CprE 308 Laboratory 1: Introduction to Linux

Department of Electrical and Computer Engineering
Iowa State University

Spring 2018

"Experience is not what happens to you; it is what you do with what happens to you."
– Aldous Huxley

Submission You will turn in your Makefile you create as well as the netid.txt file you create at the end of this lab via blackboard.

1 Purpose

In this lab, you will become familiar with the basic use of a Linux system and its development tools. You will be asked to perform a few exercises and read various manuals available in the lab.

2 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 in to 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 window by right-clicking on the desktop and selecting Open Terminal from the popup menu.

3 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. This shouldn't be confused with the terminal emulator, which simply presents a GUI for command-line programs and happens to run bash by default.

When you start a terminal, the first thing you will see is the 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]$
```

Type the commands after the prompt (`$`).
Make a new directory to contain your work for this course.

```
$ mkdir 308
```

Go to the directory.

```
$ cd 308
```

Edit a new file.

```
$ gedit practice
```

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's waiting for gedit to finish. We'd like to continue typing commands at the same time, so let's fix this.

`gedit` is currently in the foreground. On the command line, type `Ctrl-Z`. You've just suspended gedit and placed it in the background. You should have a prompt again. However, because gedit is suspended, we can't use it until it returns to the foreground or we start running it in the background. Type `bg`. 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 a list of jobs in the background, use the `jobs` command. *Note*: you can avoid this whole mess in the future by starting gedit in the background. Just type an & after the command:

```
$ gedit somefile &
```

Now that we have an editor running, type something into the window and save it. Back on the command line, type the following:

```
$ ls
```

The file `practice` should be listed. Now try:

```
$ ls -a
```

You should see two additional entries – "`.`" and "`..`". `.` is a reference to the current directory, and `..` is a reference to the parent directory. Type

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

You should see the 308 directory in the listing, since you're now in the parent directory. You need to know two more things about directories: absolute and relative paths.

An *absolute path* is a path that starts with "`/`". Because it starts with the root of the file system, it will always refer to the same file no matter what your current directory is.

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.

Type `pwd` to see the current absolute path. You should see something like `/home/username`, which indicates you're in the `username` directory under the `home` directory. This is where to find your account in Linux. It's the default current directory when you login. You can always go back to it by typing

```
$ cd
```

Now try using an absolute path to switch directories. Type (replacing username with your own username):

```
$ cd /home/username/308
```

You should now be in your 308 directory. To confirm this, type

```
$ pwd
```

To return to your home directory, type:

```
$ cd
```

Now switch to the directory again using a relative path:

```
$ cd 308
```

Again, use pwd to verify that you're in the subdirectory. So far there's no difference, right?

Suppose that you were in another user's home directory. In that case, `cd 308` would take you to their 308 directory. However, if you used the absolute path with your username in the path, your would go to your 308 directory. This difference isn't all that interesting now, but be aware of it for future use. There will be occasions when you will want to specify the file or directory you want to work with unambiguously by using an absolute path.

You can use relative and absolute paths to describe files and directories in many commands. For example, if you're in your home directory, the following two commands should be equivalent:

