Welcome to CSE 361!
Course Overview

CSE 361s: Introduction to Computer Systems

Instructor:

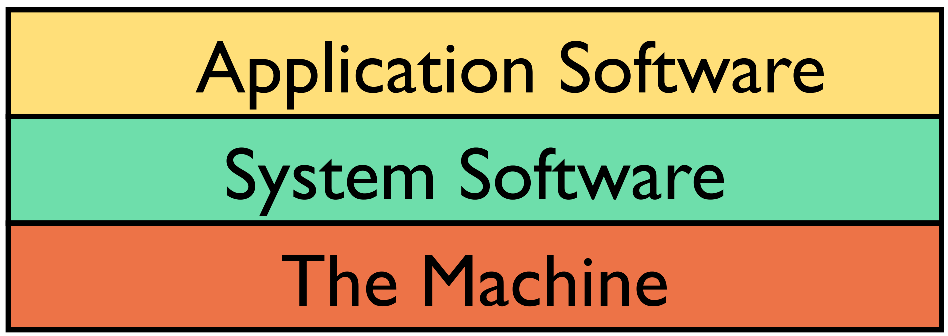
I-Ting Angelina Lee

Overview

- What this course is about
- Logistics and administrivia

What This Course Is About

- Understanding the “complete system:” the big picture



What is System Software?

- Operating System (OS)
  - Kernel, scheduler
  - File system
  - BIOS, drivers
- Utility Software
  - compiler, linker, assembler, debugger
  - libraries that provide basic facilities:
    dynamic memory allocations, file I/O, shell
  - other system calls that provide system-level services

Course Topics

- Part I: how an application program interacts with the hardware
  - program and data representation, memory hierarchy, and how these things relating to the performance of your program.
- Part II: how an application program interacts with system software and how system software serve the application and the hardware
  - linking, process, exceptional control flow, virtual memory
- Part III: how programs interact with each other
  - processes
  - threading and synchronization

A Very Loose Description of Course

"Everything you need to know about system software and hardware to write (or debug) [fast|correct|secure] software."

Course Theme:
Abstraction Is Good But Don't Forget Reality

- **Most CS courses emphasize abstraction**
  - Abstract data types
  - Asymptotic analysis
- **These abstractions have limits**
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations
- **Useful outcomes from taking this class**
  - Become more effective programmers
    - Able to find and eliminate bugs efficiently
    - Able to understand and tune for program performance
  - Prepare for later "systems" classes in CS & CoE
    - Compilers, Operating Systems, Wireless Networks, Network Security, System Security, Computer Architecture, Embedded Systems, etc.

The Five Great Realities

Code Puzzle #1

```
x = …   // initialize to some numeric value
assert((x*x) >= 0);
```

- Q: Does this assertion succeed always?

- A: depending on its data type!

Code Puzzle #2

```
float f1 = …;
float f2 = …;
float f3 = …;

assert( (f1+f2)+f3 == f1+(f2+f3) );
```

- **Q: Does this assertion succeed always?**

- **A: depending on the values of floats!**

Great Reality #1:
Ints are not Integers, Floats are not Reals

- **Example 1: Is $x^2 \geq 0$?**
  - Int's:
    - 40000 * 40000 → 1600000000
    - 50000 * 50000 → ??
  - Float's: Yes!

- **Example 2: Is (x + y) + z = x + (y + z)?**
  - Float's:
    - (1e20 + -1e20) + 3.14 → 3.14
    - 1e20 + (-1e20 + 3.14) → ??
  - Unsigned & Signed Int's: Yes!

