ITF22519: Introduction to Operating Systems

Fall Semester, 2021

Lab4: GNU Compiler and Debugging Tools

Submission Deadline: September 23th, 2021 23:59

This lab will introduce development tools for C programming in Linux and debugging tools (also called debuggers) for you to identify errors in your code. Before you start, remember to commit and push your previous lab assignment and related files to your Git repository. Then, try to pull the new lab:

```
$ cd Introduction2OS/labs
$ git pull upstream main
$ cd lab4
```

1 Development Tools

If you want to write your program, you need a develoment tool first. The development tool is a computer program that a developer uses to create, debug and maintain the program. Development tools can be of many forms such as compliers, debuggers, linkers, integrated development environments (IDEs) etc..

1.1 Compiler

When you write your program in a middle-level or a high-level programming language, you need to translate it into a languge that your machine can understand. This is where a complier comes in. A compiler takes your source code and converts it into object code or executable code.

The GNU Compiler Collection (GCC) is an optimizing compiler produced by the GNU Project and supports various programming languages, hardware architectures and operating systems. GCC is included in Linux. Several versions of the compiler such as C, C++, Fortran, Java etc are integrated. This is why we have the name `GNU Compiler Collection`. However, GCC can also refer to the "GNU C Compiler," which is the `gcc` program on the command line. To know more about `gcc`, use:

```
man gcc
```

1.2 Compiling and Linking Multiple C Files

A C program can be split up into multiple C files. This make it easier to control especially when the program is large. Splitting also allows individual C file to be compiled independently. Let's do simple practice with compiling and linking multiple C files. Create three files as the following:
`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!",
    "See you again!",
    "Goodbye!",
    "This is Lab4 assignment!",
    "This course is Introduction to Operating Systems course!"
};

void print_message() {
    int index;
    srand(time(NULL));
    index = rand()/(RAND_MAX/5);
    printf("%s\n", message[index]);
}
```

lab4.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 lab4.c
$ gcc -o lab4 lab4.o message.o
```

To run the program, type:

```
$ ./lab4
```

Note:
Why "`./`"? If you recall lab1 with the discussion of directory, you would be reminded that "`.`" 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.

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 once with the following command:

```
$ gcc -o lab4.c message.c lab4
```

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

If you want to run or update a program when certain files are updated, the `make` utility can come in handy. It 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.

Please read the GNU Make manual (`info make`) or find a tutorial online 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.

2 GNU Debugging Tool

Developers often want to find out why a program does not work as it is supposed to do. Debugging is nothing but a process of finding software errors (or bugs) in a program. There are several debugging tools (`debuggers`) which can be used for debugging. In this lab, we will practice with GNU debugging tool (`gdb`).

`gdb` is a very powerful debugger that allows single stepping through code, setting breakpoints, and viewing variables. In other words, it allows a developer to see what is going on inside a program while the program is in execution or what the program is doing at the moment it crashes.

2.1 Stepping Through Code

Let's take a look at a trival example program which reads in a comma seperated variable (CSV) file and calculates the average of all the numbers in the file. Compile the file using the command:

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

or:

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

And now run the file by:

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

If you want to understand how this code works, you could just try to read through it or use print statements. However, if your code is large and complicated, this may be impossible. Instead of reading and using printing statement, let's debug the code with (`gdb`). To do so, compile the code with the (`-g`) flag and start (`gdb`).

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

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

```
(gdb) break 46
```

You should be able to see something like

```
Breakpoint 1 at 0x12d7: file csv_avg.c, line 46.
```

Next, you can start running the program by typing `run` at the prompt. Notice, however, that it exits with a usage error because you have not told 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 you stop at the breakpoint you have set. Please be aware that execution stops right before executing the printed line. To execute the current line and go to the next line, use the command `next`. You can then print a variable using the `print` command.

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

You can type command `continue` to resume execution until a breakpoint is hit again. Repeating this a couple times so that you are scanning the file.

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

Let's set another breakpoint at line 49 now. If you use `continue` again, you still break at your previous breakpoint. Therefore, let's remove that breakpoint.

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

Next you 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
```

You 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. 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 Segmentation Faults

A Segmentation Faults (or segfaults) is the common condition that causes your pgoram to crash. Segfaults are caused when your program tries to read or write the the memory location that it is not allowed to. Segfaults can be one of the most difficult bugs to track down. Compile and run the program test `malloc.c`. You can ignore the warnings when compiling your code 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` printed and the program will exit. Next, look at the code and 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_stdin_>, 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_stdin_>,
n=<optimized out>, delim=delim@entry=10, extract_delim=extract_delim@entry=1) at iogetline.c:34
#2 0x00007ffff7e6dfeb in _IO_fgets (buf=0x0, n=<optimized out>, fp=0x7ffff7fb9a00 <_IO_2_1_stdin_>)
#3 0x000055555555518c in main (argc=1, argv=0x7fffffffe5b8) at test_malloc.c:18
```

This shows that the last function called was `test_malloc.c:18` which is line 18 of `test_malloc.c`. Since this is all you 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, 512, stdin); // get upto 512 characters from STDIN
```

Since we assume that `fgets` works, let's check the value of your 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 you want since buffer should point to the memory you 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
18 fgets(buffer, 512, stdin); // get upto 512 characters from STDIN
(gdb) print buffer
$3 = 0x0
```

So the `malloc` returned NULL. If you now check the man page for `malloc`, you would see that it returns NULL if it cannot allocate the amount of memory requested. If you look at the malloc line again, you would notice that you are trying to allocate 1¡¡31 bytes of memory, or 4GB. Therefore it is not a surprise that the `malloc` would fail. Therefore, you can change the amount of memory allocated to another suitable value that you 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`, use `quit` command.

3 Exercises

3.1 Exercise 1 (50 points)

Submit three files `message.h`, `message.c`, `lab4.c` into your `lab4` repository on GitHub.

- Run ./lab4 several times.
- Explain what the `lab4.c` program does.

3.2 Exercise 2 (20 points)

Fix the bug in `test_malloc.c`

3.3 Exercise 3 (50 points)

The program in `rand_string.c` takes a string as an input and outputs ten random characters from that string.

- Compile and run the program.
- Why the program does not work properly.
- Fix the bug.

4 What To Submit

Complete the exercises in this lab, each exercise with its corresponding `.c` files. After that, put all of files into the **lab4** directory of your repository. Make a report for Exercise 1,2, and 3. 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.