CS 429H
Assignment 1

This assignment covers material in Chapters 1 and 2 of "Computer Systems: A Programmer's Perspective". Familiarity with the reading in those chapters will help significantly.

Your assignment is to provide solutions for Chapter 2 problems 2.57, 2.60, 2.61, 2.63, 2.66, 2.67, 2.69, and 2.70.

Notes:

- Note the difference between an *expression* and a *function*. An expression is a single statement without any semicolons, e.g. $((x + y)|(z))\&x$. A function may include multiple expressions, assign expressions to variables, and have control flow (if allowed by the problem).
- Pay attention to which problems require the artificial bit-level integer coding rules, and which do not. A clarification of the rules: arbitrary integer constants are allowed, such as `0xFF` or `0x1`.

A fun additional problem.

Build the smallest 64-bit x86 program you can that deterministically returns zero. By "smallest" I mean the sum total of all the code you run should be as small as possible. Submit two files, one file called "small.S" that has your code and a Makefile that will build your executable.

Let's see what happens with the following C program, "int main(){return 0;}" which I store in the file small.c. I build it using "gcc -g small.c" and I get an executable that is 8,327 bytes long. That isn't so bad. But wait. If you type "readelf -h a.out" you will see that you have an entry point address, which is where the operating system starts executing your binary. If you run "gdb a.out" and type "x/30i 0x4003d0" (where that hex address is whatever readelf told you was your entry point), then you will find that the OS plans on executing a function called _start, not main. _start will do a bunch of work, including loading and initializing the C runtime library. Run "ldd a.out" and you will see that your 8,327 byte executable loads code from several other libraries. You can compute how much code is in each of those libraries. Oh, and fun fact ld-linux-x86-64.so.2 isn't a library at all, it is the name of Linux's dynamic linker, but that is really too advanced for this point in the course.

We thought we had built a small C program, but we built a small C program that dynamically (at runtime) loads a bunch more of the program, namely the C runtime library. Can we build an executable that does not load any code at run time?

Let's go back to the drawing board and compile our file with "gcc -g –static -o small small.c." Now I have an executable that is 88,0467 bytes long, but at least I'm not cheating anymore, my executable won't load any more code into its address space when it executes. All code my executable ever needs is in the program binary. But that binary does include the C runtime library which is linked into my executable. The initialization for the C library will run before my code in main. At one point many years ago, I measured over a million instructions executed to initialize the C runtime library before arriving at the first instruction of main.

We thought we were building a small executable by creating small.c. But it turns out that compiling a C program links in the C runtime library, which is large.

Oh, you might be wondering how to test the return code of a process. Probably the easiest way is to type "echo $?" at your shell prompt after running the program in question. Your shell might use a different

symbol, you can google something about printing the exit status of a process for your shell (running "ps" should give you the name of your shell. You can change it with chsh).

So what we really need to do is write an assembly language program and tell gcc to build a static executable without system libraries, and a minimum of other stuff (like your final executable probably won't have debug information). Good luck with that! Read about options to gcc.

One tip, store your assembly code in a file called small.S (use a capital S) and start the file with the line ".globl main" which should build with "gcc -o small small.S." The .globl directive tells the assembler that the main label is globally visible (visible to functions outside of this file). Make sure your Makefile creates an executable file named small when run without arguments (just typing, "make"). You can run standard utilities (i.e., anything installed on department machines) in your makefile in addition to gcc if you want.

One more hint. You might be tempted to make your main function a single instruction that zeros register %eax (the register that holds the return value for C function). That would work if you were building a minimal C function that deterministically returns zero. But you are building a program that returns zero and programs exit by communicating their desire to die to the operating system.

Submitting your assignment.

You will turn in your assignment using the submit function in canvas. Login to canvas and choose 'Assignments'. Choose the assignment, (eg: Assignment 1) and click Submit Assignment on the right hand side. Upload a zip file containing a pdf or text (.txt) file of the solutions to Chapter 2 problems from the book, the small.S file which is your solution to the additional question and the Makefile for building your small.S file with the appropriate gcc command line options. Your Makefile must create an executable file named small. Then click the submit assignment button to make your submsission. Please note that only electronic submissions will be accepted and corrupt or missing data will be counted as an unsubmitted assignment. Your assignment is due at 11:59 pm, Jan 28, 2014.