

UNIX 2

In this lab, over three exercises, we will develop a shell script that behaves as a terminal “screen saver”. We will:

1. Learn to use git and an ascii terminal based editor
2. Learn some basics of shell script programming

Note

At the bottom of this handout is a simple example of the kind of script we will develop. Remember, when it comes to programming there does not need to be a right answer. There can be many ways to do the same thing. Often it is a matter of taste and experience that will influence how we write things.

We encourage you to follow along and only consult the example at the bottom if you have tried several times but are still confused on how to proceed.

Setup

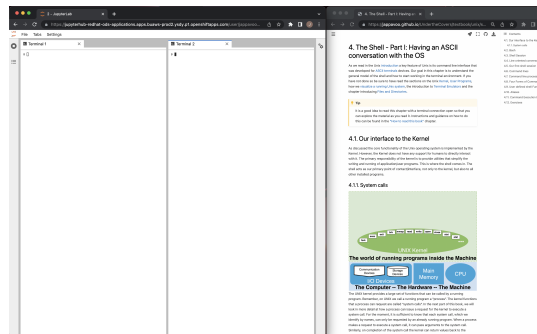
Step 1

Form partners and do a quick scan of the three exercises. Then get setup and start on Exercise 1. The TF and CA will be walking around and available to help. Note you may not get everything done in the lab session. That's ok, you are encouraged to take the handouts home and continue to play with exercises.

Using the Chrome web browser:

1. Login to the UNIX environment
2. Start your server and create two terminals arranged side by side
3. Open another web browser window with “Under the Covers: The Secret Life of Software” open to chapter “4. The Shell - Part I: Having an ASCII conversation with the OS”

Things should look something like this:



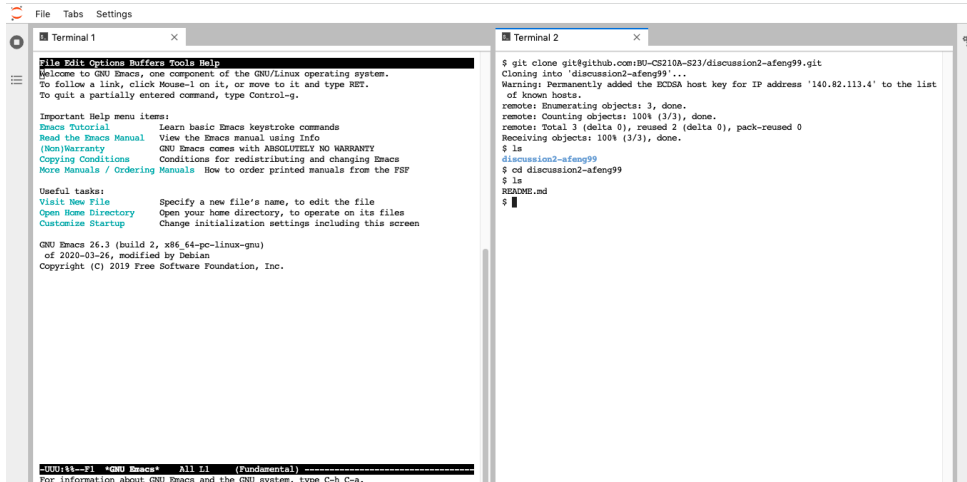
The two terminal windows we have opened each have a running independent bash shell process connected to them. The command lines we will enter into the terminal windows will be sent to the connected shells, and any responses from the commands are sent back to the terminal window that the shell is connected to. In the rest of the lab we will refer to the left terminal as terminal 1, and the right terminal as terminal 2. To switch between the two terminals you need to use your mouse and click in the window of the terminal you want to be active. Once active, the things you type at the keyboard will go to that terminal window.

Step 2

We will keep ourselves organized by using terminal 1 to run our editor and terminal 2 to run bash commands.

1. Start the emacs editor in terminal 1
2. Git setup
 - Go to piazza in another browser tab. Under resources you will find a link to the discussion repository github classroom invite link. Follow the link, accept it and copy the ssh git clone url.
 - Clone the repository in terminal 2 and `cd` into the directory that is created. Run `ls` to view the files in that directory.

