INFO 101 – Introduction to Computing and Security / Tutorial – week 7

Prof. Dr. Rui Abreu and Prof. Dr. Franz Wotawa
University of Porto, Portugal and Technical University of Graz, Austria
rui@computer.org, wotawa@ist.tugraz.at

November 26, 2020

1 Introduction

This time the tutorial has the objective to familizarize the students further with Shell scripting. This is meant as a revision, as students are supposed to know it well by now! Complement the tutorial with the slides from the lecture on Monday.

2 Shell Scripts

In this course, we need to be familiar with the "UNIX shell". We use it, whether bash, csh, tcsh, zsh, or other variants, to start and stop processes, control the terminal, and to otherwise interact with the system. Many of you have heard of, or made use of "shell scripting", that is the process of providing instructions to shell in a simple, interpreted programming language. To see what shell we are working on, type

```
echo $SHELL ---- to see the working shell in SSH
```

3 Simple Shell Scripts

The simplest scripts of all are nothing more than lists of commands. Consider the script below. You can type this into a file called: `first.sh`

```
#!/bin/sh --This line should always be the first line in your script
# A simple script
who am i
date
pwd
```

In general, anything after a # is a comment and, as such, is ignored by the shell. We see this used both as an entire line and next to each of several lines, where it shows example output.

The first line, that starts with #! (reads as "shebang"[1]) has a special meaning, however. The "#!/bin/sh" tells the shell to invoke `/bin/sh` to run the script. This is necessary because different users might be using different shells: sh, csh, bash, zcsh, tcsh, &c. And these shells have slightly different languages and build-in features. In order to ensure consistent operation, we want to make sure that the same shell is used to run the script each time. This is achieved by starting the specified shell and passing the script into its standard in.

To execute a shell script in csh, we simply type at the command prompt

[1] https://en.wikipedia.org/wiki/Shebang_(Unix)

```
% sh first.sh
```

or

```
% chmod u+x first.sh
% ./first.sh
```

This should now show the execution of the series of commands written above.

Note: The various shells are more the same than different. As a result, on many systems, there is actually one shell program capable of behaving with different personalities. On these systems, the personality is often selected by soft linking different names to the same shell binary. Then, the shell looks at `argv[0]` to observe how it was invoked, sets some flags to enable/disable behaviors, and goes from there.

The bulk of this simple script is a list of commands. These commands are executed, in turn, and the output is displayed. The commands are found by searching the standard search path `PATH`. `PATH` is a `:` delimited list of directories which should be searched for executables. You can find your path by typing:

```
echo $PATH
```

The command `which`, used as in `which ls`, will tell you which version of a command is being executed. This is useful if different versions might be in your search path. In general, the search path is traversed from left to right.

Note: Notice that "'.", the current working directory, is the last directory listed.

4 Shell Variables

`PATH` discussed above is one example of a variable. It is what is known as an environment variable. It reflects one aspect of the shell environment – where to look for executables. Changing it changes the environment in which the shell executes programs. Environment variables are special in that they are defined before the shell begins.

Environment variables, like most other variables, can be redefined simply by assigning them a new value:

```
echo $PATH
PATH=$PATH:/usr/local/apache/bin:.
echo $PATH
```

To create new variables, you simply assign them a value:

```
value ="dir"
echo $value
```

All shell script variables are untyped (well, they really are strings) – how they are interpreted often depends on what program is using them or what operator is manipulating or examining them. The shell scripts uses File system, IO, and file redirections commands among many others. A list of these commands are given below.

File system commands

- mkdir — Creates a directory
- rmdir — Deletes a directory
- ls — Lists contents of given path
- cat — Read from given file and output to STDOUT or given path
- find — Search for a given file (find <path> -name <filename>)
- chmod — Change mode/permissions
- cp — Copy files (cp sourcefile destfile)
- mv — Move/rename files (mv oldname newname)
- scp — Secure copy (Remote file copy) (scp <filename> <host>:<path>)

IO Commands

- echo — To print to stdout
- read — To obtain values from stdin

IO Redirection

- > — Output to given file
- < — Read input from given file
- << — Append output to given file

5 Command Line Arguments

Command line arguments are important part of writing scripts. Command line arguments define the expected input into a shell script. For example, we may want to pass a file name or folder name or some other type of argument to a shell script.

