Basic - Shell

EECS 201 Fall 2024

Submission Instructions

This assignment will be submitted as a repository on the EECS GitLab server (see Basic - Git 1). Create a private, blank, **README-less (uncheck that box!)** Project on it with the name/path/URL `eecs201-basic-shell` and add `brng` as a Reporter. The submission branch will be `scripts`. If this branch is not already the default initial branch, you initialize the local repo with an additional argument: `git init --initial-branch=scripts` or `git init -b scripts` if your version of Git is recent enough. Otherwise you can create a branch with this name after your first commit.

To finish the submission process and get your grade, you will need to run the autograder. You will need to SSH into the course server (see Basic - Intro):
`local$ ssh uniqname@peritia.eecs.umich.edu`
(the `local$` referring to a shell prompt on your system)

and then run `eecs201-test`:
`peritia$ eecs201-test basic-shell`
(the `peritia$` referring to a shell prompt on the server)

The repository should have the following directory structure, starting from the repository's root:

```
/
|-- get-my-info
|-- shebang-check
|-- report.txt
```

Preface

This assignment is worth a total of 9 points!. In this assignment you'll be tasked with writing a few scripts that tie together several of the concepts we've talked about in class. Recall that a (shell) script is just a plain-text file, openable in any text editor, that contains shell commands. Script files can be directly executed, taking advantage of the *shebang*, or be given as an argument to a particular shell program for it to run.

To get started, take a note of the submission branch in the submission instructions above, and create a directory. `cd` to this directory and initialize a Git repository with an initial branch of the name specified as the submission branch. You won't be provided any files this time: now it's up to you to create and edit these files. **Each script file will need to have a shebang. You are free to use `sh` or `bash`, keeping in mind the capabilities of either shell. In addition, set the execute bit with `chmod`: Git is able to track these permission/mode bits. If you happen to be using WSL and are working on this assignment on the Windows file system (e.g. stuff under `/mnt/c`), run `$ git update-index -chmod=+x <file>` to set the execute bit for Git, as the Windows file system doesn't have the same sense of these permissions as UNIX systems.**

As you work on these, remember that you can always test some things out at the command line. The commands that go into shell scripts are the commands that you enter in manually: you're just saving these commands for later when it comes to a shell script. Also, note that `#` indicates that the rest of the line is a comment: it's generally a good idea to write comments for your code, especially when shell scripting can look rather funky compared to more conventional programming languages.

1 Expansion Fun

Write a shell script named `get-my-info` (no extension) that:

1. Prints out `login: <user name>`
2. Prints out `OS: <operating system>`
3. Prints out `hostname: <computer name>`
4. Prints out `<dir>'s device: <device space used>/<total size> (<% used>)` The dir will be provided as the first argument: if there is no first argument, defaults to `/`. This will print out usage stats about the *device* that the directory is on (not the directory's actual usage).

Do not hardcode the values: this is information that will be gathered from the system at run-time. Wherever you see a space, there is only one space character.

For example:

```
# On the course server
$ ./get-my-info
login: brng
OS: Linux
hostname: peritia
/'s device: 13G/49G (27%)


# On the course server
# The /home/students directory is actually on
# a separate drive (i.e. device)
$ ./get-my-info $HOME
login: brng
OS: Linux
hostname: peritia
/home/students/brng's device: 4.7G/147G (4%)


# On my MacBook, to illustrate subtle differences in output format
# If you get something like this on your Mac, you should be fine
$ ./get-my-info /
login: brandon
OS: Darwin
hostname: pomum.local
/'s device: 15Gi/460Gi (4%)
```

I know this can look a bit daunting. Let's break this down into some managable chunks.

- You can get the username from the `logname` or `id -un` commands. If one doesn't work on your system, try the other: both work on the official testing environment.
- You can get the OS by running `uname -s`
- You can get the computer's name ("hostname") by running `uname -n`

`echo` can print strings out for you: combine that with liberal use of **command substitution** to *expand* the output of a command and with quoting you can manage how you print things out. For example, you can run `echo "welcome $(echo "$HOME" | rev)"`: this will run `echo "$HOME" | rev` inside of a subshell, capture its output, and then substitute that into the `echo "welcome $(...)"`.

