Introduction to the CLI

Homework 1

50.005 Computer System Engineering

Due: 04 Feb 08:30 AM

Natalie Agus
INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Overview

As we have learned in class, a shell is an interface to allow users to access OS services. These services are those you encounter daily when using a computer, for example: file management (rename, create, list files, delete, etc), process management (run and terminate), OS configuration, I/O operations like printing, reading from disk, communication via network, resource management (overclock speed, VM size), pro- tection (security settings), etc. It is named as 'shell' because it is the outermost layer around the OS kernel.

OS shells are usually either in a form of command-line interface (CLI, also known as terminal) or graphical user interface (GUI) a.k.a things you see on your desktop, depending on your computer and OS. In this lab, we are going to learn a little bit about the command line, bash scripting, and makefiles. **The total marks for this homework is 15.**

Submission

Please submit your answers in pdf format exactly as follows:

- Download this handout and fill in the blanks on the respective spaces. They're all color coded as blue.
- Export as pdf and **ZIP** it (not rar, or any other compression algorithm)
- **Upload** to `@csesubmitbot` telegram bot using the command `/submithw1`
- **CHECK** your submission by using the command `/checksubmission`

Command-Line Syntax

In order for us to be able to use CLI Shell, we need to be familiar with commands and their calling syntax. In particular, we are concerned with UNIX-type shells in this course. Open your terminal/command line window. The terminal window in front of you **contains a shell**, which enables you to use commands to access OS services.

[1m] Write down the name of the shell that is used in your machine below.

Your answer: Bash.

Basic Commands

Commands Without Options or Arguments

Tryout the following basic commands:

1. Type `date` and press enter. You should see today's date given to you, for example : Tue Jan 7 18:43:16 +08 2021
2. Do the same thing with `cal, pwd, ls, who, clear`.
    - Study what each command does.
    - You can use `man <command>` and press q to quit anytime.
3. **[5m]** Based on your observation, what's the purpose of each of the five commands above?

Your answer:

- **`cal`: Display a calendar layout of the current month with the current date highlighted.**
- **`pwd`: Print the name of the current working directory.**
- **`ls`: List the directory contents of the current working directory.**
- **`who`: Show who is currently logged on the machine.**
- **`clear`: Clear the terminal screen.**

Commands With Options or Arguments

The commands you have typed above are those that do not require options / more arguments (although they do accept these too). Some commands require more input arguments.

For example, the command `cd` changes your current working directory but it requires the input path to whichever directory you intend to switch to:

- Type `cd <path to your Desktop>` and press enter.
- Has your current directory changed?
- *Hint: you can confirm this by executing* ls next *and you will see that the output is the lists of files in your desktop*.

There are a whole lot of other UNIX commands that's just impossible to cover in a single lab. Below is the summary of the popular commands:

1. `open -a appname` allows you to launch a GUI app from terminal, e.g: open -a Google Chrome. If you are using WSL, you probably cannot do this unless you have installed some applications in it.
2. `mkdir <dirname>` when you want to create a folder or a directory and rmdir dirname when you want to delete an *empty* directory.
3. `rm -r <dirname>` can remove a directory that contains files. **Be careful! Deleting things from the command line doesn't allow you to retrieve it back. Unlike deleting from the GUI, it won't be found in the trash**.
4. `touch <newfilename.format>` creates a **new** file with whatever name and format you want it to be.
5. `man <command>` is used to know more about the `command` and how to use it.
6. `cp <from_path> <to_path>` copies a file from a location to another.
7. `locate <filename>` is used to locate a particular filename.
8. `cat <filepath>` is used to display the contents of a file, and `wc <filepath>` prints a count of **newlines**, **words**, and **bytes** for each input file.
9. `sudo <command>` stands for SuperUser Do, which is to execute command with **administrative privileges**
10. `df` shows the available disk space in each partition
11. `sudo apt-get install <packagename>` is used to install packages
12. `chmod +x <filepath>` is used to make a filename **executable**. However firstly, you *need to declare in your script* which **interpreter to use**.
    - You do this in the first line of the file.
    - If it's a shell script, it should be `#!/bin/sh` or `#!/bin/bash`.

- If it is a python script, the interpreter should be something like `#!/usr/bin/env python`
- This is called a shebang. In Unix-like operating systems, the shebang line provides the **path** to an **executable** program (e.g. `bash`, `python`) that can interpret the following lines as executable instructions, allowing the user **to run the text file as an executable program** by typing the name of the file directly in the shell *provided the execute permission bit is set.* See an example in page 10.

13. `hostname` and `hostname -I` returns your name in your host or network, and the latter gives your IP address in your network. For macOS, use `ifconfig`, look for IP under `en0 → inet.`
14. `ping <servername>` checks connection to a server. For example ping google.com tells you whether your connection is active or not.

