UNIT – II: SHELL PROGRAMMING

2.1 Necessity of shell programming

Shell program is series of Linux commands. Shell script is just like batch file is MS-DOS but have more power than the MS-DOS batch file. Shell script can take input from user, file and output them on screen. Useful to create our own commands that can save our lots of time and to automate some task of day today life.

One reason to use the shell for programming is that you can program the shell quickly and simply. Moreover, a shell is always available even on the most basic Linux installation, so for simple prototyping you can find out if your idea works. The shell is also ideal for any small utilities that perform some relatively simple task for which efficiency is less important than easy configuration, maintenance, and portability. You can use the shell to organize process control, so that commands run in a predetermined sequence dependent on the successful completion of each stage.

Although the shell has superficial similarities to the Windows command prompt, it's much more powerful, capable of running reasonably complex programs in its own right. Not only can you execute commands and call Linux utilities, you can also write them. The shell executes shell programs, often referred to as scripts, which are interpreted at runtime. This generally makes debugging easier because you can easily execute single lines, and there's no recompile time. However, this can make the shell unsuitable for timecritical or processor-intensive tasks.

2.2 Pipes and redirection

2.2.1 Redirecting Output

```
$ ls -l > lsoutput.txt
```

which saves the output of the ls command into a file called lsoutput.txt.

However, there is much more to redirection than this simple example reveals. You'll learn more about the standard file descriptors in Chapter 3, but for now all you need to know is that file descriptor 0 is the standard input to a program, file descriptor 1 is the standard output, and file descriptor 2 is the standard error output. You can redirect each of these independently. In fact, you can also redirect other file descriptors, but it's unusual to want to redirect any other than the standard ones: 0, 1, and 2.

The preceding example redirects the standard output into a file by using the > operator. By default, if the file already exists, then it will be overwritten. If you want to change the default behavior, you can use the command set -o noclobber (or set -C), which sets the noclobber option to

Suppose you want to use the kill command to kill a process from a script. There is always a slight risk that the process will die before the kill command is executed. If this happens, kill will write an error message to the standard error output, which, by default, will appear on the screen. By redirecting both the standard output and the error, you can prevent the kill command from writing any text to the screen.
The command

```
$ kill -HUP 1234 >killout.txt 2>killerr.txt
```

will put the output and error information into separate files.
If you prefer to capture both sets of output into a single file, you can use the >& operator to combine the two outputs. Therefore,

```
$ kill -1 1234 >killouterr.txt 2>&1
```

will put both the output and error outputs into the same file. Notice the order of the operators. This reads as "redirect standard output to the file killouterr.txt, and then direct standard error to the same place as the standard output." If you get the order wrong, the redirect won't work as you expect.

Because you can discover the result of the kill command using the return code (discussed in more detail later in this chapter), you don't often want to save either standard output or standard error. You can use the Linux universal "bit bucket" of /dev/null to efficiently discard the entire output, like this:

```
$ kill -1 1234 >/dev/null 2>&1
```

2.2.2 Redirecting Input

Rather like redirecting output, you can also redirect input. For example,

```
$ more < killout.txt
```

Obviously, this is a rather trivial example under Linux; the Linux more command is quite happy to accept filenames as parameters, unlike the Windows command-line equivalent.

2.2.3 Pipes

You can connect processes using the pipe operator (|). In Linux, unlike in MS-DOS, processes connected by pipes can run simultaneously and are automatically rescheduled as data flows between them. As a simple example, you could use the sort command to sort the output from ps.
If you don't use pipes, you must use several steps, like this:

```
$ ps > psout.txt
$ sort psout.txt > pssort.out
```

A much more elegant solution is to connect the processes with a pipe:

```
$ ps | sort > pssort.out
```

Because you probably want to see the output paginated on the screen, you could connect a third process, more, all on the same command line:

```
$ ps | sort | more
```

There's practically no limit to the permissible number of connected processes. Suppose you want to see all the different process names that are running excluding shells. You could use

```
$ ps –xo comm | sort | uniq | grep -v sh | more
```

This takes the output of ps, sorts it into alphabetical order, extracts processes using uniq, uses grep -vsh to remove the process named sh, and finally displays it paginated on the screen.