It gets more interesting when getting a directory's device's disk usage. `df -h <directory>` (e.g. `df -h /home/brandon`) takes a directory, looks at what storage device that directory is on, and prints out some disk usage stats for that storage device (not the directory). You might recall this command from the Basic - Unix assignment.

When you run it, note how it prints out a line that labels what each column is: we want the data, not the column name. The `tail` command can help us here: it gets the last lines of output. Specifically, `tail -n 1` will get the last line of output. Remember pipes from the last assignment? They're our ever-present friend :). Now we need to get our data: the `awk` utility can help us here. `awk '{print $1}'` will get the 1st field (column). Replace `1` with another number and you'd get that number's field.

Putting this together, you can form a pipeline to grab fields of data from `df`! You might find that handling these pipelines and expansions a bit cumbersome: feel free to declare variables to handle certain things, like one to hold the total disk space, one for the used disk space etc. On both Linux and macOS systems, the `Size` column

represents the total size and the `Used` column represents the amount of space used. On Linux systems, the `Use%` column is the % of space used, and on macOS this is the `Capacity` column.

You might be wondering about the "default to `/`" part: recall from class how there was a parameter/variable expansion that allows us to have a default expansion value when a variable is unset/doesn't exist. For example `${hello:-world}` will try to expand the variable `hello`, but if `hello` has not been set/is undefined, it will expand out to the string `world` instead.
Try to think of a way to take advantage of this with the potential existence of the first argument `1`. Perhaps you can have a variable that represents the what directory to look at, regardless of whether there's an argument or not?
You do not need to do any math for this: `df` directly provides the values of interest.

2 Control Flow

Write a shell script named `shebang-check` (no extension) that:

- Let's define "checking" a file's shebang as follows:

  Checks if the first two bytes are `#!`, indicating a shebang. If there is one, print out the basename of the file (filepath minus the directories i.e. the "name" of the file) and the interpreter used (i.e. the stuff after `#!`): `filename: /path/to/interpreter` e.g. `some-script: /bin/sh`
  The following bullets are cases for behavior involving this "check"

- Case: there are no arguments.

  Iterates over every file in the current directory. If the file is a normal file, "check"s it, else, does nothing.

- Case: there are arguments.

  Each argument represents a normal file or a directory. If the argument is a normal file, "checks" it. If the argument is a directory, iterates over each file in that directory and "checks" it if it is a normal file, else does nothing (just like the no argument case, but with a particular directory). **There is no recursion here: with directory arguments, only check normal files inside of them and do not traverse subdirectories.**

- In any case, the output is sorted

Let's unpack this one. Right off the bat, you're given the description of an operation that'll be used over and over again: **looks like it's a ripe opportunity to define a function!** Functions in shell scripting act pretty much like miniature shell scripts: they are called like any other command, use arguments just like a script (e.g. `$@`), and can even provide exit statuses. Remember, just like any other shell command and syntax you can define functions at the command line in case you need to test something out really quick.

Some things that might help you writing your "check":

- `head` is able to get the first lines of a file. `head -n 1` will get the first line.
- `cut` is able to grab certain parts of lines of input. `cut -b 1,2` will get the character 1 to character 2 of each line. `cut -b 3-` will get from character 3 until the end of the line.
- `test` / `[ ]` use `=` as the string comparison operator (while `-eq` is for numeric comparison)
- `basename` can get you the basename of a path: `basename /usr/bin/vim` will output `vim`

When you write your "check", you might want to take advantage of the `test` / `[ ]` structure to check of the first two bytes of the file are indeed `#!`. You'll probably need to take advantage of **command substitutions** like in the previous problem. Remember that you're always free to declare variables when these substitutions begin to become more and more cumbersome!

