CMPS 6100 Lab 07

In this lab, you explore the shell and learn the basics of C programming.

Some prompts will require you to edit various source files and others will require answers will go in `answers.md`.

Refer back to the README.md for instruction on git, how to test your code, and how to submit properly to get all the points you've earned.

Our Server

For this lab (and the following labs), we have provisioned a linux server for your use! If you are curious, this is a virtual machine provided in "the cloud" on Microsoft Azure.

Our server gives you a linux environment to interact with and a common environment in which to implement the lab problems in this and future labs. We have created users for each of you. Your username is your tulane username, and your password has been initialized to be your student ID. This isn't secure. You will have to change this!

The environment is solely text based. You will interact with it through a terminal, the most powerful interface to interact with and control computers. The first step will be to connect to it.

You can connect to the server using `ssh`. `ssh`, or Secure SHell, is a terminal program that allows you to remotely login to other computers.

Connect to our server by opening a terminal and issuing the following command (replace `username` with your user name). `$` indicates a shell prompt. You do not need to type `$`. Your command begins with `ssh`.

```
$ ssh username@cmps-lab.cs.tulane.edu
```

For example:

```
$ ssh amaus@cmps-lab.cs.tulane.edu
```

You will be prompted for your password. Your password is set by default to be your student ID. Enter it and you will see a shell prompt logged in to the server for you.

Change you password! Your default password is not secure. Once logged in, issue the command:

```
$ passwd
```

This will prompt you to enter a password. Choose and set a secure password for your account. If you don't, and we have any hackers break in though your account, we will know you didn't choose a secure password!

What is a shell?

As mentioned, a shell is the most powerful interface we have to computers. GUI and (now-a-days) voice interfaces are limited by what they are programmed to do. The shell is only limited by the capabilities of the computer itself.

A *shell* is an interactive command-line interpreter that runs programs on behalf of the user. A shell repeatedly prints a prompt, waits for a command line on stdin, and then carries out some action, as directed by the contents of the command line.

The command line is a sequence of ASCII text words delimited by whitespace. The first word in the command line is either the name of a built-in command or the pathname of an executable file. The remaining words are command-line arguments. If the first word is a built-in command, the shell immediately executes the command in the current process. Otherwise, the word is assumed to be the pathname of an executable program. In this case, the shell forks a child process, then loads and runs the program in the context of the

child. The child processes created as a result of interpreting a single command line are known collectively as a job. In general, a job can consist of multiple child processes connected by Unix pipes.

If the command line ends with an ampersand "&", then the job runs in the background, which means that the shell does not wait for the job to terminate before printing the prompt and awaiting the next command line. Otherwise, the job runs in the foreground, which means that the shell waits for the job to terminate before awaiting the next command line. Thus, at any point in time, at most one job can be running in the foreground. However, an arbitrary number of jobs can run in the background.

For example, entering the command:

```
$ jobs
```

causes the shell to execute the buit-in `jobs` command. Entering the command:

```
$ sleep 5
```

runs the sleep program for 5 seconds in the foreground. This causes a process to sleep for 5 seconds before continuing. `5` is a command line argument that modifies the behavior of `sleep`. Alternatively entering the command:

```
$ sleep 10 &
```

runs the sleep program in the background. At this point, if you run `$ jobs` you will see it listed!

Getting to know the shell

There are some basic commands that you should become familiar with in order to operate on the shell.

- `pwd`: Present Working Directory
  - Print to standard out (`stdout`) the present working directory. This is your current location in your file system
- `ls`: List files
  - List the files and directories in the present working directory
- `cd`: Change Directory
  - Change the present working directory to a new directory
- `cat`: Concatenate Files
  - Concatenate the contents of files and print them to stdout. This is most commonly used to print the contents of a single file.

Most importantly:

- `man`: Manual
  - Display the manual for a given program. e.g, `$ man ls` displays the manual for `ls`, giving a description of this program, a list of all command line arguments, and other information as well.
- `help`: Help
  - Display a list of the shell's built-in commands, or get help for an individual one of them. E.g, `$ help pwd` or `$ help cd` displays the help information about `pwd` and `cd` respectively.