Several special variables exist to help manage command-line arguments to a script:

- `$#` — represents the total number of arguments (much like argv), except command
- `$0` — represents the name of the script, as invoked
- `$1, $2, $3, .., $8, $9` — The first 9 command line arguments
- `$*` — all command line arguments
- `$@` — all command line arguments

Exercise 1: Write a simple shell script **myscript.sh** that takes a path of a directory as a command line argument and list all files and folders inside the given directory. Run the script as:
`sh myscript.sh /afs/lzu/course/info/101`

If there are more than 9 command-line arguments, there is a bit of a problem — there are only 9 positionals: `$1, $2, ..., $9`. `$0` is special and is the shell script's name. To address this problem, the shift command can be used. It shifts all of the arguments to the left, throwing away `$1`. What would otherwise have been `$10` becomes `$9` — and addressable. We will talk more about shift after we have talked about while loops.

6 Quotes, Quotes, and More Quotes

Shell scripting has three different styles of quoting — each with a different meaning:

- unquoted strings are normally interpreted
- "quoted strings are basically literals — but $variables are evaluated"
- 'quoted strings are absolutely literally interpreted'
- `commands in quotes like this are executed, their output is then inserted as if it were assigned to a variable and then that variable was evaluated`

"quotes" and 'quotes' are pretty straight-forward — and will be constantly reinforced. But, Here is an example using 'quotes' — commands in quotes

```
day=`date | cut -d" " -f1`
printf "Today is %s.\n" $day
```

The first expression finds the current date and uses cut (string tokenizer) to extract a specific part of the date. Then it assigns that to variable day. The day then is used by printf statement. You can read more about cut command later in this lecture. Cut comes handy in many shell scripts as it allows us to look at a specific token of a string.

7 Evaluating Expr

Shell scripts are not intended to do complex mathematical expressions. But expr program can be used to manipulate variables, normally interpreted as strings, as integers. Consider the following "adder" script:

```
sum=`expr $1 + $2`
printf "%s + %s = %s\n" $1 $2 $sum
```

8 A Few Other Special Variables

We'll talk a bit more about these as we get into more complex examples. For now, I'd just like to mention them:

- $? — the exit status of the last program to exit
- $$ — The shell's pid

9 Predicates

The convention among UNIX programmers is that programs should return a 0 upon success. Typically a non-0 value indicates that the program couldn't do what was requested. Some (but not all) programmers return a negative number upon an error, such as file not found, and a positive number upon some other terminal condition, such as the user choosing to abort the request.

As a result, the shell notion of true and false is a bit backward from what most of us might expect. 0 is considered to be true and non-0 is considered to be false. We can use the test to evaluate an expression. The following example will print 0 if guna is the user and 1 otherwise. It illustrates not only the test but also the use of the status variable. status is automatically set to the exit value of the most recently exited program. The notation $var, such as $test, evaluates the variable.

```
test "$LOGNAME" = guna echo $?
```

Shell scripting languages are typeless. By default everything is interpreted as a string. So, when using variables, we need to specify how we want them to be interpreted. So, the operators we use vary with how we want the data interpreted.