I/O redirection

Standard output

`echo hello` is a command that means ”output the string `hello` to standard output”. A **standard output** is a default place for output to go, also known as **stdout**. Your shell is constantly watching that output place, and whenever there’s something there, it will automatically print to your screen.

Standard input

The standard input (**stdin**) is a default place where commands **listen** for information. For example, all the commands above with no other arguments listens for input on stdin. Try typing `cat` on the command line and press enter. Notice you can type any character from your keyboard, because **it listens for input on stdin,** outputting what you type to **stdout** (and your shell is watching that output place so it is being printed on your screen), until you type an `EOF` (end of line) character: `CTRL + d.`

Standard error

The standard error (**stderr**) is the place where error messages go. Try this command that will prompt an error such as: `cat <inexistent_filepath>`. What is the output that you see? Similar to stdout, stderr is printed directly to your screen by your shell.

Similarly, we can redirect **stdin** using < operator. If we do `<command> < filename`, it means that we use the *content* of filename as an input to command. This is particularly useful for commands that only take in input stream, and is unable to read the content of a file given a filename.

One example is `tr`, a command line utility for **translating** or **deleting** characters. Do the following:

1. Create a text file with the following content: ”hello, have a good day today!”, name it test.txt and save it to your current directory.
2. Then, type the following in the command line (dont forget to navigate to your current directory first using `cd`): `tr "[a-z]" "[A-Z]" test.txt`
3. You will see usage suggestion instead:
   ```
   usage:     tr [-Ccsu] string1 string2
              tr [-Ccu] -d string1
              tr [-Ccu] -s string1
              tr [-Ccu] -ds string1 string2
   ```
   This is because `tr` cannot get input directly from a file. It only reads from **stdin**.
4. Now try `tr "[a-z]" "[A-Z]" < test.txt`. Your console should print ”`HELLO, HAVE A GOOD DAY TODAY!`”, capitalizing the content of `test.txt` file (but not changing its content).
5. **[1m]** So based on your observation, can you deduce what is the difference between `tr "[a-z]" "[A-Z]" < test.txt`
   and `tr "[a-z]" "[A-Z]" test.txt`?

Your answer: For `tr “[a-z]” “[A-Z]” < test.txt`, the content of test.txt is being redirected by the input redirection operator (<) as STDIN for the tr command. For `tr “[a-z]” “[A-Z]” test.txt`, the string “test.txt” is considered as an extra operand for the tr command, which leads to an error since based on tr’s man page, it is an invalid format (maximum of 2 SETs).

6. Now what if we want to store the capitalized content to another file?
   Try: `tr "[a-z]" "[A-Z]" < test.txt > new_test.txt`. You should find that “`HELLO, HAVE A GOOD DAY TODAY!`” exists within `new_test.txt`, since we redirect **stdout** to create this new file.
7. **[1m]** What if we want to write back to test.txt?
   Try `tr "[a-z]" "[A-Z]" < test.txt > test.txt` and comment on your observation below.

Your answer: The test.txt is empty. This is because the shell opens the test.txt file as an output file and prepares the output filehandles before executing the tr command, but this cleans up the file first before the tr command ever has a chance to even read it. As such, this one-liner clobbers the output file. This is because the output redirection command (>) is designed to overwrite the output file, compared to the appending redirection operator (>>).

Purpose of a Command Line

Why do we need to use the command line? What can you do here that you cannot do through your common graphical user interface? Well, it depends on your purpose. If you are just a basic user, i.e: browse, watch your favourite tv series, edit photos, or text your friends then chances are you don't need to use the command line. *If you're a computer science graduate who intends to work in the field then CLI is probably your new best friend.*

The most common use of the command line is "system administration" or, basically, managing computers and servers. This includes **installing and configuring software, monitoring computer resources (manage logs, setup cron jobs, daemons), setting up web servers (renaming or modifying thousands of files), and automating processes (setup databases / servers) on many hosts.** Obviously these tasks are repetitive and tedious such that it is impossible to be done manually or one by one.

Directory and Environment Variables

In a desktop environment, you have windows, menu bars, the desktop, etc to give context to what you are doing (graphically). In the command line, however, **the context is solely the file system (week 6 material)**. In fact, files and directories are what make up the command line. **Almost everything you do at the prompt will deal with files.**

When you first open a terminal window and type `pwd`, you will typically be in your home directory already (unless you use a different setup for your terminal). **Your current directory** provides context for the commands you run:

- For example, you can use the ls command to list the files in a directory thats not your current directory, e.g: `ls /Users/natalieagus/Desktop`
- When you run the command `ls` by itself, it uses your current directory as the **context**, and lists the files that are in the directory you are in.

