Module 13: Shell Scripts in UNIX

A Shell, as we remarked in module-1, offers a user an interface with the OS kernel. A user obtains OS services through an OS shell. When a user logs in, he has a login shell. The login shell offers the first interface that the user interacts with either as a terminal console, or through a window interface which emulates a terminal. A user needs this command interface to use an OS, and the login shell immediately provides that (as soon as user logs in). Come to think of it, a shell is essentially a process which is a command interpreter!! In this module we shall explore ways in which a shell supports enhancement in a user's productivity.

13.1 Facilities Offered by Unix Shells

In Figure 13.1, we show how a user interacts with any Unix shell. Note that a shell distinguishes between the commands and a request to use a tool. A tool may have its own operational environment. In that case shell hands in the control to another environment. As an example, if a user wishes to use the editor tool vi, then we notice that it has its own states like edit and text mode, etc. In case we have a built-in command then it has a well-understood interpretation across all the shells. If it is a command which is particular to a specific shell, then it needs interpretation in the context of a specific shell by interpreting its semantics.

Every Unix shell offers two key facilities. In addition to an interactive command interpreter, it also offers a very useful programming environment. The shell programming environment accepts programs written in shell programming language. In later sections in this chapter we shall learn about the shell programming language and how to make use of it for enhancing productivity. A user obtains maximum gains when he learns not only to automate a sequence of commands but also to make choices within command sequences and invoke repeated executions.

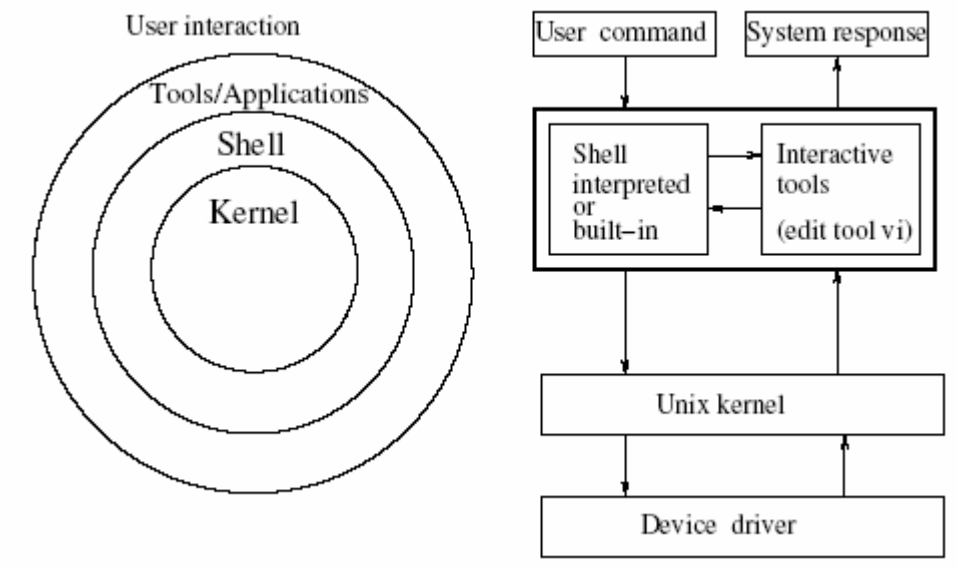

Figure 13.1: A Unix system shell.

13.1.1 The Shell Families

One of the oldest shells is the Bourne shell. In most Unix machines it is available either in its original form or as BASH which expands to Bourne Again Shell. An improvement over the Bourne shell was the Korn shell. To properly enmesh with *c* language programming environment, a new family of shells came along. The first in this family is csh, the c shell. csh has its more recent version as tchs.

The Bourne shell is the primary shell. It has all the primary capabilities. The subsequent shells provided some extensions and some conveniences. Notwithstanding the differences, all shells follow a four-step operational pattern and that is:

1. Read a command line.
2. Parse and interpret it.
3. Invoke the execution of the command line.
4. Go to step 1.

When there are no more command lines, the shell simply awaits one. As a programming environment, a shell programming language allows a user to write a program, often called *a shell script.* Once a script is presented to a shell, it goes back to its four-step operational pattern and takes commands from the script (exactly as it would have done from the terminal). So what is the advantage of the script over the terminal usage? The advantages manifest in the form of automation. One does not have to repeatedly type the

commands. One can make a batch of a set of such commands which need to be repeatedly performed and process the batch command script. We can even automate the decision steps to choose amongst alternative paths in command sequence.

