ITF22519 - Introduction to Operating Systems

Lab 2: GNU Compiler and Debugging Tool

Autumn 2020

In this lab, you will become familiar with development tools for C programming in Linux. You will be asked to perform a few exercises and read various manuals available in the lab.

Before you start, remember to `commit` and `push` your previous lab to your git repository. Then, try to `pull` the new lab:

```
$ cd ITF22519/labs
$ git pull upstream master
$ cd lab2
```

Please remember these steps for the future labs.

1 Development Tools

1.1 Compiler

A compiler converts source code into object code or executable code. The GNU Compiler Collection (gcc) is included with Linux. Several versions of the compiler (C, C++, Objective C, Fortran, Java and CHILL) are integrated; this is why we use the name "GNU Compiler Collection". GCC can also refer to the "GNU C Compiler", which is the `gcc` program on the command line. You will be using a lot of C in this course.

1.2 Compiling and Linking Multiple C Files

Create three files in your preferred editor:
`message.h` contains:

```
void print_message();
```

`message.c` contains:

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "message.h"
```

```
static const char* message[] = {
    "Hello class",
    "Goodbye class",
    "This is ITF22519 course.",
    "This is a lab section."
};

void print_message() {
    int index;
    srand(time(NULL));
    index = rand()%4;
    printf("%s\n", message[index]);
}
```

lab2.c contains:

```
#include "message.h"
int main(int argc, char** argv) {
    print_message();
    return 0;
}
```

If your code was typed in correctly, you can compile and link the two C files into an executable by typing this sequence of commands:

```
$ gcc -c message.c
$ gcc -c lab2.c
$ gcc -o lab2 lab2.o message.o
```

To run the program, type:

```
$ ./lab2
```

Note: why "`./`"? Recall the discussion of directory navigation in the previous lab. `.` is a reference to the current directory. Since the shell needs to find the file you want to run, you have to specify the complete path (or have the file in a directory stored in the `PATH` environment variable). You can think of `./` as a shortcut for the absolute path up to the current directory. Don't forget this – you'll need it to run programs for the rest of the semester.

When you use the `-c` option in gcc, the C files are compiled into object files (`.o`) that can be linked together into an executable. Because this project is quite small, it is also possible to compile all of the C files at one time with the command

```
$ gcc -o lab2 lab2.c message.c
```

Compiling the C files to object files is less memory intensive when you have hundreds of files in a project. In addition, since compiling from C to object is much more expensive than linking object files, compiling to object files will allow you to only recompile files that have changed, rather than an entire project. The `make` utility automates this process.

1.3 Make

The make utility automatically determines which pieces of a large program need to be recompiled and then issues commands to recompile them. It is easier to use than running a series of gcc commands, and less prone to typos. It also allows for flags (such as -g to add debugging information, or -O to enable optimizations) to be added to all files at one time.

We will learn more about GNU Make in the next labs. For now, please read the GNU Make manual (`info make`) or find an online tutorial to figure out how to write a Makefile that will compile and link the example in the previous section. There are multiple ways to do this experiment to see what the differences are and which way avoids recompiling files when only one of the two .c files changes.

1.4 Exercise 1 (40 points)

Submit the three files: `message.h, message.c, lab2.c` into `lab2` folder of your repository. Also, you have to submit your `lab2_report.txt` file into `lab2` folder. In this report, explain in short what this `lab2.c` program does.

2 GNU Debugging Tool

The GNU Debugging Tool (GDB) is a very powerful debugger that allows single stepping through code, setting breakpoints, and viewing variables. This section is a brief tutorial of GDB.

2.1 Stepping Through Code

Let's look at a trivial example program which reads in a comma separated variable (CSV) file and calculates the average of all the numbers in the file. Compile the file `csv_avg.c` using the command

```
$ gcc -o csv_avg csv_avg.c
```

And now run the file by running

```
$ ./csv_avg test.csv
```

If we wanted to understand how this code worked we could just try to read through it or use print statements, but as code projects get larger and more complex this becomes less viable. Instead let's debug with GDB. Compile the code with the -g flag and start GDB.

```
$ gcc -g -o csv_avg csv_avg.c
$ gdb ./csv_avg
```

First we need to set a break point which is a point where the program execution will pause allowing us to see what is going on in the program. Let's see what numbers we are reading into the buffer at line 46 by setting a break point there.

```
(gdb) break 46
```

Next we can start running the program by typing `run` at the prompt:

```
(gdb) run
```

Notice however that it exits with a usage message because we haven't told it what to use as command line arguments. Run the program again this time passing the csv file to it

```
(gdb) run test.csv
```

Notice that we stop at the breakpoint we set. Note, execution stops right before executing the printed line. To execute the current line and go to the next line use the command next. We can then print a variable using the print command.

```
(gdb) next
(gdb) print buffer[0]
```

We can type `continue` to resume execution until a breakpoint is hit again. Repeating this a couple times shows that we are scanning the file.

```
(gdb) continue
(gdb) next
(gdb) print buffer[1]
```

Let's set another breakpoint at line 49 now. If we use continue again we still break at our previous breakpoint so let's remove that breakpoint.

```
(gdb) break 49
(gdb) clear 46
(gdb) continue
```

Next we can check the value of `i` and see how many times the program looped.