As you can see, this is a much more elegant solution than a string of separate commands, each with its own temporary file. However, be wary of one thing here: If you have a string of commands, the output file is created or written to immediately when the set of commands is created, so never use the same filename twice in a string of commands. If you try to do something like

```
cat mydata.txt | sort | uniq > mydata.txt
```

you will end up with an empty file, because you will overwrite the mydata.txt file before you read it.

2.3 The Shell as a programming language

Now that you've seen some basic shell operations, it's time to move on to some actual shell programs. There are two ways of writing shell programs. You can type a sequence of commands and allow the shell to execute them interactively, or you can store those commands in a file that you can then invoke as a program.

2.3.1 Interactive Programs

Just typing the shell script on the command line is a quick and easy way of trying out small code fragments, and is very useful while you are learning or just testing things out.

Suppose you have a large number of C files and wish to examine the files that contain the string POSIX. Rather than search using the grep command for the string in the files and then list the files individually, you could perform the whole operation in an interactive script like this:

```
$ for file in *
> do
> if grep -l POSIX $file
> then
> more $file
```

> fi
> done
posix
This is a file with POSIX in it - treat it well
$
Note how the normal $ shell prompt changes to a > when the shell is expecting further input. You can type away, letting the shell decide when you're finished, and the script will execute immediately.

In this example, the grep command prints the files it finds containing POSIX and then more displays the contents of the file to the screen. Finally, the shell prompt returns. Note also that you called the shell variable that deals with each of the files to self-document the script. You could equally well have used i, but file is more meaningful for humans to read.
The shell also performs wildcard expansion (often referred to as globbing). You are almost certainly aware of the use of '*' as a wildcard to match a string of characters. What you may not know is that you can request single-character wildcards using ?, while [set] allows any of a number of single characters to be checked. [^set] negates the set — that is, it includes anything but the set you've specified. Brace expansion using {} (available on some shells, including bash) allows you to group arbitrary strings together in a set that the shell will expand. For example,

$ ls my_{finger,toe}s

will list the files my_fingers and my_toes. This command uses the shell to check every file in the current directory. We will come back to these rules for matching patterns near the end of the chapter when we look in more detail at grep and the power of regular expressions.
Experienced Linux users would probably perform this simple operation in a much more efficient way, perhaps with a command such as

$ more `grep -l POSIX *`

or the synonymous construction

*$ more $(grep -l POSIX *)*

In addition,

*$ grep -l POSIX * | more*

will output the name of the file whose contents contained the string POSIX. In this script, you see the shell making use of other commands, such as grep and more, to do the hard work. The shell simply enables you to glue several existing commands together in new and powerful ways. You will see wildcard expansion used many times in the following scripts, and we'll look at the whole area of expansion in more detail when we look at regular expressions in the section on the grep command.

Going through this long rigmarole every time you want to execute a sequence of commands is a bore. You need to store the commands in a file, conventionally referred to as a shell script, so you can execute them whenever you like.

2.3.1 Creating a script

Using any text editor, you need to create a file containing the commands; create a file called first that looks like this:

```
#!/bin/sh
# first
# This file looks through all the files in the current
# directory for the string POSIX, and then prints the names of
# those files to the standard output.
for file in *
do
if grep -q POSIX $file
then
echo $file
fi
done
exit 0
```

Comments start with a # and continue to the end of a line. Conventionally, though, # is kept in the first column. Having made such a sweeping statement, we next note that the first line, #!/bin/sh, is a special form of comment; the #! characters tell the system that the argument that follows on the line is the program to be used to execute this file. In this case, /bin/sh is the default shell program.

Since the script is essentially treated as standard input to the shell, it can contain any Linux commands referenced by your PATH environment variable.

The exit command ensures that the script returns a sensible exit code (more on this later in the chapter). This is rarely checked when programs are run interactively, but if you want to invoke this script from another script and check whether it succeeded, returning an appropriate exit code is very important. Even if you never intend to allow your script to be invoked from another, you should still exit with a reasonable code. Have faith in the usefulness of your script: Assume it may need to be reused as part of another script someday.

A zero denotes success in shell programming. Since the script as it stands can't detect any failures, it always returns success. We'll come back to the reasons for using a zero exit code for success later in the chapter, when we look at the exit command in more detail.

