LINUX SHELL SCRIPTING

Scripting Languages Versus Compiled Languages

- Most medium and large-scale programs are written in a compiled language, such as Fortran, Ada, Pascal, C, C++, or Java.
- The benefit of compiled languages is that they're efficient.
- Their disadvantage is that they usually work at a low level, dealing with bytes, integers, floating-point numbers, and other machine-level kinds of objects.
- For example, it's difficult in C++ to say something simple like "copy all the files in this directory to that directory over there."

- So-called scripting languages are usually interpreted.
- The advantage to scripting languages is that they often work at a higher level than compiled languages, being able to deal more easily with objects such as files and directories.
- Examples of scripting languages include awk, Perl, Python, Ruby, and the shell.

reasons to use a shell script

- **Simplicity**
  - The shell is a high-level language; you can express complex operations clearly and simply using it.
- **Portability**
  - By using just POSIX-specified features, you have a good chance of being able to move your script, unchanged, to different kinds of systems.
- **Ease of development**
  - You can often write a powerful, useful script in little time.

A Simple Script

- Let's start with a simple script. Suppose that you'd like to know how many users are currently logged in. The who command tells you who is logged in:

```
$ who
george   pts/2    Dec 31 16:39   (valley-forge.example.com)
betsy    pts/3    Dec 27 11:07   (flags-r-us.example.com)
```

  - On a large multiuser system, the listing can scroll off the screen before you can count all the users, and doing that every time is painful anyway.
  - This is a perfect opportunity for automation.

- What's missing is a way to count the number of users. For that, we use the wc (word count) program.
  - `$ who | wc -l` *Count users*
- The next step is to make this pipeline into a separate command. You do this by entering the commands into a regular file, and then making the file executable, with chmod, like so:

```
$ cat > nusers                  Create the file, copy terminal input with cat
who | wc -l                     Program text
^D                              Ctrl-D is end-of-file
$ chmod +x nusers               Make it executable
$ ./nusers                      Do a test run
      6                         Output is what we expect
```

USING VARIABLES

Environment variables

- You have already seen how we can view and set the environment variables.
- You can also use them in your scripts.

An Example

```
$ cat test2
#!/bin/bash
# display user information from the system.
echo "User info for userid: $USER"
echo UID: $UID
echo HOME: $HOME
$
```


A Drawback

- Look at this Example:
- $ echo "The cost of the item is $15“
- The cost of the item is 5
- The correct technique:
- $ echo "The cost of the item is \ $15"
- The cost of the item is $15

User variables

- User variables can be any text string of up to 20 letters, digits, or an underscore character. User variables are case sensitive.
- Here are a few examples of assigning values to user variables:
  - var1=10
  - var2=-57
  - var3=testing
  - var4="still more testing"

Example

```
$ cat test3
#!/bin/bash
# testing variables
days=10
guest="Katie"
echo "$guest checked in $days days ago"
days=5
guest="Jessica"
echo "$guest checked in $days days ago"
$
```

Using backtick

- The backtick allows you to assign the output of a shell command to a variable.

```
$ cat test5
#!/bin/bash
# using the backtick character
testing=`date`
echo "The date and time are: " $testing
$
```