10 Operators for strings, ints and files

Operators for strings, ints, and files						
string	x = y, comparison: equal	x != y, comparison: not equal	x, not null/not 0 length	-n x, is null		
ints	x -eq y, equal	x -ge y, greater or equal	x -le y, lesser or equal	x -gt y, strictly greater	x -lt y, strictly lesser	x -ne y, not equal
File	-f x, is a regular file	-d x, is a directory	-r x, is readable by this script	-w x, is writeable by this script	-x x, is executible by this script	
logical	x -a y, logical and, like && in C (0 is true, though)			x -o y, logical or, like && in C (0 is true, though)		

[Making the Common Case Convenient]

We've looked at expressions evaluated as below:

```
test -f somefile.txt
```

Although this form is the canonical technique for evaluating an expression, the shorthand, as shown below, is universally supported — and much more reasonable to read:

```
[ -f somefile.txt ]
```

You can think of the [] operator as a form of the test command. But, one very important note — there must be a space to the inside of each of the brackets. This is easy to forget or mistype. But, it is quite critical.

11 IF-THEN-ELSE

Like most programming languages, shell script supports the if statement, with or without an else. The general form is below:

```
if command
then
    command
    command
    ...
    command
else
    command
    command
    ...
    command
fi
```

The `else` branch is optional:

```
if command
then
    command
```

```
     command
     ...
     command
fi
```

The command used as the predicate can be any program or expression. The results are evaluated with a 0 return being true and a non-0 return being false.

The following is a nice, quick example of an if-else:

```
     if [ "$LOGNAME"="guna" ]
     then
       printf "%s is logged in" $LOGNAME
     else
       printf "Intruder! Intruder!"
     fi
```

The elif construct Shell scripting also has another construct that is very helpful in reducing deep nesting. It is unfamilar to those of us who come from languages like C and Perl. It is the elif, the “else if”. This probably made its way into shell scripting because it drastically reduces the nesting that would otherwise result from the many special cases that real-world situatins present — without functions to hide complexity (shell does have functions, but not parameters — and they are more frequently used by csh shell scripters than traniditonalists).

```
if command
     command
     command
     ...
     command
then
     command
     command
     ...
     command
elif command
then
     command
     command
     ...
     command
elif command
then
     command
     command
     ...
     command
fi
```

12 The switch statement

Much like C, C++, or Java, shell has a case/swithc statement. The form is as follows:

```
case var
in
```

```
pat) command
        command
        ...
        command
        ;; # Two ;;'s serve as the break
pat) command
        command
        ...
        command
        ;; # Two ;;'s serve as the break
pat) command
        command
        ...
        command
        ;; # Two ;;'s serve as the break
esac
```

Here's a quick example:

```
#!/bin/sh

case "$2"
in
"+") ans=`expr $1 + $3`
     printf "%d %s %d =
     ;;
"-") ans=`expr $1 - $3`
     printf "%d %s %d = %d\n" $1 $2 $3 $ans %d\n" $1 $2 $3 $ans
     ;;
"\*") ans=`expr "$1 * $3"`
      printf "%d %s %d = %d\n" $1 $2 $3 $ans
      ;;
"/") ans=`expr $1 / $3`
     printf "%d %s %d = %d\n" $1 $2 $3 $ans
     ;;
# Notice this: the default case is a simple *
*) printf "Don't know how to do that.\n"
   ;;
esac
```

Run the program like: `sh myscript.sh 2  3+`

13 The Foor Loops

The for loop provides a tool for processing a list of input. The input to the for loop is a list of values. Each iteration through the loop it extracts one value into a variable and then enters the body of the loop. The loop stops when the extract fails because there are no more values in the list.

Let's consider the following example which prints each of the command line arguments, one at a time. We'll extract them from ""$@" into $arg:

```
for var in "$@"
do
    printf "%s\n" $var
done
```

Much like C or Java, shell has a break command, also. As you might guess, it can be used to break out of a loop. Consider this example which stops printing command line arguments, when it gets to one whose value is “quit”:

```
for var in "$@"
do
    if [ "$var" = "quit" ]
then
  break
fi
    printf "%s\n" $var
done
```

Similarly, shell has a continue that works just like it does in C or Java.

```
for var in "$@"
do
  if [ "$var" = "me" ]
  then
    continue
  elif [ "$var" = "mine" ]
  then
    continue
  elif [ "$var" = "myself" ]
  then
continue fi
  if [ "$var" = "quit" ]
  then
break fi
  printf "%s\n" $var
done
```