In later sections, we shall study the syntax and associated semantics of shell programming language. We shall use Bourne shell as that is almost always available regardless of which of the Unix family OSs is being used.

13.1.2 Subshells

Since we have mentioned so many shells from different families, it is natural to be curious about the shell currently in use. The following Unix command tells us which is our current shell environment:

echo $SHELL

We shall examine the above command for some of the concepts associated with shell programming. The first part is the command *echo* and the second part is the argument $SHELL. The latter, in this case, is an *environmental variable.* First the command “echo". This command asks the shell environment to literally echo some thing based on the text that follows. We shall make use of this command very frequently in shell scripts either to prompt to ourselves some intermediate message, or show a value of interest.

Every OS has some environmental variables. To see the values associated with various environmental variables just give the Unix command set or env .

- set /* shows values of all the environmental variables in use */
- env /* shows values of all the environmental variables in use */

In the response, we should get the value of $SHELL as the name of shell currently in use. This should be the same as the value which our echo command prompted. Next, the second part of the echo command. The $ in $SHELL yields a value. Try giving the echo command without a $ in front of SHELL and you will see that echo promptly responds with SHELL. That then explains the role of the leading $ on variable names. One may have user defined variables, as we shall see later, whose values too can be echoed using a leading $ symbol.

In Table 13.1 we show some typical settings for Bourne shell environmental variables. One may open a new shell within an existing shell. In the new shell, one may define variables or organize a control flow for a sequence of shell commands to be executed.

The variables defined in sub-shells scope to their nested level only. The nesting shell variables may be reassigned within the nested shell for local usage.

The environmental variable	The description of this variable	Its default value
$HOME	Users Home directory	Usually from passwd file
$IFS	Internal field separator	Its white space
$LANG	Directory containing some information language dependent SW	
$MAIL	Path containing user's mailbox	It is set on login
$PATH	Colon separated list of directories	/usr/bin
$PS1	Prompt for interactive shells	
$PS2	Prompt for multiline command shells	
$SHELL	Login shell environment	/bin/sh
$TERM	Terminal type	vt100

Table 13.1: A partial list of environmental variables.

Also, one can use a particular shell of his choice. For instance, suppose we wish to use scripts that are specific to Korn shell. We could enter a Korn shell from the current shell by giving the command ksh. To check the location of a specific shell use:

which shell-name

where *which* is a Unix command and *shell-name* is the name of the shell whose location you wished to know. To use scripts we need to create a file and use it within a new shell environment.

13.2 The Shell Programming Environment

Shell files may be created as text files and used in two different ways. One way is to create a text file with the first line which looks like the following:

#! /bin/sh

The rest of the file would be the actual shell script. The first line determines which shell shall be used. Also, this text file's mode must be changed to executable by using +x in *chmod command.*

Another way is to create a shell file and use a -f option with the shell command like *ksh -f file_name*, with obvious interpretation. The named shell then uses the file identified as the argument of the script file.

13.3 Some Example Scripts

In this section we shall see some shell scripts with accompanying explanations. It is hoped that these examples would provide some understanding of shell programming. In addition it should help a learner to start using shell scripts to be able to use the system

Option chosen	The effect of choice
-v	view the file being executed
-x	view each command as it gets executed
-n	avoid any side effects from an erroneous command

Table 13.2: The options with their effects.

efficiently in some repetitive programming situations. All examples use the second method discussed above for text files creation and usage as shell scripts. To execute these the pattern of command shall be as follows:

sh [options] <file_name> arg1 arg2 ....

The options may be [vxn]. The effect of the choice is as shown in Table 13.2.