Notice that this script does not use any filename extension or suffix; Linux, and UNIX in general, rarely makes use of the filename extension to determine the type of a file. You could have used .sh or added a different extension, but the shell doesn't care. Most preinstalled scripts will not have any filename extension, and the best way to check if they are scripts or not is to use the file command — for example, file first or file /bin/bash. Use whatever convention is applicable where you work, or suits you

2.3.2 Making a Script Executable

Now that you have your script file, you can run it in two ways. The simpler way is to invoke the shell with the name of the script file as a parameter:

$ /bin/sh first

This should work, but it would be much better if you could simply invoke the script by typing its name, giving it the respectability of other Linux commands. Do this by changing the file mode to make the file executable for all users using the chmod command:

$ chmod +x first

You can then execute it using the command

$ first

You may get an error saying the command wasn't found. This is almost certainly because the shell environment variable PATH isn't set to look in the current directory for commands to execute. To change this, either type PATH=$PATH:. on the command line or edit your .bash_profile file to add this command to the end of the file; then log out and log back in again. Alternatively, type ./first in the directory containing the script, to give the shell the full relative path to the file.

Specifying the path prepended with ./ does have one other advantage: It ensures that you don't accidentally execute another command on the system with the same name as your script file

Once you're confident that your script is executing properly, you can move it to a more appropriate location than the current directory. If the command is just for your own use, you could create a bin directory in your home directory and add that to your path. If you want the script to be executable by others, you could use /usr/local/bin or another system directory as a

convenient location for adding new programs. If you don't have root permissions on your system, you could ask the system administrator to copy your file for you, although you may have to convince them of its worth first. To prevent other users from changing the script, perhaps accidentally, you should remove write access from it. The sequence of commands for the administrator to set ownership and permissions would be something like this:

```
# cp first /usr/local/bin
# chown root /usr/local/bin/first
# chgrp root /usr/local/bin/first
# chmod 755 /usr/local/bin/first
```

Notice that rather than alter a specific part of the permission flags, you use the absolute form of the chmod here because you know exactly what permissions you require.

If you prefer, you can use the rather longer, but perhaps more obvious, form of the chmod command:

```
# chmod u=rwx,go=rx /usr/local/bin/first
```

2.4 Shell Syntax

The shell is quite an easy programming language to learn, not least because it's easy to test small program fragments interactively before combining them into bigger scripts. You can use the bash shell to write quite large, structured programs. The next few sections cover the following:

- ❑ Variables: strings, numbers, environments, and parameters
- ❑ Conditions: shell Booleans
- ❑ Program control: if, elif, for, while, until, case
- ❑ Lists
- ❑ Functions
- ❑ Commands built into the shell
- ❑ Getting the result of a command
- ❑ Here documents

2.4.1 Variables

You don't usually declare variables in the shell before using them. Instead, you create them by simply using them (for example, when you assign an initial value to them). By default, all variables are considered and stored as strings, even when they are assigned numeric values. The shell and some utilities will convert numeric strings to their values in order to operate on them as required. Linux is a case-sensitive system, so the shell considers the variable foo to be different from Foo, and both to be different from FOO.

Within the shell you can access the contents of a variable by preceding its name with a $. Whenever you extract the contents of a variable, you must give the variable a preceding $. When you assign a value to a variable, just use the name of the variable, which is created dynamically if necessary. An easy way to check the contents of a variable is to echo it to the terminal, preceding its name with a $.
On the command line, you can see this in action when you set and check various values of the variable salutation:

```
$ salutation=Hello
$ echo $salutation
Hello
$ salutation="Yes Dear"
$ echo $salutation
Yes Dear
$ salutation=7+5
$ echo $salutation
7+5
```

You can assign user input to a variable by using the read command. This takes one parameter, the name of the variable to be read into, and then waits for the user to enter some text. The read normally completes when the user presses Enter. When reading a variable from the terminal, you don't usually need the quote marks:

```
$ read salutation
Wie geht's?
$ echo $salutation
Wie geht's?
```

2.4.2 Conditions

Fundamental to all programming languages is the ability to test conditions and perform different actions based on those decisions. Before we talk about that, though, let's look at the conditional constructs that you can use in shell scripts and then examine the control structures that use them.

