Lab 1: Setup & Useful Tools

1 Linux Review

Most of this section should be review. Please make sure you are comfortable using the Linux command line and all the commands in this section before moving on with lab. If you feel you are already a Linux expert please create a directory to work in and move to the next section.

1.1 Logging In

Linux is a multi-user operating system – it allows multiple users to have accounts and to be logged in at one time. Your ISU Net-ID is the username and password used to log into the OS.

Linux presents a desktop environment that enables users to easily use and configure their computers. Once you feel marginally comfortable with the desktop, open a terminal by right-clicking on the desktop and selecting Open Terminal from the popup menu.

1.2 Command Line

The command line in UNIX is actually just another program, usually called a shell. The shell allows you to run other programs or scripts, and is generally useful for managing your computer once you know which commands to use. Most Linux distributions ship with bash as the default shell.

When you start a terminal, the first thing you will see is the shell prompt (often called the command line). As its name would suggest, it is prompting you to enter a command. It should look something like this:

```
username@host currentdirectory $
```

In this and future labs commands to be entered in the terminal shall be preceded by the `$` and typeset as shown below. Start by making a directory to save your work for this course in called cpre308:

```
$ mkdir cpre308
```

And change into that directory using

```
$ cd cpre308
```

Note: Bash has tab-complete feature which allows you to type the first few characters and press tab to auto fill in the rest. Pressing tab twice will show all possible values what can be entered.

Next create a new file and edit in the Gedit text editor

```
$ gedit newfile
```

A window should open for you to type in. Go back to the command line. Notice anything odd? The prompt didn't reappear. That's because it is waiting for gedit to finish. We would like to continue typing commands at the same time, so let's fix this.

Gedit is currently running in the foreground. To suspend gedit, return to the terminal and type Ctrl+Z. You should have a prompt again; however, because gedit is suspended we can't use it until it is returned to the foreground or we move it to the background. type bg in the terminal. Gedit will now be running in the background while we continue to type commands. If we wanted to return it to the foreground, we could have typed `fg`.

To list all jobs in the background of a given terminal use the `jobs` command.

To start a program in the background type an `&` at the end of the command. For example, to start gedit in the background type

```
$ gedit somefile &
```

In your editor type something and save the file. Next list the contents of the directory using the `ls` command:

```
$ ls
```

To list all contents of a directory include a `-a` flag, and to show file sizes in a human readable format include a `-l -h`

```
$ ls -a -l -h
```

or

```
$ ls -alh
```

You should see two additional entries: `.` and `..`. The single dot is a reference to the current directory and the two dots is a reference to the parent directory. Change to the parent directory by typing

```
$ cd ..
$ ls
```

You should see the cpre308 directory in the listing, since you are now in the parent directory. There are two more important points to know about directories: absolute and relative paths.

An absolute path is a path that starts with `/` or `~`. Paths of the form `/some/path/to/file` will always point to the same file because it is referenced to the root of the file system. Paths of the form `~/local/path/to/file` will point to a file relative to the users home directory. A users home directory is usually `/home/login/` on Linux.[1]

A relative path is a path that starts with any other character. It is relative to the current directory, and will refer to different files depending on the current directory.

You can always find your current absolute path name by using the print working directory comand: `pwd`.

1.3 Some tips

1.3.1 Recommended Editors

This can spark a heated debate among many people, and this list is no where near exaustive, but here is some text editors that you may choose to use throughout this class: `gedit`, `gvim`, `kwrite`, `vim`, `nano`, `emacs`, and `xemacs`.

[1]Calling cd with no arguments will always change directory to the users home directory.

1.3.2 Useful Commands

- `mkdir` – make a directory
- `rmdir` – remove a directory
- `cd` – change current directory
- `ls` – list files and directories
- `mv` – move/rename a file/directory
- `rm` – remove a file/directory
- `cp` – copy a file/directory
- `less` – view a file (use arrow keys to scroll, use `q` to quit).
- `pushd` – save as cd, but save current directory on a stack
- `popd` – change to the directory you were in before the pushd command
- `grep` – search a file, or directory, for a string(useful for finding specific error messages, for example)

To learn more about these or any other command in linux check its manual page. For example, to learn more about ls type

```
$ man ls
```

Type `q` to exit the man page and return to the terminal.

