CSI3660 – System Administration

Prof. Fredericks

Bash Scripting Pt. 1

Outline

- Intro to bash scripting
- IN-CLASS LAB

Bash Scripting

- Toot toot off to Ch 11/12 we go!
- (Chapter 10 talks about Nano / Vim / Emacs if you want more information)
- Brief overview of scripting to support lab today

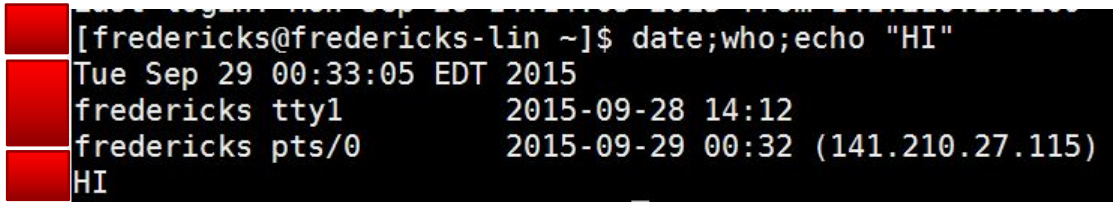
Some Details

```
[user]$ <command>

    - will represent work on the command line

----

#!/bin/bash

<commands>

    - will represent work in a bash script file
```

Command Chaining

- Run multiple commands
  - Can be used to send output from one to another…

- `[user]$ date ; who ; echo "HI"`



- That was a shell script!

Script Files

- Text file that specifies the shell and a series of commands
  - Used so you don't enter it by hand each and every time

- Structure of file:
  - Start with a shebang!
    - Or…sha-bang, hashbang, hashpling…
    - Character sequence that defines executing shell

- `#!/bin/bash`
  - *Run as program interpreted by bash*

Commenting

- # denotes a comment for a line
  - Aside from the first line

```
#!/bin/bash

# Hi everybody this is a comment how neat is that?

# List the current directory

ls

# Output the date

date
```

Creating Output

- **echo** displays simple string of text
  - `[user]$ echo "HELLO"`
    - → HELLO
  - `[user]$ echo HELLO`
    - → HELLO
  - `[user]$ echo That frank is Frank's frank`
    - → *waiting for input….*
  - `[user]$ echo That frank is Frank\'s frank`
  - `[user]$ echo "That frank is Frank's frank"`
    - → That frank is Frank's frank

Running Scripts

- 2 ways…
  - Run as script
    - `[user]$ bash <script>.sh`
  - Run as program (in current directory)
    - `[user]$ ./script.sh`
      - Executes in subshell
    - Doesn't work?!?!?!
      - No execute permission!!
      - `[user]$ chmod u+x script.sh`

Environment Variables

- Environment variables
  - **set**
    - List current variables
    - e.g., $PATH, $BASH, etc.
    - Can set environment variables in .bash_profile to take effect at startup
      - (Use **source** command if you want changes to take immediate effect) → **source** ~/.bash_profile

User Variables

- User variables
  - Temporarily store data while script is active
  - Case sensitive

```
#!/bin/bash

myvar=10

myvar2="Hello"

myvar3="ls -la"
```

Variables

- Access in script by placing $ in front of it

```
#!/bin/bash

myvar=10

echo $myvar
```

Arrays

- Store multiple values in single variable
- Indexing starts at 0 (index references element of array)

[“hello” “this” “is” “a” “sample” “array”]

```
#!/bin/bash

array1=(hello this is a sample array)

OR

array2[0]=hello

array2[1]=this

array2[2]=is

…
```

Arrays

```
array1=(hello this is a sample array)


[user]$  echo ${array1[4]}

?


Array has a length of 6 elements, but is indexed
from 0-5
```

Getting Output from Command

- Use backtick `
- Use $()

- `[user]$ thecurrentdate=`date``
- `[user]$ thecurrentdate=$(date)`

- This puts the output of the **date** command into **thecurrentdate**

- `[user]$ **echo** $thecurrentdate`

Command Substitution Scripted

- Let's say we want to make a backup of our .bashrc file, timestamped
  - .bashrc contains user preferences for our shell

```
#!/bin/bash

thedate=$(date +%y%m%d)

cp ~/.bashrc ~/.bashrc.$thedate
```