Computer Arithmetic

- **Does not generate random values**
  - Arithmetic operations have important mathematical properties
- **Cannot assume all “usual” mathematical properties**
  - Due to finiteness of representations
  - Integer operations satisfy “ring” properties
    - Commutativity, associativity, distributivity
  - Floating point operations satisfy “ordering” properties
    - Monotonicity, values of signs
- **Observation**
  - Need to understand which abstractions apply in which contexts
  - Important issues for compiler writers and serious application programmers

Code Puzzle #3:

```
typedef struct {
  int a[2];
  long l;
} struct_t;

long fun(int i) {
  volatile struct_t s;
  s.l = 999;
  s.a[i] = 1000; /* Possibly out of bounds */
  return s.l;
}
```

- **Q: What does function 'fun' return?**
- **A: depending on its input!**

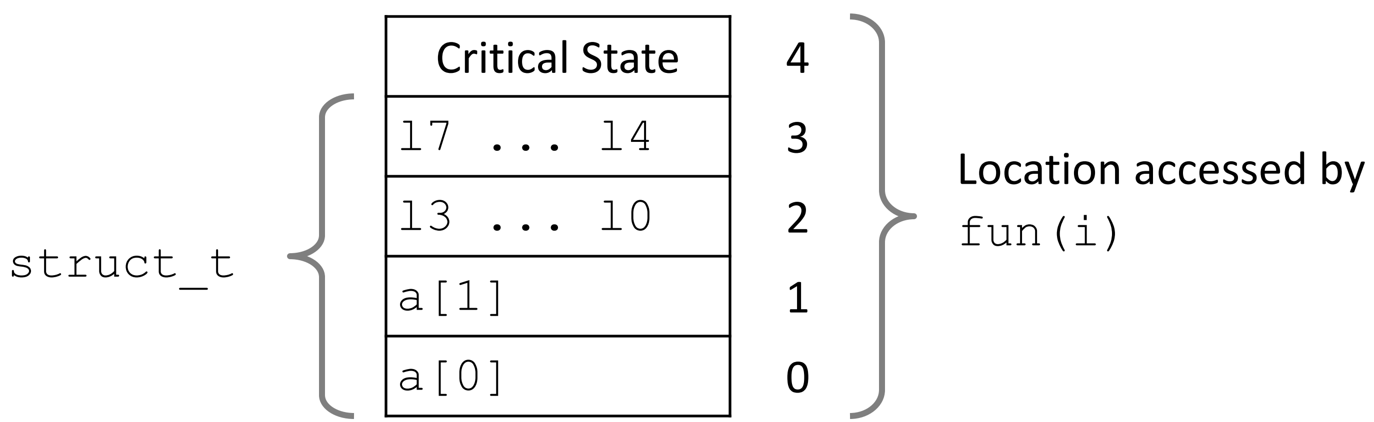
```
fun(0)  →            999
fun(1)  →            999
fun(2)  →           1000
fun(3)  →  4294967296999
fun(4)  →  Segmentation fault
```

Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  long l;
} struct_t;
```

`fun(0)`	→	999
`fun(1)`	→	999
`fun(2)`	→	1000
`fun(3)`	→	4294967296999
`fun(4)`	→	Segmentation fault

Explanation:



Result is system specific!

Great Reality #2: You've Got to Know Assembly

- **Chances are, you'll never write programs in assembly**
  - Compilers are much better & more patient than you are
- **But: Understanding assembly is key to machine-level execution model**
  - Behavior of programs in presence of bugs
    - High-level language models break down
  - Tuning program performance
    - Understand optimizations done / not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing system software
    - Compiler has machine code as target
    - Operating systems must manage process state
  - Creating / fighting malware
    - x86 assembly is the language of choice!

Great Reality #3: Memory Matters
Random Access Memory Is an Unphysical Abstraction

- **Memory referencing bugs especially pernicious**
  - Effects are distant in both time and space
- **Memory is not unbounded**
  - It must be allocated and managed
  - Many applications are memory dominated
- **Memory performance is not uniform**
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements

Memory Referencing Errors

- **C and C++ do not provide any memory protection**
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free
- **Can lead to nasty bugs**
  - Whether or not bug has any effect depends on system and compiler
  - Bug may manifest much later down the execution
    - Corrupted object logically unrelated to one being accessed
    - Effect of bug may be first observed long after it is generated
- **How can I deal with this?**
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (e.g. gdb, Valgrind)

Code Puzzle #4

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

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

- **Q: Do these two programs behave similarly?**
- **A: Their performance differ greatly!**

2.7ms* 45ms*

17 times performance difference!

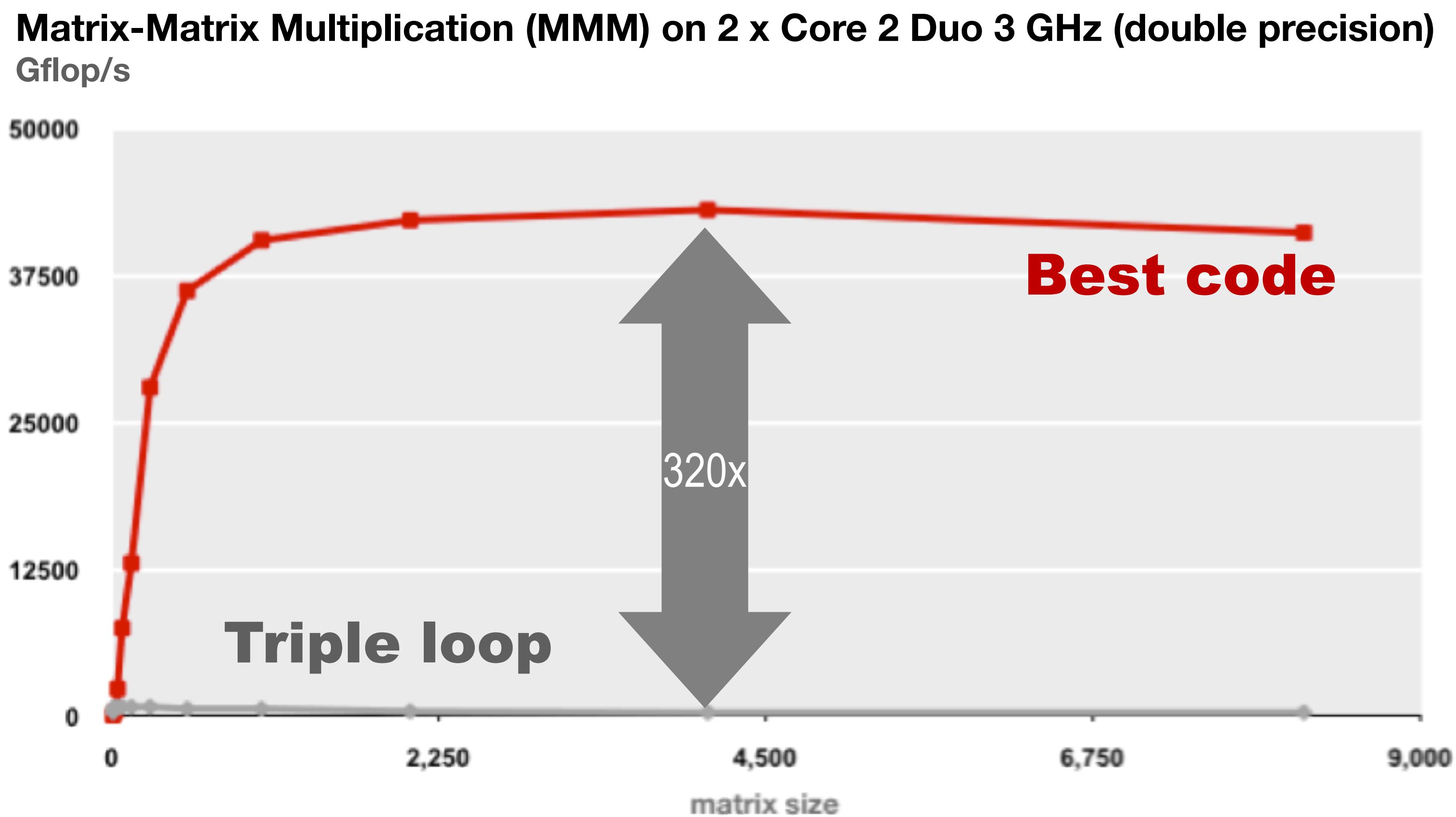
