LINUX SHELL SCRIPTING | Essentials

Diving into the world of commands and automation for UNIX / Linux

Compiled by John Rey A. Goh for Core360 IT Services
johnreygoh@gmail.com

Chapter 1: Introducing the Shell

Things you should know before using the shell

The shell environment helps users to interact with and access core functions of the operating system. The term scripting is more relevant in this context. Scripting is usually supported by interpreter-based programming languages. Shell scripts are files in which we write a sequence of commands that we need to perform and are executed using the shell utility.

In this training we are dealing with Bash (Bourne Again Shell), which is the default shell environment for most GNU/Linux systems. Since GNU/Linux is the most prominent operating system on Unix-style architecture, most of the examples and discussions are written by keeping Linux systems in mind.

Commands are typed and executed in a shell terminal. When a terminal is opened, a prompt is available which usually has the following format:

```
username@hostname$
```

Or:

```
root@hostname #
```

or simply as $ or #.

$ represents regular users and **#** represents the administrative user root. Root is the most privileged user in a Linux system.

Note: Avoid using the shell as the root user (administrator) to perform tasks. This is because your commands have the potential to do more damage when your shell has more privileges. It is recommended to log in as a regular user, and then use tools such as `sudo' to run privileged commands. Running a command such as sudo <command> <arguments> will run it as root.

A shell script is a text file that typically begins with a shebang, as follows:

```
#!/bin/bash
```

Shebang is a line on which #! is prefixed to the interpreter path. /bin/bash is the interpreter command path for Bash.

Execution of a script can be done in two ways. Either we can run the script as a command-line argument to bash or we can grant execution permission to the script so it becomes executable.

The script can be run with the filename as a command-line argument as follows (the text that starts with # is a comment, you don't have to type it out):

```
$ bash script.sh # Assuming script is in the current directory.
```

Or:

```
$ bash /home/path/script.sh # Using full path of script.sh.
```

If a script is run as a command-line argument for bash, the shebang in the script is not required.

If required, we can utilize the shebang to facilitate running the script on its own. For this, we have to set executable permissions for the script and it will run using the interpreter path that is appended to #! to the shebang. This can be set as follows:

```
$ chmod a+x script.sh
```

This command gives the script.sh file the executable permission for all users. The script can be executed as:

```
$ ./script.sh #./ represents the current directory
```

Or:

```
$ /home/path/script.sh # Full path of the script is used
```

The kernel will read the first line and see that the shebang is #!/bin/bash. It will identify /bin/bash and execute the script internally as:

```
$ /bin/bash script.sh
```

When a shell is started, it initially executes a set of commands to define various settings such as prompt text, colors, and much more. This set of commands are read from a shell script at ~/.bashrc (or ~/.bash_profile for login shells) located in the home directory of the user.

The Bash shell also maintains a history of commands run by the user. It is available in the ~/.bash_history file.

In Bash, each command or command sequence is delimited by using a semicolon or a new line. For example:

```
$ cmd1 ; cmd2
```

This is equivalent to:

```
$ cmd1
$ cmd2
```

Finally, the # character is used to denote the beginning of unprocessed comments.

Printing in the terminal

echo puts a newline at the end of every echo invocation by default:

```
$ echo "Welcome to Bash"
Welcome to Bash
```

Simply, using double-quoted text with the echo command prints the text in the terminal.
Similarly, text without double quotes also gives the same output:

```
$ echo Welcome to Bash
Welcome to Bash
```

Another way to do the same task is by using single quotes:

```
$ echo 'text in quotes'
```

These methods may look similar, but some of them have a specific purpose and side effects too. Consider the following command:

```
$ echo "cannot include exclamation - ! within double quotes"
```

This will return the following output:

```
bash: !: event not found error
```

Hence, if you want to print special characters such as !, either do not use them within double quotes or escape them with a special escape character (\) prefixed with it, like so:

```
$ echo Hello world !
```

Or:

```
$ echo 'Hello world !'
```

Or:

```
$ echo "Hello world \!" #Escape character \ prefixed.
```

The side effects of each of the methods are as follows:

- When using echo without quotes, we cannot use a semicolon, as it acts as a delimiter between commands in the Bash shell
- **echo hello; hello** takes echo hello as one command and the second hello as the second command
- Variable substitution, which is discussed in the next recipe, will not work within single quotes

Another command for printing in the terminal is printf. It uses the same arguments as the printf command in the C programming language. For example:

```
$ printf "Hello world"
```

printf takes quoted text or arguments delimited by spaces. We can use formatted strings with printf. We can specify string width, left or right alignment, and so on. By default, printf does not have newline as in the echo command. We have to specify a newline when required, as shown in the following script:

```
#!/bin/bash
#Filename: printf.sh
printf "%-5s %-10s %-4s\n" No Name Mark
printf "%-5s %-10s %-4.2f\n" 1 Sarath 80.3456
printf "%-5s %-10s %-4.2f\n" 2 James 90.9989
printf "%-5s %-10s %-4.2f\n" 3 Jeff 77.564
```

%s, %c, %d, and %f are format substitution characters for which an argument can be placed after the quoted format string.

%-5s can be described as a string substitution with left alignment (- represents left alignment) with width equal to 5. If - was not specified, the string would have been aligned to the right. The width specifies the number of characters reserved for that variable.

For Name, the width reserved is 10. Hence, any name will reside within the 10-character width reserved for it and the rest of the characters will be filled with space up to 10 characters in total.

For floating point numbers, we can pass additional parameters to round off the decimal places.

For marks, we have formatted the string as %-4.2f, where .2 specifies rounding off to two decimal places. Note that for every line of the format string a newline (\n) is issued.

Escaping newline in echo

By default, echo has a newline appended at the end of its output text. This can be avoided by using the -n flag. echo can also accept escape sequences in double-quoted strings as an argument. When using escape sequences, use echo as echo -e "string containing escape sequences". For example:

```
echo -e "1\t2\t3"
1 2 3
```

Using variables and environment variables

In Bash, the value for every variable is string, regardless of whether we assign variables with quotes or without quotes. Furthermore, there are variables used by the shell environment and the operating environment to store special values, which are called environment variables.