Command Input and Output

- Three file descriptors:
  - Standard input (**stdin**)
    - Refers to information processed by the command during execution
    - Often user input typed on keyboard
  - Standard output (**stdout**)
    - Refers to normal output of the command
    - Displayed on terminal
  - Standard error (**stderror**)
    - Refers to error messages generated by the command
    - Displayed on terminal

Command Input and Output

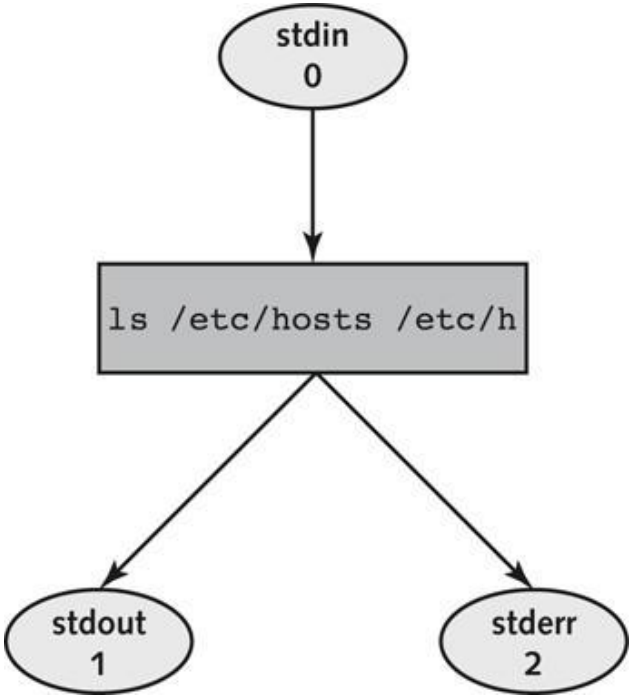


Figure 7-1: The three common file descriptors

Redirection

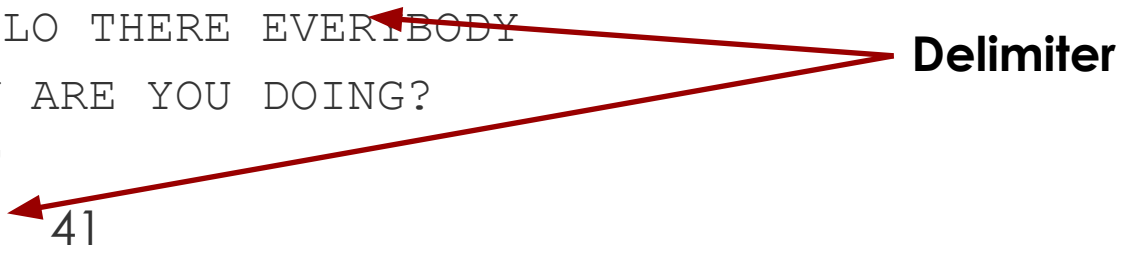
- We've talked about output redirection: **>**
  - Redirect output to a file
- and even about appending output redirection: **>>**
  - Append output to a file
- but you can also do input redirection: **<**
- and redirect from a command line: **<<**

Input Redirection

- <
  - **wc :** count the number of lines/words/bytes in a file
  - `[user]$ wc < lorem_text`
    - 9 484 3210
    - # of lines # of words # of bytes

- <<
  - `[user]$ wc << EOF`
  - `> HELLO THERE EVERYBODY`
  - `> HOW ARE YOU DOING?`
  - `> EOF`

    2 7 41



Redirecting File Descriptors

- `[user]$ ls 1> ls_output.txt`
  - Standard output redirect
  - Sends contents of 'ls'
  - Same as >
- `[user]$ sh script.sh 2> sh_output.txt`
  - Send standard error to file
  - Standard error is printed if the script has a problem
- `[user]$ sh script.sh 2>&1 sh_output.txt`
  - Send standard error to standard out to a file
  - Combine error and output in a single stream

Redirection

Command	Description
`command 1>file` `command >file`	The Standard Output of the command is sent to a file instead of to the terminal screen.
`command 2>file`	The Standard Error of the command is sent to a file instead of to the terminal screen.
`command 1>fileA 2>fileB` `command >fileA 2>fileB`	The Standard Output of the command is sent to fileA instead of to the terminal screen, and the Standard Error of the command is sent to fileB instead of to the terminal screen.