- Hierarchical memory organization
- Performance depends on access patterns

*on a 1.2 GHz Intel Xeon Processor

Great Reality #4: There's more to performance than asymptotic complexity

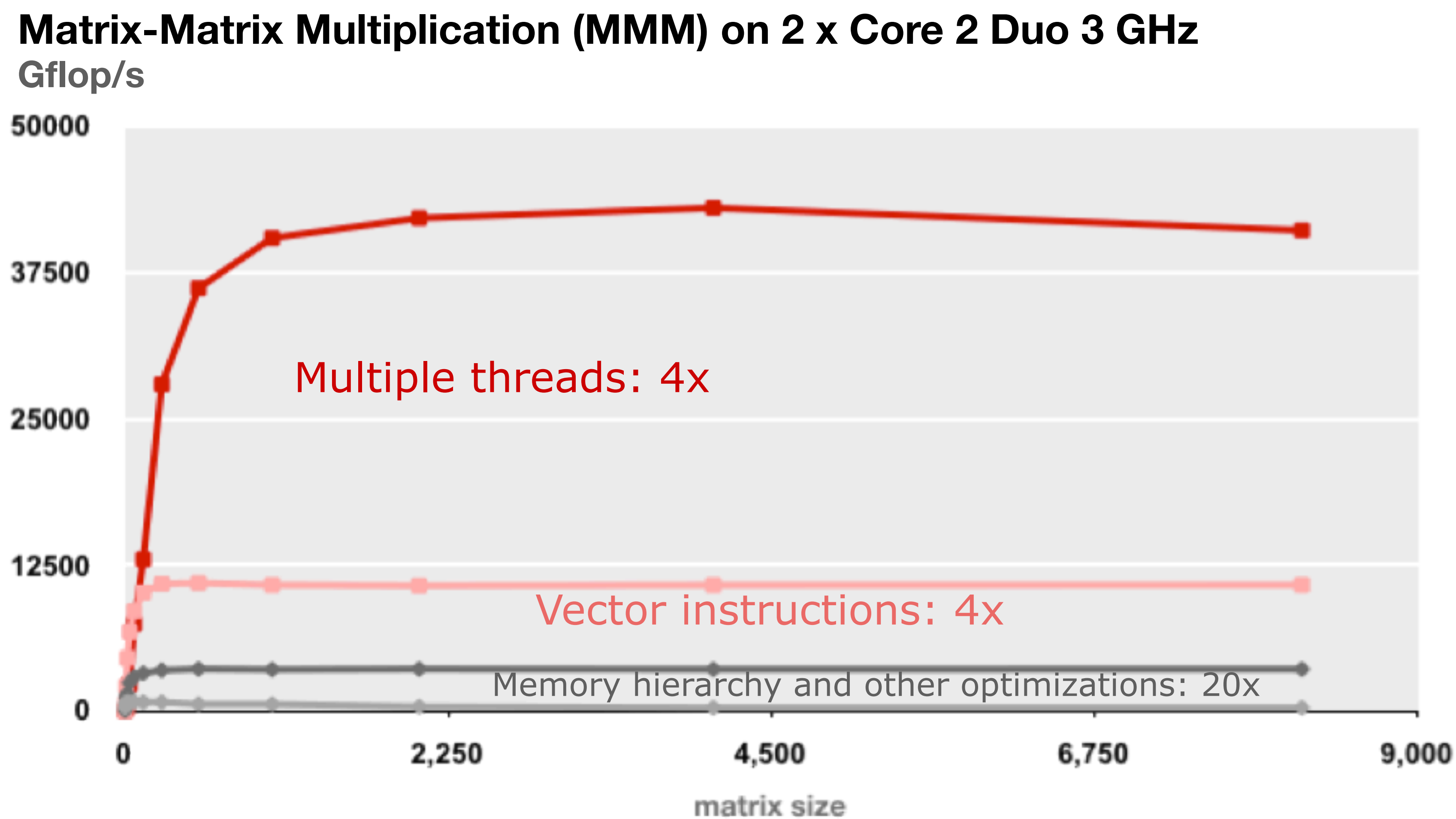
- **Constant factors matter too!**
- **And even exact op count does not predict performance**
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- **Must understand system to optimize performance**
  - How programs compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Example Matrix Multiplication



- Standard desktop computer, vendor compiler, using optimization flags
- Both implementations have exactly the same operations count ($2n^3$)
- What is going on?

MMM Plot: Analysis



- Reason for 20x: Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice
- Reason for another 16x: parallelism

Great Reality #5: Parallelism / Concurrency Matters

- **Instruction-Level Parallelism:**
  - Hardware schedules instructions out or order
  - Vectorization unit
- **The notion of processes:**
  - The OS manages the sharing of resources and maintains the illusion that your program is the only one running.
- **Parallelism among threads:**
  - Utilize the underlying hardware to finish a computation faster / process more data in shorter amount of time.
  - Manage sharing of resources at the user level.
  - Complex thread interaction can lead to programming errors that are difficult to detect / debug.
  - More complex performance issues.

Course Perspective

- Most Systems Courses are Builder-Centric
  - Computer Architecture
    - Design pipelined processor in Verilog
  - Operating Systems
    - Implement sample portions of operating system
  - Compilers
    - Write compiler for simple language
  - Networking
    - Implement and simulate network protocols

Course Perspective (Cont.)

- Our Course is **Programmer-Centric**
  - Show that by knowing more about the underlying system, one can be more effective as a programmer
  - Enable you to:
    - Write programs that are more reliable and efficient
    - Incorporate features that require hooks into OS