At this point your UNIX environment should look something like this



One of the nice properties of the shell is that anything you can do interactively, you can write a script to do. Similarly, this means that an easy way to develop a script is to test out your script in an interactive shell session. To demonstrate this we will use terminal 2 to test out command lines, in addition to running our git commands. As we figure out useful command lines for our script, we will use emacs running in terminal 1 to add them to our script file.

Exercise 1: Exploring Variables

In this exercise we will explore how shell environment variables work to make writing our screen saver easier.

To write our screen saver we need to know how “big” the terminal window is in lines and columns. We can use the UNIX `tput` command as follows:

```
$ tput lines
20
$ tput cols
80
$
```

Try these `tput` commands out in terminal 2. Notice that you might get different numbers based on how big your terminal window is.

Cool. In our script we will use two variables to hold the number of lines and rows of the display. Let's play around with how to do this in terminal 2.

Try setting a variable to the values you get back from `tput line` and `tput cols`.

Eg.

```
$ lines=20
$ columns=80
$
```

Now use echo to explore how the shell will expand `$lines` and `$columns` into the values they are set to.

i Shell variable expansion

When the shell encounters a word that begins with `$` it will try to perform several types of expansions. The one we care about right now is simple ["Variable Expansion"](#) (See Parameter and Variable Expansion in the textbook). So, if we write `echo $hello`, the shell, before it runs the echo command, will replace `$hello` with the string value of the `hello` shell variable if one exists. If one does not exist, it will substitute the empty string. Each time we use the syntax `var=value` we are introducing or updating a shell variable whose name is `var` to the string `value`.

Eg.

```
$ echo $hello
$ hello="nice string"
$ echo $hello
nice string
$ hello="bad string"
$ echo hello is currently set to: $hello
```

So given this, what will `echo $lines` do? Remember we can also do things like: `echo number of lines: $lines`

Starting our script

Now using emacs in terminal 1 lets start developing our script which we will call `saver`. First thing to do is to tell emacs we want to start working on a new file. To do this use the menu bar at the top of the emacs terminal. To activate it press the `F10` key. If your keyboard does not have an `F10` key then ask your TF/CA for help running the `menu-bar-open` emacs command.

Now use the arrow keys to navigate to the `Visit New File...` menu option and press return.

i Note

Many of the menu options list the keyboard shortcut you can use to do the same function without having to open the menus. In the case of creating a new file it is `C-x C-f` which corresponds to the following key sequence: press and release the control and the x key followed by the control and the f key.

At this point, in the lower portion of the emacs terminal called the `mini-buffer`, you will be prompted for the path of the new file. In this case give it the path of a file named `saver` within the repository directory. Eg. `~/discussion2-jappavoo/saver`.

At this point, the things you type into emacs can be saved to the path of the file you specified.

See the "how to use emacs" howto on piazza for some pointers on using emacs.

Lets add a few starting lines to our script.

The first line we want will tell the kernel to start a bash process and have it read and execute the commands from the file

```
#!/bin/bash
```

see UCSLS 4.7.8.2. Non-Interactive Mode (Script Mode), specifically the hint for more information about this line.

Now let's add two lines that set our lines variable, `lines=20`, and columns variable `columns=80`.

