csce215 — UNIX/Linux Fundamentals
Spring 2022 — Assignment 7

This assignment is intended to provide some practice and additional content for the material covered in lecture on Monday, March 21. You'll practice working with jobs and processes, and also experiment with permissions. The assignment is meant to be started in the lab sessions on Wednesday, March 23 and Thursday, March 24. It must be submitted by 11:59pm on Sunday, March 27. A total of 92 points are available.

1 Get started

The format for this assignment is the same as the earlier ones. Just like always, you should record your work using `recbash`. And similar to the previous few assignments, you should make and navigate to a directory called `assignment7`, and then copy a set of files from the shared class directory into your own new directory:

```
cp -rv /class/215/assignment7/* .
```

Finally, as a reminder, if you attend the scheduled lab session, don't forget to give yourself credit for attendance by scanning the QR code displayed on the screens.

Record your attendance using the QR code. Record your terminal session using `recbash`. Create a directory called `assignment7`, and copy the provided files into it.

2 Permission denied

First, let's practice examining and changing permissions. Navigate to the `perm` directory that was copied into your `assignment7` directory. Notice the three files here. Use `ls -l` to examine their permissions. You may want to consult the notes from Chapter 7 to remind yourself about what each column in the permissions list means.

Use `cd` to navigate to the `perm` directory. Use `ls -l` to display the permissions of the files in this directory. *5 points*

We want to see what happens when you don't have adequate permissions to work with a file. So let's start by setting things up so that there are no permissions at all; no one is allowed to read, write, or execute any of the files here. (This is a somewhat strange thing to do, but will help to illustrate what each of those permission bits allows us to do.) To make this change, use the `chmod` command:

```
chmod -v a-rwx *-me
```

This might be a good time to recall `chmod` from your notes. Here the `-v` stands for verbose; without it, `chmod` will just change the permissions without producing any output, making it harder to tell what's happening. The `a` means 'all users', which include the owner ('user', `u`), members of the same group ('group', `g`) and other users ('world', `w`). The `-` means to remove permissions and `r`, `w`, and `x` refer to read, write, and execute permissions respectively. So the command says, for all users, to remove the read, write, and execute permissions.

The `*-me` at the end tells which files should have their permissions changed. It is a wildcard expression like we saw in Chapter 4, expanded by the shell to a list of all of the files ending in `-me`, i.e. all three files here: `read-me`, `write-me`, and `execute-me`.

Use the command above to remove all permissions from the three files in the `perm` directory. *3 points*

Now let's work with the file `read-me`. Before we change anything, first confirm that you cannot see the contents of the file by (attempting to) use the `cat` command to see it. You should see a `permission denied` error.

Then, use `chmod` to give yourself permission to read this file. (But don't add any other permissions.) Use `ls -l` to verify that the permissions have been changed to `-r--------`. Use `cat` again, this time successfully, to see the file's contents.

Try to display the contents of `read-me` and receive a permission denied error. Enable user read permission on this file using `chmod`. (Successfully) display the contents of the file using `cat`. Use `ls -l` to show the corrected permissions of the file. *8 points*

Next, let's try the file `write-me`. Try to edit this file with `vim`. When `vim` starts, you'll see a problem right away: Because we don't have read permission for the file, `vim` cannot read its contents to display them for editing.

Return to the command prompt by exiting `vim` and use `chmod` to add both read and write permissions for the owner on the file `write-me`. Use `ls -l` to verify that the permissions for this file are `-rw------`.

Now you can return to `vim` and things should work correctly. Modify the file `write-me` in some way. (The specific contents you add to the file are not important. Perhaps you'd like to follow its instructions and compose a computing-related haiku?) Then save, quit, and use `cat` to display the modified file to confirm that it has indeed been changed.

Try to edit the contents of `write-me` in `vim`. Exit or suspend `vim` and use `chmod` to add read and write permission for the user to this file. Edit the file in some way, then save and quit. Use the command `cat write-me` to display the modified file. *8 points*

The last file here is `execute-me`, which is a small shell script. (That is, a small program consisting of shell commands stored in a file.) To execute it, we'd use this command:

```
./execute-me
```

In this case, because the shell needs to find the program `execute-me` in your current directory (rather than one of the directories that normally would be searched for programs to execute), you'll need to type `./` before its name to execute it. More details about how the shell finds the programs it runs, including about the `./` thing, are coming in Chapter 8.

Attempt to run the program now. You should receive a permission denied error, because you do not yet have execute permission on this file.

Actually, you'll need both read and execute permissions here, because the system needs to read and process the commands in the file, which requires read permissions, before it

can execute them. So use `chmod` to add read and execute permissions for the user to the `execute-me`. Then verify that the permissions are `-r-x------` using `ls -l`. Then run the program again, this time successfully.

Try to execute `execute-me` and fail due to insufficient permissions. Use `chmod` to add read and execute permission for the user to this file. Run `execute-me` successfully. *8 points*

(We'll be learning more about shell scripts like this `execute-me` example in the coming weeks. If you are curious for a preview, perhaps you'd like to take a look at the contents of this file to see how it works?)

