COMP 206 – Introduction to Software Systems

Lecture 4 – More Advanced Shell Programming

Recall: the simplest shell programming style

- We have already seen. It involves simply listing several shell commands in order, perhaps with the input/output redirection and pipe operators to make them work together:

```
#!/bin/bash
ls > dir_contents.txt
grep elephant dir_contents.txt > animals.txt
grep hippo dir_contents.txt >> animals.txt
echo "I found the following animals:"
cat animals.txt
```

- Note that we are using files on the disk to hold temporary data, as if they were variables. It gets us started, but we can go much further!

Shell Variables

- The shell keeps track of a set of parameter names and values.
  - Assignment just with equals: # my_var=Hello
- Some of these are special parameters determine the behavior of the shell. Others are simply to be used to build program logic.
- We can access these variables and use them in many ways when composing shell programs:
  - Access with the dollar sign character: # echo $my_var

Shell Variables

NOTE: I will use the symbol ”#" to mean your command-prompt for the next small section. This is to avoid confusion with "$" that has a special meaning for shell variables. In a minute we will see how to swap the prompt.

- The shell keeps track of a set of parameter names and values.
  - Assignment just with equals: # my_var=Hello
- Some of these are special parameters determine the behavior of the shell. Others are simply to be used to build program logic.
- We can access these variables and use them in many ways when composing shell programs:
  - Access with the dollar sign character: # echo $my_var

Displaying Shell Variables

- Prefix the name of a shell variable with "$".
- The **`echo`** command will do:

```
# echo $HOME
# echo $PATH
```

- You can use these variables on any command line:

```
# ls -al $HOME
# var=ls
# $var -al $HOME
# options=-al
# $var $options $HOME
```

Setting Shell Variables: Details

- Variable with an assignment command is a shell *builtin* command:

```
# HOME=/etc
# PATH=/usr/bin:/usr/etc:/sbin
```

- There cannot be any spaces in the variable name, between the variable name and the equals, between the equals and the value, or within the value.

- However, it is possible to use values with spaces by enclosing them in quotes:

```
# NEWVAR="blah blah blah"
```

Special Variables for Shell Programs

- Since we can run a whole shell program at once, it can be given its own arguments and these are made available to us through:
  - $# the number of arguments given
  - $0 the shell program's name
  - $1 the first argument
  - … …
  - $9 the ninth argument
  - $@ the full argument string, $1 up to the end

`set` command (shell builtin)

- The **`set`** command with no parameters will print out a list of all the shell varibles.

- You'll probably get a pretty long list…

- Depending on your shell, you might get other stuff as well...

Shell Variables with special meaning

PWD	*current working directory*
PATH	*list of places to look for commands*
HOME	*home directory of user*
MAIL	*where your email is stored*
TERM	*what kind of terminal you have*
HISTFILE	*where your command history is saved*
PS1	*the string to be used as the command prompt*

Example $PS1

- The **PS1** shell variable is your command line prompt. It's a string.
- By changing PS1 you can change the prompt.
- E.g.
  - `# PS1="Next command: "`
  - `# PS1="# "`

Fancy **`bash`** prompts

Bash supports some fancy stuff in the prompt string:

`\t` is replace by the current time

`\w` is replaced by the current directory

`\h` is replaced by the hostname

`\u` is replaced by the username

`\n` is replaced by a newline

Example **bash** prompt

```
======= [foo.cs.rpi.edu] - 22:43:17 =======
/cs/hollingd/introunix echo $PS1
======= [\h] - \t =======\n\w
```

Capturing command output in a variable

- We saw that “>” stored the command’s output in a file
- Sometimes we want to skip the filesystem (efficiency of memory vs disk) and re-use the output within our BASH program
- The back-tick operator allows this:
  - Format: # variable=`command`
  - Anything that # command alone would output to the terminal is now stored as the value of variable. Access it with $variable. (of course this name is just an example, we can pick any other, e.g. # daves_var=`comand`)
  - A nearly equivalent syntax is # variable=$(command)

Example Backup Program:

```
1  #!/bin/bash
2
3  # This bash script is used to backup a user's home directory to /tmp/.
4
5  user=$(whoami)
6  input=/home/$user
7  output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz
8
9  tar -czf $output $input 2> /dev/null
10 echo "Backup of $input completed! Details about the output backup file:"
11 ls -l $output
```

Math

- The shell can do simple arithmetic.
- Enclose your computation in $((computation))
- You can use this in quite flexible ways:
  - # a=$((3+5))
  - # echo There are $((60*60)) seconds in an hour

Expanding to larger programs: Control Structures

- Each of the familiar looping constructs are available in the shell, but they often have to be used with a slightly different syntax, and are tailored for use in the usual jobs of shell programming:
  - Looping through files, interacting with other commands, starting and stopping programs

- There are many different formats and syntax for each, and you may prefer a different one that we show in lectures. But, the versions shown here will be the only ones to appear on tests, and are enough to let you do everything.

Shell Conditionals: if

```
if    program
then
commands
fi
```

Execute the body if the program returns an exit code of zero (success).

Program exit codes

- Every process returns an integer value when it terminates. This is part of the Linux process specification.