The test or [Command

In practice, most scripts make extensive use of the [or test command, the shell's Boolean check.
On some systems, the [and test commands are synonymous, except that when the [command is
used, a trailing] is also used for readability. Having a [command might seem a little odd, but within the code it does make the syntax of commands look simple, neat, and more like other programming languages

We'll introduce the test command using one of the simplest conditions: checking to see whether a file exists. The command for this is test -f , so within a script you can write

```
if test -f fred.c
then
...
Fi
```

The test command's exit code (whether the condition is satisfied) determines whether the conditional code is run.

We're getting ahead of ourselves slightly, but following is an example of how you would test the state of the file /bin/bash, just so you can see what these look like in use:

```
#!/bin/sh
if [ -f /bin/bash ]
then
echo "file /bin/bash exists"
fi
if [ -d /bin/bash ]
then
echo "/bin/bash is a directory"
else
echo "/bin/bash is NOT a directory"
fi
```

Before the test can be true, all the file conditional tests require that the file also exists. This list contains just the more commonly used options to the test command, so for a complete list refer to the manual entry.

2.4.3 Control structures

The shell has a set of control structures, which are very similar to other programming languages.

if

The if statement is very simple: It tests the result of a command and then conditionally executes a group of statements:

```
if condition
then
statements
else
statements
fi
```

A common use for if is to ask a question and then make a decision based on the answer:

```
#!/bin/sh
echo "Is it morning? Please answer yes or no"
read timeofday
if [ $timeofday = "yes" ]; then
echo "Good morning"
else
echo "Good afternoon"
fi
exit 0
```

This would give the following output:

```
Is it morning? Please answer yes or no
yes
Good morning
$
```

This script uses the [command to test the contents of the variable timeofday. The result is evaluated by the if command, which then allows different lines of code to be executed.

for

Use the for construct to loop through a range of values, which can be any set of strings. They could be simply listed in the program or, more commonly, the result of a shell expansion of filenames.

The syntax is simple:

```
for variable in values
do
statements
done
```

The values are normally strings, so you can write the following:

```
#!/bin/sh
for foo in bar fud 43
do
echo $foo
done
exit 0
```

That results in the following output:

```
bar
fud
43
```

while

Because all shell values are considered strings by default, the for loop is good for looping through a series of strings, but is not so useful when you don't know in advance how many times you want the loop to be executed. When you need to repeat a sequence of commands, but don't know in advance how many times they should execute, you will normally use a while loop, which has the following syntax:

while condition do
statements
done

For example, here is a rather poor password-checking program:

```
#!/bin/sh
echo "Enter password"
read trythis
while [ "$trythis" != "secret" ]; do
echo "Sorry, try again"
read trythis
done
exit 0
```

An example of the output from this script is as follows:

```
Enter password
password
Sorry, try again
secret
$
```

Clearly, this isn't a very secure way of asking for a password, but it does serve to illustrate the while statement. The statements between do and done are continuously executed until the condition is no longer true. In this case, you're checking whether the value of trythis is equal to secret. The loop will continue until $trythis equals secret. You then continue executing the script at the statement immediately following the done

2.4.3 Functions

You can define functions in the shell; and if you write shell scripts of any size, you'll want to use them to structure your code

To define a shell function, simply write its name followed by empty parentheses and enclose the statements in braces:

```
function_name () {
statements
}
```

Example:

```
#!/bin/sh
foo() {
echo "Function foo is executing"
}
echo "script starting"
foo
echo "script ended"
exit 0
```

Running the script will output the following:

```
script starting
Function foo is executing
script ending
```

2.4.4 Commands

You can execute two types of commands from inside a shell script. There are "normal" commands that you could also execute from the command prompt (called external commands), and there are "built-in" commands (called internal commands), as mentioned earlier. Built-in commands are implemented internally tothe shell and can't be invoked as external programs. However, most internal commands are also provided as standalone programs — this requirement is part of the POSIX specification. It generally doesn't matter if the command is internal or external, except that internal commands execute more efficiently.

