ITF22519: Introduction to Operating Systems

Fall Semester, 2022

Lab4: GNU Compiler, Debugging Tools, and SEGFAULT

Submission Deadline: September 20th, 2022 23:59

You need to get at least 50pts to pass this lab assignment.

This lab introduces development tools for C programming in Linux and debugging tools (also called debuggers) for you to find out why your code does not behave as expected. You will also learn about segmentation fault, or SEGFAULT, which you may encounter while writing a C program. 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 OS2022/labs
$ git pull main main
$ cd lab4
```

1 Development Tools

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

1.1 GNU Compiler

When you write your program in a middle-level programming language (for example C) or a high-level programming language (for example Java, Python), you need to translate it into a language that your machine can understand. This is where a compiler 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 be referred to the "GNU C Compiler," which is the `gcc` program on the command line. When you use the `gcc` command, the most important thing while compiling a source code file (i.e, a .c file) is the name of the source file. The rest is option. The simplest form of `gcc` command is:

```
gcc source_file.c
```

Here, *source_file* is the name of a *.c* file you want to compile. The command compiles the *source_file* file and give the output file as **a.out** file - the default name of output file given by gcc compiler. The **a.out** file can be executed using

```
./a.out
```

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.

- **-o:** option

Example:

```
gcc source_file.c -o out
```

This command compiles the *source_file* file but instead of giving default name **a.out**, you have **out** - whatever you want to name - executable file. Execution is done by using command

```
./out
```

- **-c:** option

Example:

```
gcc -c source_file.c
```

This command compiles the *source_file* and gives the object file - *source_file.o* file- as output, which is used to make libraries.

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:

File *message.h* contains:

```
void print_message();
```

File *message.c* contains:

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

static const char* message[] = {
    "Hello",
    "Goodbye",
    "See you again",
    "This is a lab section."
};
```

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

File *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
```

Recall that 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 file. 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 lab4.c message.c
```

In the above command, `lab4` is the name of the executable file output - whatever you name it- which you can run later on by `./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 Makefile and Make

On UNIX systems, there is a program called `make` (with numerous variants such as *gmake*, *pmake*) that reads the **Makefile**, which tells it which files are dependent on which other files. What `make` does is see which object files are needed to build the binary and for each one, check to see if any of the files it depends on (the code and headers) have been modified subsequent to the last time the object file was created. If so, that object file has to be recompiled. When `make` has determined which *.c* files have to recompiled, it then invokes the C compiler to recompile them, thus reducing the number of compilations to the bare minimum [1]. 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.
To practice, in your lab4 repository, make a **Makefile** file as follows:

```
CC=gcc
CFLAGS=-I.

lab4: lab4.o message.o
    $(CC) -o lab4 lab4.o message.o
```

Delete *lab4.o* and *message.o*. Now, run

```
$ make
```

in the command line to see what happens. Next, modify *message.c* and run `make` command again. You can also practice with the following content of the **Makefile**:

```
lab4: lab4.c message.c
     gcc -o lab4 lab4.c message.c -I.
```

When you type `make` on the command line, it will execute the compile command as you have written it in the **Makefile**. If you have error, then you need tab in the second line of the **Makefile** file.

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 (bugs) in a program or why the program does not behave as expected. 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 separated 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
```

The output of the above steps will be more or less as the followings:

```
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./csv_avg...
(gdb) break 46
Breakpoint 1 at 0x12df: file csv_avg.c, line 46.
(gdb) run
Starting program: /home/ttdinh/OS2022/labs/lab4/csv_avg
Usage:
        /home/ttdinh/OS2022/labs/lab4/csv_avg csv-file
[Inferior 1 (process 2102734) exited with code 0377]
(gdb) run test.csv
Starting program: /home/ttdinh/OS2022/labs/lab4/csv_avg test.csv

Breakpoint 1, main (argc=2, argv=0x7fffffffe578) at csv_avg.c:46
46                      rv = fscanf(fp, "%d, ", &buffer[i++]);
(gdb) next
39              while(!feof(fp))
(gdb) print buffer[0]
$1 = 19
(gdb) continue
Continuing.

Breakpoint 1, main (argc=2, argv=0x7fffffffe578) at csv_avg.c:46
46                      rv = fscanf(fp, "%d, ", &buffer[i++]);
(gdb) next
39              while(!feof(fp))
(gdb) print buffer[1]
$2 = 9324
(gdb) break 49
Breakpoint 2 at 0x555555555324: file csv_avg.c, line 49.
(gdb) clear 46
Deleted breakpoint 1
(gdb) continue
Continuing.