Other useful commands:

- `mkdir`: Make Directory
- `touch`: Create an empty file
- `history`: Display the history of all commands you have entered
- `exit`: Exit your shell. This will close your ssh session.
- `vi`: Edit the specified file with the Vim text editor. Vim Basics Tutorial
  - `$ vi file.txt` opens file.txt in the vim text editor.
- `git`: Issue any of the git commands to interact with a git repository.

For an introduction to the shell, from the shell itself, issue the command:

```
$ man intro
```

Secrets

In your home directory, I have hidden a secret directory with a secret file.

Find this directory and file.

The file contains the assembly code for a simple C program. In it, you will find a section with the instructions that correspond to the main method. Other sections have to do with the initialization and running of the program.

In the main method, you see that a value is stored in a memory location, the location given as `[rbp-0x4]`. Operations are performed on this value and the value in `[rbp-0x4]` at the end of the program is the final value of the operations.

1. What value is stored in `[rbp-0x4]`, what operations are performed on it, and what is the final result of the operations? In assembly, the values are given in hexadecimal. Give your answers in decimal (base-10).

   Enter your answers in answers.md (4 pts)

Git on the Shell

For the remainder of this lab, you will be implementing small C programs. To get the starter source, you will need to clone your repository to our `cmps-lab` machine.

In your home directory is a `labs`. This is where you can clone this and the following labs for this course.

Clone your repository

To clone your repository for this lab:

- Navigate into your `labs` directory
- Copy your `HTTPS` clone link from GitHub.
- Inside of your labs directory, issue the following command:

  ```
  $ git clone YOUR-CLONE-LINK
  ```

A directory will be created and all files will be downloaded into this directory.

Updating GitHub

As your implement the following programs and answer the questions, you should be committing and pushing your changes to GitHub.

Let's make a first commit right now.

Add your name to the top of `answers.md`

Using Vi

- Navigate into your repository.
- Open `answers.md` using `vi`.

  ```
  $ vi answers.md
  ```

- Vi has two basic modes, Command and INSERT mode. It starts in Command mode. To enter INSERT mode, hit the `i` key on your keyboard. You should now see –INSERT– at the bottom of the window.

- Navigate your cursor, using the arrow keys on your keyboard, to the location you want to type your name and type your name.
- Leave –INSERT– mode by hitting the `ESC` key.
- Save and quit the file by typing `:` to pull up a command prompt at the bottom of the screen. Type `wq` and press enter. This will "write" and "quit" your file.

Now let's commit and push this change to GitHub.

Making a Commit

Being a text only interface, you will interact with git through text commands.

Enter the command:

```
$ git status
```

This will print out the current status of your repository. You should see that `answers.md` has changed.

Stage your commit by issuing the command:

```
$ git add answers.md
```

Issue `git status` again to see that it has been done.

Make a local commit by issuing the command:

```
$ git commit -m "Add name to answers.md"
```

The message that follows `-m` is required. You should customize each message to the context of your commit.

If you don't give a message, git will open a text editor (likely vi) where it will want you to enter a message, then save and quit the file.

With the commit made, push it to GitHub:

```
$ git push origin main
```

You will follow this process to push every change to GitHub. You should do this often. Individual commits should be small.

C Programming

In this section of the lab, you will become acclimated with writing C programs. While we have solely focussed on Python so far this semester, after learning one language, picking up a second is much easier. C is lower level than Python. It lacks Python's flexibility and ease of use, but in exchange, C is very powerful and gives us direct access and control of memory and simple, low level access to the Operating System.

In these programming prompts, you will start small and built up to implementing your own shell in the next lab.

All source files for this lab are in the `src` directory in this repository.

As you learn C Programming and work through problems, an excellent resource for the C libraries is Harvard's CS50 Manual Page. Bookmark it!

Hello, World!

