Course Wrap-Up
CSE 333 Summer 2020

Instructor: Travis McGaha

Teaching Assistants:
Jeter Arellano    Ramya Challa    Kyrie Dowling
Ian Hsiao         Allen Jung      Sylvia Wang
Administrivia

❖ hw4 due tomorrow (8/20)
  ▪ Submissions accepted until Sunday (8/23)
  ▪ If you want to use late day(s), you **MUST** let staff know. Make a private post on ed or send an email to staff letting us know you want to use late day(s).

❖ Course evaluations due Friday night
  ▪ Please fill these out! <3

❖ Grades for various assignments have been posted. **PLEASE CHECK THESE** and contact staff if something seems incorrect!!!
So what have we been doing for the last 9 weeks?

Ideally you would know everything I am talking about in this lecture, but the red stars indicate things you really should leave the course knowing.
Course Goals

❖ Explore the gap between:

- The computer is a magic machine that runs programs!
- The computer is a stupid machine that executes really, really simple instructions (really, really fast).
Lecture Outline

❖ Systems Programming: The What
❖ Systems Programming: The Why
Systems Programming: The What

- The programming skills, engineering discipline, and knowledge you need to build a system

- **Programming:** C / C++

- **Discipline:** design, testing, debugging, performance analysis

- **Knowledge:** long list of interesting topics
  - Concurrency, OS interfaces and semantics, techniques for consistent data management, distributed systems algorithms, ...
  - Most important: a deep understanding of the “layer below”
Main Topics

❖ C
  ▪ Low-level programming language

❖ C++
  ▪ The 800-lb gorilla of programming languages
  ▪ “better C” + classes + STL + smart pointers + ...

❖ Memory management

❖ System interfaces and services

❖ Networking basics – TCP/IP, sockets, ...

❖ Concurrency basics – POSIX threads, synchronization
The C/C++ Ecosystem

❖ System layers:
  ▪ C/C++
  ▪ Libraries
  ▪ Operating system

❖ Building Programs:
  ▪ Pre-processor (cpp, #include, #ifndef, ...)
  ▪ Compiler: source code → object file (.o)
  ▪ Linker: object files + libraries → executable

❖ Build tools:
  ▪ make and related tools
  ▪ Dependency graphs
Structure of C Programs

- Standard types and operators
  - Primitives, extended types, structs, arrays, typedef, etc.

- Functions
  - Defining, invoking, execution model

- Standard libraries and data structures
  - Strings, streams, etc.
  - C standard library and system calls, how they are related

- Modularization
  - Declaration vs. definition
  - Header files and implementations
  - Internal vs. external linkage

- Handling errors without exception handling
  - `errno` and return codes
C++ (and C++11)

- A “better C”
  - More type safety, stream objects, memory management, etc.
- References and const
- Classes and objects!
  - So much (too much?) control: constructor, copy constructor, assignment, destructor, operator overloading
  - Inheritance and subclassing
    - Dynamic vs. static dispatch, virtual functions, vtables and vptrs
    - Pure virtual functions and abstract classes
    - Subobjects and slicing on assignment
- Copy semantics vs. move semantics
C++ (and C++11)

- C++ Casting
  - What are they and why do we distinguish between them?
  - Implicit conversion/construction and explicit

- Templates – parameterized classes and functions
  - Similarities and differences from Java generics
  - Template implementations via expansion

- STL – containers, iterators, and algorithms
  - vector, list, map, set, etc.
  - Copying and types

- Smart Pointers
  - unique_ptr, shared_ptr, weak_ptr
  - Reference counting and resource management
Program Execution

Mostly review from 351....

❖ What’s in a process?

- Address space
- Current state
  - SP, PC, register values, etc.
- Thread(s) of execution
- Environment
  - Arguments, open files, etc.
Memory

❖ Object scope and lifetime
  - Static, automatic, and dynamic allocation / lifetime
❖ Pointers and associated operators (&, *, ->, [ ])
  ▪ Can be used to link data or fake “call-by-reference”
❖ Dynamic memory allocation
  ▪ malloc/free (C), new/delete (C++)
  ▪ Who is responsible? Who owns the data? What happens when (not if) you mess this up? (dangling pointers, memory leaks, ...)
❖ Tools
  ▪ Debuggers (gdb), monitors (valgrind)
  ◼ Most important tool: thinking!
The Operating System

- Operating System has more permissions
  - User must ask OS to handle restricted operations
  - Only OS can directly interact with hardware, read from disk, ...