Breakpoint 2, main (argc=2, argv=0x7fffffffe578) at csv_avg.c:49
```

```
49              printf("Average = %f\n", average(buffer, i));
(gdb) print i
$3 = 17
(gdb) step
average (buffer=0x55555555a4c0, count=17) at csv_avg.c:8
8               long sum = 0;
(gdb) print sum
$4 = 0
(gdb) next
9               for(i=0;i<count;i++)
(gdb) next
11                      sum += buffer[i];
(gdb) print sum
$5 = 0
(gdb) finish
Run till exit from #0  average (buffer=0x55555555a4c0, count=17) at csv_avg.c:11
main (argc=2, argv=0x7fffffffe578) at csv_avg.c:49
49              printf("Average = %f\n", average(buffer, i));
Value returned is $6 = 1629.41174
(gdb) continue
Continuing.
Average = 1629.411743
[Inferior 1 (process 2102773) exited normally]
(gdb) quit
```

3 Segmentation Fault (SEGFAULT)

3.1 Introduction

A Segmentation Faults (or SEGFAULT) is the common condition that causes your program 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. The followings are some examples of segmentation faults:

3.1.1 Example 1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){
        char *s;

        s[0] = 0;
        return 0;
}
```

Since `s` is a poiter, you cannot assign `s[0] = 0`.

3.1.2 Example 2

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main(){
        char s[100] = "";
        int count= 0;

        printf("\nEnter a string:");
        scanf(" %[^\n]s",s);
        printf("You have entered: %s\n", s);
        printf("The size of the string is %d ", strlen(s));
        printf("Printing %c ", s[1000000]);

        return 0;
}
```

In this code, `s` is a string initiated with the size of 100. But the code tries to access `s[1000000]` which does not belong to `s`.

3.1.3 Example 3

```
#include <stdio.h>
#include <stdlib.h>

int main(){
   int *array;
   array[10] = 10;

   return 0;
}
```

The reason for segmentation bug in this example is the same as the Example 1. The code can be fixed as

```
#include <stdio.h>
#include <stdlib.h>

int main(){
   int *array = (int*)malloc(20*sizeof(int));
//   arr[10] = 10;
   *(array + 10) = 100;
   printf("Element array [10] is %d and stored at the address %p \n", *(array + 10), array + 10);

   return 0;
}
```

3.2 Debugging Segmentation Faults

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
```

You will see something like:

```
#0  0x00007ffff7e6ba71 in __GI__IO_getline_info (fp=fp@entry=0x7ffff7fb4980 <_IO_2_1_stdin_>, buf=buf@
    at iogetline.c:77
#1  0x00007ffff7e6bb58 in __GI__IO_getline (fp=fp@entry=0x7ffff7fb4980 <_IO_2_1_stdin_>, buf=buf@entry=
#2  0x00007ffff7e6aa56 in _IO_fgets (buf=0x0, n=1024, fp=0x7ffff7fb4980 <_IO_2_1_stdin_>) at iofgets.c
#3  0x00005555555551a8 in main (argc=1, argv=0x7fffffffe568) at test_malloc.c:10
(gdb)
```

This shows that the last function called was `test_malloc.c:10` which is line 10 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
\end{Verbatim
You should see something like:
\begin{Verbatim}
#3  0x00005555555551a8 in main (argc=1, argv=0x7fffffffe568) at test_malloc.c:10
10              fgets(buffer, 1024, stdin); // get upto 1024 characters from STDIN
(gdb)
```

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 8 to check the buffer:

```
(gdb) break 8
```

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
```

You should see something like the following:

```
Starting program: /home/ttdinh/OS2022/labs/lab4/test_malloc

Breakpoint 1, main (argc=1, argv=0x7fffffffe568) at test_malloc.c:8
8               buffer = malloc(1<<31); // allocate a new buffer
(gdb)
```

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
9               printf("Please enter your name:\n");
```

Print buffer to check what is currently in the buffer.

```
(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.

```
(gdb) quit
A debugging session is active.

        Inferior 1 [process 2175528] will be killed.

Quit anyway? (y or n) y
```

4 Exercises

4.1 Exercise 1 (50 points)

- Submit three files `message.h`, `message.c`, and `lab4.c` into your `lab4` repository on GitHub.
- Use **Makefile** and **make** to link `lab4.c`, `message.c`, and `hello.c` and make executable output `lab4` file. (You may have to change source files also)
- Call the function implemented in `hello.c`, change the content in `message.c` and `hello.c` files.
- Run ./lab4 several times and copy screen capture in the report.
- Explain what the `lab4.c` program does.

4.2 Exercise 3 (50 points)

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

- Compile and run the program.
- Why there is segmentation fault.
- Fix the bug.
- When the number of characters in the input string is less than 10, the string output always has 10 characters. Explain why
- Change the code in `rand_string.c` so that the input string and output string has the same length.

5 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 and 2. 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.

References

[1] Andrew S. Tanenbaum and Herbert Bos.: Modern Operating Systems. I4th Edition, 2015.