Performing Maths

- The expr command
- Example:
  - $ expr 5 * 2
  - 10
  - $

Operator	Description
ARG1 | *ARG2*	Return *ARG1* if neither argument is null or zero; otherwise, return *ARG2*.
ARG1 & *ARG2*	Return *ARG1* if neither argument is null or zero; otherwise, return 0.
ARG1 < *ARG2*	Return 1 if *ARG1* is less than *ARG2*; otherwise, return 0.
ARG1 <= *ARG2*	Return 1 if *ARG1* is less than or equal to *ARG2*; otherwise, return 0.
ARG1 = *ARG2*	Return 1 if *ARG1* is equal to *ARG2*; otherwise, return 0.
ARG1 != *ARG2*	Return 1 if *ARG1* is not equal to *ARG2*; otherwise, return 0.
ARG1 >= *ARG2*	Return 1 if *ARG1* is greater than or equal to *ARG2*; otherwise, return 0.
ARG1 > *ARG2*	Return 1 if *ARG1* is greater than *ARG2*; otherwise, return 0.
ARG1 + *ARG2*	Return the arithmetic sum of *ARG1* and *ARG2*.
ARG1 - *ARG2*	Return the arithmetic difference of *ARG1* and *ARG2*.
ARG1 * *ARG2*	Return the arithmetic product of *ARG1* and *ARG2*.
ARG1 / *ARG2*	Return the arithmetic quotient of *ARG1* divided by *ARG2*.
ARG1 % *ARG2*	Return the arithmetic remainder of *ARG1* divided by *ARG2*.

Example

- #!/bin/bash
- # An example of using the expr command
- var1=10
- var2=20
- var3=`expr $var2 / $var1`
- echo The result is $var3

- To assign the result of a mathematical equation to a variable, you have to use the backtick character to extract the output from the expr command

A much easier way

- Using brackets

```
$ var1=$[1 + 5]
$ echo $var1
6
$ var2 = $[$var1 * 2]
$ echo $var2
12
$
```

An Example Script

```
#!/bin/bash
var1=100
var2=50
var3=45
var4=$[$var1 * ($var2 - $var3)]
echo The final result is $var4
$
```

Commenting and Documenting Scripts

- In Unix shells, comment lines begin with a hashmark character (#). Anything that comes after the hashmark is taken to be a comment and is not interpreted as a command.

```
# Here is a comment that
# spans multiple lines.
```

- In addition, a hashmark can be inserted into the middle of a line, and everything to the right of it will be considered a comment:
  - some_command # This is a comment explaining the command

Invoking the Shell

- Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the shebang construct. For example,
  - #!/bin/bash
- tells the system that the commands that follow are to be executed by the bash shell.

Example

- To create a script containing these commands, you put the shebang line first and then add the commands:

```
#!/bin/bash
pwd
ls
```


Reading Input from the Keyboard

- In addition to assigning a variable within a script, you can assign a variable by reading its value from the keyboard.
- For example, you might write the simple script:

```
echo “What is your name?”
read PERSON
echo “Hello, $PERSON”
```

Performing Floating Calculations

- Floating point calculations can be performed using the bc command.
- Example:
  - echo "4 * 0.56" | bc
  - 2.24
- Specifying decimal precision (scale): In the following example the scale=2 parameter sets the number of decimal places to 2. Hence the output of bc will contain a number with two decimal places:
  - echo "scale=2;3/8" | bc
  - 0.37

Special Variables

Variable	Function
?	The previous command's exit status.
$	The PID of the current shell process.
-	Options invoked at start-up of the current shell.
!	The PID of the last command that was run in the background.
0	The filename of the current script.
1-9	The first through ninth command-line arguments given when the current script was invoked: $1 is the value of the first command-line argument, $2 the value of the second, and so forth.
_	The last argument given to the most recently invoked command before this one.

$# Represents the Number of Command Line Arguments

Exit Status

- The $?variable represents the exit status of the previous command.
- Exit status is a numerical value returned by every command upon its completion.
- As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

Example

```
$ date
Sat Sep 29 10:01:30 EDT 2007
$ echo $?
0
$
```

Example

- An exit status value of 126 indicates that the user didn't have the proper permissions set to exe-cute the command:

```
$ ./myprog.c
-bash: ./myprog.c: Permission denied
$ echo $?
126
$
```


Exit Codes

Code	Description
0	Successful completion of the command
1	General unknown error
2	Misuse of shell command
126	The command can't execute
127	Command not found
128	Invalid exit argument
128+x	Fatal error with Linux signal x
130	Command terminated with Ctl-C
255	Exit status out of range

The exit command

- You can change that to return your own exit status code. The exit command allows you to specify an exit status when your script ends:

```
#!/bin/bash
# testing the exit status
var1=10
var2=30
var3=$[ $var1 + var2 ]
echo The answer is $var3
exit 5
$
```

Variables can also be used with exit

```
#!/bin/bash
# testing the exit status
var1=10
var2=30
var3=$[ $var1 + var2 ]
exit $var3
$
```

Limitations

```
#!/bin/bash
# testing the exit status
var1=10
var2=30
var3=$[ $var1 * var2 ]
echo The value is $var3
exit $var3
$
```

exit codes can go up to 255.