Variables are named with the usual naming constructs. When an application is executing, it will be passed a set of variables called environment variables. To view all the environment variables related to a terminal, issue the env command. For every process, environment variables in its runtime can be viewed by:
cat /proc/$PID/environ

Set PID with a process ID of the process (PID always takes an integer value).

For example, assume that an application called gedit is running. We can obtain the process ID of gedit with the pgrep command as follows:

```
$ pgrep gedit
12501
```

You can obtain the environment variables associated with the process by executing the following command:

```
$ cat /proc/12501/environ
GDM_KEYBOARD_LAYOUT=usGNOME_KEYRING_PID=1560USER=slynuxHOME=/home/slynux
```

If you can substitute the \0 character with \n, you can reformat the output to show each variable=value pair in each line. Substitution can be made using the tr command as follows:

```
$ cat /proc/12501/environ | tr '\0' '\n'
```

A variable can be assigned as follows:

```
var=value
```

var is the name of a variable and value is the value to be assigned. If value does not contain any space character (such as space), it need not be enclosed in quotes, otherwise it is to be enclosed in single or double quotes.

Note that **var = value** and **var=value** are different. It is a usual mistake to write **var = value** instead of **var=value**. The later one is the assignment operation, whereas the earlier one is an equality operation.

Printing contents of a variable is done using by prefixing $ with the variable name as follows:

```
var="value" #Assignment of value to variable var.
echo $var
```

Or:

```
echo ${var}
```

We will receive an output as follows:

```
value
```

We can use variable values inside printf or echo in double quotes:

```
#!/bin/bash
#Filename :variables.sh
fruit=apple
count=5
echo "We have $count ${fruit}(s)"
```

The output will be as follows:

```
We have 5 apple(s)
```

Environment variables are variables that are not defined in the current process, but are received from the parent processes. For example, HTTP_PROXY is an environment variable.

This variable defines which proxy server should be used for an Internet connection.
Usually, it is set as:

```
HTTP_PROXY=192.168.1.23:3128
export HTTP_PROXY
```

The export command is used to set the env variable. Now any application, executed from the current shell script, will receive this variable. We can export custom variables for our own purposes in an application or shell script that is executed. There are many standard environment variables that are available for the shell by default.

For example, PATH. A typical PATH variable will contain:

```
$ echo $PATH
/home/slynux/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

When given a command for execution, the shell automatically searches for the executable in the list of directories in the PATH environment variable (directory paths are delimited by the ":" character). Usually, $PATH is defined in /etc/environment or /etc/profile or ~/.bashrc. When we need to add a new path to the PATH environment, we use:

```
export PATH="$PATH:/home/user/bin"
```

Or, alternately, we can use:

```
$ PATH="$PATH:/home/user/bin"
$ export PATH
$ echo $PATH
/home/slynux/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/home/user/bin
```

Here we have added /home/user/bin to PATH.

Some of the well-known environment variables are HOME, PWD, USER, UID, SHELL, and so on.
When using single quotes, variables will not be expanded and will be displayed as is. This means:
$ echo '$var' will print $var

Whereas, $ echo "$var" will print the value of the $var variable if defined or nothing at all if it is not defined.

Finding the length of a string

Get the length of a variable value using the following command:

```
length=${#var}
```

For example:

```
$ var=123456789012345678905
echo ${#var}
20
```

Identifying the current shell

To identify the shell which is currently being used, we can use the SHELL variable, like so:

```
echo $SHELL
```

Or:

```
echo $0
```

For example:

```
$ echo $SHELL
/bin/bash
```

```
$ echo $0
/bin/bash
```

Checking for super user

UID is an important environment variable that can be used to check whether the current script has been run as a 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 value for the root user is 0.

Computations with the shell

The Bash shell environment can perform basic arithmetic operations using the commands **let**, **(( ))**, and **[]**. The two utilities expr and bc are also very helpful in performing advanced operations.

1. A numeric value can be assigned as a regular variable assignment, which is stored as a string. However, we use methods to manipulate as numbers:

```
#!/bin/bash
no1=4;
no2=5;
```

2. The let command can be used to perform basic operations directly. While using let, we use variable names without the $ prefix, for example:

```
let result=no1+no2
echo $result
```

- Increment operation:

```
$ let no1++
```

- Decrement operation:

```
$ let no1--
```

- Shorthands:

```
let no+=6
let no-=6
```

- Alternate methods:
  The [] operator can be used in the same way as the let command as follows:

```
result=$[ no1 + no2 ]
```

  Using the $ prefix inside [] operators are legal, for example:

```
result=$[ $no1 + 5 ]
```

  (( )) can also be used. $ prefixed with a variable name is used when (()) operator is used, as follows:

```
result=$(( no1 + 50 ))
```

  expr can also be used for basic operations:

```
result=`expr 3 + 4`
result=$(expr $no1 + 5)
```

  All of the preceding methods do not support floating point numbers, and operate on integers only.

3. bc, the precision calculator is an advanced utility for mathematical operations. It has a wide range of options. We can perform floating point operations and use advanced functions as follows:

```
echo "4 * 0.56" | bc
2.24

no=54;
result=`echo "$no * 1.5" | bc`
echo $result
81.0
```

Additional parameters can be passed to bc with prefixes to the operation with semicolon as delimiters through stdin.

- Decimal places scale with bc: 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
```

- Base conversion with bc: We can convert from one base number system to another one. Let us convert from decimal to binary, and binary to octal:

```
#!/bin/bash
Desc: Number conversion

no=100
echo "obase=2;$no" | bc
1100100

no=1100100
echo "obase=10;ibase=2;$no" | bc
100
```

- Calculating squares and square roots can be done as follows:

```
echo "sqrt(100)" | bc #Square root
echo "10^10" | bc #Square
```

Using file descriptors and redirection

File descriptors are integers associated with an opened file or data stream. File descriptors 0, 1, and 2 are reserved as follows:

- 0: stdin (standard input)
- 1: stdout (standard output)
- 2: stderr (standard error)

1. Redirecting or saving output text to a file can be done as follows:

```
$ echo "This is a sample text 1" > temp.txt
This would store the echoed text in temp.txt by truncating the file, the contents will be emptied before writing.
```

2. To append text to a file, consider the following example:

```
$ echo "This is sample text 2" >> temp.txt
```

3. You can view the contents of the file as follows:

```
$ cat temp.txt
This is sample text 1
This is sample text 2
```

4. Let us see what a standard error is and how you can redirect it. stderr messages are printed when commands output an error message. Consider the following example:

```
$ ls +
ls: cannot access +: No such file or directory
```

Successful and unsuccessful commands

When a command returns after an error, it returns a nonzero exit status. The command returns zero when it terminates after successful completion. The return status can be read from special variable $? (run echo $? immediately after the command execution statement to print the exit status).

The following command prints the stderr text to the screen rather than to a file (and because there is no stdout output, out.txt will be empty):

```
$ ls + > out.txt
ls: cannot access +: No such file or directory
```

In the following command, we redirect stderr to out.txt:

```
$ ls + 2> out.txt # works
```

You can redirect stderr exclusively to a file and stdout to another file as follows:

```
$ cmd 2>stderr.txt 1>stdout.txt
```

It is also possible to redirect stderr and stdout to a single file by converting stderr to stdout using this preferred method:

```
$ cmd 2>&1 output.txt
```

Or the alternate approach:

```
$ cmd &> output.txt
```

5. Sometimes, the output may contain unnecessary information (such as debug messages). If you don't want the output terminal burdened with the stderr details then you should redirect the stderr output to /dev/null, which removes it completely. For example, consider that we have three files a1, a2, and a3. However, a1 does not have the read-write-execute permission for the user. When you need to print the contents of files starting with a, we use the cat command. Set up the test files as follows:

```
$ echo a1 > a1
$ cp a1 a2 ; cp a2 a3;
$ chmod 000 a1 #Deny all permissions
```

While displaying contents of the files using wildcards (a*), it will show an error message for file a1 as it does not have the proper read permission:

```
$ cat a*
cat: a1: Permission denied
a1
a1
```

Here, cat: a1: Permission denied belongs to the stderr data. We can redirect the stderr data into a file, whereas stdout remains printed in the terminal. Consider the following code:

```
$ cat a* 2> err.txt #stderr is redirected to err.txt
a1
a1
$ cat err.txt
cat: a1: Permission denied
```

Take a look at the following code:

```
$ cmd 2>/dev/null
```

When redirection is performed for stderr or stdout, the redirected text flows into a file. As the text has already been redirected and has gone into the file, no text remains to flow to the next command through pipe (|), and it appears to the next set of command sequences through stdin.

6. However, there is a way to redirect data to a file, as well as provide a copy of redirected data as stdin for the next set of commands. This can be done using the tee command. For example, to print stdout in the terminal as well as redirect stdout into a file, the syntax for tee is as follows:

```
command | tee FILE1 FILE2
```

In the following code, the stdin data is received by the tee command. It writes a copy of stdout to the out.txt file and sends another copy as stdin for the next command. The cat -n command puts a line number for each line received from stdin and writes it into stdout:

```
$ cat a* | tee out.txt | cat -n
cat: a1: Permission denied
1a1
2a1
```

Examine the contents of out.txt as follows:

```
$ cat out.txt
a1
a1
```

Note that cat: a1: Permission denied does not appear because it belongs to stderr. The tee command can read from stdin only.

By default, the tee command overwrites the file, but it can be used with appended options by providing the -a option, for example, $ cat a* | tee -a out.txt | cat -n.

Commands appear with arguments in the format: command FILE1 FILE2 … or simply command FILE.

7. We can use stdin as a command argument. It can be done by using - as the filename argument for the command as follows:

```
$ cmd1 | cmd2 | cmd -
```

For example:

```
$ echo who is this | tee -
who is this
who is this
```

Alternately, we can use /dev/stdin as the output filename to use stdin. Similarly, use /dev/stderr for standard error and /dev/stdout for standard output. These are special device files that correspond to stdin, stderr, and stdout.

A command that reads stdin for input can receive data in multiple ways. Also, it is possible to specify file descriptors of our own using cat and pipes, for example:

```
$ cat file | cmd
$ cmd1 | cmd
```

Redirection from a file to a command

By using redirection, we can read data from a file as stdin as follows:

```
$ cmd < file
```

Redirecting from a text block enclosed within a script

Sometimes we need to redirect a block of text (multiple lines of text) as standard input. Consider a particular case where the source text is placed within the shell script.

A practical usage example is writing a log file header data. It can be performed as follows:

```
#!/bin/bash
cat<<EOF>log.txt
LOG FILE HEADER
This is a test log file
Function: System statistics
EOF
```

The lines that appear between cat <<EOF >log.txt and the next EOF line will appear as the stdin data. Print the contents of log.txt as follows:

```
$ cat log.txt
LOG FILE HEADER
This is a test log file
Function: System statistics
```

Visiting aliases

An alias is basically a shortcut that takes the place of typing a long-command sequence.

There are various operations you can perform on aliases, these are as follows:

1. An alias can be created as follows:

   ```
   $ alias new_command='command sequence'
   ```

   Giving a shortcut to the install command, apt-get install, can be done as follows:

   ```
   $ alias install='sudo apt-get install'
   ```

   Therefore, we can use install pidgin instead of sudo apt-get install pidgin.

2. The alias command is temporary; aliasing exists until we close the current terminal only. To keep these shortcuts permanent, add this statement to the ~/.bashrc file. Commands in ~/.bashrc are always executed when a new shell process is spawned:

```
$ echo 'alias cmd="command seq"' >> ~/.bashrc
```

3. To remove an alias, remove its entry from ~/.bashrc (if any) or use the unalias command.
4. As an example, we can create an alias for rm so that it will delete the original and keep a copy in a backup directory:

```
alias rm='cp $@ ~/backup && rm $@'
```

There are situations when aliasing can also be a security breach. See how to identify them.