3 *Doing the job*

Now leave the `perm` directory and use `cd` to navigate to the other subdirectory, called `nope`. Here you'll see a small Python program called `nope.py` that we'll use to illustrate a few of the concepts of jobs and processes. Take a look at the contents of this file to get an idea of what it will do. Notice that it prints messages periodically (just like the `alive.py` example we saw in class) but also mentions some things about a 'terminate signal' that we'll see a bit later. The idea is to have a simple program that we can execute in several different ways.

Before we can execute `nope.py`, however, we need to ensure that its execute permission is set. Use `chmod` to give the file's owner, i.e. yourself, execute permission for `nope.py`. Use the `-v` option for `chmod` to get an output describing the change in permissions.

Using `chmod -v`, enable user execute permission on the file `nope.py`. *5 points*

Note that, because the file `nope.py` is in your current directory (rather than one of the directories that normally would be searched for programs to execute), you'll need to type `./` before its name to execute it.

Now we can run the program and experiment with using our shell to control its status. Recall the diagram in the notes for Chapter 7 showing the three states that a job can be in (foreground, background, and stopped). This diagram shows two arrows for two different ways to start a job, and four arrows showing ways to change the job between states after it's been started.

Try out each of these six arrows on a job that executes `none.py`. In addition, you should use `Ctrl-C` when the job is in the foreground to terminate the job — In this case, you should see a message from Python about a `KeyboardInterrupt` when that happens. As you go, you should use the `jobs` command liberally to keep track of the current state of things.

Hint: You can tell if `nope.py` is running by observing whether its periodic messages are appearing. You can tell if it's running in the foreground or background based on whether you get a command prompt or not. And don't forget that if outputs from a background job mess up the display of a command you are typing, you can press `Ctrl-L` to declutter the screen.

Do these seven things at least once each, not necessarily in this order:

1. Start `nope.py` as a foreground job.
2. Start `nope.py` as a background job.
3. Use `fg` to switch `nope.py` from a background job to a foreground job.
4. Use `fg` to switch `nope.py` from a stopped job to a foreground job.
5. Use `bg` to switch `nope.py` from a stopped job to a background job.
6. Use `Ctrl-Z` to switch `nope.py` from a foreground job to a stopped job.
7. Use `Ctrl-C` when `nope.py` is a foreground job, to terminate it.

25 points

4 *Processing the experience*

Let's try one more thing with `nope.py`: Using `ps` and `kill` to terminate it. To start, make sure you have no stopped nor background jobs, and then fire up `nope.py` as a background job again. Run `jobs` one more time to see that the program is indeed running in the background.

Establish `nope.py` as a background job, using either '&' or `bg`. Use `jobs` to verify that the job is in a background state. *10 points*

This time, instead of bringing the job to the foreground and terminating it with `Ctrl-C`, we'll use `ps` and `kill` to terminate it directly.

To use `kill`, we need to know the process ID of the process we want to terminate. Use the `ps` command to see a list of processes associated with the current terminal. You should see several processes in the list. In this list, the `CMD` column shows the program that each process is running. Look in this column for `python3`, which is the name of the interpreter that `nope.py` uses. Take note of the `PID` for this process.

Use `ps` to see a list of processes associated with the current terminal. *5 points*

Now that we know the process ID, we can (try to) kill the process. Use the `kill` command with the `PID` of the `python3` process as its argument. Then use `ps` again to see an updated list of processes. If you've done this correctly, you should see a rude message from `nope.py` when you run `kill`, and `ps` should show that the process still exists.

Use `kill` to send a terminate signal to the `python3` process that is running `nope.py` and receive an insult in reply. Use `ps` again to display a list of processes associated with the current terminal. *8 points*

What's happening here? Why didn't `kill` work to end this process? Doing these steps for most programs will convince them to end, either immediately or after short clean up period for things like deleting temporary files, closing network connections, and so on. But if there's some malfunction or misfeature in the program, it may not actually end when it receives the terminate signal. Our example program `nope.py` is designed to illustrate this case: If you use `kill` on it, it responds by printing a message instead of by ending itself.

This is exactly the situation that `kill -9` is designed for. It sends a termination signal that cannot be ignored, forcibly terminating the process. Use `kill` with the `-9` option to finally terminate the `nope.py` process. Be sure to give the `-9` option before the process ID. When this works correctly, `bash` will notice that the process for one of its background jobs has

been killed, and show a message telling you that's happened. Use `ps` one last time to verify that that `nope.py` process is really gone.

Use the `kill` command with the `-9` argument to actually kill the `nope.py` process running in the background. Verify that the process has ended using `ps`. *7 points*

5 *Fin*

When you've completed all of the tasks above, submit as usual. End the `recbash` recording using `Ctrl-D` or `exit`. Make sure that `recbash` has really exited— you might need to use `Ctrl-D` or `exit` twice in a row if you have stopped or background jobs. Check the transcript to verify that it shows all of the steps mentioned in the green boxes. Then upload and submit.

Upload your recording(s) and submit.