Pipes, streams, and the user environment

Hands-On Unix System Administration DeCal

Lab 2 — 10 September 2012

Due 17 September at 6:10 PM

1 Environment variables and bash startup files

1.1 .bashrc and .bash_profile

Bash uses a collection of startup files to create an environment. For example, you can initialize parameters, set your `PATH`, change what your prompt looks like, and much more. In this section we'll look at `.bashrc` and `.bash_profile`.

What's the difference? `.bashrc` and `.bash_profile` are invoked in different settings. `.bash_profile` is executing for *login shells*, while `.bashrc` is executed for interactive, non-login shells (See `man bash`). System-wide shell initialization files are typically in `/etc`, owned by root. They are `/etc/profile` and `/etc/bashrc`. Users' profile and shell rc files are owned by the user, and they reside in their home directories. (`~/.bash_profile` and `~/.bashrc`)

Why the distinction? Say, for example, there are certain things you'd want to see upon login only (such as system information, like load average, or number of current users, etc.). You'd want to edit `.bash_profile` to do that. If you put it in your `.bashrc`, you'd see it every time you launch a new terminal/shell.

1. Edit your `.bash_profile` and add a variable called `MYNAME`.

   ```
   MYNAME="Bob"
   ```

   Edit your `.bashrc` and add a variable, `MYFAVCOLOR`.

   ```
   MYFAVCOLOR="red"
   ```

   Save the two files, logout, and login again. (To be clear, we're expecting you to be using a login shell for the following exercises).

   Tip: You can emulate a login shell by running `bash --login`.

2. Now type the following.

   ```
   $ echo $MYNAME
   ```

   ```
   $ echo $MYFAVCOLOR
   ```

   What happens? Explain why.

3. Type `bash`, do the same thing.

   ```
   $ bash
   $ echo $MYNAME
   $ echo $MYFAVCOLOR
   ```

   What happens? Explain why.

4. You can see if you are using a login shell by running `shopt | grep login_shell`. How does the value of this option (on/off) correlate with the behavior you just saw?

1.2 Playing with environment variables

5. Open a terminal and define the following variables.

   ```
   $ FOO="foo" ; export FOO
   $ BAZ="baz"
   ```

   Now let's try accessing our newly defined variables

   ```
   $ echo $FOO
   $ echo $BAZ
   $ bash
   $ echo $FOO
   $ echo $BAZ
   ```

   What happens, and why? Explain the difference between `FOO` and `BAZ`.

6. Define a PS1 variable that includes the following:

   - The day and month
   - The current time in 12-hour am/pm format
   - your username
   - the hostname
   - the current working directory

7. Let's add a new directory to PATH.

   Before we do anything, note the output of `ls`

   ```
   $ ls
   ```

   Create a new directory.

   ```
   $ mkdir stuff
   ```

   Let's put a simple shell script in there.

   ```
   $ echo 'echo "herro"' > stuff/ls
   $ chmod u+x stuff/ls
   ```

Now alter your PATH variable such that the `stuff` directory is listed first.

```
$ PATH=~/stuff:$PATH
```

Now when you run `ls`, what happens? Explain why.

How does the order of `PATH` matter? What would happen if we had placed `~/stuff` at the end of the `PATH`?

8. Type the following, and note the output.

   ```
   $ echo $SHELL
   ```

   ```
   $ echo '$SHELL'
   ```

   What do you notice? Why does this happen?

9. Explain the behavior of the following command.

   ```
   $ echo '`find \$HOME -type d -print | wc -l`' > directories
   ```

   Modify it so it works as intended (i.e., count the number of directories in `$HOME`)

2 Wildcards

10. Match the filenaames lecture01, lecture09, lecture13, lecture26 with a wildcard expression.

3 Streams, redirection, and pipes

11. What is /dev/null usually used for?

12. What happens when you use

    (a) cat > foo, if foo contains data

    (b) who >> foo, if foo doesn't exist

    (c) cat foo > foo

    (d) echo 1> foo

13. Write a simple pipeline using `getent passwd`, and the `cut` utility (see `man cut`), to extract usernames and real names. You may want to run `getent passwd` to get a feel for what it does first.

3.1 Named pipes

14. Name one difference between named pipes and unamed pipes (other than the obvious).

15. Let's create a named pipe with `mkfifo`.

    ```
    $ mkfifo pipe
    ```

    If you run `ls -l pipe`, you'll see it looks just like a file. It could be deleted with rm, like so:

    ```
    $ rm pipe
    ```

    Now try running this command:

    ```
    $ ps -e > pipe
    ```

    Instinctively, you'd expect to be returned to a prompt, but it appears to "hang". Explain why.

    Now open another terminal, and run the following command in the second terminal.

    ```
    $ cat pipe
    ```

    What is the output of this command? Is the command you had typed in the first terminal still "hanging"? Explain the interaction between these two processes in a sentence or two.

4 Extra for Experts™!

4.1 Pipe madness

Using two commands, think of a way to generate a loop using named pipes. What happens to the CPU usage?