13.3.1 Example Shell Scripts

We shall first see how a user-defined variable is assigned a value. The assignment may be done within a script or may be accepted as an inline command argument. It is also possible to check out if a variable has, or does not have a definition. It is also possible to generate a suitable message in case a shell variable is not defined.

file sh_0.file: We shall begin with some of the simplest shell commands like *echo* and also introduce how a variable may be defined in the context of current shell.

```
---------------------------------------------------------------------
# file sh_0.file
echo shellfile is running /* just echos the text following echo */
defineavar=avar /* defines a variable called defineavar */
echo $defineavar /* echos the value of defineavar */
echo "making defineavar readonly now" /* Note the text is quoted */
readonly defineavar
echo "an attempt to reassign defineavar would not succeed"
defineavar=newvar
---------------------------------------------------------------------
```

file sh_1.file One of the interesting things which one can do is to supply parameters to a shell script. This is similar to the command line arguments given to a program written in a high level programming language like c.

```
# file sh_1.file
# For this program give an input like "This is a test case" i.e. 5 parameters
echo we first get the file name
```

echo $0 /* $0 yields this script file name */

Variable	Interpretation
$$	Process number of the current process
$!	Process number of the last background process
$?	Exit value of the last command
$#	The number of command line arguments
$n	The n th command line argument (maximum 9)
$*	All command line arguments

Table 13.3: A partial list of special variables.

echo we now get the number of parameters that were input

echo $# /* yields the number of arguments */

echo now we now get the first three parameters that were input

echo $1 $2 $3 /* The first three arguments */

shift /* shift cmd could have arguments */

echo now we now shift and see three parameters that were input

echo $1 $2 $3 /* Three arguments after one shift */

A partial list of symbols with their respective meanings is shown in Table 13.3.

file sh_1a.file

this is a file with only one echo command

echo this line is actually a very long command as its arguments run \

on and on and on Note that both this and the line above and below \

are part of the command Also note how back slash folds commands

In most shells, multiple commands may be separated on the same line by a semi-colon (;). In case the command needs to be folded this may be done by simply putting a back slash and carrying on as shown.

file sh_2.file: If a variable is not defined, no value is returned for it. However, one can choose to return an error message and check out if a certain variable has indeed been defined. A user may even generate a suitable message as shown in scripts sh 2a and sh 2b.

file sh_2.file

This is to find out if a certain parameter has been defined.

echo param is not defined so we should get a null value for param

echo ${param}

echo param was not defined earlier so we got no message.

```
echo Suppose we now use "?" option. Then we shall get an error message
echo ${param?error}
```

```
# file sh_2a.file
# This is to find out if a certain parameter has been defined.
echo param is not defined so we should get a null value for param
echo ${param}
# echo param is not defined with "?" option we get the error message
# echo ${param?error}
echo param is not defined with "-" option we get the quoted message
echo ${param-'user generated quoted message'}
```

```
# file sh_2b.file
# This is to find out if a certain parameter has been defined.
echo param is not defined so we should get a null value for param
echo ${param}
# echo param is not defined with "?" option we get the error message
# echo ${param?error}
echo param is not defined with "=" option we get the quoted message
echo ${param='user generated quoted message'}
```

file sh_3.file: Now we shall see a few scripts that use a text with three kinds of quotes - the double quotes, the single forward quote and the single backward quote. As the examples in files sh 3, sh 3a and sh 3b show, the double quotes evaluates a string within it as it is. The back quotes result in substituting the value of shell variables and we may use variables with a $ sign pre-fixed if we wish to have the string substituted by its value. We show the use of back quotes in this script in a variety of contexts.

```
# file sh_3.file
echo the next line shows command substitution within back quotes
echo I am `whoami`              /* every thingin back quotes is evaluated   */
echo I am 'whoami'              /* nothing in back quotes is evaluated      */
echo "I am using $SHELL"        /* Variables evaluated in double quotes     */
```

```
echo today is `date`              /* one may mix quotes and messages     */
echo there are `who | wc -l` users at the moment /* using a pipe          */
echo var a is now assigned the result of echo backquoted whoami
a=`whoami`
echo we shall output its value next
echo $a
echo also let us reassign a with the value for environment var HOME
a=`echo $HOME`
echo $a
echo a double dollar is a special variable that stores process id of the shell
echo $$
echo the shell vars can be used to generate arguments for Unix commands
echo like files in the current directory are
cur_dir=.
ls $cur_dir
echo list the files under directory A
ls $cur_dir/A
----------------------------------------------------------------------
# file sh_3a.file
# In this file we learn to use quotes. There are three types of quotes
# First use of a single quote within which no substitution takes place
a=5
echo 'Within single quotes value is not substituted i.e $a has a value of $a'
# now we look at the double quote
echo "Within double quotes value is substituted so dollar a has a value of $a"
echo Finally we look at the case of back quotes where everything is evaluated
echo `$a`
echo `a`
echo Now we show how a single character may be quoted using reverse slash
echo back quoted a is \`a and dollar a is \$a
echo quotes are useful in assigning variables values that have spaces
```

```
b='my name'
echo value of b is = $b
```

```
# file sh_3b.file
# In this file we shall study the set command. Set lets you
# view shell variable values
echo ---------out put of set --------------
set
echo use printenv to output variables in the environment
echo ---------output of printenv --------------
printenv
```