Escaping aliases

The alias command can be used to alias any important command, and you may not always want to run the command using the alias. We can ignore any aliases currently defined by escaping the command we want to run. For example:

```
$ \command
```

The \ character escapes the command, running it without any aliased changes. While running privileged commands on an untrusted environment, it is always good security practice to ignore aliases by prefixing the command with \. The attacker might have aliased the privileged command with his/her own custom command to steal the critical information that is provided by the user to the command.

Terminal Handling

tput and **stty** are utilities that can be used for terminal manipulations. Let us see how to use them to perform different tasks.

There are specific information you can gather about the terminal as shown in the following list:

- Get the number of columns and rows in a terminal by using the following commands:

```
tput cols
tput lines
```

- To print the current terminal name, use the following command:

```
tput longname
```

- To move the cursor to a 100,100 position, you can enter:

```
tput cup 100 100
```

- Set the background color for the terminal using the following command:

```
tputsetb n
n can be a value in the range of 0 to 7.
```

- Set the foreground color for text by using the following command:

```
tputsetf n
n can be a value in the range of 0 to 7.
```

- To make text bold use this:

```
tput bold
```

- To start and end underlining use this:

```
tput smul
tput rmul
```

- To delete from the cursor to the end of the line use the following command:

```
tputed
```

- While typing a password, we should not display the characters typed. In the following example, we will see how to do it using stty:

```
#!/bin/sh
#Filename: password.sh
echo -e "Enter password: "
stty -echo
read password
stty echo
echo
echo Password read.
```

  The -echo option in the command disables the output to the terminal, whereas echo enables output.

Getting and setting dates and delays

Many applications require printing dates in different formats, setting date and time, and performing manipulations based on date and time. Delays are commonly used to provide a wait time (such as 1 second) during the program execution. Scripting contexts, such as monitoring a task every 5 seconds, demands the understanding of writing delays in a program.

It is possible to read the dates in different formats and also to set the date. This can be accomplished with these steps:
1. You can read the date as follows:

```
$ date
Thu May 20 23:09:04 IST 2010
```

2. The epoch time can be printed as follows:

```
$ date +%s
1290047248
```

We can find out epoch from a given formatted date string. You can use dates in multiple date formats as input. Usually, you don't need to bother about the date string format that you use if you are collecting the date from a system log or any standard application generated output. Convert the date string into epoch as follows:

```
$ date --date "Thu Nov 18 08:07:21 IST 2010" +%s
1290047841
```

The --date option is used to provide a date string as input. However, we can use any date formatting options to print the output. Feeding the input date from a string can be used to find out the weekday, given the date.
For example:

```
$ date --date "Jan 20 2001" +%A
Saturday
```

3. Use a combination of format strings prefixed with + as an argument for the date command to print the date in the format of your choice. For example:

```
$ date "+%d %B %Y"
20 May 2010
```

4. We can set the date and time as follows:

```
# date -s "Formatted date string"
```

For example:

```
# date -s "21 June 2009 11:01:22"
```

5. Sometimes we need to check the time taken by a set of commands. We can display it using the following code:

```
#!/bin/bash
#Filename: time_take.sh
start=$(date +%s)
commands;
statements;
end=$(date +%s)
difference=$(( end - start))
echo Time taken to execute commands is $difference seconds.
```

To write a date format to get the output as required, use the following table:

Date component	Format
Weekday	%a (for example, Sat)
	%A (for example, Saturday)
Month	%b (for example, Nov)
	%B (for example, November)
Day	%d (for example, 31)
Date in format (mm/dd/yy)	%D (for example, 10/18/10)
Year	%y (for example, 10)
	%Y (for example, 2010)
Hour	%I or %H (For example, 08)
Minute	%M (for example, 33)
Second	%S (for example, 10)
Nano second	%N (for example, 695208515)
Epoch Unix time in seconds	%s (for example, 1290049486)

Producing delays in a script

To delay execution in a script for a particular period of time, use sleep:$ sleepno_of_seconds. For example, the following script counts from 0 to 40 by using tput and sleep:

```
#!/bin/bash
#Filename: sleep.sh
echo -n Count:
tput sc

count=0;
while true;
do
if [ $count -lt 40 ];
then
let count++;
sleep 1;
tput rc
tput ed
echo -n $count;
else exit 0;
fi
done
```

In the preceding example, a variable count is initialized to 0 and is incremented on every loop execution. The echo statement prints the text. We use tput sc to store the cursor position. On every loop execution we write the new count in the terminal by restoring the cursor position for the number. The cursor position is restored using tput rc. This clears text from the current cursor position to the end of the line, so that the older number can be cleared and the count can be written. A delay of 1 second is provided in the loop by using the sleep command.

Debugging the script

Debugging is one of the critical features that every programming language should implement to produce race-back information when something unexpected happens. Debugging information can be used to read and understand what caused the program to crash or to act in an unexpected fashion. Bash provides certain debugging options that every sysadmin should know.

We can either use Bash's inbuilt debugging tools or write our scripts in such a manner that they become easy to debug, here's how:

1. Add the -x option to enable debug tracing of a shell script as follows:

```
$ bash -x script.sh
```

Running the script with the -x flag will print each source line with the current status. Note that you can also use sh -x script.

2. Debug only portions of the script using set -x and set +x. For example:

```
#!/bin/bash
#Filename: debug.sh
for i in {1..6};
do
set -x
echo $i
set +x
done
echo "Script executed"
```

In the preceding script, the debug information for echo $i will only be printed, as debugging is restricted to that section using -x and +x.

3. The aforementioned debugging methods are provided by Bash built-ins. But they always produce debugging information in a fixed format. In many cases, we need debugging information in our own format. We can set up such a debugging style by passing the _DEBUG environment variable.

