CSC 412 – Operating Systems

Programming Assignment 01, Fall 2018

Thursday, September 6th, 2018

Due date: Tuesday, September 18th, 11:55pm.

1 What this Assignment is About

1.1 Objectives

The objectives of this assignment are for you to

- Setup the environment that you are going to use for all assignments this semester;
- Learn some simple Unix commands (if you were not already familiar with Unix/Linux);
- Get started with C programming and bash scripting;
- Make sure that you follow properly assignment specifications.

This is an individual assignment.

1.2 Handout

There is no handout for this assignment besides this document.

1.3 On Unix vs. Linux

Throughout this assignment, I say "On Unix,. . . " Please be aware that strictly and legally speaking, Linux is *not* Unix™, but merely "Unix-like," because it has not been certified by the Open Group (unlike, say, Apple's macOS, formerly known as MacOS X). This being said, over 99% of terminal commands are found on both system, with only minor variations in the list of options.

2 Part I: Setup the Environment

As mentioned in the syllabus, all projects will be graded on the—as of today—latest version of Linux Ubuntu running in a VirtualBox virtual machine. Of course, we can't control what environment you use for development, but please be aware that there are subtle differences in the execution of system calls, and even regular C code, on Linux, Windows, and macOS. Just because your code builds and runs fine in Xcode on macOS is no guarantee that it will do so on Ubuntu Linux. It is *your* responsibility to test your code on Ubuntu before you submit it. The graders will not grant any partial credit for "but it runs in <whatever OS>" excuses.

2.1 Virtual Box

Download and install VirtualBox at `https://www.virtualbox.org`.

2.2 Linux Ubuntu

Go to the Ubuntu web site `https://www.ubuntu.com` and download the latest stable version of Ubuntu Desktop. At the time of this writing, this would be Ubuntu 18.04 LTS.

2.3 Create a virtual machine and install Ubuntu on it

When you create your virtual machine in VirtualBox, make sure to give it settings in memory and disk space that are at least as large as the minimum requirements to install Ubuntu. Follow the directions[1].

> **Step 1.** *Take a screenshot of the settings of the Ubuntu virtual machine that you just created. Name the screenshot file* `step1.pdf` *(of course, convert the file to pdf if this is not the native format for screenshots on your system).*

3 Part II: Simple Linux Walkthrough

This part of the assignment will consist in a leisurely walk through some common and useful Linux commands, documented by screenshots that you will take as you complete each step.

3.1 Printing the documentation

You can printout the documentation of any Unix command by typing `man` (for *manual*) followed by the name of the command. For example, `man cd` will print out documentation about the `cd` (for *change directory*) command, which lets you navigate the folder hierarchy on your system.

You can also search for command names by typing `man -k` followed by a keyword for the search. Obviously, if you put a completely generic keyword such as `file` you will get a ridiculously long list of commands.

3.2 Create a working directory

If you have moved away from your home folder, you can return there by executing the `cd` command with no parameter. Create a new folder using the `mkdir` command. Move to that folder and printout the path this (current) working directory by using the `pwd` command.

[1] No, I won't give hand-holding directions for every single subtask of this assignment. You are computer scientists and you have to be able to follow use online documentation to complete some tasks assigned to you. If you don't know already how to do that, this assignment is a good a opportunity to start learning how to do it.

> **Step 2.** *Take a screenshot of the terminal after you executed the* `pwd` *command. Name the screenshot file* `step2.pdf` *(of course, convert the file to pdf if this is not the native format for screenshots on your system).*

Print out again the path to the current working directory, but instead of getting the output in the console, `redirect` it to file using the `>` operator: `pwd > path.txt`. You can view the content of the file either by typing `pr path.txt` or `more path.txt` (look at the man pages to see the differences between these two commands.

> **Step 3.** *Take a screenshot of the terminal showing the content of the* `path.txt` *file after you executed the commands listed in this subsection. Name the screenshot file* `step3.pdf` *(of course, convert the file to pdf if this is not the native format for screenshots on your system).*

3.3 Create and delete a text file

You can create a new (empty) text file with the terminal command `touch`. You can then verify in the GUI explorer that the file has indeed been created, and confirm this by listing all the files in the current working directory. If you haven't done so already, you should probably spend some quality time getting acquainted with the various options of the `ls` command. I personally find the variant `ls -la` particularly useful. We will see in class during the semester what all the information in the list output means, and how to exploit and modify it. Now that you see that the file has been created, delete it using the `rm` command.

> **Step 4.** *Take a screenshot of the terminal and GUI explorer that you have completed this step in the assignment. Name the screenshot file* `step4.pdf` *(of course, convert the file to pdf if this is not the native format for screenshots on your system).*

3.4 Edit a file in a text editor

There is an old and very silly dispute among Unix users as to which text editor, `vi`/`vim` or `emacs`, is *the* one and true text Unix editor. You can pick your camp in this tired old battle, or you can simply eschew it by opting for one of the newer text editors (Robert, our TA, advocates `nano`, which I have nothing polite to say about, being a `vi` person myself, like anybody with proper taste[2]).

Use the text editor of your choice to open a next text file, type in the customary line `"Hello World!"` and save the file.

[2] I never claimed that old, silly disputes were beneath me.

