Alternative OS – Lecture 10

Plan:

1. Recall last lecture – basic shell script elements
2. Predefined shell variables
3. The conditional control structure **if**
4. The **while** loop
5. The **for** loop
6. Conclusions

1. Recall last lecture – basic script elements

a. Every script starts with the "shbang" line:

#!/bin/sh

b. Comments begin with **#**

This is a comment

c. We use **echo** command to give output:

echo "Hello, world!"

d. We can introduce variables at any place in the script. We do not have to declare them like in Pascal or C. All variables are of string type in **Bourne shell**. We use **$** to extract values from variables.

number="1265-6216-3341"
echo "My bank account number is *$number***"**

e. We can read the user input using **read** command.

echo " Enter your bank account:"
read *number*
echo " Your bank account number is *$number*"

2. Predefined shell variables

Before we recall the **if** control structure we got to know in the end of the last lecture, let us introduce some special variables available in every **Bourne** shell script:

$# number of arguments on the command line
$- options supplied to the shell
$? exit value of the last command executed
$$ process number of the current process
$! process number of the last command done in background
$n argument on the command line, where n is from 1 through 9, reading left to right
$0 the name of the current shell or program
$* all arguments on the command line ("$1 $2 ... $9")
$@ all arguments on the command line, each separately quoted ("$1" "$2" ... "$9")

For example:

Consider a script called *sample.sh* that is being executed with an arguments list:

% ./sample.sh *one two three four*

The values of the *shell variables* would be:

$# *4*
$-
$? *0*
$$ *11223*
$!
$2 *two*
$0 *./sample.sh*
$* *one two three four*
$@ *one two three four*

We shall use these variables in the later examples.

3. The conditional control structure if

The **if** statement is used to test if the condition is *true* (exit status is 0, success), and execute either the“ then” part or the“ else” part.

Syntax:

```
if condition
then
     commands
     ……
elif condition; then
     commands
     ……
else
     commands
     ……
fi
```

The *semicolon* is used to separate the command that tests a condition from the ***then*** *operator*.

Examples:

1)

```
#!/bin/sh
if [ "$SHELL" = "/bin/bash" ]; then
  echo "your login shell is bash (Bourne again shell)"
else
  echo "your login shell is not bash but $SHELL"
fi
```

2)

```
#!/bin/sh
if [ $# -ge 2 ]
 then
  echo $2
 elif [ $# -eq 1 ]; then
  echo $1
 else
  echo “ There are no arguments”
```

fi

We gave some of the *options* of the `test` command last time. Today we give a bigger list:

For **filenames** the options to ***test*** are given with **the syntax**:

-option *filename*

The options available for the ***test*** operator for **files** include:

-r	true if it exists and is readable
-w	true if it exists and is writable
-x	true if it exists and is executable
-f	true if it exists and is a regular file (or for csh, exists and is not a directory)
-d	true if it exists and is a directory
-e	true if it exists regardless of type
-L	true if it exists and is a symbolic link
-c	true if it exists and is a character special file (i.e. the special device is accessed one character at a time)
-b	true if it exists and is a block special file (i.e. the device is accessed in blocks of data)
-p	true if it exists and is a named pipe (fifo)
-u	true if it exists and is setuid (i.e. has the set-user-id bit set, s or S in the third bit)
-g	true if it exists and is setgid (i.e. has the set-group-id bit set, s or S in the sixth bit)
-k	true if it exists and the sticky bit is set (a t in bit 9)
-s	true if it exists and is greater than zero in size

Remark:

In addition to usual modes of the **chmod** command there are so-called *special modes:*

1000: The sticky bit is set by adding 1000 to the permissions number, like 1777=1000+777. It is applied to a directory. If set then only the file owner, the directory owner, or superuser can

delete a file in that directory. For example, if a directory has permissions 0770, then the directory owner or anyone in the directory's group can add files or delete any files (regardless of who the file's owner is). If the sticky bit is set, so the permissions are 1770, then anyone in the group can add files to the directory, but each user can only delete his or her own files.

2000: This octal permission code sets the set group ID bit. When a directory has this permission, files created within the directory have the group ID of the directory, rather than that of the default group setting for the user who created the file. Note that some operating systems don't allow you to use this numerical value, instead forcing you to use the symbolic alternative.

