

COSC 301: Operating Systems

Lab 6: Testing and Debugging

In this lab, you'll learn about facilities and programs for testing and debugging programs. Your main task for this lab is to fix a piece of broken code. You will use the `gdb` and `valgrind` tools to do this (described below). Once you've fixed the code, you can simply demo your fixes to me. Clone the git repo at https://github.com/jsommers/cosc301_lab06 to get started.

In the repo there is a file named `broken.c`, which is the file you'll need to read, debug, and fix. There's a `Makefile` in the repo to build the file - just type `make`. You should:

- fix the bug(s) in `broken.c`.
- eliminate any memory leaks in the program.

The correct output from the program should be:

```
1: apple
2: banana
3: coconut
4: date
5: fig
6: grape
I finished! (but you still need to check memory leaks!)
```

Described below are three techniques and tools for debugging programs. I won't describe "printf debugging", since you are probably quite familiar with that technique!

Using `assert` in C

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(message, a, b) \
    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("\tttest passed.\n"); \

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

It is a good idea to use `assert` as you are developing your code to verify that certain conditions hold, or are true. For example, if in a function you expect each pointer parameter to be non-NULL, you can write `assert` expressions to test that. If something ever happens (a bug!) that causes a NULL value to be passed to the function, you'll immediately find out.

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`. Type `help` within `gdb` to access internal `gdb` help.

Before you leave the lab

1. Demo/show your fixed code to me. Show me `valgrind` output that confirms that you have eliminated the memory leaks.
 - You'll need to show me exactly the code differences between what you started with, and your fixed version. You can use `git diff` for that purpose.