If the code exceeds this number, the result will be the code modulo 256

Basic I/O Redirection

- The default for standard input, standard output, and standard error is the terminal. This can be seen with cat:

```
$ cat
now is the time
now is the time
for all good men
```

With no arguments, read standard input, write standard output
Typed by the user
Echoed back by cat

Redirection and pipelines

- Change standard input with <
  - Use program < file to make program's standard input be file:
    - tr -d '\r' < dos-file.txt …
- Change standard output with >
  - Use program > file to make program's standard output be file:
    - tr -d '\r' < dos-file.txt > unix-file.txt
- The > redirector creates the destination file if it doesn't exist. However, if the file does exist, then it is truncated; all existing contents are lost.

- Append to a file with >>
  - Use program >> file to send program's standard output to the end of file.
  - Like >, the >> operator creates the destination file if it doesn't exist.
  - However, if it already exists, instead of truncating the file, any new data generated by the running program is appended to the end of the file:

- Create pipelines with |
  - Use program1 | program2 to make the standard output of program1 become the standard input of program2.
- Putting everything together:
  - tr -d '\r' < dos-file.txt | sort > unix-file.txt

Basic Command Searching

- The shell searches for commands along the search path, $PATH.
- This is a colon-separated list of directories in which commands are found.
- The default path varies from system to system. It will contain at least /bin and /usr/bin.
- For example:
  - $ echo $PATH
    /bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin

- The term “bin” for directories that hold executables is short for binary.
- When you write your own scripts, it would be nice to have your own bin in which to place them, and have the shell find them automatically. For Example:

```
$ cd                              Change to home directory
$ mkdir bin                       Make a personal "bin" directory
$ mv nusers bin                   Put our script there
$ PATH=$PATH:$HOME/bin            Append our bin directory to PATH
$ nusers                          Test it out
      6                           The shell finds it
```


Simple Execution Tracing

- One way to get some idea of what your program is doing is to turn on execution tracing.
- This causes the shell to print out each command as it’s executed, preceded by “+ ”— that is, a plus sign followed by a space.

```
$ sh -x nusers                    Run with tracing on
+ who                             Traced commands
+ wc -l
       7                          Actual output
```

The basics of bc

- The bash calculator allows you to enter floating-point expressions at a command line, then interprets the expressions, calculates them, and returns the result. The bash calculator recognizes:
  - Numbers (both integer and floating point)
  - Variables (both simple variables and arrays)
  - Comments (lines starting with a pound sign or the C language /* */ pair
  - Expressions
  - Programming statements (such as if-then statements)
  - Functions

Using Scale

- The floating-point arithmetic is controlled by a built-in variable called scale.
  - $ bc -q
  - 3.44 / 5
  - 0
  - scale=4
  - 3.44 / 5
  - .6880
  - quit
  - $

Variables in bc

- Besides normal numbers, the bash calculator also understands variables:

```
$ bc -q
var1=10
var1 * 4
40
var2 = var1 / 5
print var2
2
quit
$
```


Using bc in scripts

- The basic format to use is:
  - variable= echo "options; expression" | bc
    - The first portion, options, allows us to set variables.
      - If you need to set more than one variable,separate them using the semicolon.
      - The expression parameter defines the mathematical expres-sion to evaluate using bc.

