Homework 2
Shells and Your Environment

Assigned: Friday, January 15, 11:00AM

Due: Friday, January 22, 11:00AM (Hard Deadline)

Submission Instructions

Submit this assignment on Gradescope. You must submit every page of this PDF. We recommend using the free online tool PDFescape to edit and fill out this PDF. You may also print, handwrite, and scan this assignment.

1 Understanding your environment

When you open a new terminal window, a lot has actually happened behind the scenes before your prompt shows up. In lecture we introduced the files `~/.bashrc` and `~/.bash_aliases` as some files that are read when a new session starts, however, there more than just those.

List all of the files read by your shell during startup that contribute to your environment. List files in the order that they are read. *Hint: Don't forget about global files, such as those in* `/etc`

> This was not a good question, I apologize. The intent was for you to explore some of the things that happen "behind the scenes" every time you start things up.
>
> The "ideal" way to answer this would have been to check out the `FILES` section of `man bash`. Then, in the spirit of exploration, add `echo "Currently in XXX"` to each of these files to see what happens and figure out the ordering.
>
> The open-ended lectures did not end up going in a direction to help you with this, nor did the question give you enough hints to get here. I apologize for that and am aiming to correct things moving forward.

Run the command `set`.[1] This command prints your current environment, that is, all of the variables currently set to any value. Also try the command `env`, which only includes variables that have been "exported", that is variables that will be set for child processes. Why do you think so many variables are set, but not exported?

Pick a variable that is set but not exported (something printed by `set` but not by `env`). Explain why you think that variable is useful for the shell process, but not for a child process spawned (created) by the shell:

> There are tons of these. One example is variables such as `HISTFILE` and `HISTCONTROL` that affect the behavoir of bash itself, and don't make sense to export to subprocesses.

While some variables should be familiar from the first part of this question, there are many more variables in your environment than those set by `.bashrc` and friends.

Pick a variable not set during shell startup. Explain where that variable comes from (what sets it) and what its value means *(do not use the same variable as the previous question)*:

> This question was not worded particularly well either. What I wanted was for you to understand that many environment variables are set before bash ever starts up – that your terminal/bash inherit a fairly complex environment. Many things that are exported (things printed by `env`) fall into this category. Things related to the window manager (`DISPLAY`, `XAUTHORITY`, `GDMSESSION`) or the login controller (`MAIL`) are examples.

[1] That dumps a lot of text to the screen. Try running `set | less` to get a scrollable view or `set > output.txt` to save the output to a file for easier viewing.

2 Special Variables

Bash has quite a few special variables that can be very useful when writing scripts or while working at the terminal.

What does the variable `$?` do? Give an example where this value is useful

`$?` holds the return value of the last command run. By convention, this is 0 if the command succeeded and nonzero if the command failed. It can be useful when writing scripts, especially if you want to have fairly complex things happen depending on success or failure (making it hard to just use `&&` or `||`).

Try writing a simple program and varying the return value from `main`. Run your program and look a the value of `echo $?`.

What does the variable `$_` do? Give an example where this value is useful

Several people answered this question by explaining what the `$` character does in bash (it's responsible for *variable substituion*, it replaces what followed with the contents of the variable).

The question asked about the underscore variable, accessed via `$_`. It holds the argument of the previous command,[2] and is useful as a shortcut when working in the terminal. Homework 4 gives an example of its use.

3 Understanding your `PATH`

In a terminal, type `PATH=` (just hit enter after the equal sign, no space characters anywhere). Try to use the terminal like normal (try running `ls`). What happened?

Give an example of a command that used to work but now doesn't:

Most things don't work `mv, cp, rename, gcc, make, ....` I even gave you one example in the question itself (`ls`).

Can you still run this command with an empty `PATH`? How?

Lecture showed off the `which` command, but that may have flown by.

This is a great example of something that should have been managable to figure out on your own. The first google result for "ls command not found", is a stack exchange post where (admittedly the second response) says "To check that it is indeed a problem with your path, what's the result of `/bin/ls` ?" (The query "what does PATH do" is equally fruitful).

Another good way of problem-solving this would be to open a new terminal and look at the original contents (i.e. `echo $PATH`). The colon-separated list is a little cryptic, but exploring those directories would hopefully let you figure this out.

Give an example of a command that works the same even with an empty `PATH`. Why does this command still work?