2. In `hello.c`, print to stdout "Hello, World!". Compile and run your program.

   To compile `hello.c`:

   ```
   $ gcc -g hello.c -o hello
   ```

This creates an executable named `hello` for your program

To run `hello`:

`$ ./hello`

Pro tip: You can compile an run on the same line;

`$ gcc -g hello.c -o hello && ./hello`

This compiles `hello.c` and runs it if the compilation suceeded.

The `-g` argument tells gcc to compile with debugging symbols. I recommend compiling all programs for our class with debugging symbols enabled.

C Variables and Pointers

There are two major differences between C's variables and Python's variables. In C, every variable must have a type, that is, before using a variable, we need to declare it, stating what it will hold.

For example:

```
int num;
double pi;
char letter;

num = 5;
pi = 3.14159265;
letter = 'A';
```

Jargon wise, the top three lines are **declarations** and the bottom three are **intializations**. Each of these variables is restricted to only holding the types declared.

The second difference is that C exposes the memory locations of our variables. We get a variable's memory location using the `&` operator.

E.g,

```
int num = 7; // declare and initialize on one line
// printf is C's print function. It works similar to python's
// formatted printing. %p is a placeholder for a value to be
// printed. The 'p' indicates that the value should be formatted
// as a memory address. \n indicates to print a new line character
// after the value.
printf("%p\n", &num);
```

We can store memory addresses in variables known as **pointers**. We declare a pointer using the `*` operator.

e.g,

```
int num = 7;
// declare a pointer to an int and store in it
// the memory address where num is stored.
int* ptr = &num;
printf("%p\n", ptr);
```

We can also use the `*` operator to "dereference" a pointer to access the value at that memory location.

```
int num = 7;
int* ptr = &num;
// deference ptr and print the value stored there
// will print 7. The placeholder '%d' indicates to
// formate the value as an int. The 'd' stands for
```

```
// decimal number, that is, as a base-10 number (as
// opposed to a binary, octal, or hexadecmial number).
printf("%d\n", *ptr);
```

3. We'll start our exploration of C by playing with variables and pointers. In `pointer.c`, declare and initialize two ints. Call them `num1` and `num2` and given them arbitrary different values. **(2.5 pts)**

    a. Print out the memory addresses of `num1` and `num2`. What values did you assign to `num1` and `num2`? What are the memory addresses of `num1` and of `num2`?

    Enter your answers in answers.md

    b. Declare two pointers to ints, `ptr1` and `ptr2` and assign into them the memory addresses of `num1` and `num2`. Using the dereference operator, print out the values pointed to by `ptr1` and `ptr2`.

    Enter your output from this step in your program to answers.md

    c. It is possible to increment or decrement a pointer. Increment `prt2` by 1. Print out the memory address now stored in `ptr2`. How does this relate to `ptr1`?

    Enter this memory address and your answer to answers.md

    d. Finally, using the dereference operator, update the value pointed to by `ptr2` to be a new value. Now print out `num1`. What happened?

    Enter your output and answer to answers.md.

    e. What does all of this tell you about memory safety in C? Does C protect variables from indirect access via pointers?

    Enter your answers in answers.md

Arrays

Let's now explore arrays in C. You are already somewhat familiar with arrays even if you haven't heard of them. Python's Lists are higher level versions of an array.

An array is just a contiguous set of memory locations. In C, we can declare an array as follows:

```
// declare an array that can hold 10 ints.
int arr[10];
```

Once declared, an array's size is set. Arrays can neither grow nor shrink in size after being created. The reason for this is simple. Once memory is allocated for them, the memory after them is allocated for other purposes and so they can not grow. They do not shrink for simplicity. Managing a bunch of holes in allocated memory would be hard.

By the way, what to do if you need more space? Create a new larger array and copy everything over. This is how python gives the illusion that lists can grow arbitrarily large.

Just like in Python (Python stole the syntax!), individual elements can be accessed via index notation:

```
// print out the first thing in the array
printf("%d\n", arr[0])
// print out the last thing in the array
printf("%d\n", arr[9])
```

4. In `array.c` the above array have been declared for you. Print out it's contents using a `for` loop. **(2.5 pts)**

    a. Record the contents of your array in answers.md.