At this point in terminal 1 we should have something like the following (remember to set the value to the size of your terminal not the ones in the example)

```
#!/bin/bash

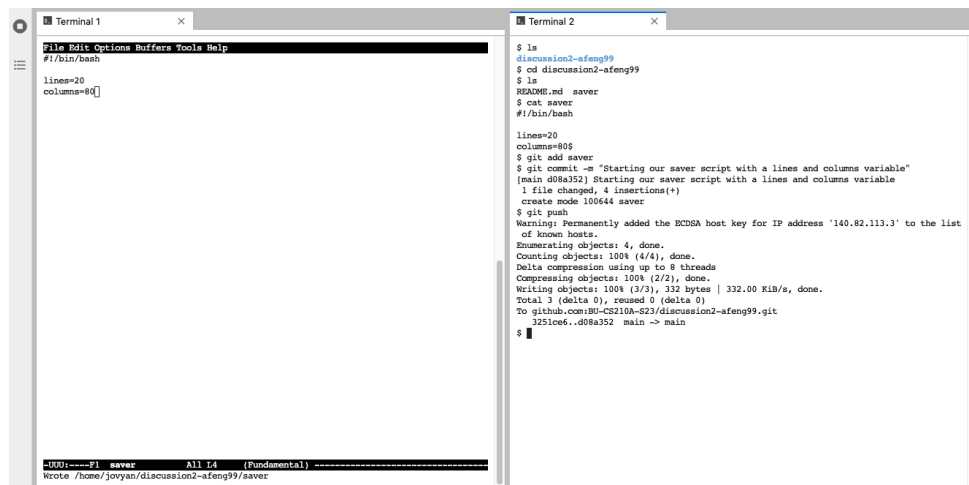
lines=20
columns=80
```

Now let's save what's in emacs to the file. Use the file menu again but select the **Save** option (notice its shortcut `C-x C-s`).

Now in terminal 2, in the repository directory, use `ls` to confirm the file was created. Make sure you are inside the directory before using the command. Now use `cat` to copy the file's contents to your terminal. Does it look right?

Now let's use `git` to record this version of the file and to save a copy to our github main repository.

1. `git add saver`
2. `git commit -m "Starting our saver script with a lines and columns variable"`
3. `git push`



```
Terminal 1
File Edit Options Buffers Tools Help
#!/bin/bash

lines=20
columns=80

Terminal 2
$ ls
discussion2-afeng99
$ cd discussion2-afeng99
$ ls
README.md saver
$ cat saver
#!/bin/bash

lines=20
columns=80
$ git add saver
$ git commit -m "Starting our saver script with a lines and columns variable"
[main d08a352] Starting our saver script with a lines and columns variable
1 file changed, 4 insertions(+)
create mode 100644 saver
$ git push
Warning: Permanently added the BCDSA host key for IP address '140.82.113.3' to the list
of known hosts.
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 332 bytes | 332.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:BU-CSD10A-823/discussion2-afeng99.git
3251ce6..d08a352 main -> main
$
```

If you run into any errors, poke around and get some help from the TF/CA to understand what went wrong. Once you get these commands to succeed, use a web browser window to visit your main repository and confirm that your `saver` file is saved there and its contents are correct.

Now to practice, add some comments to the file and repeat the above process. Don't forget to add a small meaningful message to the commit (eg. "add comments").

The following is an example of how you might update the contents of "saver"

```
#!/bin/bash

# My simple terminal screen saver script

# lines holds the size of the terminal window in lines
lines=20
# columns holds the size of the terminal in columns
columns=80
```

Testing our script

In terminal 2, let's try running our script as it stands so far.

1. First we must mark the `saver` file as executable with the following command `chmod +x saver`

2. Now we can run it by using the shell command `./saver`

At some point you should read “4.7.6.1.3.2. Commands as files within the PATH list” to understand what these steps are doing.

Of course at this point the script does not really seem to do much but its a place for us to start.

Exercise 2: Loops and arithmetic

Ok now that we know the basics of how to develop a shell script, let's play around with various things that will be useful in developing our script.

First, here are a few external commands that we will use (remember you can use the `man` command to learn more about these

1. `clear` - clears the terminal of all text and places the cursor at the first row and column (top left)
2. `sleep n` - where n is a integer number. Does nothing for the n seconds
3. `date` - prints the current date and time
4. `true` - this is actually an external command – what does it do?

In terminal 2 play with the above commands and get a sense for what they do.

Try this one liner.

```
date; sleep 2; clear; date; sleep 2; clear
```

Does it give you any ideas?

Add the above line to your script, save, test, add, commit and push (same steps as before). Remember this is our pattern for doing work ... get used to it.

An infinite loop

Try the following one liner and figure out what happens. Note you can send the interrupt execution signal by pressing the Control and the c key (C-c) at the terminal.

```
while true; do date; done
```

Yikes!!! What's happening. Remember you can press control-c to end the loop

Cool, combine this with your script. You might do something like this

```
while true
do
    date; sleep 2; clear;
done
```

A for loop

Let's now assume that we want our screen saver to progressively change the line on which the date appears by one line. Once we reach the bottom, we should reset and start printing the date back at the top.