```
$ ls /
$ ls ../..
```

Why does this work? Since the first command uses an absolute path (it starts with /), then it will always return the same result no matter where you are. If you're in the directory /home/username, then ../.. resolves to the parent of the parent of the current directory, or /. If you were in /home/username/308 and typed ls ../.., you would see the directory listing for /home instead.

Here's a list of commands you might find helpful. Try a few out, but be careful with the ones that delete files! You don't have permission to delete anything other that your own files, but there isn't a recycling bin to recover your work from!

- mkdir – make a directory
- rmdir – remove a directory
- cd – change current directory
- ls – lists files and directories
- mv – move/rename a file/directory
- rm – remove a file
- cp – copy a file
- less – view a file (use arrow keys or space to scroll). Use q to quit.
- man – display the manual for a subject or command. Similar to less.

A few more general tips:

- Linux has two clipboards to allow more convenient copy/paste while keeping the features of standard cut/copy/paste. When you select text in the terminal or any application, it is copied to the "selection" clipboard; use the middle mouse button (wheel button) to paste into an application. The second clipboard is usually accessed through Ctrl-C and similar shortcuts; however, these do not work in the terminal as they are sent to the program in the terminal.
- You can use the tab key to finish commands and paths. Hit tab twice to see a list of potential matches.
- You can use the arrow keys to look through recently typed commands. In bash, `Ctrl-R` will search your command history for the text you type.

- Your account uses the same space as the U: drive under Windows, so you can access it from the department labs. You can use ssh, scp, or sftp to access the files from home if you run a Unix system. From windows, SSH clients such as PuTTY or SFTP clients such as FileZilla can be used in addition to Windows' shares. The systems `linux-1.ece.iastate.edu` through `linux-6.ece.iastate.edu` are accessible on-campus for running applications.

3.1 Editors

If you aren't already proficient with a Unix editor, you should pick one to become proficient with. The editor gedit is a useful graphical editor for those new to the command line. There are several other editors available, many with more features. More seasoned programmers tend to use emacs or vi, which are console-based editors. Emacs, or its GUI version `xemacs`, is another editor that some have found to be simpler and more flexible than vi.

4 Development Tools

4.1 Compiler

A compiler converts source code into object code or executable code. The GNU Compiler Collection (gcc) is included with Linux. Several versions of the compiler (C, C++, Objective C, Fortran, Java and CHILL) are integrated; this is why we use the name "GNU Compiler Collection." GCC can also refer to the "GNU C Compiler," which is the `gcc` program on the command line. You will be using a lot of C in this course; if you are rusty it might help to to review your CprE 211 notes.

4.2 Compiling and Linking Multiple C Files

Create a three files in your preferred editor:
message.h contains:

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

void print_message();
```

message.c contains:

```
#include "message.h"

static const char* message[] = {
    "Hello Iowa!",
    "Goodbye Iowa!",
    "The penguins are coming!",
    "Caffeine is my friend!"
};

void print_message() {
    int index;
    srand(time(NULL));
    index = rand()/(RAND_MAX/4);
    printf("%s\n", message[index]);
```

```
}
```

lab1.c contains:

```
#include "message.h"
int main(int argc, char** argv) {
    char buf[1024];
    print_message();
    printf("You ran me as:%s\n", argv[0]);
    getcwd(buf, 1024);
    printf("Current working directory:%s\n", buf);
    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 lab1.c
$ gcc -o lab1 lab1.o message.o
```

To run the program, type:

```
$ ./lab1
```

Note: why "`./`"? Recall the discussion of directory navigation in the previous section? `.` 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. Don't forget this – you'll need it to run programs for the rest of the semester.

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 one time with the command

```
$ gcc -o lab1 lab1.c message.c
```

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.

4.3 Make

The make utility 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.

12.5 pts After you figure out how Makefile works, write a Makefile that will compile the *lab1.c* into *lab1* then test it. Turn in this makefile through Blackboard.

5 Additional Information

5.1 Manual pages

Manual pages are your best friends in Unix; they provide information you can digest on almost all commands, applications, system calls, programming languages, etc. For example, to view the manual page for the ping command, type `man ping` in your terminal. You will see something like this:

```
PING(8)              System Manager's Manual: iputils                PING(8)

NAME
ping, ping6 - send ICMP ECHO_REQUEST to network hosts

SYNOPSIS
ping [ -LRUbdfnqrvVaAB] [ -c count] [ -i interval] [ -l preload] [
-p pattern] [ -s packetsize] [ -t ttl] [ -w deadline] [ -F flowlabel]
[ -I interface] [ -M hint] [ -Q tos] [ -S sndbuf] [ -T
timestamp option] [ -W timeout] [ hop ...] destination

DESCRIPTION
ping uses the ICMP protocol's mandatory ECHO_REQUEST datagram to elicit an ICMP
ECHO_RESPONSE from a host or gateway. ECHO_REQUEST datagrams ("pings") have an
IP and ICMP header, followed by a struct timeval and then an arbitrary number of
"pad" bytes used to fill out the packet.
```

There may also be more help in info pages. The compiler's manual can be reached using `info gcc`. When they exist, the info pages for a given command may contain more information about what the program does rather than simply a description of its syntax.