Example

```
#!/bin/bash
var1=`echo " scale=4; 3.44 / 5" | bc`
echo The answer is $var1
$
```


Example

```
#!/bin/bash
var1=100
var2=45
var3=`echo "scale=4; $var1 / $var2" | bc`
echo The answer for this is $var3
$
```

- And of course, once a value is assigned to a variable, that variable can be used in yet another calculation:

```
#!/bin/bash
var1=20
var2=3.14159
var3=`echo "scale=4; $var1 * $var1" | bc`
var4=`echo "scale=4; $var3 * $var2" | bc`
echo The final result is $var4
$
```


Another Technique

```
#!/bin/bash
var1=10.46
var2=43.67
var3=33.2
var4=71
var5=`bc << EOF
scale = 4
a1 = ( $var1 * $var2)
b1 = ($var3 * $var4)
a1 + b1
EOF
`
echo The final answer for this mess is $var5
$
```

Flow Control

- Flow control can generally be broken down into two types:
  - conditional and
  - iterative.
- Both types describe how scripts set conditions and how the scripts react when the given conditions are met or not met.

CONDITIONAL FLOW CONTROL

The if-then Statement

- The heart of conditional flow control is the if-then statement. In general terms, an if-then statement looks like this:

```
if some_condition
then
    something happens
fi
```


Example

```
#!/bin/bash
# testing the if statement
if date
then
echo "it worked"
fi
```

Example

```
#!/bin/bash
echo “Guess the secret color”
read COLOR
if [ $COLOR=”purple” ]
then
echo “You are correct.”
fi
```

- You’re not limited to just one command in the then section.

```
#!/bin/bash
# testing multiple commands in the then section
testuser=ankur
if grep $testuser /etc/passwd
then
echo The bash files for user $testuser are:
ls -a /home/$testuser/.b*
fi
```

The if-then-else Statement

- The if-then-else statement provides another group of commands in the statement:

```
if command
then
commands
else
commands
fi
```


Adding an else

```
#!/bin/bash
echo “Guess the secret color”
read COLOR
if [ $COLOR=”purple” ]
then
echo “You are correct.”
else
echo “Your guess was incorrect.”
fi
```

```
#!/bin/bash
# testing the else section
testuser=badtest
if grep $testuser /etc/passwd
then
echo The files for user $testuser are:
ls -a /home/$testuser/.b*
else
echo "The user name $testuser doesn't exist on this
 system"
fi
```


Nesting ifs

- The elif continues an else section with another if-then statement:

```
if command1
then
commands
elif command2
then
more commands
fi
```

Using elif to specify another condition

```
#!/bin/bash
echo “Guess the secret color”
read COLOR
if [ $COLOR=”purple” ]
then
echo “You are correct.”
elif [ $COLOR=”blue” ]
echo “You’re close.”
else
echo “Your guess was incorrect.”
fi
```


The test Command

- The test command is used to evaluate conditions.
- The square brackets are syntactically equal to the test command.
- For example, the preceding script could have been written:
  - if (test $COLOR=”purple”)

Syntax

```
if test condition
then
commands
fi
```

- This concept is important because the test command has a number of options that can be used to evaluate all sorts of conditions, not just simple equality.
- For example, use test to see whether a particular file exists:
  - if (test -e filename)
  - In this case, the test command would return a value of true, or 0, if the file exists, and false, or 1, if it
  - doesn't.

Three Classes

- There are three classes of conditions the test command can evaluate:
  - Numeric comparisons
  - String comparisons
  - File comparisons

Numeric comparisons

```
#!/bin/bash
# using numeric test comparisons
val1=10
val2=11
if [ $val1 -gt 5 ]
then
echo "The test value $val1 is greater than 5"
fi
if [ $val1 -eq $val2 ]
then
echo "The values are equal"
else
echo "The values are different"
fi
```

The test Numeric Comparisons

Comparison	Description
n1 -eq *n2*	Check if *n1* is equal to *n2*.
n1 -ge *n2*	Check if *n1* is greater than or equal to *n2*.
n1 -gt *n2*	Check if *n1* is greater than *n2*.
n1 -le *n2*	Check if *n1* is less than or equal to *n2*.
n1 -lt *n2*	Check if *n1* is less than *n2*.
n1 -ne *n2*	Check if *n1* is not equal to *n2*.