`command 1>file 2>&1`
`command >file 2>&1`
`command 1>&2 2>file com-mand >&2 2>file`
	Both the Standard Output and the Standard Error are sent to the same file instead of to the terminal screen.

`command 1>>file` `command >>file`	The Standard Output of the command is appended to a file instead of being sent to the terminal screen.
`command 2>>file`	The Standard Error of the command is appended to a file instead of being sent to the terminal screen.
`command 0<file` `command <file`	The Standard Input of a command is taken from a file.

Table 7-1: Common redirection examples

Piping

- Send output from one command to another's input

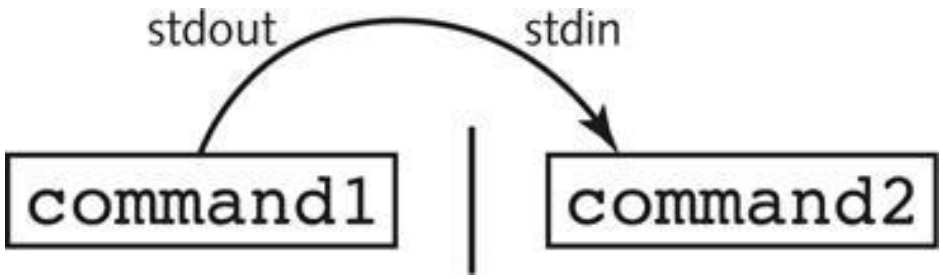


Figure 7-2: Piping information from one command to another

Pipes

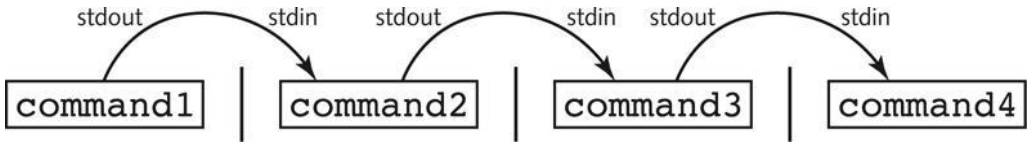


Figure 7-3: Piping several commands

- You can also combine piping with redirection!
  - `[user]$ rpm -qa | sort | tail > end_of_sorted_pkg_listing.txt`

Math / Strings

- Slightly awkward in bash…

- **expr :** recognizes math/string operations

  - `[user]$ expr 1+5`
    - Doesn't work! → need the spaces : **`expr`** `1 + 5`

  - `[user]$ expr 1 + 40 – 2 * 5`
    - Doesn't work!
  - `[user]$ expr  1 + 40 – 2 \* 5`
    - Need to escape the asterisk (wildcard operator otherwise)

Easier Math

- Use brackets…
- `[user]$ echo $[1 + 5]`
- `[user]$ echo $[1+5]`
  - Both work!
- `[user]$ myvar=$[1+5*2+3/9]`
- `[user]$ echo $myvar`
- Problem: bash only supports integer math
  - Need to use built-in calculator: bc
    - # yum install bc

Strings

- We can also do string manipulation

```
[user]$ stringvar=“HELLO”
[user]$ echo ${#stringvar}
5
```

Exit Status

- $? → status of last command executed

```
[user]$    pwd
/home/fredericks
[user]$    echo $?
0
[user]$ aj;lskedrjfpoqiwejklfoajsdfkajsdfasf
127 → Command not found
```

Table 11-2 (page 293)

Loops

- Will cover more detail on loops next lecture…
- For loop : iterate through list series of values

```
#!/bin/bash

for i in {1..5}

do

   <commands>

done
```

In-Class Lab

- Follow lab report and submit results
  - Required submission format:
    - Name each script: <lastname>_Script#.sh
      - Each script must be properly commented (see handout)
    - Name each output file: <lastname>_Script#_Output.txt
  - Create a tar.gz file that contains all script files and output files.
  - Submit to Moodle