CSE 3100 Systems Programming
Lab #1

Welcome to Lab 1! At the start of each lab, you will receive a document like this detailing the tasks you are to complete. Read it carefully. Try your best to finish labs during the lab session. However, labs are not due right after each lab period. **Lab assignments are due in two days, which means that Lab 1 is due September 5 at midnight**

This lab is intended to be interactive. Feel free to discuss the assignment with other students. However, please recall that all of your code should be your own! If you have trouble, feel free to raise your hand and ask your TA for guidance!

After lab 1, you should be able to access your account on Mimir, use basic commands (e.g. to copy/remove/rename files) in a shell, edit text files like C source files, compile and run simple C code, and submit your work.

1 Mimir

You will use Mimir to do all the work for this course. So you need to learn the environment very well. We will learn how to start Mimir IDE, use shell in a terminal, and work on C code, and submit your work. You may like to Watch the video and read the help file listed below. Ask questions if you have trouble.

- CSE 3100: Lab0 Demo on YouTube.
- How To Use The Terminal In The Mimir IDE To Run Code on Mimir. Focus on terminal, shell, and instructions on C.

In Mimir, select the project lab1. In the "Sandbox" tab, click "Open in Mimir IDE". You should see an editor and a virtual terminal. You can then try commands in shell.

2 Shell

2.1 Manual Pages

One of the most useful commands is `man`, which allows you to access manual pages for different programs and functions. We generally refer to these manual pages as *manpages*, for short. You can open up the manpage for `man` with the following command:

```
$ man man
```

The man pages provide documentation for most programs available on your system, as well as many functions found in the C libraries. This can lead to conflicts: sometimes a library function and program can have the same name. This is where sections come in handy. You'll notice that the output of `man man` has a table of sections. In the Mimir VM, it looks like this:

```
1   Executable programs or shell commands
2   System calls (functions provided by the kernel)
3   Library calls (functions within program libraries)
4   Special files (usually found in /dev)
5   File formats and conventions, e.g. /etc/passwd
6   Games
7   Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8   System administration commands (usually only for root)
9   Kernel routines [Non standard]
```

Now, run the following command:

```
$ man -f printf
```

You should see the following output:

```
printf (3)           - formatted output conversion
printf (1)           - format and print data
```

The `(3)` and `(1)` indicate the section number; they correspond to Library functions, and shell commands, respectively. If we run:

```
$ man -S3 printf
```

This shows the documentation for the `printf` family of functions, which includes `printf`, `fprintf`, and `snprintf` functions. I recommend you spend **a minute** to look over the DESCRIPTION part of this manpage. Now, if we run the following:

```
$ man -S1 printf
```

This shows the documentation for the `printf` *command*, which, similar to the `printf` *function*, takes a set of arguments and a format string, constructs a string from them, and prints it to the terminal.

2.2 Navigating the File System

I'm going to step you through using some basic commands to navigate the file system. The first command is `pwd`, which stands for "print working directory":

```
$ pwd
/home/jordan/cse3100_fall_2020/lab1
```

The directory structure on Linux and Unix systems is essentially a tree, with the root being the `/` directory. We can navigate to the root as follows:

```
$ cd /
```

This is the `cd` command, which stands for "change directory". Then, we can use the "ls" command to list the files in the `/` directory:

```
$ ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  src  srv
```

For now, we'll be concerned with the `home` directory. We can go into that directory by typing the following, **but do not press enter after typing it**:

```
$ cd h
```

Press the TAB key. It should complete to `cd home`:

```
$ cd home
```

This is called tab-completion. Now, if we run `ls`, we should see a single directory:

```
$ ls
jordan
```

My first name is Jordan, so Mimir gave me the home directory `/home/jordan`. I will refer to your home directory as `/home/NAME` from here on out; when I say NAME, you should substitute your directory name. Also, `~` refers to your home directory. Now, we can use `cd` to navigate into that directory, and list the files in it:

```
$ cd NAME
$ ls
```

The exact output of `ls` will depend on the contents of your home directory. If you've never used Mimir before, it should be relatively empty. However, there should at least be a `cse3100_fall_2020` directory in the output; within that, there should be a `lab1` directory. To go into `lab1`, we can do the following:

```
$ cd cse3100_fall_2020/lab1
```

There are two options `ls` that are quite useful. The first is the "-l" option, which shows additional information about each file/directory in the output:

```
$ ls -l
-rw------- 1 root root    328 Jul 29 17:24 floyd.c
```

This shows file permissions, file ownership, and last modification time. The other option is "-a". In Linux, files/directories that begin with a period are normally hidden in the output of `ls`. However, the "-a" option shows those files:

```
$ ls -a
.  ..  floyd.c
```