Look at the following example code:

```
#!/bin/bash
function DEBUG()
{
[ "$_DEBUG" == "on" ] && $@ || :
}
for i in {1..10}
do
DEBUG echo $i
done
```

We can run the above script with debugging set to "on" as follows:

```
$ _DEBUG=on ./script.sh
```

We prefix DEBUG before every statement where debug information is to be printed. If _DEBUG=on is not passed to the script, debug information will not be printed. In Bash, the command " **:** " tells the shell to do nothing.

The -x flag outputs every line of script as it is executed to stdout. However, we may require only some portions of the source lines to be observed such that commands and arguments are to be printed at certain portions. In such conditions we can use set built in to enable and disable debug printing within the script.

- set -x: This displays arguments and commands upon their execution
- set +x: This disables debugging
- set -v: This displays input when they are read
- set +v: This disables printing input

Shebang hack

The shebang can be changed from

```
#!/bin/bash
```

to

```
#!/bin/bash -xv
```

to enable debugging without any additional flags (-xv flags themselves).

Functions and arguments

We can create functions to perform tasks and we can also create functions that take parameters (also called arguments) as you can see in the following steps:

1. A function can be defined as follows:

```
function fname()
{
statements;
}
```

Or alternately,

```
fname()
{
statements;
}
```

2. A function can be invoked just by using its name:

```
$ fname ; # executes function
```

3. Arguments can be passed to functions and can be accessed by our script:

```
fname arg1 arg2 ; # passing args
```

Following is the definition of the function fname. In the fname function, we have included various ways of accessing the function arguments.

```
fname()
{
echo $1, $2; #Accessing arg1 and arg2
echo "$@"; # Printing all arguments as list at once
echo "$*"; # Similar to $@, but arguments taken as single entity
return 0; # Return value
}
```

Similarly, arguments can be passed to scripts and can be accessed by script:$0
(the name of the script):

- $1 is the first argument
- $2 is the second argument
- $n is the nth argument
- "$@"expands as "$1" "$2" "$3" and so on
- "$*" expands as "$1c$2c$3", where c is the first character of IFS
- "$@" is used more often than "$*"since the former provides all arguments as a single string

Exporting functions

A function can be exported—like environment variables—using export, such that the scope of the function can be extended to subprocesses, as follows:

```
export -f fname
```

Reading the return value (status) of a command We can get the return value of a command or function in the following way:

```
cmd;
echo $?;
$? will give the return value of the command cmd.
```

The return value is called exit status. It can be used to analyze whether a command completed its execution successfully or unsuccessfully. If the command exits successfully, the exit status will be zero, otherwise it will be a nonzero value.

We can check whether a command terminated successfully or not by using the following script:

```
#!/bin/bash
#Filename: success_test.sh
CMD="command" #Substitute with command for which you need to test the
exit status
$CMD
if [ $? -eq 0 ];
then
echo "$CMD executed successfully"
else
echo "$CMD terminated unsuccessfully"
fi
```

Passing arguments to commands

Arguments to commands can be passed in different formats. Suppose -p and-v are the options available and -k N is another option that takes a number. Also, the command takes a filename as argument. It can be executed in multiple ways as shown:

- $ command -p -v -k 1 file
- $ command -pv -k 1 file
- $ command -vpk 1 file
- $ command file -pvk 1

Reading the output of commands

One of the best-designed features of shell scripting is the ease of combining many commands or utilities to produce output. The output of one command can appear as the input of another, which passes its output to another command, and so on. The output of this combination can be read in a variable.

Input is usually fed into a command through stdin or arguments. Output appears as stderr or stdout. While we combine multiple commands, we usually use stdin to give input and stdout to provide an output.

In this context, the commands are called filters. We connect each filter using pipes, the piping operator being |. An example is as follows:

```
$ cmd1 | cmd2 | cmd3
```

Here we combine three commands. The output of cmd1 goes to cmd2 and output of cmd2 goes to cmd3 and the final output (which comes out of cmd3) will be printed, or it can be directed to a file.

We typically use pipes and use them with the subshell method for combining outputs of multiple files. Here's how:

1. Let us start with combining two commands:

   ```
   $ ls | cat -n > out.txt
   ```

   Here the output of ls (the listing of the current directory) is passed to cat –n, which in turn puts line numbers to the input received through stdin. Therefore, its output is redirected to the out.txt file.

2. We can read the output of a sequence of commands combined by pipes as follows:

   ```
   cmd_output=$(COMMANDS)
   ```

   This is called subshell method. For example:

   ```
   cmd_output=$(ls | cat -n)
   echo $cmd_output
   ```

   Another method, called back quotes (some people also refer to it as back tick) can also be used to store the command output as follows:

   ```
   cmd_output=`COMMANDS`
   ```

   For example:

   ```
   cmd_output=`ls | cat -n`
   echo $cmd_output
   ```

   Back quote is different from the single-quote character. It is the character on the ~ button in the keyboard.

Running a command until it succeeds

When using your shell for everyday tasks, there will be cases where a command might succeed only after some conditions are met, or the operation depends on an external event (such as a file being available to download). In such cases, one might want to run a command repeatedly until it succeeds.

Define a function in the following way:

```
repeat()
{
while true
do
$@ && return
done
}
```

Or, add this to your shell's rc file for ease of use:

```
repeat() { while true; do $@ && return; done }
```

A faster approach

On most modern systems, true is implemented as a binary in /bin. This means that each time the aforementioned while loop runs, the shell has to spawn a process. To avoid this, we can use the : shell built-in, which always returns an exit code 0:

```
repeat() { while :; do $@ && return; done }
```

Adding a delay

Let's say you are using repeat() to download a file from the Internet which is not available right now, but will be after some time. An example would be:

```
repeat wget -c http://www.example.com/software-0.1.tar.gz
```

In the current form, we will be sending too much traffic to the web server at www.example.com, which causes problems to the server (and maybe even to you, if say the server blacklists your IP for spam). To solve this, we can modify the function and add a small delay as follows (This will cause the command to run every 30 seconds):

```
repeat() { while :; do $@ && return; sleep 30; done }
```

Field separators and iterators

The internal field separator (IFS) is an important concept in shell scripting. It is very useful while manipulating text data. We will now discuss delimiters that separate different data elements from single data stream. An internal field separator is a delimiter for a special purpose.