2 Git Subversioning

Much of this and following labs will use the git subversioning system. Git allows multiple people to work on the same code base at once, each with their own entire version of the source tree.

2.1 GitHub

If you do not already have a GitHub account please go to http://www.github.com and create a free account now.

2.1.1 Create SSH Keys

GitHub uses a secure shell (SSH) connection to push and pull the Git repository to and from the cloud. Public/Private keys are used to secure the connection between your local computer and GitHub. We will learn later in this course how public/private keys work, but for now let's just create a new key pair. First check to see if SSH keys already exist by running on the terminal

```
$ ls -al ~/.ssh
```

If you see a file listed called `id_rsa.pub` then skip to the Install SSH Keys section; otherwise, please continue to create new keys. Run the following command in the terminal replacing the email address with your email address:

```
$ ssh-keygen -t rsa -C "your_email@iastate.edu"
```

When asked which file to save the key to press `Enter` to accept the default. When asked for a passphrase you may optionally choose to supply one, or you may press `Enter` to not use a passphrase. Passphrases greatly improve the security of the SSH key pair.

2.1.2 Install SSH Keys

Once you have generated SSH keys you must install the public key to GitHub. Open the public key in a text editor

```
$ gedit ~/.ssh/id_rsa.pub
```

And copy all of the contents of the file into the clipboard. Next in a web browser log into GitHub and click the **Settings** button in the upper right corner. Next click the **SSH keys** option. Click **Add SSH key** and enter `UDrive Key` as the title of the key. Next paste the contents of the key file into **Key** and click **Add key**.

Next we will test the key. Enter the following command in the terminal:

```
$ ssh -T git@github.com
```

If succesful you should be asked if you want to continue. Type `yes` and press `Enter`. If you setup a passphase you will now be asked to enter the passphase. If everything works you should get a welcome message from GitHub. If you get an error please ask your TA for assistance.

Note: If you would like to do development work on your own computer you will need to follow the steps again giving the key a different title. The key generated here will work from any on campus linux computer.

2.2 Access Class GitHub

This information will be given during your lab section. To gain access to the class GitHub, navigate to the `Labs` folder, under `Course Content` on the class Blackboard page, and download the script `setup_labs.sh` that is posted there. Save the script in the `cpre308` folder you created. Once the download is complete, on the terminal, navigate to the `cpre308` folder and run the follwing command:

```
$ chmod u+x setup_labs.sh
```

This command changes the permissions of the file `setup_labs.sh` to allow the user who owns the script to be able to execute it. Now run script by typing into the terminal:

```
$ ./setup_labs.sh
```

This will execute the script. If you are interested in what the script does, and how it does it, feel free to open up the script using either `less` or the editor of your choice. If you get any errors from running the script ask your TA for help.

2.3 Using The Class GitHub

For all future labs you will get the code and other material for the lab by issuing the command in your labs directory:

```
$ git pull labn master
```

For example, to get the code and other material for lab 1, run:

```
$ git pull lab1 master
```

To commit your labs in the lab directory run the command

```
$ git status
```

To see which files have changed or been created. Add the lab you want to submit by typing

```
$ git add lab1/
```

Next run `git status` again and make sure all the files that should be submitted show up. Next commit your changes to your local repository with a meaningful commit message using

```
$ git commit
```

If and only if you are ready to submit the lab for gradding run the following command to create a tag

```
$ git tag lab1
```

Now that you have your local repository up-to-date you need to push your changes to the GitHub. To do this run the command

```
$ git push --tags
```

And finally check the GitHub website to make sure your changes show up correctly. Note, you can (and should) commit your changes often and push as often as you like. We will only grade your **tagged** push, and by commiting often you have the ability to roll back changes or recover files if something happens.