Comparison Operators

Operator	Example	Test Condition
=	string a = string b	Text string a is the same as text string b.
!=	string a != string b	Text string a is not the same as text string b.
>	string a > string b	Text string a is greater than text string b.
<	string a < string b	Text string a is lesser than text string b.

if [$COLOR != “purple”]

String comparisons

The test Command String Comparisons

Comparison	Description
str1 = *str2*	Check if *str1* is the same as string *str2*.
str1 != *str2*	Check if *str1* is not the same as *str2*.
str1 < *str2*	Check if *str1* is less than *str2*.
str1 > *str2*	Check if *str1* is greater than *str2*.
-n *str1*	Check if *str1* has a length greater than zero.
-z *str1*	Check if *str1* has a length of zero.

Example

```
$ cat test8
#!/bin/bash
# testing string equality
testuser=baduser
if [ $USER != $testuser ]
then
echo "This isn't $testuser"
else
echo "Welcome $testuser"
fi
```

String order

- Trying to determine if one string is less than or greater than another is where things start getting tricky:
  - The greater-than and less-than symbols must be escaped, or the shell will use them as redirection symbols, with the string values as filenames

This will not run

```
#!/bin/bash
# mis-using string comparisons
val1=baseball
val2=hockey
if [ $val1 > $val2 ]
then
echo "$val1 is greater than $val2"
else
echo "$val1 is less than $val2"
fi
```

Correct

```
#!/bin/bash
# mis-using string comparisons
val1=baseball
val2=hockey
if [ $val1 \> $val2 ]
then
echo "$val1 is greater than $val2"
else
echo "$val1 is less than $val2"
fi
```


String size

```
$ cat test10
#!/bin/bash
# testing string length
val1=testing
val2=''
if [ -n $val1 ]
then
echo "The string '$val1' is not empty"
else
echo "The string '$val1' is empty"
fi
if [ -z $val2 ]
then
echo "The string '$val2' is empty"
else
echo "The string '$val2' is not empty"
fi
```

test command File Comparision

Comparison	Description
`-d file`	Check if *file* exists and is a directory.
`-e file`	Checks if *file* exists.
`-f file`	Checks if *file* exists and is a file.
`-r file`	Checks if *file* exists and is readable.
`-s file`	Checks if *file* exists and is not empty.
`-w file`	Checks if *file* exists and is writable.
`-x file`	Checks if *file* exists and is executable.
`-O file`	Checks if *file* exists and is owned by the current user.
`-G file`	Checks if *file* exists and the default group is the same as the current user.
file1 `-nt` *file2*	Checks if *file1* is newer than *file2*.
file1 `-ot` *file2*	Checks if *file1* is older than *file2*.

File comparisons

- Checking directories
- The -d test checks if a specified filename exists as a directory on the system

```
#!/bin/bash
# look before you leap
if [ -d $HOME ]
then
echo "Your HOME directory exists"
cd $HOME
ls -a
else
echo "There's a problem with your HOME directory"
fi
```

Checking if an object exists

- The -e comparison allows you to check if a file or directory object exists before you attempt to use it in your script:

```
if [ -e $HOME ]
then
echo "OK on the directory, now let's check the file"
# checking if a file exists
if [ -e $HOME/testing ]
then
# the file exists, append data to it
echo "Appending date to existing file"
date >> $HOME/testing
```


Checking file date

```
#!/bin/bash
# testing file dates
if [ ./test19 -nt ./test18 ]
then
echo "The test19 file is newer than test18"
else
echo "The test18 file is newer than test19"
fi
if [ ./test17 -ot ./test19 ]
then
echo "The test17 file is older than the test19 file"
fi
```