    Enter the output of your program in answers.md

    b. Run your program several times. Do the values change? Speculate about what you are seeing.

Enter your answers in answers.md

c. Modify your `for` loop to continue printing values past the end of the array. Access and print the values from your array for indices 0-14. Does C allow you to do this?

Include the output of your program and enter your answer in answers.md

d. In C, an array variable is a pointer to the first memory address of the array. Where is the array `arr` stored in memory? Print out the memory address of the beginning of `arr`. What is the memory address where `arr[0]` is stored? What is the memory address where arr`[1]` stored?

Enter your answers in answers.md

e. What does the difference between the memory addresses `arr[0]` and `arr[1]` tell you about the size of an integer? How many bytes of memory does an int require?

Strings in C

In C, a string is an array of characters:

```
char str[10];
str[0] = 'A';
str[1] = ' ';
str[2] = 's';
str[3] = 't';
str[4] = 'r';
str[5] = 'i';
str[6] = 'n';
str[7] = 'g';
str[8] = '!';
str[9] = '\0';
printf("%s\n", str);
```

All strings end in the special `'\0'` character. If this looks, too cumbersome, yes! C does have a `string.h` library that gives us a slightly better way of initializing a string.

```
char str[10];
strcpy(str,"A string!");
printf("%s\n", str);
```

`strcpy` copies the string into the array given.

An even easier way is to use pointers. As we've seen, an array is a pointer to the beginning of the array. In C, we can declare a pointer to a `char` and store a string there.

```
char* str = "A string!"
```

The C compiler automatically creates an array large enough for `str` based on the number of characters in the string literal.

Based on this idiom, it is easy to create an array of strings:

```
char* lst[4];
lst[0] = "The";
lst[1] = "quick";
lst[2] = "brown";
lst[3] = "fox";
```

Command Line Arguments

In is possible in C (and very common!) for a program to take in command line arguments as input and/or to modify its behavior.

A C program which takes in command line arguments has a main method declared as follows:

```
int main(int argc, char* argv[]) {

}
```

`argv` is an array of strings. The 0^{th} string is command given to run this program and all subsequent ones are the command line arguments that follow it.

E.g, if `myprogram` is a C program:

```
$ ./myprogram --verbose --input file.txt
```

It's `argv` array will have the following contents:

```
argv[0]: "./myprogram"
argv[1]: "--verbose"
argv[2]: "--input"
argv[3]: "file.txt"
```

`argc` or "argument count" is the number of strings in `argv`, in this case, 4.

5. Implement a program, `cmdargs.c` which takes in command line arguments and prints each one to stdout on a new line. **(1 pt)**

   Example output:

   ```
   $ ./cmdargs 5 19 a 13
   ./cmdargs
   5
   19
   a
   13
   ```

String Splitting

In the next lab, you will implement your own tiny shell. One of the key subroutines is to take in a command entered as a string on the command line, split it into its arguments, and store them in an array so that they can be passed into a program as an `argv` array.

Being used to implement a shell, you function will have a couple of important stipulations.

First, it should treat any command line arguments passed in within single quotes as a single arguments.

For example:

```
"./myprogram 2 'input string' 7"
```

Should be stored in an array as:

```
args[0]: "./myprogram"
args[1]: "2"
args[2]: "input string"
args[3]: "7"
```

Second, it needs to detect if the command given should run in the background or foreground (see What is a Shell? above). Your function should return `1` if the command should be run in the background, `0` if it should be run in the foreground.

That's it.

6. Implement the function `parseline`. **(15 pts)**

   `parseline` takes in two parameters: `const char* cmdline` and `char* args[]`. `cmdline` is the command line to parse. `const` indicates that this string may not be modified. `args` is the array where you will store each argument of the command line, including the program to execute (see above).