An internal field separator is an environment variable that stores delimiting characters. It is the default delimiter string used by a running shell environment. Consider the case where we need to iterate through words in a string or comma separated values (CSV). In the first case we will use IFS=" " and in the second, IFS=",". Let us see how to do it.

Consider the case of CSV data:

```
data="name,sex,rollno,location"
```

To read each of the item in a variable, we can use IFS.

```
oldIFS=$IFS
IFS=, now,
for item in $data;
do
echo Item: $item
done
IFS=$oldIFS
```

```
The output is as follows:
Item: name
Item: sex
Item: rollno
Item: location
```

The default value of IFS is a space component (newline, tab, or a space character).

When IFS is set as comma (,) the shell interprets the comma as a delimiter character, therefore, the $item variable takes substrings separated by a comma as its value during the iteration.

If IFS is not set as comma (,) then it would print the entire data as a single string.

Let us go through another example usage of IFS by taking the /etc/passwd file into consideration. In the /etc/passwd file, every line contains items delimited by ":".

Each line in the file corresponds to an attribute related to a user.

Consider the input: root:x:0:0:root:/root:/bin/bash. The last entry on each line specifies the default shell for the user. To print users and their default shells, we can use the IFS hack as follows:

```
#!/bin/bash
#Desc: Illustration of IFS
line="root:x:0:0:root:/root:/bin/bash"
oldIFS=$IFS;
IFS=":"
count=0
for item in $line;
do
[ $count -eq 0 ] && user=$item;
[ $count -eq 6 ] && shell=$item;
let count++
done;
IFS=$oldIFS
echo $user\'s shell is $shell;
```

The output will be:

```
root's shell is /bin/bash
```

Loops are very useful in iterating through a sequence of values. Bash provides many types of loops. Let us see how to use them:

1. Using a for loop:

```
for var in list;
do
commands; # use $var
done
```

list can be a string, or a sequence.

We can generate different sequences easily.

echo {1..50}can generate a list of numbers from 1 to 50. echo {a..z} or{A..Z} or {a..h} can generate lists of alphabets. Also, by combining these we can concatenate data.

In the following code, in each iteration, the variable i will hold a character in the range a to z:

```
for i in {a..z}; do actions; done;
```

The for loop can also take the format of the for loop in C. For example:

```
for((i=0;i<10;i++))
{
commands; # Use $i
}
```

2. Using a while loop:

```
while condition
do
commands;
done
```

For an infinite loop, use true as the condition.

3. Using a until loop:

A special loop called until is available with Bash. This executes the loop until the given condition becomes true. For example:

```
x=0;
until [ $x -eq 9 ]; # [ $x -eq 9 ] is the condition
do
let x++; echo $x;
done
```

Comparisons and tests

Flow control in a program is handled by comparison and test statements. Bash also comes with several options to perform tests that are compatible with the Unix system-level features. We can use if, if else, and logical operators to perform tests and certain comparison operators to compare data items.

- Using an if condition:

```
if condition;
then
commands;
fi
```

- Using else if and else:

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

- Performing mathematical comparisons: Usually conditions are enclosed in square brackets []. Note that there is a space between [or] and operands. It will show an error if no space is provided. An example is as follows:

```
[$var -eq 0 ] or [ $var -eq 0]
```

Performing mathematical conditions over variables or values can be done as follows:

```
[ $var -eq 0 ] # It returns true when $var equal to 0.
[ $var -ne 0 ] # It returns true when $var is not equal to 0
```

Other important operators are as follows:

```
-gt: Greater than
-lt: Less than
-ge: Greater than or equal to
-le: Less than or equal to
```

- Filesystem related tests: We can test different file system-related attributes using different condition flags as follows:

```
[ -f $file_var ]: This returns true if the given variable holds a regular file path or filename
[ -x $var ]: This returns true if the given variable holds a file path or filename that is executable
[ -d $var ]: This returns true if the given variable holds a directory path or directory name
[ -e $var ]: This returns true if the given variable holds an existing file
[ -c $var ]: This returns true if the given variable holds the path of a character device file
[ -b $var ]: This returns true if the given variable holds the path of a block device file
[ -w $var ]: This returns true if the given variable holds the path of a file that is writable
[ -r $var ]: This returns true if the given variable holds the path of a file that is readable
[ -L $var ]: This returns true if the given variable holds the path of a symlink
```

An example of the usage is as follows:

```
fpath="/etc/passwd"
if [ -e $fpath ]; then
echo File exists;
else
echo Does not exist;
fi
```

- String comparisons: While using string comparison, it is best to use double square brackets, since the use of single brackets can sometimes lead to errors.

Two strings can be compared to check whether they are the same in the following manner:

```
[[ $str1 = $str2 ]]: This returns true when str1 equals str2, that is, the text contents of str1 and str2 are the same
[[ $str1 == $str2 ]]: It is an alternative method for string equality check
```

We can check whether two strings are not the same as follows:

```
[[ $str1 != $str2 ]]: This returns true when str1 and str2 mismatch
```

We can find out the alphabetically smaller or larger string as follows:

```
[[ $str1 > $str2 ]]: This returns true when str1 is alphabetically greater than str2
[[ $str1 < $str2 ]]: This returns true when str1 is alphabetically lesser than str2
```

Note that a space is provided after and before (=), if it is not provided, it is not a comparison, but it becomes an assignment statement.

Chapter 2: Using more commands

Concatenating with cat

cat is one of the first commands that a command-line warrior must learn. It is usually used to read, display, or concatenate the contents of a file, but cat is capable of more than just that. We even scratch our heads when we need to combine standard input data, as well as data from a file using a single-line command. The regular way of combining the stdin data, as well as file data, is to redirect stdin to a file and then append two files. But we can use the cat command to do it easily in a single invocation.