The "." and ".." entries are both directories. This might seem strange, but the "." refers to the current directory, and ".." refers to the parent directory. The "." will become important later in this document, but we can use ".." to go up in the directory structure:

```
$ pwd
/home/jordan/cse3100_fall_2020/lab1
$ cd ..
$ pwd
/home/jordan/cse3100_fall_2020
$ ls
lab1
```

2.3 Viewing Files

In this section, I will describe three important commands: cat, less and grep. First, go back into your `lab1` directory using `cd`. Weu'll use the `cat` command to view the contents of `floyd.c`

```
$ cat floyd.c
#include <stdio.h>

int main(void)
{
  int n = 0;
  printf("Enter number of rows: ");
  /* TODO: Read in the number of rows from the user, and place it in n */
  for(int i = 1; i <= n; i++){
    /*
```

```
      TODO: Print row i of the Floyd triangle.
     */

    //This printf prints a newline.
    printf("\n");
  }
  return 0;
}
```

This `cat` command is short for "concatenate". It takes in a list of files, concatenates them together, and prints the result to the terminal. Since we only specified a single file, it just prints the `floyd.c` file to the terminal. Now, imagine you had a much larger file, and you wanted to scroll through it. You can use the `less` command for that:

```
$ less floyd.c
```

This will display the contents of the `floyd.c` file on screen. If the file was long enough, you would be able to scroll through it. Now, you can press `q` to exit. What if you want to search a file? You can use the `grep` utility for that. Suppose I wanted only the lines that contain a ";" in them. Then, I could do the following:

```
$ grep ";" floyd.c
  int n = 0;
  printf("Enter number of rows: ");
  for(int i = 1; i <= n; i++){
    printf("\n");
  return 0;
```

2.4 Manipulating Files and Directories

We'll create a file called "hi.txt" that contains the text "hello" using the following command:

```
$ echo "hello" > hi.txt
```

You can view the contents of `hi.txt` using `cat` or `less`. Now, if you want to delete the file, you can use the `rm` command:

```
$ rm hi.txt
```

Similarly, you can create and remove directories using the `mkdir` and `rmdir` commands:

```
$ mkdir test
$ ls
floyd.c  test
$ rmdir test
$ ls
floyd.c
```

2.5 Creating Archives

In this section, you will learn how to create and extract archive files on Linux. These are used in the same way as a ZIP file. First, we'll create a directory, and add two files to it:

```
$ mkdir archiveTest
$ echo "hello" > archiveTest/thing.txt
$ echo "bye" > archiveTest/bye.txt
```

Now, we'll create an archive named `archive.tar.gz`, which contains `archiveTest`

```
$ tar -zcvf archive.tar.gz archiveTest
archiveTest/
archiveTest/bye.txt
archiveTest/thing.txt
$ ls
archive.tar.gz  archiveTest  floyd.c
```

The `tar` command creates a *tape archive* file out of the specified directory, and then compresses it using the `gzip` program. So, by convention, these files have the suffix `.tar.gz`. Here are what each of the arguments in **zcvf** mean:

- **z** - Compress the tar archive using `gzip`
- **c** - Create a new archive
- **v** - Verbose, so it prints out each file that's added to the archive
- **f** - Names the output `archive.tar.gz`, per the following argument

Now, we'll create another directory, move the archive into it using the `mv` command, and extract the archive:

```
$ mkdir archiveExtract
$ mv archive.tar.gz archiveExtract
$ cd archiveExtract
$ ls
archive.tar.gz
$ tar -zxvf archive.tar.gz
archiveTest/
archiveTest/bye.txt
archiveTest/thing.txt
$ ls
archive.tar.gz  archiveTest
```

The one argument we changed was **c** to **x**, which causes `tar` to *extract*, rather than *create*.

2.6 File Integrity

Now, we'll use a program called `md5sum` to compute the MD5 hash of a file.

```
$ md5sum archive.tar.gz
b9c5220332846d766aa42b7758291046  archive.tar.gz
```

This is useful because it allows you to quickly check if two files are identical; when downloading or transferring large files, it's common for the author of the file to provide the MD5 Hash, so that you can make sure that the file wasn't corrupted during transfer. While it's technically possible for non-identical files to have the same MD5 Hash, it's extremely unlikely that corruption during transfer would be such that the corrupted file has the same hash as the original file.

3 Text editors and your first C program!

To work on a project, you need to go to the corresponding directory in the virtual terminal. Normally Mimir goes to the correct directory when you start a virtual terminal. Under the project directory, you can modify existing files and/or create new files (.c files, .h files, readme's, make files, etc.).