   > `args` will declared outside of your function and passed in to it. Modifications you make to it will persist outside of your function. You do not need to and should not return `args`. See `parseline_tests.c` for usage of your function.

   Your function should return `1` if the command should be run in the background, that is, if the last argument passed in is `'&'`, `0` otherwise.

   Any argument enclosed in single quotes should be treated as a single argument.

   > Advise: Start small and work your way up. Given the string `"hello world"`, can you split it into two strings by appropriately placing a `\0` character (to indicate the end of "hello") and getting pointers to be beginning of each word?
   >
   > Likewise, can you detect a `'` character in a string? What about splitting the string `"a 'b c'"` into two strings `"a"` and `"b c"`?
   >
   > It is good practice to experiment with little tasks like this in a test program, then, once you have them working, incorporate them into your main program.

   Do not add a main method to your `parseline.c` source file. You will break the automated tests if you do so. Instead, a runner for your `parseline` function has been provided for you: `parseline_runner.c`. It includes a few examples so that you can see how your function will be called and for you to check your behavior. You can compile and run it like normal:

   ```
   $ gcc -g parseline_runner.c -o parseline_runner && ./parseline_runner
   ```

   The automated test, `parseline_tests.c` is provided in the `tests` directory for you to verify that your program will pass the tests. See `tests/README.md` for instructions on how to run it.

Dealing with Segmentation Faults

Is your program crashing with a `Segmentation Fault`? This is often due to a null pointer reference involving strings, or more generally a memory error relating to the misuse of pointers. Fortunately there is a good tool to find out exactly where the problem lies in your code: `gdb`

I have a program that I know crashes: `foo_bar`. I compiled it with debugging symbols enabled.

If I run it, I see:

```
$ ./foo_bar
Segmentation fault
```

To debug:

```
$ gdb ./foo_bar
Reading symbols from ./foo_bar...
(gdb)
```

You are given the prompt `(gdb)`. Enter the command `run` to execute your program.

```
(gdb) run
Starting program: /home/amaus/foo_bar

Program received signal SIGSEGV, Segmentation fault.
__strlen_evex () at ../sysdeps/x86_64/multiarch/strlen-evex.S:77
```

```
77  ../sysdeps/x86_64/multiarch/strlen-evex.S: No such file or directory.
(gdb)
```

To find out where the SEGFAULT was generated from, issue the command `backtrace`:

```
(gdb) backtrace
#0  __strlen_evex () at ../sysdeps/x86_64/multiarch/strlen-evex.S:77
#1  0x00007ffff7e58f76 in __vfprintf_internal (s=0x7ffff7fc06a0 <_IO_2_1_stdout_>,
    format=0x555555556004 "%s\n", ap=ap@entry=0x7fffffffe250,
    mode_flags=mode_flags@entry=0) at vfprintf-internal.c:1688
#2  0x00007ffff7e43d9b in __printf (format=<optimized out>) at printf.c:33
#3  0x000055555555517c in read () at foo_bar.c:6
#4  0x0000555555555198 in main (argc=1, argv=0x7fffffffe458) at foo_bar.c:11
(gdb)
```

Look for the highest line that refers to your source file and that shows you which line caused the SEGFAULT to be issued. In this case, line 6. In your program, investigate that line and those before it that relate to it. This will help you identify and fix the issue.

> `gdb` is able to see exactly which line caused the SEGFAULT only if you comile your programs using the `-g` option (as in `$ gcc -g foo_bar.c -i  foo_bar`)

To quit gdb:

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

    Inferior 1 [process 103285] will be killed.

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

`foo_bar.c` is included in this repository.

Epilogue

In this lab, you have been granted access to a linux virtual machine so that you have access to a linux environment to explore and learn in.

You've been introduced to the shell, and started learning basic commands to interact with it.

You've also started learning C Programming. The programs in this lab have given you a foundation and scaffolding for our next lab.

In the next lab, you will implement your own tiny shell. This will require some C programming (of course!) and also an understanding of the process model. You will use `fork` and `exec` to spin up chil processes to complete the jobs instructed on the command line.