The general syntax of cat for reading contents is:

```
$ cat file1 file2 file3 ...
```

This command concatenates data from the files specified as command-line arguments.

- To print contents of a single file:

```
$ cat file.txt
This is a line inside file.txt
This is the second line inside file.txt
```

- To print contents of more than one file:

```
$ cat one.txt two.txt
This is line from one.txt
This is line from two.txt
```

The cat command can not only read from files and concatenate the data, but can also read the input from the standard input.

To read from the standard input, use a pipe operator as follows:

```
OUTPUT_FROM_SOME COMMANDS | cat
```

Similarly, we can concatenate content from input files along with standard input using cat.
Combine stdin and data from another file, as follows:

```
$ echo 'Text through stdin' | cat - file.txt
```

In this example, - acts as the filename for the stdin text.

Sometimes text files may contain two or more blank lines together. If you need to remove the extra blank lines, use the following syntax:

```
$ cat -s file
```

Finding files and file listing

To list all the files and folders from the current directory to the descending child directories, use the following syntax:

```
$ find base_path
```

base_path can be any location from which find should start descending (for example, /home/slynux/).

An example of this command is as follows:

```
$ find . -print
# Print lists of files and folders
```

. specifies current directory and .. specifies the parent directory. This convention is followed throughout the Unix filesystem.

The -print argument specifies to print the names (path) of the matching files. When –print is used, '\n' will be the delimiting character for separating each file. Also, note that even if you omit -print, the find command will print the filenames by default.

The -print0 argument specifies each matching filename printed with the delimiting character '\0'. This is useful when a filename contains a space character.

Search based on filename or regular expression match
The -name argument specifies a matching string for the filename. We can pass wildcards as its argument text. The *.txt command matches all the filenames ending with .txt and prints them. The -print option prints the filenames or file paths in the terminal that matches the conditions (for example, -name) given as options to the find command.
$ find /home/slynux -name "*.txt" -print

The find command has an option -iname (ignore case), which is similar to -name but it matches filenames while ignoring the case.

For example:

```
$ ls
example.txt EXAMPLE.txt file.txt

$ find . -iname "example*" -print
./example.txt
./EXAMPLE.txt
```

If we want to match either of the multiple criteria, we can use OR conditions as shown in the following:

```
$ ls
new.txt some.jpg text.pdf
```

```
$ find . \( -name "*.txt" -o -name "*.pdf" \) -print
./text.pdf
./new.txt
```

The previous command will print all of the .txt and .pdf files, since the find command matches both .txt and .pdf files. **\(** and **\)** are used to treat -name "*.txt" -o -name "*.pdf" as a single unit.

The -path argument can be used to match the file path for files that match the wildcards.

-name always matches using the given filename. However, -path matches the file path as a whole. For example:

```
$ find /home/users -path "*/slynux/*" -print
```

This will match files as following paths.

```
/home/users/list/slynux.txt
/home/users/slynux/eg.css
```

Negating arguments

find can also exclude things that match a pattern using (**!**):

```
$ find . ! -name "*.txt" -print
```

This will match all the files whose names do not end in .txt. The following example shows the result of the command:

```
$ ls
list.txt new.PY new.txt next.jpg test.py
```

```
$ find . ! -name "*.txt" -print
.
./next.jpg
./test.py
./new.PY
```

Using xargs

We use pipes to redirect stdout (standard output) of a command to stdin (standard input) of another command. For example:

```
cat foo.txt | grep "test"
```

Some of the commands accept data as command-line arguments rather than a data stream through stdin (standard input). In that case, we cannot use pipes to supply data through command-line arguments.

We should try alternate methods. xargs is a command that is very helpful in handling standard input data to the command-line argument conversions. It can manipulate stdin and convert to command-line arguments for the specified command. Also, xargs can convert any one-line or multiple-line text inputs into other formats, such as multiple lines (specified number of columns) or a single line and vice versa.

The xargs command can supply arguments to a command by reformatting the data received through stdin. xargs can act as a substitute that can perform similar actions as the -exec argument in the case of the find command. Let's see a variety of hacks that can be performed using the xargs command.

- Converting multiple lines of input to a single-line output: Multiple-line input can be converted simply by removing the newline character and replacing with the " " (space) character. '\n' is interpreted as a newline character, which is the delimiter for the lines. By using xargs, we can ignore all the newlines with space so that multiple lines can be converted into a single-line text as follows:

```
$ cat example.txt # Example file
1 2 3 4 5 6
7 8 9 10
11 12
```

```
$ cat example.txt | xargs
1 2 3 4 5 6 7 8 9 10 11 12
```

- Converting single-line into multiple-line output: Given a maximum number of arguments in a line = n, we can split any stdin (standard input) text into lines of n arguments each. An argument is a piece of a string delimited by " " (space). Space is the default delimiter. A single line can be split into multiple lines as follows:

```
$ cat example.txt | xargs -n 3
1 2 3
4 5 6
7 8 9
10 11 12
```

Finding and deleting duplicate files

Duplicate files are copies of the same files. In some circumstances, we may need to remove duplicate files and keep a single copy of them. Identification of duplicate files by looking at the file content is an interesting task. It can be done using a combination of shell utilities.

We can identify the duplicate files by comparing file content. Checksums are ideal for this task, since files with exactly the same content will produce the same checksum values.

We can use this fact to remove duplicate files.

1. Generate some test files as follows:

```
$ echo "hello" > test ; cp test test_copy1 ; cp test test_copy2;
$ echo "next" > other;
# test_copy1 and test_copy2 are copy of test
```

2. The code for the script to remove the duplicate files is as follows:

```
#!/bin/bash
#Filename: remove_duplicates.sh
#Description: Find and remove duplicate files and keep one sample of each file.
ls -lS --time-style=long-iso | awk 'BEGIN {
    getline; getline;
    name1=$8; size=$5
}
{
    name2=$8;
    if (size==$5)
    {
    "md5sum "name1 | getline; csum1=$1;
    "md5sum "name2 | getline; csum2=$1;
      if ( csum1==csum2 )
      {
        print name1; print name2
      }
     };

     size=$5; name1=name2;
}' | sort -u > duplicate_files
cat duplicate_files | xargs -I {} md5sum {} | sort | uniq -w 32 |
awk '{ print "^"$2"$" }' | sort -u > duplicate_sample
echo Removing..

comm duplicate_files duplicate_sample -2 -3 | tee /dev/stderr | xargs rm
echo Removed duplicates files successfully.
```

3. Run it as:

```
$ ./remove_duplicates.sh
```

Working with file permissions, ownership, and the sticky bit

In Linux systems, each file is associated with many types of permissions. Out of these permissions, three sets of permissions (user, group, and others) are commonly manipulated.

