Bash Scripts

Class 6

Bash is a Language

- the bash shell is a program
- it has a built-in language interpreter
- just as powerful as JavaScript or Python
- has variables, branching, loops, and subroutines
- optimized for interacting with the Unix operating system

Shell Commands

- some commands that you give at the $ prompt are built-in
- they are functions within the bash program code itself
- examples are cd, echo, pwd, read, test

- some commands that you give at the $ prompt invoke external programs
- the shell sees that it has no function named "ls"
- it looks externally and finds the program /bin/ls and runs it
- examples are grep, less, cat, echo, mkdir, and test
- usually they are in /bin or /usr/bin
- or /usr/local/bin if you installed a 3rd-party program

- notice that echo and test are built-in and external commands — confusing!

Shell Commands

- echo and test are both built-in and external commands

- if you type `$ echo foo` you get the built-in version
- if you type `$ /bin/echo foo` you get the external version
- they're different

Execution of Built-In Command

- assume the shell is executing, waiting for another command at the $ prompt
- the running shell is a process
- the user issues the built-in command `$ pwd`
- the shell executes the internal function pwd and returns to the $ prompt, waiting for another command

Execution of External Command

- an external command is a separate program
- running it requires a new process
- the shell creates a new process using the C library function fork
- fork creates a new process that is an exact duplicate of the original process (i.e., another bash shell)
- the original process is the parent process; the new one is the child process

- the parent process goes into the wait state
- the child process becomes ready

Execution of External Command cont

- now the child needs to stop being a clone bash shell and become the requested external program
- the child shell does this by invoking the exec C library function
- exec allows a process to overwrite itself with the executable code of another program
- when that program finishes running, the child process is finished
- the child process terminates with a status code that is returned to the parent process
- when the parent process receives the exit status from its child, it "wakes up" and resumes its own execution (in this case, returning to the $ prompt)

Execution of Shell Script File

- you ran the script from the first homework like this:
  `$ ./filesize.sh`
- filesize.sh is not a built-in command, so it must be an external program
- but it is not a compiled, binary program
- to run this script, the parent process executes fork to create a child shell process, just like a binary external command
- but the child does not call exec because there is no binary program to overwrite itself with
- instead, the child process executes the shell commands in the script file exactly as though they were issued at the keyboard
- (any commands in the script file that are external commands cause a new sub-child process to be forked; this can go to arbitrary depth)
- when the child shell reaches end-of-file in the script file, it terminates, sending its exit code back to the parent shell

Subshells

- on the last slide from Wednesday, we saw this code:

  ```
  $ (head -n 1 contacts.csv && tail -n +2 contacts.csv |
     sort +2) > output.csv
  ```
- in bash, shell commands within parentheses are run in a subshell
- the subshell's standard output is given to the parent shell

Getting Help

- for (almost) every program on the system
- there is a manual page that describes it
- when someone posts a question online like to slashdot, such as “what do the options of ps do?”, a common response is RTFM, which loosely translates as “read the manual”

- to read the manual for a command use the man command:
  ```
  $ man ls
  ```
- this kicks you into less (remember, q to quit) which lets you browse up and down through all the information about ls
- you should look at the man pages for grep, ls, etc.
- they are always correct for your computer: man ls on a Mac gives different information than on a Linux box

Getting Help

- to get help on an external command, use man commandname
- to get help on a built-in bash command, use man bash and then search for that command
- `$ man echo`
- `$ man bash` /echo n n n . . .

Shell Variables

- a variable is a named memory storage location for a value
- two types of variables
  - environment variables — set when the shell process is created and passed to every forked child process; some can be changed, while some are read-only
  - programmer-defined variables — local or global within the script in which they are defined

Shell Variables

- several commands let you view variables and their values
- if you know the name of a variable, you can view its value with echo (either built-in or external):

  ```
  $ echo $foobar
  ```

- to see all environment variables currently defined:

  ```
  $ printenv
  ```

- to see all current variables:

  ```
  $ set
  ```

Startup Config Files

- how do the environment variables get set in the first place?
- some are created as the bash program starts up
- others are set in init or config files
- there are several files that bash may consult upon starting, depending on how it starts

- bash may be interactive, meaning it starts with input connected to a keyboard and output connected to the screen
- a non-interactive shell is not associated with a “terminal” like when running a shell script
- an interactive shell may be a login or a non-login shell
- a login shell is invoked from a physical console (rare) or via ssh from another computer
- a shell is non-login if invoked by running a virtual terminal from a windowed login (e.g., clicking the Terminal icon after connecting via vnc)