Here we'll cover only the main commands, both internal and external, that we use when we're programming scripts. As a Linux user, you probably know many other commands that are valid at the command prompt. Always remember that you can use any of these in a script in addition to the built-in commands presented here

break

Use break for escaping from an enclosing for, while, or until loop before the controlling condition has been met. You can give break an additional numeric parameter, which is the number of loops to break out of, but this can make scripts very hard to read, so we don't suggest you use it. By default, break escapes a single level.

```
#!/bin/sh
rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4
```

```
for file in fred*
do
if [ -d "$file" ]; then
break;
        fi
done
echo first directory starting fred was $file
rm -rf fred*
exit 0
```

The : Command

The colon command is a null command. It's occasionally useful to simplify the logic of conditions, beingan alias for true. Since it's built-in, : runs faster than true, though its output is also much less readable.

You may see it used as a condition for while loops; while : implements an infinite loop in place of the more common while true.

The : construct is also useful in the conditional setting of variables. For example,

```
: ${var:=value}
```

Without the :, the shell would try to evaluate $var as a command.

```
#!/bin/sh
rm -f fred
if [ -f fred ]; then
:
else
echo file fred did not exist
fi
exit 0
```

continue

Rather like the C statement of the same name, this command makes the enclosing for, while, or until loop continue at the next iteration, with the loop variable taking the next value in the list:

```
#!/bin/sh
rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4
for file in fred*
do
```

```
if [ -d "$file" ]; then
echo "skipping directory $file"
continue
fi
echo file is $file
done
rm -rf fred*
exit 0
```

continue can take the enclosing loop number at which to resume as an optional parameter so that you can partially jump out of nested loops. This parameter is rarely used, as it often makes scripts much harder to understand. For example

```
for x in 1 2 3
do
echo before $x
continue 1
echo after $x
done
```

The output for the preceding will be

```
before 1
before 2
before
```

echo

Despite the X/Open exhortation to use the printf command in modern shells, we've been following common practice by using the echo command to output a string followed by a newline character.

A common problem is how to suppress the newline character. Unfortunately, different versions of UNIX have implemented different solutions.

The common method in Linux is to use

```
echo -n "string to output"
```

but you'll often come across

```
echo -e "string to output\c"
```

The second option, echo -e, ensures that the interpretation of backslashed escape characters, such as \c for suppressing a newline, \t for outputting a tab and \n for outputting carriage returns, is enabled. In older versions of bash this was often set by default, but more recent versions often default to not interpreting backslashed escape characters. See the manual pages for details of the behavior on your distribution.

2.4.5 Command Execution

When you're writing scripts, you often need to capture the result of a command's execution for use in the shell script; that is, you want to execute a command and put the output of the command into a variable.

You can do this by using the $(command) syntax introduced in the earlier set command example. There is also an older form, `command`, that is still in common usage.

All new scripts should use the $(...) form, which was introduced to avoid some rather complex rules covering the use of the characters $, `, and \ inside the backquoted command. If a backtick is used within the `...` construct, it must be escaped with a \ character. These relatively obscure characters often confuse programmers, and sometimes even experienced shell programmers are forced to experiment to get the quoting correct in backticked commands.

The result of the $(command) is simply the output from the command. Note that this isn't the return status of the command but the string output, as shown here:

```
#!/bin/sh
echo The current directory is $PWD
echo The current users are $(who)
exit 0
```

Since the current directory is a shell environment variable, the first line doesn't need to use this command execution construct. The result of who, however, does need this construct if it is to be available to the script.

If you want to get the result into a variable, you can just assign it in the usual way:

```
whoisthere=$(who)
echo $whoisthere
```

The capability to put the result of a command into a script variable is very powerful, as it makes it easy to use existing commands in scripts and capture their output. If you ever find yourself trying to convert a set of parameters that are the output of a command on standard output and capture them as arguments for a program, you may well find the command xargs can do it for you. Look in the manual pages for further details.

A problem sometimes arises when the command you want to invoke outputs some white space before the text you want, or more output than you require. In such a case, you can use the set command as shown earlier.

Arithmetic Expansion

We've already used the expr command, which enables simple arithmetic commands to be processed, but this is quite slow to execute because a new shell is invoked to process the expr command.

A newer and better alternative is $((…)) expansion. By enclosing the expression you wish to evaluate in
$((…)), you can perform simple arithmetic much more efficiently:

```
#!/bin/sh
x=0
while [ "$x" -ne 10 ]; do
echo $x
x=$(($x+1))
done
exit
```