3 GNU Debugging Tool

The GNU Debuggin 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; if you already feel comfortable with using GDB you may skip this section. Much of this section is based on the work of Ryan Schmidt of the University of Toronto.

3.1 Stepping Through Code

Let's 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 `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. Notice however that it exits with a usage error 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. 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) clear 46
(gdb) continue
```

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

3.2 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 part-b.c

```
$ gcc -o part-b part-b.c
$ ./part-b
```

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 part-b part-b.c
$ gdb part-b
```

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 recieved a signal SIGSEGV. To see what functions were last called run a backtrace:

```
(gdb) backtrace
#0  __GI__IO_getline_info (fp=0x7ffff7dd4640 <_IO_2_1_stdin_>, buf=0x0, n=1022, delim=10, extract_delim
#1  0x00007ffff7a82d46 in _IO_fgets (buf=0x0, n=0, fp=0x7ffff7dd4640 <_IO_2_1_stdin_>) at iofgets.c:56
#2  0x0000000000400634 in main (argc=1, argv=0x7fffffffe0e8) at part-b.c:22
```

This shows that the last function we wrote that was called was part-b.c:21 which is line 21 of part-b.c. Since this is all we are interested in let's switch the stack frame to frame 2 and see where the program crashed:

```
(gdb) frame 2
#2  0x0000000000400634 in main (argc=1, argv=0x7fffffffe0e8) at part-b.c:22
22      fgets(buffer, 1024, stdin); // get upto 1024 characters from STDIN
```

Since we assume that fgets works let's check the value of our arugment. `stdin` is a global variable created by `stdio` library so we assume it is ok. 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`. Next set a breakpoint at line 20:

```
(gdb) break 20
```

Now run the program again:

```
(gdb) run
Breakpoint 1, main (argc=1, argv=0x7fffffffe0e8) at part-b.c:20
20      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 assinged. Let's step over the `malloc` line and print buffer again:

```
(gdb) next
22      fgets(buffer, 1024, stdin); // get upto 1024 characters from STDIN
(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 the 1024 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.

4 Valgrind Memory Tool

Another useful tool is Valgrind. Valgrind tracks dynamic memory allocations and can detect memory leaks or other memory problems such as buffer overflows. To compile a program for Valgrind again use the `-g` flag.

4.1 Simple Malloc Test

Compile the source malloc_test.c

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

Next run Valgrind on the program using the command

```
$ valgrind --leak-check=yes ./malloc_test
```

You will see that we get a lot of information. Let's go over each part.

4.1.1 Invalid Write

The first message we see is invalid write of size 1. This is because we tried to write 1 byte past the end of the buffer.

4.1.2 Heap Summary

The heap summary shows the number of allocates and frees used over the life of the program, and the number of bytes allocated at exit.

This section also shows information about the blocks which were use at exit such as size and line they were allocated on.

4.1.3 Leak Summary

The last section is the leak summary which summarizes how many bytes and blocks were lost when the program exited.

Valgrind is much more powerful than shown here, but takes a little bit to get the hang of. Check out the online documentation at http://valgrind.org/docs/manual/ to learn more.

5 Test Your Knowledge

Compile and run the program `rand_string.c`. This program takes a string as an input and outputs ten random characters from the string. Please use the techniques from this lab to figure out why the program is not working correctly.

6 What To Submit

Write a report summerizing what you have learned in this lab in either **markdown** or **PDF** format. Please include the report in the `lab1` folder of your GitHub submission and add it by running `git add lab1/report.md` or `git add lab1/report.pdf`. Also please make sure to add your modified `rand_string.c` file by running `git add lab1/rand_string.c`. Run `git status` to ensure the files have been added and commit the changes by running `git commit -m "Commit Message"`. Next you need to tag this state of the code as your submission by running `git tag lab1`. Finally, submit your code to GitHub by running `git push --tags`. Check the GitHub website to make sure all files have been submitted. This lab is due at the begining of next weeks lab, and will be automatically pulled at that time.

7 License