Startup Files

- /etc/profile — if this is a login shell, read this file first
- ∼/.bash_profile — if this is a login shell, read this file second
- ∼/.bashrc — read this file if this is a non-login and interactive shell
  (Note: historically, a startup configuration file's name ends in "rc", for "Run these Commands when the program starts")
- ∼/.profile is read by sh, the predecessor to bash; since bash is backwards-compatible with sh, all login bash shells read this file if it exists

Creating Shell Variables

- you can create a shell variable with an assignment:

```
$ name=Fred
```

- if the value contains spaces, you must use quotes:

```
$ name='Fred Flintstone'
```

- single and double quotes work differently
- there can be no space before or after the assignment operator
- to see the variable you just created, use echo with the dereferencing $:

```
$ echo $name
```

Quotes

```
$ cd states
$ name=New*
$ echo $name # shell sees the metacharacter
New_Hampshire New_Jersey New_Mexico New_York

$ echo "$name" # interpolates, but protects the metacharacter
New*

$ echo ’$name’ # literal: no interpolation
$name
```

A Command

- if the value of a variable is a valid command, invoking the variable runs the command:

  ```
  $ command=date
  $ $command
  Sun 30 Aug 2020 02:59:35 PM CDT
  ```

-
  ```
  $ command=foobar
  $ command=foobar $ $command
  foobar:  command not found
  ```

Command Substitution

- often we do not want the command just to run
- rather, we wish to capture the command's output
- this is accomplished with command substitution (in a subshell)

```
$ datestring=date
$ echo $datestring
date

$ datestring=$(date)
$ echo $datestring
Sun 30 Aug 2020 03:07:34 PM CDT

$ command=date
$ datestring=$($command)
$ echo $datestring
Sun 30 Aug 2020 03:08:12 PM CDT

$ datestring="The current timestamp is $($command)"
$ echo $datestring
The current timestamp is Sun 30 Aug 2020 03:12:50 PM CDT
```

Unsetting a Variable

- if you need a variable to no longer exist, unset it
-
  ```
  $ unset datestring
  $ echo $datestring

  $
  ```
- a variable that has never been assigned, or one that has been unset, can be referred to without error
- it has the null value

Exporting

- when a variable is created in a shell, it is local to the shell
- the export command causes a variable to become an environment variable available to subsequent subshells

Without Export:

```
$ ps
    PID TTY          TIME CMD
 126563 pts/5    00:00:00 bash
 185352 pts/5    00:00:00 ps
$ datestring=$(date)
$ echo $datestring
Sun 30 Aug 2020 03:25:52 PM CDT
$ bash
$ ps
    PID TTY          TIME CMD
 126563 pts/5    00:00:00 bash
 185342 pts/5    00:00:00 bash
 185343 pts/5    00:00:00 ps
$ echo $datestring

$ exit
$ ps
    PID TTY          TIME CMD
 126563 pts/5    00:00:00 bash
 185414 pts/5    00:00:00 ps
$ echo $datestring
Sun 30 Aug 2020 03:25:52 PM CDT
```

With Export:

```
$ ps
    PID TTY          TIME CMD
 126563 pts/5    00:00:00 bash
 185352 pts/5    00:00:00 ps
$ datestring=$(date)
$ echo $datestring
Sun 30 Aug 2020 03:25:52 PM CDT
$ export datestring
$ bash
$ ps
    PID TTY          TIME CMD
 126563 pts/5    00:00:00 bash
 185342 pts/5    00:00:00 bash
 185343 pts/5    00:00:00 ps
$ echo $datestring
Sun 30 Aug 2020 03:25:52 PM CDT
$ exit
$ ps
    PID TTY          TIME CMD
 126563 pts/5    00:00:00 bash
 185414 pts/5    00:00:00 ps
$ echo $datestring
Sun 30 Aug 2020 03:25:52 PM CDT
```

Script Arguments

- a script file is often run with arguments
- we did this with filesize.sh
- the arguments are passed to the script (which means they are passed to the bash shell that is running the script)
- they are passed similar to an array, and also as individual variables
- most scripts use the individual variables

- look at arguments_demo.sh

  ```
  $ rsync -vupz
  user@sand.truman.edu:/tmp/arguments_demo.sh .
  ```

  or

  ```
  $ rsync -vupz /tmp/arguments_demo.sh .
  ```