      - – E.g., concurrency, signal handlers, multithreading
  - Not just a course for dedicated hackers
    - *It a course for everyone who wants to be a power programmer!*

Know How and When to Use Your Tools

- linux: command line
- git: (command line)
- text editor for coding: (vi / vim / emacs)
- compilation: makefile, gcc
- debugging: gdb, valgrind

- Exercises to familiarize yourself with basic linux tools and intro to C will be released sometime this week.

Logistics and Administrivia

CSE 361 Course Staff

- Instructor: I-Ting Angelina Lee
  - Office hours: after class on Mon/Wed, 5:20-6:20pm or by appointment

- TAs:

Zihao Chen,
Noah Goldstein (head TA),
Yuchen Han,
Clayton Knittel,
Michael Liu,
Connor Monahan,
Richard Wu,
Charles Yang,
Yiheng Yao

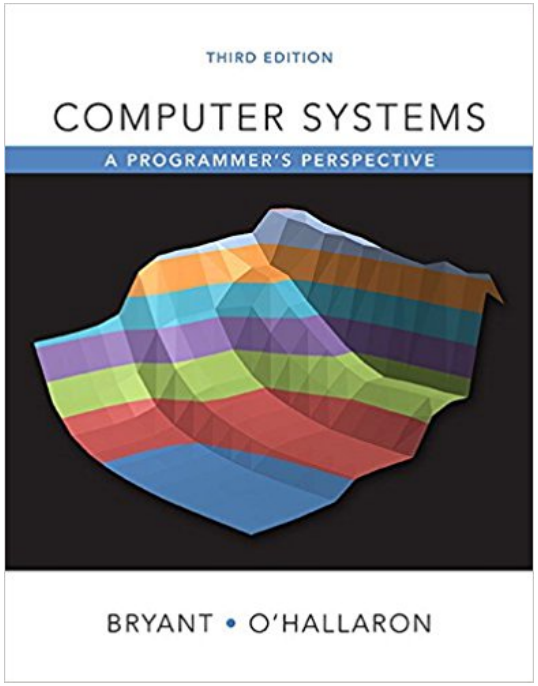
CSE 361 Online

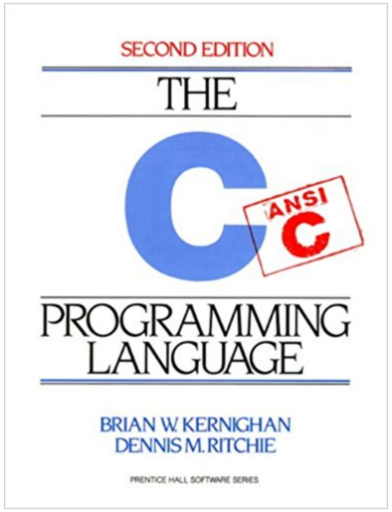
- Webpage:
  https://www.cse.wustl.edu/~angelee/cse361
  - Syllabus, textbook, etc.
  - Course schedule (lab due dates, exam dates)
  - Lab hours (TBD)

- Piazza:
  https://piazza.com/wustl/fall2019/cse361/home
  - Q & A **(do *not* use email)**
    - Answer someone else's question!
  - Lecture notes, lab assignments, sample exams

Textbooks

- Randal E. Bryant and David R. O'Hallaron,
  - "Computer Systems: A Programmer's Perspective, Third Edition" (CS:APP3e), Pearson, 2015
  - http://csapp.cs.cmu.edu
  - This book really matters for the course!
    - How to solve labs
    - Practice problems typical of exam problems





- Brian Kernighan and Dennis Ritchie,
  - "The C Programming Language, Second Edition", Prentice Hall, 1988

Course Components

- Lectures
  - Lopata 101
  - Higher level concepts, practice exercises (participate!)
- Lab Hours
  - starting 2nd week of class; check course page for hours
- Labs (5) + one extra credit lab
  - The heart of the course
  - 2-3 weeks each except for Lab 5 (~3.5 weeks)
  - Provide in-depth understanding of an aspect of systems
- Exams (midterm + final)
  - Test your understanding of concepts & principles
  - Crib sheets allowed.

Grading