Manpages are divided into several sections; notice `PING(8)` in the above example means ping is in section 8. A description of the different sections is in `man man`. Some important sections are: system calls are in section 2, library functions are in section 3, system programs are in section 1 (or 8 for administrative tools). For example, `man read` will not show the documentation for the system call `read()`; you must run `man 2 read` to get the correct entry.

In many of the future labs, you will be expected to look up important information in man pages. Take some time now to look up a few things. Try looking up the man page for "ls" and "signal". When you're finished with an info or man page, just type q to quit.

5.2 Outside Sources

If you are new to Unix or need to review, there's plenty of Linux information available online at the Linux Documentation Project: http://www.tldp.org

6 Adding your program to the path

Bash makes use of lots of environment variables to make user's job easier. Environment variables are like global varibales for a given environment. An environment is where you run your program. For example, if you run your program from a Bash instance then your environment is that Bash shell. Therefore, everything that ran in that instance of the shell can access to the same environment variables and they can modify it as well. This means that every instnce of Bash may potentially have different values in their environment variables. To see what environment variables you have and what is in them in your current Bash, you can use `printenv` command.

There are many different uses for environment variables. Each environment variable usually has its own usage defined. You can Google the specific variables to get further information about them. The one we are going to use in this lab is named `PATH`. The `PATH` variable stores a colon sepearted list of directories. You can see what is currently inside your `PATH` variable by using `echo $PATH` command in your bash. `PATH` is generally used by Bash itself.

Notice that when you enter `ls` in your bash it works but when you enter `myprogram` (where "myprogram" is your compiled code) it complains. To make it work you need to enter some type of directory information (i.e, `./myprogram`) What is different between "ls" and you "myprogram"? Nothing as far as bash is concerned, they are both compiled programs. Then why can't we run our code like without a directory information? That is what `PATH` is used for. Bash needs to know where your executable is to be able to run it. Therefore, if user does not enter any directory information then bash starts to check every directory inside `PATH` variable and runs the first found program. The "ls" program generally lies in "/bin/ls" which should be in your `PATH` variable. Therefore, bash does not require any additional directory information from user. Whereas your program lies in some other drectory that is not in `PATH` which is why Bash complains if no information about where is given.

In short, if we want to run our code in like "ls" then we need to put its directory inside `PATH` variable. This can be done by `export PATH=$PATH:/new/directory/` command. This will append "/new/directory/" at the end of the `PATH`. As for the "export" that is so that Bash will treat this as an environment variable rather than a normal variable.

There is one other thing you need to know which is how does `PATH` get filled in the first place. After adding something to `PATH` try closing your Bash and opening a new instance, you'll see that your addition is gone. The reason is that when BAsh start it starts with no environment variables but then runs bunch of Bash commands defined in file in the background and then it prints the first prompt for the user. As you can guess these commands are theones that fills/create your environment variables. These commands are stored in several places and you can only access the ones stored inside your home directory "~/". The one file that you can modify is `~/.profile`. This is the file to put extra commands the user needs for himself or herself only. After Bash executes the system wide defined commands it looks into the user's (who is logging in) home dirctory for the `.profile` file, if it exists then it executes every command in it before printing the prompt for the user. IN short if you make the addition to `PATH` here then the addition would be permenant.

- Create in your home directory, a new directory called `cpre308/bin`. Move the executable lab1 file to this new directory.
- Add your `cpre308/bin` folder to your `PATH` permanently by modifying your `.profile` script.
- Logout from the system, and then log back in then Immediately execute `lab1 > netid.txt` where netid should be replaced with your username.
- The textttnetid.txt file will have the output of your program.

12.5 pts You are required to turn in your `netid.txt` through Blackboard.