Another way of providing context is through something called **environment variables**. Tryout command: `cd $HOME`

The `$HOME` part is a reference to the `HOME` variable, and is replaced by the path to your home directory when the command is run. In other words, running `cd $HOME` is the same as running `cd <actual path to your home>`

To find out your current environment variables, do the following:

1. Enter the command `env`
2. Find the value for `HOME` and `PATH`
3. Now type echo `$HOME` and echo `$PATH`
4. **[1m]** Are the answers from part (2) and (3) the same?

   Your answer: Yes, they are the same.

5. You can make your own environment variables using the command export, i.e: `export MESSAGE1 = "This is message 1"`. You can now do echo `$MESSAGE1` and the message ”`This is message 1`” will appear in the console.

One of the most important environment variables you'll work with on the command line is `PATH`. This is the key on how the shell knew which file to execute for commands like `cd` or `echo` or other built-in or installed programs. The `PATH` variable provides the additional **context** that the command line needs to figure out where that particular file is in the system.

- **[1m]** Examine the value for $PATH and write it below.

  Your answer: [REDACTED]

- Open your folder (from your Desktop GUI) and navigate to that path. **You may need to enable viewing of hidden files.** If you are using WSL, you need to cd to this path as there's no GUI.

- See whether you can find the location of some of your system programs such as ls, cd, echo, etc.

This means that you can effectively execute any **binaries** if you have added its location to the `$PATH` environment variable:

- To add to path, type export `PATH=$PATH:<your new directory path>`. This **appends** your new path to the existing paths.
- To test, type `echo $PATH`
- Now you can directly launch any executable in that directory without having to navigate to that directory first using `cd`
- **[1m]** Close your terminal and open a new session. Type echo $PATH. Does your new path still exist?

  Your answer: No, the new path entry disappears since it was only added temporarily to the context of that specific terminal session (unless you add it permanently to the ~/.bashrc file).

Bash Scripting

Now we have learned a bit about shell, let's move on to bash. Bash is a default interpreter on many systems, so you have basically used it when you typed all the commands in the previous section.

To see your default shell **interpreter**, type `echo $SHELL`.

So what is scripting? It is simply lines of code for the CLI to interpret. Imagine you want to create thousands of text files (using `touch`). You wouldn't want to type the command one by one, but rather just run a script that does this task in one shot.

Do the following:

Open a text editor and type the following:

```
#!/bin/bash

echo "Hello world"
```

1. Save it as `helloworld.sh` in a directory that you know.
2. `cd` to that directory
3. Then do: `chmod +x helloworld.sh`
4. Execute the script by typing `./helloworld.sh`

You have just created and run a super simple bash script. Note that the first line is the **shebang**. Similar to coding in any other language, you can use variables, functions, conditional statements, loops, comparisons, etc in your bash script. We do not have enough time to run them all, but the following bash script shows some of the common operations:

1. Open a text editor and type the following:

```
#!/bin/bash
greeting="Welcome"
user=$(whoami)
day=$(date +%A)

echo "$greeting back $user! Today is $day, which is the best day
of the entire week!"
echo "Your Bash shell version is: $BASH_VERSION. Enjoy!"
```

```
string_a="UNIX"
string_b="GNU"

echo "Are $string_a and $string_b strings equal?"
[ $string_a = $string_b ]
echo $?

num_a=100
num_b=200

echo "Is $num_a equal to $num_b ?"
[ $num_a -eq $num_b ]
echo $?

if [ $num_a -lt $num_b ]; then
    echo "$num_a is less than $num_b!"
fi

for i in 1 2 3; do
        echo $i
done
```

2. Save it as `myscript.sh`
3. Make it executable using `chmod +x myscript.sh` command and execute it by typing `./myscript.sh`
    - The shebang line `#!/bin/bash` assigns the **bash** as the interpreter of the script automatically.
4. You should see such output on your screen:

```
Hello World
Welcome back (username)! Today is Thursday, which is the best day
of the entire week!
Your Bash shell version is: 3.2.57(1)-release. Enjoy! Are UNIX and
GNU strings equal?
1
Is 100 equal to 200 ?
1
100 is less than 200!
1
2
```

```
3
```

Debug: you need to type out EXACTLY the bash script as shown, including the spaces, i.e: in the following, the leading and trailing space after [and before] are intentional.

[$string_a = $string_b]

[1m] What is the purpose of `echo $?`

Your answer: It is to print the exit status of the last/previous command in the script to the display screen/console/terminal. Commands on successful completion (without any errors) would most likely exit with an exit status of 0.