- System Calls
  - OS provides an interface for User Processes to request the OS to complete a protected operation.
    - Library calls (fread/fwrite/...) will also have to go through the OS via system calls.

- I/O
  - Reading/Writing to disk takes a LONG time
    - (relative to other operations)
  - Strategies like buffering should be used to minimize number of disk accesses.
Network Programming

Client side

1) Get remote host IP address/port
2) Create socket
3) Connect socket to remote host
4) Read and write data
5) Close socket

Server side

1) Get local host IP address/port
2) Create socket
3) Bind socket to local host
4) Listen on socket
5) Accept connection from client
6) Read and write data
7) Close socket

- Error handling
- Blocking vs. non-blocking calls
Concurrency

❖ Why or why not?
  ▪ Better throughput, resource utilization (CPU, I/O controllers)
  ▪ Tricky to get right – harder to code and debug

❖ Threads – “lightweight”
  ▪ Address space sharing; separate stacks for each thread
  ▪ Standard C/C++ library: pthreads

❖ Processes – “heavyweight”
  ▪ Isolated address spaces
  ▪ Forking functionality provided by OS

❖ Synchronization
  ▪ Data races, locks/mutexes, how much to lock...
Processes vs Threads on One Slide

OS kernel [protected]

Stack_{parent}

Stack_{child}

Shared Libraries

Heap (malloc/free)

Read/Write Segment .data, .bss

Read-Only Segment .text, .rodata

Parent

PC_{parent}

PC_{child}

Child

OS kernel [protected]

Stack

Shared Libraries

Heap (malloc/free)

Read/Write Segment .data, .bss

Read-Only Segment .text, .rodata

Parent

PC

PC

Child

fork()
What is your Favourite Topic In CSE 333?
Lecture Outline

- Systems Programming: The What
- Systems Programming: The Why
Systems Programming: The Why

- The programming skills, engineering discipline, and knowledge you need to build a system
  1) Understanding the “layer below” makes you a better programmer at the layer above
  2) Gain experience with working with and designing more complex “systems”
  3) Learning how to handle the unique challenges of low-level programming allows you to work directly with the countless “systems” that take advantage of it
So What is a System?

❖ “A **system** is a group of interacting or interrelated entities that form a unified whole. A system is delineated by its spatial and temporal boundaries, surrounded and influenced by its environment, **described by its structure and purpose and expressed in its functioning.**”
  - Still vague, maybe still confusing

❖ But hopefully you have a better idea of what a system in CS is now
  - What kinds of systems have we seen...?
Software System

- Writing complex software systems is *difficult*!
  - Modularization and encapsulation of code
  - Resource management
  - Documentation and specification are critical
  - Robustness and error handling
  - Must be user-friendly and maintained (not write-once, read-never)

 Discipline: cultivate good habits, encourage clean code

- Coding style conventions
- Unit testing, code coverage testing, regression testing
- Documentation (code comments, design docs)
- If programmer discipline is interesting to you, take CSE 331!
The Computer as a System

- Modern computer systems are increasingly complex!
  - Networking, concurrency/parallelism, distributed systems
  - Buffered vs. unbuffered I/O

---

OS / app interface
(system calls)

HW/SW interface
(x86 + devices)

operating system

hardware

C application
C standard library (glibc)

C++ application
C++ STL/boost/standard library

Java application
JRE

CPU memory storage network
GPU clock audio radio peripherals
A Network as a System

- A networked system relies heavily on its connectivity
  - Depends on materials, physical distance, network topology, protocols

Conceptual abstraction layers

- Physical, data link, network, transport, session, presentation, application
- Layered *protocol* model
  - We focused on IP (network), TCP (transport), and HTTP (application)

- Network addressing
  - MAC addresses, IP addresses (IPv4/IPv6), DNS (name servers)

- Routing
  - Layered packet payloads, security, and reliability
Congratulations!

❖ Look how much we learned!

❖ Lots of effort and work, but lots of useful takeaways:
  - Debugging practice and metacognition (gdb, bug journals)
  - Reading documentation
  - Tools (git, valgrind, makefiles)
  - C and C++ familiarity, including multithreaded and networked code

❖ No exam to study for, but go forth and build cool systems!

❖ Tomorrow’s Lecture: Future Classes, Course Thanks, and AMA!