Step 5. *Take a screenshot of open editor window. Name the screenshot file* `step5.pdf` *(of course, convert the file to pdf if this is not the native format for screenshots on your system).*

3.5 Command history

Unix keeps track of the commands that you have executed recently. You can display your recent history by typing the command `history`. You can use the up and down arrow keys to bring back to the current terminal line prompt a command that you have already execute. This is particularly convenient when you have to execute multiple times long commands with multiple parameters (coming in future assignments). You can also execute a specific command by typing `!` followed directly (no space) by the number of that command in your history. Note that if you have multiple terminal windows open, they typically will report different command histories because they correspond to different executions of a `shell` (we will explain this term next week in class).

Step 6. *Take a screenshot showing that you have experimented with the history features of Unix. Name the screenshot file* `step6.pdf` *(of course, convert the file to pdf if this is not the native format for screenshots on your system).*

3.6 Final step (wrap-up)

Save your command history to a file named `history.txt`, edit this file to add a first line containing your name. Save the `history.txt` file.

4 Part III: A Little C program

4.1 What the program should compute

Your program will take either one or two strictly positive integer arguments. If the program was launched with a single argument, then you should compute and output the list of all prime divisors of the argument. For example, if your executable is named `prog` and is launched from the console with the command

```
./prog 36
```

Then your program should produce the following output:

```
The list of divisors of 36 is: 1, 2, 3, 4, 6, 9, 12, 36.
```

If the program was launched with two arguments, then your program should compute and output the gcd of these two numbers.For example, if your executable is named `prog` and is launched from the console with the command

```
./prog 36 60
```

Then your program should produce the following output:

```
The gcd of 36 and 60 is 12.
```

4.2 Data validation and error reports

You should check the argument(s) of your program and report an eventual error using one of the following messages (again assuming that you built an executable named `prog`):

- `prog launched with no argument.`
  `Proper usage:  prog m [n]`
- `prog launched with too many arguments.`
  `Proper usage:  prog m [n]`
- `prog's argument is not a strictly positive integer.`
- `prog's first argument is not a strictly positive integer.`
- `prog's second argument is not a strictly positive integer.`

Needless to say, the name of the executable should not be hard-coded in your C program.

4.3 Regarding output specifications

In this and future assignments, when I give specifications for the format of the output, you are expected to follow these specifications and will be penalized if you don't.

5 Part IV: a first simple `bash` script

5.1 What it is about

Here again, nothing fancy: We want you to write a `bash` script that launches an executable, multiple times.

5.2 The script

Your script should take as arguments a desired name for the executable of the divisor program, as well as a list of strictly positive integers. Then your script should do the following:

- Launch the divisor program for each of the integer arguments, to get the list of divisors of that number.
- Launch again the divisor program for each pair of arguments (ignoring order, so if you called `./prog 20 12`, don't call `./prog 12 20`, to compute the gcd of these two numbers.

5.3 Output of the script

When the numerical arguments to your script are all strictly positive integers, then the entire output is provided by the divisor program. For example, if your script is launched with the command

```
sh ./myScript.sh 36 10 37
```

then the output should be:

```
 The list of divisors of 36 is: 1, 2, 3, 4, 6, 9, 12, 36.
The list of divisors of 10 is: 1, 2, 5, 10.
The list of divisors of 37 is: 1, 37.
The gcd of 36 and 10 is 2.
The gcd of 36 and 37 is 1.
the gcd of 10 and 37 is 1.
```

On the other hand, if any the script was launched with an invalid argument list, your script should simply terminate and report:

```
Invalid argument list.
Proper usage: myscript m1 [m2 [m3 [...]]]
```

5.4 Extra credit: up to 8 points

Provide a more "granulated" error report: too few arguments, which argument is invalid, missing executable, etc.

5.5 Extra credit: 7 points

If a value is repeated in the list of arguments to your script, then use this value only once to launch the divisor program.

6 What to submit

6.1 The pieces

If you have kept track of the various steps so far, you should have six screenshots documenting the progressive steps through the assignment, an edited text file listing the history of your commands in the terminal, a C source file, and the code of a `bash` scrip.

6.2 Organization

All the screenshots, named as specified earlier, should go into a folder/directory named `Screenshots`, while the C program and the script should go into a folder named `Code`. These two folders and the `history.txt` file should be placed into a folder named `Assignment_01`. Compress this folder into a `zip` archive and upload the archive to your submit folder before the submission deadline.

6.3 Grading

- Screenshot for each of the steps in the walk-through: 2.5% each (15% total)
- History file: 10%
- Code quality: 25%
  - C program: 15%
  - `bash` script: 10%
- Execution: 30%
  - C program: 15%
  - `bash` script: 15%
- Folder organization: 20%

6.3.1 Execution

We are going to test your program and script with different list of arguments, some valid some not. Your grade for this section will reflect to what extent you produce the desired output for all the test cases. So, make sure that you test your program and script before you submit; don't simply replicate the examples of this handout.

6.3.2 Code quality

For this part of the grade we look at things such as:

- Proper indentation;
- Good comments;
- Good choice of identifiers (chances are that `a`, `b`, and `c` are not good identifiers, particularly if the first two are strings and the third one an integer;
- Consistent choices of identifiers. For example, if you have a variable `int beanCount` that does what it claims to do, then it would be a bad idea to have another variable `int num_of_boxes`.
- Good implementation decision: avoid super long functions (at your current skill level, if your function occupies more than one page of your screen, it's probably too long); select proper data types; create your own when appropriate.

Your grade for this section will reflect how much you actually implemented. A superbly commented and indented "Hello World!" program won't bring you many "code quality" points.

6.3.3 No syntax error

Code with syntax errors (compiler errors for the C program) will get a grade of 0 for execution and quality. Comment out the part with syntax errors and explain in the comments what you were trying to do.