Review Notes, CS 107

Major topics on the final exam:

1. Floating Point
2. x86-64 Assembly
3. Runtime Stack
4. Managing the heap / heap allocation

Minor topics on the final exam:
1. Program optimization (manual and compiler)
2. git
3. gcc / ELF format
4. Makefiles

Possible topics from before the midterm:
1. void * arrays
2. function pointers
3. bits/bytes

Major topic big ideas:

Floating Point

- You should understand the 32-bit floating point format to the point where you can identify normalized, denormalized, and exceptional floating point values by looking at a 32-bit binary floating point value.
- You should be able to write out the floating point format for special numbers such as +/-0, +/-0 infinity, FLT_MAX, FLT_MIN.
- You should be able to convert simple numbers to and from the 32-bit floating point format. E.g., numbers such as low integers (2, 5, 8, etc.), numbers that have small negative powers of to components (e.g., 0.5, 0.625, 0.75).
- You should understand associativity to recognize when addition / subtraction will not produce exact results because of magnitude differences.
- Given two floating point numbers in decimal, you should be able to determine if the result of an operation on those numbers will be able to be represented exactly in floating point format. E.g., 1.0 / 2.0: produces 0.5, which can be represented exactly; 1/10: produces 0.1, which cannot be represented exactly.

x86-64 Assembly

- You should be able to reverse-engineer an assembly program to produce the C code it came from, given some scaffolding (e.g., we will provide an outline of the code, which may or may not have details like while statements, or if statements).
- The above bullet point also requires you to know how the assembly instructions listed on the 1-page assembly handout work (we will provide that to you for the exam)

Runtime Stack

- You need to understand how the stack works in a function written in x86-64 assembly, to include information about the stack frame for a function.
- You should understand buffer overflows and how they can cause behavior that the original programmer would not expect (e.g., seg faults, or compromised code).
- You should be able to explain the reason for caller-owned and callee-owned registers, and demonstrate how they are used in an assembly language program.
- You should recognize and be able to explain recursive functions in x86-64 assembly.

**Managing the heap / heap allocation**

- You should understand bump allocators, implicit free lists, and explicit free lists as they relate to heap allocation.
- You should be able to write basic heap allocation functions for a heap that is well-defined (e.g., we can make a slightly different type of heap, and you should be able to write coalesce functions, expand functions, etc., for that heap — see the practice exam from last quarter for an example).

  **Minor topic big ideas:**

**Program optimization (manual and compiler)**

- You should understand the five different types of compiler optimizations (Constant folding, Common subexpression elimination, Strength reduction, Code motion, Dead code elimination), and be able to describe how the compiler has modified code to perform those optimizations.

**git**

- You should understand how to make a commit in git, and how to include a commit message.
- You should understand the “git log” command.
- You should understand how to see the status of your files with the “git status” command.

**gcc / ELF format**

- You should understand what the pre-processor does for `#define` and `#include` statements.
- You should be able to lightly describe the different types of sections in the ELF format (.text, .data, .rodata, .bss, .comment). E.g., “The .text section is the executable code in your program).
- You should understand what the difference between static linking and dynamic linking.
- You should be able to describe the four parts of the program creation process, in light detail (Preprocessing, Compiling, Assembling, Linking)

**Makefiles**

- Given a very basic Makefile, you should be able to point out the target, the dependencies, and the command that will run.
- You should be able to describe when a Makefile will run commands for a target (e.g., when the timestamp for a dependency is older than the target)
- You should be able to describe some of the benefits of using Makefiles