Can you read it

```
#!/bin/bash
# testing if you can read a file
pwfile=/etc/shadow
# first, test if the file exists, and is a file
if [ -f $pwfile ]
then
# now test if you can read it
if [ -r $pwfile ]
then
tail $pwfile
else
echo "Sorry, I'm unable to read the $pwfile file"
fi
else
echo "Sorry, the file $file doesn't exist"
fi
```


Checking ownership

```
#!/bin/bash
# check file ownsership
if [ -O /etc/passwd ]
then
echo "You're the owner of the /etc/passwd file"
else
echo "Sorry, you're not the owner of the
 /etc/passwd file"
fi
```

Multiple Conditions

```
if [ condition1 ]
then
    if [ condition2 ]
    then
    some action
    fi
fi
```


- Or you could simplify it by using a logical and operator (&&):

```
if [ condition1 && condition2 ]
    then
        some action
fi
```

- There is also a logical oroperator (||):

```
if [ condition1 || condition2 ]
    then
    some action
fi
```

Example

```
#!/bin/bash
# testing compound comparisons
if [ -d $HOME ] && [ -w $HOME/testing ]
then
echo "The file exists and you can write to it"
else
echo "I can't write to the file"
fi
```


Advanced if-then Features

- There are two relatively recent additions to the bash shell that provide advanced features that you can use in if-then statements:
  - Double parentheses for mathematical expressions
  - Double square brackets for advanced string handling functions

Using double parentheses

- The double parentheses command allows you to incorporate advanced mathematical formulas in your comparisons.
- The format of the double parentheses command is:
  - ((expression))

Example

```
#!/bin/bash
# using double parenthesis
val1=10
if (( $val1 ** 2 > 90 ))
then
(( val2 = $val1 ** 2 ))
echo "The square of $val1 is $val2"
fi
```

The Double Parentheses Command Symbols

Symbol	Description
val++	post-increment
val--	post-decrement
++*val*	pre-increment
--*val*	pre-decrement
!	logical negation
~	bitwise negation
**	exponentiation
<<	left bitwise shift
>>	right bitwise shift
&	bitwise Boolean AND
|	bitwise Boolean OR
&&	logical AND
||	logical OR

Using double brackets

- The double bracket command provides advanced features for string comparisons. The double bracket command format is:
  - [[expression]]

Example

```
#!/bin/bash
# using pattern matching
if [[ $USER == r* ]]
then
echo "Hello $USER"
else
echo "Sorry, I don't know you"
fi
```


The case Statement

```
case expression in
    pattern1)
        action1
    ;;
    pattern2)
        action2
    ;;
    pattern3)
        action3
    ;;
esac
```

Example

```
#!/bin/bash
# using the case command
case $USER in
ankur | root)
echo "Welcome, $USER"
echo "Please enjoy your visit";;
testing)
echo "Special testing account";;
jessica)
echo "Don't forget to log off when you're done";;
*)
echo "Sorry, you're not allowed here";;
esac
```


Example

```
case `date +%a` in
  "Mon")
      BACKUP=/home/myproject/data0
      TAPE=/dev/rft0
# Note the use of the double semi-colon to end each option
      ;;
# Note the use of the "|" to mean "or"
  "Tue" | "Thu")
      BACKUP=/home/myproject/data1
      TAPE=/dev/rft1
      ;;
  "Wed" | "Fri")
      BACKUP=/home/myproject/data2
      TAPE=/dev/rft2
      ;;
# Don't do backups on the weekend.
  *)
BACKUP="none"
      TAPE=/dev/null
      ;;
esac
```

Some Examples

- Identifying the current shell:
- Display the currently used shell as follows:
  - echo $SHELL
  - Or, you can also use:
  - echo $0

Check for super user

- UID is an important environment variable that can be used to check whether the current script has been run as root user or regular user. For example:
  - if [$UID -ne 0]; then
  - echo Non root user. Please run as root.
  - else
  - echo "Root user"
  - fi
- The UID for the root user is 0.