COSC 301: Operating Systems
Lab 3: Testing and Debugging

In this lab, you'll learn about facilities and programs for testing and debugging programs. I'll walk through debugging and testing one program in the classroom, then you'll apply similar techniques to debug two broken programs.

In the class github repo, there are broken1.c and broken2.c, which you will need to debug and fix. After you debug the programs, you'll just need to demo them to me, including showing me the code.

When you get started, make a new directory to do the debugging, and **put the two files in git**. The repo you make does not need to be stored on github - a local repo will be fine. So, create a new directory, copy the two `.c` files in the directory (along with the Makefile), and do a `git init`, `git add .`, and `git commit -am "initial commit"`.

The reason to put the two files in the repo is to be able to easily perform a `diff` on the original state of the files. After you've made the fixes, you'll need to demo your fixes to me - the easiest way is to use `git diff` to compare the initial state with the final state.

One more thing: to use `gdb` and `valgrind` effectively, you need to have compiled your program with `-g`, which includes "debugging symbols". Without the debugging symbols, the tools are almost useless.

An overview of the techniques we'll use:

Testing: `assert` and feeding in standard input

`assert` If you have `#include <assert.h>` at the top of our C file, you can use the `assert` function. This function accepts one Boolean expression as a parameter. If the expression yields non-zero (`True`), the function won't do anything else. If the expression yields `False`, the `abort` function will be called to terminate your program. Using `assert` provides a way to check whether a condition holds or not, and thus gives a very basic way to unit test functions.

`assert` isn't awesome by itself, but I often create a macro that wraps the `assert` call to do something more useful, like:

```
#define myassertstr(a, b, message) \
    printf("%s: %s ?= %b:\n", message, a, b); \
    if (!a) \
        printf("\tOops: a is NULL\n"); \
    if (!b) \
        printf("\tOops: b is NULL\n"); \
    assert(0 == strcmp(a,b)); \
    printf("\ttest passed.\n"); \

// call the macro function
// assumes that strings one and two have been defined
myassertstr("Comparing one and two", one, two);
```

Creating a helper program for feeding stdin Many C programs take input from the console (standard input). For testing these programs, I often create a separate "driver" program to create specific input streams that I want to test. I then use a shell pipeline to feed the output from my test driver into the program I'm testing:

```
$ python mkinput.py 1 | ./program
```

More often, I run the test program under the valgrind tool, like:

```
$ python mkinput.py 1 | valgrind ./program
```

`valgrind`

Valgrind is a tool that can detect memory corruption, memory leaks, and other bad behaviors. You can run it by simply saying:

```
$ valgrind ./program
```

There are many different options you can give to valgrind to modify its behavior. The two most useful options are `--leak-check=full`, and `--track-origins=yes`. The first option will do detailed checks for any memory leaks your program may have. The second option is useful when tracking down the origin of a memory allocation that later is implicated in a problem. To use one of the options, just say something like:

```
$ valgrind --leak-check=full ./program
```

Type `valgrind --help` for all available options.

`gdb`

`gdb` is a symbolic debugger than can be used to step through a program line by line, and to inspect or modify anything in the address space of the running program. `gdb` is more complex to use than `valgrind`, but can be incredibly helpful for tracking down and verifying the source of a problem. To use `gdb`, type:

```
$ gdb ./program
```

At the `gdb` prompt, you can type `help` to get lots of help. Here are a set of commands that I find most useful:

`run` Restart the program from the beginning. You can say `run x y z` if your program takes arguments from the command line.

`info stack` Print a representation of the stack. You can type `up` to go up one stack frame and inspect any variables, or `down` to go down a stack frame.

`list` List the source code of your running (or crashed) program, near where the debugger is currently stopped.

`print x` Where `x` is a variable name, gdb will print the value currently stored in that variable. There's also the `inspect` command, which is more complicated, but can do things that print cannot.

`break function` If `function` is a valid function name in your program, gdb will stop the program when you enter that function. You can then inspect any variables and find out what's going on. `continue` will cause the program to start running again, from the currently stopped point.

`next` Execute the next source code line of the program, and stop. After you hit a breakpoint, it's often useful to step the program forward line by line and print things.

There are many, many other commands in gdb. See the help menus within gdb, or use Google to find out more.

Before you leave the lab

1. Demo/show your fixed code (both broken1.c and broken2.c) to me.
   - You'll need to show me the differences between your starting and ending code with `git diff`, so be prepared.