---

file sh_4.file

One of the interesting functions available for use in shell scripts is the \eval" function. The name of the function is a give away. It means \to evaluate". We simply evaluate the arguments. As a function it was first used in functional programming languages. It can be used in a nested manner as well, as we shall demonstrate in file sh_4.file.

```
# file sh_4.file
# this file shows the use of eval function in the shell
b=5
a=\$b
echo a is $a
echo the value of b is $b
eval echo the value of a evaluated from the expression it generates i.e. $a
c=echo
eval $c I am fine
d=\$c
echo the value of d is $d
eval eval $d I am fine
```

---

file sh_5.file: In the next two files we demonstrate the use of a detached process and also how to invoke a sleep state on a process.

```
# file sh_5.file
# This file shows how we may group a process into a detached process
# by enclosing it in parentheses.
# Also it shows use of sleep command
echo basically we shall sleep for 5 seconds after launching
echo a detached process and then give the date
(sleep 5; date)
```

---

file sh_6.file

```
# file sh_6.file
# Typically << accepts the file till the word that follows
# in the file. In this case the input is taken till
# the word end appears in the file.
#
# This file has the command as well as data in it.
# Run it : as an example : sh_6.file 17 to see him 2217 as output.
# $1 gets the file argument.
grep $1<<end /* grep is the pattern matching command in Unix */
me 2216
him 2217
others 2218
end
```

---

file sh_7.file: Now we first show the use of an if command. We shall check the existence of a .ps file and decide to print it if it exists or else leave a message that the file is not in this directory. The basic pattern of the “if” command is just like in the programming languages. It is:

```
if condition
    then
```

```
        command_pattern_for_true
else
       command_pattern_for_false
fi
```

We shall now use this kind of pattern in the program shown below:

```
# file sh_7.file
if ls my_file.ps
then lpr -Pbarolo-dup my_file.ps /* prints on printer barolo on both sides */
else echo "no such file in this directory"
fi
```

Clearly a more general construct is the case and it is used in the next script.

```
# file sh_7a.file
# This file demonstrates use of case
# In particular note the default option and usage of selection
# Note the pattern matching using the regular expression choices.
case $1 in
[0-9]) echo "OK valid input : a digit ";;
[a-z]|[A-Z]) echo "OK valid input : a letter ";;
*) echo "please note only a single digit or a letter is valid as input";;
esac
```

---

file sh_8.file: We shall now look at an iterative structure. Again it is similar to what we use in a programming language. It is:

```
for some_var in the_list
    do
       the_given_command_set
   done /* a do is terminated by done */
```

We shall show a use of the pattern in the example below:

```
# file sh_8.file
# In this file we illustrate use of for command
# It may be a good idea to remove some file called
```

