Parallelism

- An extremely important approach to going fast is to do more than one thing at once
  - Gobally: Distributed Systems
  - Each System: Processes
  - Each Process: Threads
  - Each CPU: Cores
  - Each Core: Multiple instructions in execution at once
  - Each Instruction: Pipelining; Dynamic rewriting to break dependences
Course Outcomes

- Understanding the fundamentals of what is happening in going from creating a source file to running a program and obtaining its output

- Understanding some of the abstractions that exist between programs and the hardware they run on, why they exist, and how they build upon each other

- Knowledge of key details of underlying implementations

- Become better at thinking about problem solving in ways that have proven effective in computing