- "if" checks this so we need to know how common programs decide what to return:
  - For example, "ls" returns 0 when the file you ask it to list was present, otherwise it returns false. E.g., > ls elephant
  - Check the return code from a program by accessing the special shell variable "$?", which is set for each and every command in a shell program

Full if structure

- More generally we can have:

```
if pgm
then
   commands
elif  otherpgm
   commands
else
   commands
fi
```

Example using if

- How do we check if today is a Monday?
  - If it’s a Monday, print “another week starts”.
  - The "date" command prints the current date on standard output
  - The "grep" command filters its input for a specified pattern

Option 1, avoid if, use file-based programming

```
date > junkfile
grep Mon junkfile
```

- This gets us started, it will print the date only when it contains a Monday
  - But… it doesn’t yet do what was specified!

Option 2: almost correct

```
date > junkfile
if grep Mon junkfile
then
   echo Another week starts.
fi
```

- But this printed an extra line:

  Mon Jan 24 13:53:28 EST 2022
  Another week starts.

Option 2: almost correct

```
date > junkfile
if grep Mon junkfile
then
   echo Another week starts.
fi
```

Here we use the return code of grep:
0 (true) if some pattern found, 1 (false) otherwise

- But this printed an extra line:

  Mon Jan 24 13:53:28 EST 2022
  Another week starts.

Still ugly, but finally correct

```
date > junkfile
if grep Mon junkfile > junkfile2
then
    echo Another week starts.
fi
```

- It prints:

  ```
  Another week starts.
  ```

  on Mondays only.

  However, it leaves 2 files behind

  Worse yet, it clobbers any prior files called junkfile or junkfile2

Refinements: making it elegant

- Instead if using a junk file, use /dev/null, which is a special "file" specifically for the purpose of deleting whatever is put into it (the black hole of the file system!)

```
date > junkfile
if grep Mon junkfile > /dev/null
then
    echo Another week starts.
fi
```

Why use a file at all?

- Remember the "back-tick" character, `, which captures the standard output of a command. E.g:

  x=`date`

- When you think about such a line, imagine crossing out `date` and replacing it with date's output, surrounded in double quotes:

  x="Tue Mar 13 14:02:59 EDT 2018"

Why use a file at all?

- Put the temporary result in a variable:

```
x=`date`
```

- Now, how to get the variable's contents to be considered by grep:

```
if echo $x | grep Mon > /dev/null
then
    echo Another week starts.
fi
```

We can even skip the variable

```
if date | grep Mon > /dev/null
then
    echo Another week starts.
fi
```

What about general conditions?

- In the previous example, we relied here on the fact that grep produced a nice output code that worked well with if, but this was a "bonus" on top of its feature as a text filter. What if we weren't so lucky?

- The built-in "test" program provides a more complete set of logic operations, and is made for the special purpose of working with shell conditional statements

```
if b = a
then
   echo equals
fi
```

test: examples

- Syntax: test flags(s) arguments
  - test -r file
    - is the file readable?
  - test -w file
  - test arg1 = arg 2
    - are the strings identical?
    - test arg1 != arg2: are the NOT equal?
  - -gt, -le, -eq etc: numerical tests: greater than, less than or equal to, equal, ...

Test syntax

- Test is so connected to shell conditionals that it has a special syntax:

```
if [[ b = a ]]
then
    echo hi
fi
```

- Note the spaces between each of the [[, each argument, and the]]
- Similar syntax for test patterns with one argument:

```
if [[ -r my_file.txt ]]
then
    echo I can do something with the file!
fi
```

Shell conditional summary

- Shell conditionals are slightly different than other programming languages. They rely on a program to run and give an output code that can be evaluated.
  - Contrast with C, the if statement looks directly at built-in variables with equality, greater than, less than operators all a core part of the language
- They are much used and give us our first building block to make larger programs and deploy our code for software systems
- We will continue on next time to consider loops and other programming constructs

While

Our first core looping construct. Similar idea to if:

```
while program    < Check return code of program
 do
    list_of_commands    < Execute the whole list each time
done
```

First try at while

- Does this work based on what we know now?

```
x=1
while $x > 10
do
  print $x
done
```

While

while pgm < **Boolean test based on return code of program**

do list

done

eg.

```
x=1
while $x > 
 do
 print $x
 done
```

WRONG this must be:

```
x=1
while test $x -lt 10
      do
      echo $x
       x=`expr $x + 1`
      done
```

For

```
for variable in wordlist
do
        stuff
done
```

The variable takes on the values of the items
from the worldlist as the iterations proceed

Example: for

```
for i in this is a test
do
    echo $i
done
```

Prints:

```
this
is
a
test
```

Should we delete all the files? (Example)

- ```
  for fn in *
    do
    echo Should I delete file $fn
    read ok
    if test $ok != “no”

        then
        echo deleting $fn
        sleep 2
        rm $fn
        fi
    done
  ```
  ```

Should we delete all the files? (Example)

```
for fn in *
  do
  echo Should I delete file $fn
  read ok
  if test $ok
       then
       echo del
       sleep 2
       rm $fn
       fi
  done
```

WRONG!

```
== "yes"    <- NEVER DO THE
               UNRECOVERABLE
               THING BY DEFAULT.
```