- Exams (45%):
  - 20% (midterm: Wednesday, Oct 16, in class)
  - 25% (final: Friday, Dec 13, 6-8pm)
  - If you have special arrangement with Cornerstone, or it conflicts with another class, please make PRIOR arrangements with me
- Labs (55%):
  - Lab grades are weighted by difficulty:
    Lab 1: 10%, Lab 2-4: 11%, Lab 5 12%
  - Extra credit will be worth 1.5% of your overall grade.
  - You get 2-day extension per lab automatically
- Exercises:
  - Won't be graded; for your benefit only

- Grades will be posted through Canvas.

Facilities and Getting Help

- Linuxlab:
  - ssh into shell.cec.wustl.edu and do `qlogin`
  - Linuxlab desktop sessions: https://linuxlab.seas.wustl.edu/equeue/

- You can also use:
  - Your own Linux box
  - Terminal or X11 on your Mac
  - Cygwin or or secure shell from your Windows machine
  - But these options won't be supported by class staff

Facilities and Getting Help (Cont)

- Go to Lab Hours for help on projects.
- Come to office hour for lecture materiel (though I can answer questions on labs, too).

- Do the exercises!
- We will hold only limited hours during the 2nd week and official lab hours will start on the 3rd week.
  - Will post on Piazza / webpage for 2nd week hours
  - Will update on Piazza / webpage for 3rd week hours right before the week starts

Academic Integrity

- What is cheating?
  - **Searching online for possible solution**
  - **Copying code** from previous terms or from the web
    - Only allowed to use code **we** supply
  - **Sharing code:** by copying, retyping, looking at, or supplying a file
  - **Coaching:** helping your friend to write a lab, line by line
  - Warning: once you see a solution, it is **very hard to un-see**
- What is NOT cheating?
  - Explaining how to use systems or tools
  - Helping others with high-level design issues

TO DO

- Make sure that you are signed up for Piazza
- Read Chap 1 of the textbook
- If you are not already familiar with Linux command lines / C, plan to work through exercise 1 before starting lab 1.
  - Exercise 1 will be posted soon on Piazza

Welcome and Enjoy!