Anything built-in to bash would still work. `cd` is the easy answer here (why doesn't it make sense for `cd` to be an extrenal command?). For a full list, you can type `help` and see a list of built-ins (incidentally, `help` itself is a built-in and would have been a good answer to this question).

A interesting last question to add would have been, *Why do you need to type `./a.out` to run your programs instead of simply `a.out`? What is the dot in `./` doing?*

[2] At least, that's what it often is / is useful for in an interactive shell session (interactive session == you typing stuff), more precisely: "The underscore variable is set at shell startup and contains the absolute file name of the shell or script being executed as passed in the argument list. Subsequently, it expands to the last argument to the previous command, after expansion. It is also set to the full pathname of each command executed and placed in the environment exported to that command. When checking mail, this parameter holds the name of the mail file.", from `http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_03_02.html`

4 Basic Scripting

Recall from lecture that scripting is really just programming, only in a very high-level language. Interestingly, `sh` is probably one of the oldest languages in regular use today.

`make` is a good tool for build systems, but we can actually use some basic scripting to accomplish a lot of the same things. First, write a simple C program that prints "Hello World!". Write a shell script named `build.sh` that performs the following actions:

1. Compile your program
2. Runs your program
3. Verifies that your program outputs exactly the string "Hello World!"
4. Prints the string "All tests passed." if the output is correct, or prints "Test failed. Expected output >>Hello World<<, got output >>{the program output}<<".

I think this question would have been improved with a little more direction to help you get started, specifically, I would add:

To get you started, here's a simple shell script. Try copying it and running it.

```
#!/usr/bin/env bash

echo "Hello, I am a shell script"

# Anything after a # character is a comment

# Shell scripts are like working in a terminal, only the script types the
# commands automatically for you

pwd
ls

# You won't break anything with shell scripts. The best way to write one is
# to add some commands and run it until the script does what you want.
```

Recall that your shell script began life a simple text file, it's up to you to let the operating system know that this is actually now an executable program. (← this is a pointer to `chmod +x` without actually saying it)

~~**Copy the output of `cat build.sh` here:**~~ **Show that your script works as expected for both correct and buggy hello world programs. Copy the contents of your script below:**

As for a solution, there are **many** ways of answering this question. This is true of scripting in general. Scripts are (usually) not meant to be "pretty" programs. There is no need to have the shortest or most elegant solution. That's a waste of everyone's time. They are the kind of tool where you simply find something that works and move on.

To that end, I've copied in several different solutions from several different students that show off the various ways this could be done on the next page.

```
Probably my favorite submission, as the citation history at the bottom shows
how this student identified the three challenges (command output -> variable,
conditionals in bash, and string comparison in bash), found examples for each,
and synthesized a result:

#!/bin/bash
#1. Compile the program
gcc main.c -o hello_world

#2. Run the program
output_val=$(./hello_world)
expected_val='Hello World!'

#3. Verify the program outputs 'Hello World!'
if [ "$output_val" == "$expected_val" ]
then
  echo "All tests passed"
else
  echo "Test failed. Expected output >>Hello World<<, got output >>$output_val<<"
fi

Sources:
http://www.cyberciti.biz/faq/unix-linux-bsd-appleosx-bash-assign-variable-command-output/
http://www.thegeekstuff.com/2010/06/bash-if-statement-examples/
http://stackoverflow.com/questions/4277665/how-do-i-compare-two-string-variables-in-an-if-statement-in-bash

-------------------------------------------------------------------------------

#!/bin/bash
# Compile and make executable
gcc hello.c -o hello_exec

# Set the desired value in a temp variable
test="Hello World!"

# Store the output in a temp variable
output="$(./hello_exec)"

# Run diff against the desired string
diff <(echo $output) <(echo $test) &>/dev/null

# Check the exit code of diff
if [ $? = 0 ]; then
  echo "All tests passed."
else
  echo "Test failed. Expected output >>$test<<, got output>>$output<<"
fi

-------------------------------------------------------------------------------

$ cat build.sh
g++ hello.cpp -o hello
"Hello World!" > correct
./hello > output
cmp --silent output correct\
  && echo "All tests passed"\
|| echo "Test Failed. Expected output >>Hello World!<<, got output >>$(cat output)<<"

-------------------------------------------------------------------------------

gcc main.c -o main
./main > file1
echo -n "Hello World!" > file2
if diff -q file1 file2 > /dev/null; then
  echo "All tests passed."
else
  echo -n "Test failed. Expected output >>Hello World!<<, got output >>"
  cat file1
  echo "<<"
fi
```