You can use the web editor, or an editor in the virtual terminal, to edit text files, which include your C source code. If you decide to learn an editor in the terminal, there are three primary options: `vim`, `emacs`, and `nano`. It is highly recommended to learn one of them, in addition to the web editor.

Use your favorite editor to open `lab1.c`. Here are different ways to do it in a terminal. Remember `open` is for web editor and only works in Mimir. Other three editors are available in many systems.

```
open lab1.c
vi lab1.c
emacs lab1.c
nano lab1.c
```

Add the following C code in `lab1.c`. Remember to save the file after editing.

```
#include <stdio.h>

int main(void)
{
     printf("Hello, World!\n");
     return 0;
}
```

This is your first C program! Before you can run it, you must compile it. You can do so by typing the following command in shell. If you have closed the terminal, you can start a new one (by choosing menu File then New Terminal, or using the keyboard shortcut).

```
cc lab1.c
```

This will compile your C code in `lab1.c` and create an executable called `a.out`. If the process fails, read the error messages, fix the error in your C file (using the text editor), and try again. If there is no error, you will see `a.out` in your current directory.

You can optionally specify the name of the executable that is generated by the compiler.

```
cc -o lab1 lab1.c
```

To run your program, type

```
./lab1
```

3.1 The PATH

Each command you type is, itself, a program, which resides somewhere in the filesystem. How does your shell know where to look for these programs? It uses the PATH variable. You can examine its value as follows:

```
$ echo $PATH
/src/bin:/usr/games:/usr/bin/games:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

This is a list of colon seperated directories [1]. These directories are searched sequentially until a program with the name of the command is found. Let's examine the `/usr/games` directory:

```
$ ls /usr/games
cowsay  cowthink  fortune  LS  pacman4console  sl  sl-h
```

Let's run the `fortune` program:

```
$ fortune
Stay away from hurricanes for a while.
```

That's brilliant advice. Now, let's try to run `lab1` this way:

```
$ lab1
bash: lab1: command not found
```

None of the directories in the PATH contain a program called `lab1`. This is why you have to type "./lab1" in order to run the `lab1` program. The "." references the current directory, and the "/lab1" references a file called "lab1" within the current directory. If we type `cd` into the shell, it will take us to our home directory, and then we can type `cse3100_fall_2020/lab1/lab1` to run the `lab1` program:

```
$ cd
$ pwd
/home/jordan
$ cse3100_fall_2020/lab1/lab1
Hello, World!
```

4 Floyd's Triangle

Open `floyd.c`. Here, you will add code to create what's known as "Floyd's Triangle". The first 4 rows are as follows:

```
1
2 3
4 5 6
7 8 9 10
```

Your program should read an integer n in, and print out the first n rows of Floyd's Triangle. See video 6 from week 1 for a demonstration of how to read in integer, print integers, and use loops.

5 Submission

If you think your code works, you can submit it by clicking on the "Submit" button. You need to submit `lab1.c` and `floyd.c` in this lab. Check if your code passes the tests. Although the submission process will test your code, you should NOT rely on it to find bugs in your code. You should try your best to test/debug your code before submission.

Please remember to submit your work before the deadline for each assignment and exam.

Enjoy!

[1] The $ indicates that PATH is a variable; if you did `echo PATH`, it would simply echo "PATH"

Exit from an editor in terminal

There are many tutorials on nano, vi/vim, and emacs. Feel free to explore and share with other students. In case you are stuck in an editor, here are the commands you can exit from the program.

nano

nano is very user friendly. You can see the help at the bottom of the screen.

```
nano a.txt          Start nano to edit file a.txt
Ctrl-x              Exit from nano. See the prompt at the bottom of the screen
```

There are many tutorials on the Internet, for example, here.

emacs

Commands in emacs are control characters or prefixed a set of characters.

```
emacs a.txt         Start emacs to edit file a.txt
Ctrl-x Ctrl-s       Save file
Ctrl-x Ctrl-c       Exit from emacs
```

Here and here are examples of cheat sheets and tutorials on emacs.

vi/vim

vi has three modes. Normally, you are in the normal mode when vi just starts. Press `i` to enter the insert mode, where you can insert/type text. Press `Esc` to exit from the insert mode and get back to the normal mode. If you get lost, press `Esc` many times to get back to the normal mode.

```
vi a.txt            Start vi/vim to edit file a.txt
:wq                 Save the file and then exit from vi. Work in the normal mode
:q!                 Exit from vi without saving file. Work in the normal mode
```

Here is an example cheat sheet.