4000: This permission sets the set user ID bit. When a directory has this permission, files created within the directory have the user ID of the directory, rather than that of the user who created the file. Note that some operating systems don't allow you to use this numerical value, instead forcing you to use the symbolic alternative.

4. The while loop

The **if** control structure allows to choose an action depending on some condition. Similarly we can execute commands as long as the *condition* is true using **while** loops.

Syntax:

while *condition*
do
    commands
    …….
    [*break*]
    [*continue*]
done

Example:

#!/bin/sh

```
x="1"
while [ $x –le 10 ]
do
        echo "Iteration number $x"
        x=`expr $x+1`
done
```

The **break** and **continue** commands are optional. If **break** is used then the loop is broken regardless of the *condition* value. The shell executes the instruction next to the **done** command.

If **continue** is used then the next loop iteration is started right away. That is, the shell executes the instruction next to the **while** command.

Example (shiftarg.sh):

```
#!/bin/sh
while [ $# -gt 0 ]
do
        echo $1
        shift
done
```

The usage example:

%./shiftarg.sh One Two Three Four

The output:

```
One
Two
Three
Four
```

This example illustrates the use of the **shift** command. Using it the script outputs all given arguments. This command simply replaces the value of the $1 with the value of $2, the value of the $2 with the value of $3 … the value of the $8 with the value of $9, and the value of $9 is set to the empty string. In other words, this commands shifts the arguments list one position left, discarding the current value of $1.

5. The for loop

Another way of sequential iteration of the same set of instructions is the **for** command. It is designed for looping through a list of string values.

Syntax:

```
for variable [ in list_of_values ]
do
        commands
        ……..
done
```

Where the list_of_values is a list of strings separated with spaces.

Example:

```
#!/bin/sh
echo "We make a sandwich of "
for x in bread lettuce cheese
do
        echo "Lovely $x"
done
```

If some of the items in the list_of_values contain spaces we must protect them with quotes.

Example:

```
#!/bin/sh
echo "We make a sandwich of"
for x in "wholegrain bread" "lettuce" "cheddar cheese"
do
        echo "lovely $x"
done
```

The output is:

```
We make a sandwich of
Lovely wholegrain bread
```

Lovely lettuce
Lovely cheddar cheese

The **for** loops are specifically powerful when working with files, because they allow filenames *globbing* – the *wildcards!*

Every *wildcard* expression is expanded by the shell to the space separated list of filenames that match this *wildcard* expression.

For example:

% echo *.txt

The command shown above displays the list of all text files filenames found in the current working directory.

The same way we can use wildcard expressions in the **for** loops.

Example:

```
#!/bin/sh
for file in *.txt
do
        echo $file
done
```

Example (extren script):

```
#!/bin/sh
for file in *.$1
do
        newfile=`basename $file $1`
        mv $file $newfile$2
done
```

This simple script takes two arguments and replaces the extension given in the first argument with the extension given in the second argument for all files in the current working directory.

In that script we use *command substitution* by calling an external command within *backticks (backquotes)*. Particularly we call the **basename** command that way. The *command substitution* feature

allows us to take the output of the command and treat it as though it was written on the line. For example, we can set it as a value to some variable. As we did it with the *newfile* variable.

Concerning the **basename** command. It has a simple syntax:

basename *string suffix*

The output is the *string* with the *suffix* removed.

For example:

Suppose current working directory contains files *file1.txt, file2.txt, file3.txt, file4.txt, file5.txt*. Suppose also we saved the program above as *extern.sh. Then*

% ./extern.sh txt text

will replace the files ending with *"txt"* with *file1.text, file2.text, file3.text, file4.text, file5.text*.

6. Conclusions

There are predefined shell variables that we can use in our own programs. Their values contain the list of command line arguments, the number of arguments, the filename of the script, its PID and many more. Using these variables gives us more flexibility and makes writing scripts easier.

We learnt how to use conditional structure **if** and two kinds of loops: **while** and **for.** The while loop can be often used with the same **test** command as the conditional structure **if**. The **for** loop is used to iterate through a space separated list of string values. Using *wildcards* to specify the list of values in the **for** loop makes it real easy to perform file-oriented tasks.

Acknowledgements

Some of the examples were taken from *"Introduction to Unix"* by Frank G. Fiamingo, Linda DeBula and Linda Condron. Some other ideas are due to "*Shell scripts in 20 pages"* by Russel Quong.