To do this we can nest a for loop within the while loop. Our goal is to develop a for loop which we can use after we clear the terminal to move to a particular row. To move down one line we will simply use `echo` to display a blank line.

For example, what does this do?

```
clear
for ((j=0; j<10; j++)); do echo; done
date
```

Lets now assume that we have a variable `i` that holds the number of lines to go down.

Eg.

```
i=10
```

How can we modify our for loop to use `i` as the limit?

```
clear
for ((j=0; j<i; j++)); do echo; done
date
```

Cool, so now we have a way to get to a row on the screen. With this knowledge modify the script as follows:

```
i=10
while true
do
  clear;
  for ((j=0; j<i; j++)); do echo; done
  date;
  sleep 2
done
```

Now do the standard steps to save, commit and push to the repository.

Arithmetic and shell variables

You might have noticed something strange when we using bash's arithmetic support Eg. math enclosed in double parenthesis `((x=y+5))`. We don't need to use `$` to expand the value of a shell variable (eg we don't need to say `((x=$y+5))`). That is because the arithmetic command of bash is a special case where it knows that names (not numbers) should automatically be treated as variables and substituted with their current value.

Random number and modulus

Bash has several special variables that you can use. One of them provides you with a random number – `RANDOM`.

Eg.

```
echo $RANDOM
```

Play with this.

Arithmetic expressions have support for modulus. Combining a random number and modulus operation lets us generate numbers from 0 to `n-1`.

```
for ((i=0; i<100; i++)); do echo $((RANDOM % 10 )); done
```

Can you write a loop that uses this ability to move the cursor a random number of spaces along a line? Remember passing `-n` to echo will suppress the printing of a newline. So to print a single space on the current line we could do the following

```
echo -n ' '
```

Argument position parameters

When we invoke a command, the shell has the kernel pass the arguments to it. In a shell script these "positional" command line arguments are accessible via numbered variables starting at 1.

Try the following:

1. Add the following two lines to the beginning of your script

```
echo $1 $2 $3
exit
```

1. Now run your script as follows: `./saver hello 4 foo`
2. Notice what gets printed. Try running it a few more times with different arguments.
3. Now you can remove these lines and you can use the positional parameter variables in your script to make it more flexible.

Note

The positional variable 0 expands to the name of the command. Eg. `echo $0` will print the name of the command in the command line. Additionally, the variable `#` will expand to the number of arguments that were specified. Eg. `echo $#` will expand to 3 in a script that was invoked with three arguments.

Exercise 3: Putting it all together

Now change your script to take in, as arguments, the number of lines, columns and how long to display the date for before clearing the screen.

Then add an if statement that exits if the script was not invoked with three arguments. To do this try to use the test operator `.`. See `help test` and `help [`.

```
if [[ -z $x ]]; then echo "x is zero length (empty)"; else echo "x has non zero
length and is $x"; fi
```

Finally, modify your script so that it progressively moves down the screen and randomly picks a column to start printing the date. Read the example code to see what we mean.

From here, modify to your hearts content:

- personalize what it displays
- have it randomly pick a row
- try your hand at ascii art
- command expansion could make this script much nicer
 - what does this do `echo $(tput lines)`
 - what does this do ``echo -n $(date)``

Example `saver` script

Note a final version would have more comments.

```
#!/bin/bash

lines=$1
columns=$2
delay=$3

if [[ -z $lines || -z $columns || -z $delay ]]; then
    echo "USAGE: saver <lines> <columns> <delay>"
    exit -1
fi

while true; do
    for ((i=0;i<lines;i++)); do
        clear
        for ((j=0; j<i; j++)); do echo; done
        cols=$(( $RANDOM % $columns ))
        for ((j=0; j<cols; j++)); do echo -n ' '; done
        date
        sleep $delay
    done
done
```

By Jonathan Appavoo

© Copyright 2021.