Do not worry about Bash programming language. The purpose of this lab is just to expose you to fundamental basic concepts about shell and bash. You will not be tested with materials that are out of the scope of the lab.

Unix Makefile

Download this and look at the files inside `makeFileDemo`. Read all the .c and .h files and get an understanding of what each file is supposed to do. To compile the files and run the executable:

1. Navigate to this directory and type the command `gcc -o myexecoutprog.o main.c hello.c factorial.c binary.c`
2. And then execute by typing `./myexecoutprog`
3. Experiment with the program a little bit. You should see this prompt:

```
Hello World!
Key in a number to obtain its factorial:
```

Basically, `gcc` **compiles** all the input argument files: main.c, hello.c, factorial.c, binary.c and produces a binary output (this is what -o means) named myexecoutprog.o which you can **execute** using `./myexecprogout.o`

In this context, it is feasible to type out the source file one by one each time you want to compile your program. However in a **large** scale project with thousands of files, it is very **tedious** to type the compilation command all the time.

Hence, the make command allows us to compile these files more easily. It requires a special file called the `makefile`.

1. Now instead of typing gcc and all that above, type `make` instead
2. After executing `make`, realise that myexecoutprog.o is made. You can run the executable in the terminal by typing `./myexecoutprog.o` or by simply clicking that executable in your shell GUI (your desktop).

Let's now examine how `makefile` is made.

1. Open `makefile` with your text editor.
2. The first four lines are the **MACROS**, which are convenient shorthands you can make to make your life easier when typing these codes.
3. Afterwards, there's a bunch of **explicit rules** that you can call using `make`. So in this makefile, you can try calling these in sequence and observe what each rule do:
   (a) `make myexecout`
   (b) `make myexecoutFromMacro`

```
(c) make clean
(d) make cleanFromMacro
```

Recompiling

Run this command consecutively:

- make clean
- make myexecout2.

You should see the following output on your terminal as a result of make myexecout2:

```
gcc -c -o main.o main.c
gcc -c -o hello.o hello.c
gcc -c -o factorial.o factorial.c
gcc -c -o binary.o binary.c
gcc -o myexecoutprog2 main.o hello.o factorial.o binary.o
```

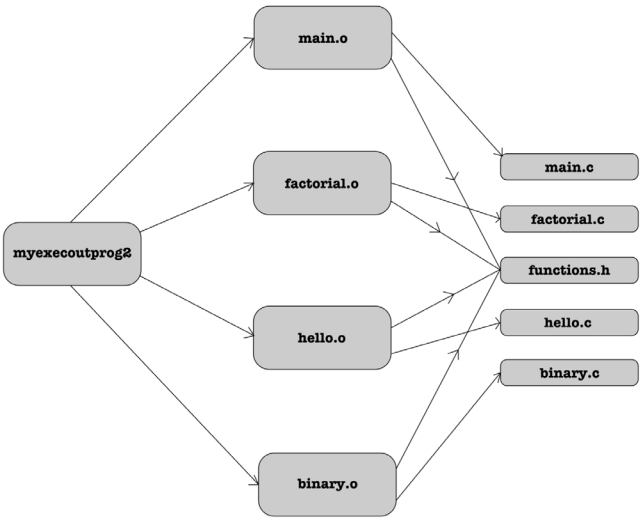
Now open `binary.c` **in the text editor and add another function in it**. You can write any function you want, the point is just to modify the file. Save your changes, and run make myexecout2.

Observe: The output will show compiling only on files concerning binary.c, not all files are recompiled.

```
gcc -c -o binary.o binary.c
gcc -o myexecoutprog2 main.o hello.o factorial.o binary.o
```

Scroll down to the end of the makefile, and notice there's **implicit** rules there to determine dependency. This gives more **efficient** compilation. **It only recompiles parts that are changed.** The Figure below shows the data dependency between files that are specified in the `makefile`.

Note: File_1 → File_2 means that File_1 ***depends*** *on File_2.*



Now answer the following questions:

1. **[1m]** Which file you should change in order to force gcc to recompile everything?

   You answer: functions.h

2. **[1m]** What is the advantage of using a makefile as opposed to just typing the `gcc` commands yourself?

   You answer: A makefile allows someone to describe and automate the parameters/steps of the compilation workflow of some C code. A makefile also allows someone to perform an incremental build of the C code workspace, whereas simply using gcc directly and typing the gcc commands ourselves will re-compile everything again.

3. **[1m]** Why is it more efficient to recompile parts that are changed as opposed to recompiling the whole program?

   You answer: Particularly in a large codebase, when the programmer only implemented some small changes and modifications, they do not need to re-compile and rebuild the whole C codebase with all of the different packages/features. Instead, they can just focus on rebuilding their immediate part of concern. As such, the build times would be much faster.