For the main logic, you'll need to check if there are variables in order to decide what to do. The big part of this is iteration: you'll probably want to use a `for`-loop for this. Now the big question is how to get the list that you iterate over: if you want to try something new, you could try using a **filename expansion (glob/wildcard)** (e.g. `dir/*`) to populate a list of files for you, or you could use the output of `ls`: your choice (though I do highly recommend filename expansion). Another neat thing about `test` / `[ ]` is that it can do more than binary comparisons: you can also check if files exist or if they're normal files: `[ -f filepath ]` can check if a file is a normal file and `[ -d filepath ]` can check if a file is a directory. Try to put out some stepping stones: write a `for` loop that iterates over files, write an `if` that checks what a file is etc. Remember you can test things out at the command line: whatever you can put in a file you can type out!

Our last neat trick is these big multi-line "compound commands" like `for` and `if` are themselves commands: you can redirect the output of them or pipe them to things like `sort` ;). For example:

```
for x in 5 4 3 2 1; do
  echo $x
done | sort
```

Try it out!

How you approach your logic will be up to you. Maybe you'll find another place where you can declare and use a function. Will you get the files in a directory via `ls` or by filename expansion? There are a lot of choices. What I have presented here and on the videos and slides are all that you'll really need: you'll have to put together the puzzle. That being said, you can try other commands and try things however you wish as long as the course server supports it.

If you have trouble coming up with the overall structure, perhaps you can write a function that just performs the check on a file path provided as an argument. Perhaps you could also write another function that iterates through a directory and calls the first function to perform checks on regular files. It could look something like this:

```
# define a function to check a file

# define a function to check a directory
  # loop through files in directory
    # if file is a regular file, call the file check function

# if there are args, iterate through each arg
  # if the arg is a regular file, call the file check function on it
  # else if the arg is a directory, call the directory check function
# else if there are no args, call the directory check function on the current directory
```

You don't have to do it exactly this way, but it does allow you to reuse code via calling functions.

Gotchas

- Remember that whitespace separation happens after expansion: this can cause some "interesting" behavior inside of a `[ ]` testing command. Remember that you can quote!
- Remember how specific whitespace (or lack thereof) is for variable assignment!
- If you use `ls`, remember that `ls` lists its contents: it doesn't provide full paths for you to use. Remember outputs are just strings: they're not magical references to files, and keep in mind what your current directory is.

3 Conclusion

1. Create a file called `report.txt`.
2. On the first line of the file, put down an estimate of how long you took to do this assignment in minutes as an **integer (e.g. "37", "84": just numbers, no letters)**.
3. On the second line and onwards of the file, put down what you learned (if anything) by doing this assignment. **If you already knew how to do all of this, just put "N/A"**.
4. Stage and commit this `report.txt` on the submission branch.
5. Use `git push` to push this commit to your EECS GitLab repository.
6. Test your submission out on the autograder!

Symbolic links

This section is ungraded and is more for your personal edification. Do it in a random directory outside of your submission repo.
A "symbolic link" is a special type of file that "points" to another file. (This is similar to the idea of a shortcut to a file). In this question we'll be exploring the use of symbolic links ("symlinks").

1. Use `mkdir` to create a directory called `symlink` and `cd` into it. Use `touch` to create an empty file named `hello`.
2. Using `mkdir`, make a directory called `links`.
3. Let's make a symbolic link to `hello` called `world`:
   run `$ ln -s hello world`

4. Open the `world` file that gets created in a text editor, writing "`Hello world!`" in it and saving it.
5. Open the `hello` file. Note how the changes to `world` appear here.
6. What `ln -s` did is create a special file called `world` that links to a file called `hello` in the *current directory*.
7. We can see what a symlink points to by running `$ readlink world`. Alternatively `$ ls -l` will also show where symlinks point. Note how `world` points to `hello` by itself.
8. Using `mv`, move the `world` file into `links` directory.
9. Now run `$ cat links/world`. Note how it can't resolve the target file as the link refers to a file named `hello` in the same directory as it. You can run `$ readlink -f links/world` to see the exact ("canonical") path that the symlink resolves to.
10. Now run `$ ln -s hello links/foo` and `$ ln -sr hello links/bar`
    Now run `$ ls -l links`. Note the difference in where `links/foo` and `links/bar` point. The `-r` flag makes the pointer to the target file **r**elative to the link's location. You can also provide absolute addresses (starting from root `/`) as targets to link (with all of the issues that they come with: what would happen if I cloned this repo and tried to access that symlink?).