```
(gdb) print i
```

You can see the value of `i` is 17 since there are 16 numbers inside the `test.csv` file. Next lets check the `average` function. To step into that function type the command

```
(gdb) step
```

We are now at the begining of the `average` function. Type `next` a couple of times to go through one iteration of the loop. Now print `sum` to see what the sum is.

```
(gdb) print sum
(gdb) next
(gdb) next
(gdb) print sum
...
```

To step out of a function use the command

```
(gdb) finish
```

To finish execution to the end type `continue`. To stop debugging type the command `quit`.

```
(gdb) continue
(gdb) quit
```

2.2 Debugging Segfaults

Segfaults are an invalid memory access error, and can be one of the most difficult bugs to track down. Compile and run the program `test_malloc.c`. You can **ignore the warning** when compiling the file for now.

```
$ gcc -o test_malloc test_malloc.c
$ ./test_malloc
```

The program is waiting for input from standard in. Type any string and press enter. You should now see `Segmentation fault` print and the program will exit. Next Look at the code in an editor. Everything seems to make sense so let's try debugging with GDB.

```
$ gcc -g -o test_malloc test_malloc.c
$ gdb ./test_malloc
```

First let's just run it in GDB and see what happens:

```
(gdb) run
```

Again, type any random string and press enter. We see that the program received a signal `SIGSEGV`. To see what functions were last called run a `backtrace`:

```
(gdb) backtrace
#0  0x00007ffff7e6ef60 in __GI__IO_getline_info (fp=fp@entry=0x7ffff7fb9a00 <_IO_2_1
n=1023, delim=delim@entry=10, extract_delim=extract_delim@entry=1, eof=eof@entry=0x0
#1  0x00007ffff7e6f038 in __GI__IO_getline (fp=fp@entry=0x7ffff7fb9a00 <_IO_2_1_stdi
n=<optimized out>, delim=delim@entry=10, extract_delim=extract_delim@entry=1) at iog
#2  0x00007ffff7e6dfeb in _IO_fgets (buf=0x0, n=<optimized out>, fp=0x7ffff7fb9a00 <
#3  0x000055555555518c in main (argc=1, argv=0x7fffffffe5b8) at test_malloc.c:18
```

This shows that the last function we wrote that was called was `test_malloc.c`:18 which is line 18 of `test_malloc.c`. Since this is all we are interested in let's switch the stack frame to frame 3 and see where the program crashed:

```
(gdb) frame 3
#3  0x000055555555518c in main (argc=1, argv=0x7fffffffe5b8) at test_malloc.c:18
18              fgets(buffer, 1024, stdin); // get upto 1024 characters from STDIN
```

Since we assume that `fgets` works let's check the value of our argument. `stdin` is a global variable created by `stdio` library so we assume it is alright. Let's check the value of `buffer`:

```
(gdb) print buffer
$1 = 0x0
```

The value of `buffer` is `0x0` which is a `NULL` pointer. This is not what we want since `buffer` should point to the memory we allocated using `malloc`. Let's now check the value of `buffer` before and after the `malloc` call. First kill the currently running session by issuing the `kill` command and answering y.

```
(gdb) kill
```

Next set a breakpoint at line 16:

```
(gdb) break 16
```

Now run the program again:

```
(gdb) run

Breakpoint 1, main (argc=1, argv=0x7fffffffe5b8) at test_malloc.c:16
16              buffer = malloc(1<<31); // allocate a new buffer
```

Check the value of `buffer` by issuing `print buffer`. It may or may not be garbage since it has not yet been assigned.

```
(gdb) print buffer
$2 = 0x0
```

Let's step over the `malloc` line and print buffer again:

```
(gdb) next
17              printf("Please enter your name:\n");
(gdb) print buffer
$3 = 0x0
```

So `malloc` returned `NULL`. If we now check the man page for `malloc` we see that it returns `NULL` if it cannot allocate the amount of memory requested. If we look at the `malloc` line again we notice we are trying to allocate `1<<31` bytes of memory, or 4GB. Therefore it is not a surprise that the `malloc` would fail. So we can change the amount of memory allocated to 1024 bytes that we are actually using and the program executes as expected. Also, ALWAYS check the return values of system calls and make sure that they are as expected. To quit GDB issue the `quit` command.

2.3 Exercise 2 (10 points)

Fix the bug in `test_malloc.c`, recompile and check if there is any error and submit your corrected file.

2.4 Exercise 3 (50 points)

Compile and run the program `rand_string.c`. This program takes a string as an input and outputs ten random characters from that string. Please use the techniques from this lab to figure out why the program is not working correctly. In `lab2_report.txt` file, explain in short how you figure out the bug and submit your correct version of `rand_string.c`.

Hint: The bug is related to the **first for loop**. This is an **index overflow exception** and the program tries to write to an **illegal memory location** of its stack. That is why there is **"stack smashing detected"** when you use gdb to debug the program. So, next time you encounter this kind of message, pay attention to the index of the array.

3 What To Submit

Complete the the exercises in this lab and put everything into the **lab2** directory of your repository. Run `git add .` and `git status` to ensure the file has been added and commit the changes by running `git commit -m "Commit Message"`. Finally, submit your files to GitHub by running `git push`. Check the GitHub website to make sure all files have been submitted.