14 The While and until Loops

Shell has a while loop similar to that seen in C or Java. It continues until the predicate is false. And, like the other loops within shell, break and continue can be used. Here’s an example of a simple while loop:

```
# This lists the files in a directory in alphabetical order
# It continues until the read fails because it has reached the end of input
ls | sort |
while read file
do
    echo $file
done
```

In the above code, | called a “pipe” directs output from one process to another process. For example, ls | sort takes the output from ls command, and sort the output stream received. Pipe is a form of interprocess communication which we will discuss later.

There is a similar loop, the until loop that continues until the condition is successful — in other words, while the command failes. This will pound the user for input until it gets it:

```
  printf "ANSWER ME! "
  until read $answer
  do
    printf "ANSWER ME! "
  done
```

15 Writing Shell Scripts

he decision to write a shell script is mostly based on the task on hand. Tasks that require, directory and file manipulations, reorganizing files, setting permissions, running other processes are typically good tasks for using a shell script. System administrators mostly use shell scripting for their routine tasks.

cut

`cut` is a quick and dirty utility that comes in handy across all sorts of scripting. It selects one portion of a string. The portion can be determined by some range of bytes, some range of characters, or using some delimiter-field_list pair.

The example below prints the first three characters (-c) of each line within the file:

```
cat file | cut -c1-3
```

The next example uses a :-colon as a field delimiter and prints the third and fifth fields within each line. In this respect lines are treats as records:

```
cat file | cut -d: -f3,5
```

In general, the ranges can be expressed as a single number, a comma-separated list of numbers, or a range using a hyphen (-).

16 Exercises

Do the following exercises and check the answers only after trying it out.

Exercise 2: Write a shell script sum.sh that takes an unspecified number of command line arguments (up to 9) of ints and finds their sum. Modify the code to add a number to the sum only if the number is greater than 10

```
#! /bin/sh sum=0
for var in "$@" do
 if [ $var -gt 10 ]
   then
    sum=`expr $sum + $var`
   fi
done
printf "%s\n" $sum
%sh2.1.sh245 --runthescriptas
```

Exercise 3: Write a shell script takes the name a path (eg: `/afs/lzu/course/info/101/handin`), and counts all the sub directories (recursively).

```
#! /bin/sh
ls -R $1 | wc -l

run the script as:
% sh 2.2.sh /afs/lzu/course/info/101/handin
```

Exercise 4: Write a shell script that takes a file where each line is given in the format: F08,guna,Gunawardena,Ananda,SCS,C ,G. Creates a folder for each user name (eg: guna) and set the permission to rw access for user only (hint: use fs sa).

```
#! /bin/sh
cat $1 |
while read line
do
  userid=`echo $line | cut -d’’,’’ -f2`
  mkdir $userid
  fs sa $userid $userid rw
done
```

Exercise 5: Write a shell script that takes a name of a folder as a command line argument, and produce a file that contains the names of all sub folders with size 0 (that is empty sub folders).

```
#! /bin/sh
ls $1 |
while read folder
do
  if [ -d $1/$folder ]
  then
   files=`ls $1/$folder | wc -l`
   if [ $files -eq 0 ]
     then
        echo $folder >> output.txt
fi fi
done
```

Exercise 6: Write a shell script that takes a name of a folder, and delete all sub folders of size 0.

```
#! /bin/sh
ls $1 |
while read folder
do
   files=`ls $folder | wc -l`
   if [ files -eq 0 ]
     then
        rmdir $folder
fi done
```

Exercise 7: Write a shell script that will take an input file and remove identical lines (or duplicate lines from the file)

```
#! /bin/sh
cat $1 | sort |
while read line
do
```

```
 if [ $prev!=$line ]
 then
   echo $line >> sorted.txt
 fi
 prev=$line
done
```