Test format	Value returned
.b file_name	True when file_name exists as a blocked special file
.c file_name	True when file_name exists as a character special file
.d file_name	True when file_name exists and is a directory
.f file_name	True when file_name exists and is a regular file
.g file_name	True when file_name exists and its set groupID bit is set
.h file_name	True when file_name exists and is a symbolic link
.H file_name	True when file_name exists and is a hidden directory
.k file_name	True when file_name exists and its sticky bit is set
.p file_name	True when file_name exists and is a named pipe
.r file_name	True when file_name exists and is readable
.s file_name	True when file_name exists and has non zero size
.u file_name	True when file_name exists and its setuserID bit is set
.w file_name	True when file_name exists and is writable
.x file_name	True when file_ exists and is executable

Table 13.4: A list of test options.

```
# dummy in the current directory as a first step.
#echo removing dummy
rm dummy
for i in `ls`; do echo $i >> dummy; done
grep test dummy
```

File tests may be much more complex compared to the ones shown in the previous example. There we checked for only the existence of the file. In Table 13.4 we show some test options.

In the context of use of test, one may perform tests on strings as well. The table below lists some of the possibilities.

Test Operation	Returned value
str1 = str2	True when strs are equivalent
str != str2	True when not equivalent
-1 str	True when str has length 0
-n str	True when str has nonzero length
string	True when NOT the null string.

In addition to for there are while and until iterators which also have their do and done. These two patterns are shown next.

```
while condition
   do
      command_set
```

```
  done
```

Alternatively, we may use until with obvious semantics.

```
until condition
   do
      command_set
  done
```

A simple script using until may be like the one given below.

```
count=2
until [ $count -le 0 ]
    do
      lpr -Pbarolo-dup file$count /* prints a file with suffix = count */
      count=`expr $count - 1`
   done
```

Note that one may nest these commands, i.e. there may be a until within a while or if or case.

file sh_9.file: Now we shall demonstrate the use of expr command. This command offers an opportunity to use integer arithmetic as shown below.

```
b=3
echo value of b is = $b
echo we shall use as the value of b to get the values for a
echo on adding two we get
a=`expr $b + 2`
echo $a
```

---

file sh 9a.file We shall combine the use of test along with expr. The values of test may be *true or false* and these may be combined to form relational expressions which finally yield a logical value.

```
# file sh_9a.file
# this file illustrates the use of expr and test commands
b=3
echo on adding two we get
```

```
a=`expr $b + 2`
echo $a
echo on multiplying two we get
a=`expr $b \* 2`            /* Note the back slash preceding star */
# We shall see the reason for using back slash before star in the next example
echo $a
test $a -gt 100
$?
test $a -lt 100
$?
test $a -eq 6
$?
test $a = 6
$?
test $a -le 6
$?
test $a -ge 6
$?
test $a = 5
$?
if (test $a = 5)
    then echo "found equal to 5"
    else echo "found not equal to 5"
fi
test $a = 6
if (test $a = 6)
then echo "the previous test was successful"
fi
```

---

file sh 10.file Now we shall use some regular expressions commonly used with file names.

```
# file sh_10.file
# in this program we identify directories in the current directory
echo "listing all the directories first"
for i in *
do
if test -d $i
then echo "$i is a directory"
fi
done
echo "Now listing the files"
for i in *
do
if test -f $i
  then
    echo "$i is a file"
fi
done
echo "finally the shell files are"
ls | grep sh_
```

```
file sh 11.file
# file sh_11.file
# In this file we learn about the trap command. We will first
# create many files with different names. Later we will remove
# some of these by explicitly trapping
touch rmf1
touch keep1
touch rmf2
touch rmf3
touch keep2
touch rmf4
```

```
touch keep3
echo "The files now are"
ls rmf*
ls keep*
trap `rm rm*; exit` 1 2 3 9 15
echo "The files now are"
ls rmf*
ls keep*
```

---

file sh_12:.file Now we assume the presence of files of telephone numbers. Also, we demonstrate how Unix utilities can be used within the shell scripts.

```
# file sh_12.file
# In this file we invoke a sort command and see its effect on a file
# Also note how we have used input and output on the same line of cmd.
sort < telNos > stelNos
# We can also use a translate cmd to get translation from lower to upper case
tr a-z A-Z < telNos > ctelNos
```

In this module we saw the manner in which a user may use Unix shell and facilities offered by it. As we had earlier remarked, much of Unix is basically a shell and a kernel.