The user is the owner of the file. The group is the collection of users (as defined by the system administrator) that are permitted some access to the file. Others are any entities other than the user or group owner of the file.

Permissions of a file can be listed by using the ls -l command:

```
-rw-r--r--    1    slynux slynux  2497    2010-02-28 11:22    bot.py
drwxr-xr-x    2    slynux slynux  4096    2010-05-27 14:31    a.py
-rw-r--r--    1    slynux slynux  539     2010-02-10 09:11    cl.pl
```

The first column of the output specifies the following, with the first letter corresponding to:

```
- – if it is a regular file
d – if it is a directory
c – for a character device
b – for a block device
l – if it is a symbolic link
s – for a socket
p – for a pipe
```

Let's take a look at what each of these three character sets mean for the user, group, and others:

- User (permission string: rwx------): The first letter in the three letters specifies whether the user has read permission for the file. If the read permission is set for the user, the character r will appear as the first character. Similarly, the second character specifies write (modify) permission (w) and the third character specifies whether the user has execute (x) permission (the permission to run the file). The execute permission is usually set for executable files. The user has one more special permission called setuid (S), which appears in the position of execute (x). The setuid permission enables an executable file to be executed effectively as its owner, even when the executable is run by another user. An example for a file with setuid permission set is -rwS------.

- Group (permission string: ---rwx---): The second set of three characters specifies the group permissions. The interpretation of permissions rwx is the same as the permissions for the user. Instead of setuid, the group has a setgid (S) bit. This enables the item to run an executable file with an effective group as the owner group. But the group, which initiates the command, may be different. An example of group permission is ----rwS---.

- Others (permission string: ------rwx): Other permissions appear as the last three character set in the permission string. Others have the same read, write, and execute permissions as the user and group. But it does not have permission S (such as setuid or setgid).

Directories have a special permission called a sticky bit. When a sticky bit is set for a directory, only the user who created the directory can delete the files in the directory, even if the group and others have write permissions. The sticky bit appears in the position of execute character (x) in the others permission set. It is represented as character t or T. The

t character appears in the position of x if the execute permission is unset and the sticky bit is set. If the sticky bit and the execute permission are set, the character T appears in the position of x. For example:
------rwt , ------rwT

A typical example of a directory with sticky bit turned on is /tmp. In each of the ls -l output lines, the string slynux slynux corresponds to the owned user and owned group. Here, the first slynux is the user and the second slynux is the group owner.

In order to set permissions for files, we use the chmod command. Assume that we need to set permission: rwx rw- r—.

This could be set using chmod as follows:

```
$ chmod u=rwx g=rw o=r filename
```

Here:

u – specifies user permissions
g – specifies group permissions
o – specifies others permissions

Use + to add permission to a user, group, or others and use - to remove the permissions.

Add the executable permission to a file, which is already having the permission rwx rw- r— as follows:
$ chmod o+x filename

This command adds the x permission for others.

Add the executable permission to all permission categories, that is, for user, group, and others as follows:
$ chmod a+x filename

Here a means all.

In order to remove a permission, use -. For example:
$ chmod a-x filename

Alternatively, permissions can also be denoted by three-digit octal numbers in which each of the digits corresponds to user, group, and other in that order. Read, write, and execute permissions have unique octal numbers as follows:

r-- = 4
-w- = 2
--x = 1

We can get the required combination of permissions by adding the octal values for the required permission sets. For example:

rw- = 4 + 2 = 6
r-x = 4 + 1 = 5

The permission rwx rw- r-- in the numeric method is as follows:

rwx = 4 + 2 + 1 = 7
rw- = 4 + 2 = 6
r-- = 4

Therefore, rwx rw- r-- is equal to 764, and the command for setting the permissions using octal values is:

```
$ chmod 764 filename
```

Changing ownership

In order to change ownership of files, use the chown command as follows:

```
$ chown user.group filename
```

For example:

```
$ chown slynux.slynux test.sh
```

Here, slynux is the user, as well as the group.

Printing the directory tree

The following is a sample Unix filesystem tree to show an example:

```
$ tree ~/unixfs
unixfs/
|-- bin
| |-- cat
| `-- ls
|-- etc
| `-- passwd
|-- home
| |-- pactpub
| | |-- automate.sh
| | `-- schedule
| `-- slynux
|-- opt
|-- tmp
`-- usr
8 directories, 5 files
```

HTML output for tree

It is possible to generate an HTML output from the tree command. For example, use the following command to create an HTML file with the tree output:

```
$ tree PATH -H http://localhost -o out.html
```

Replace http://localhost with the URL where you are planning to host the file. Replace PATH with a real path for the base directory. For the current directory use . as PATH.

Chapter 4: Text Manipulation

Searching and mining a text inside a file with grep

The grep command is the magic Unix utility for searching in text. It accepts regular expressions, and can produce output in various formats. Additionally, it has numerous interesting options. Let's see how to use them:

1. To search for lines of text that contain the given pattern:

```
$ grep pattern filename
this is the line containing pattern
```

Or:

```
$ grep "pattern" filename
this is the line containing pattern
```

2. We can also read from stdin as follows:

```
$ echo -e "this is a word\nnext line" | grep word
this is a word
```

3. Perform a search in multiple files by using a single grep invocation, as follows:

```
$ grep "match_text" file1 file2 file3 ...
```

4. We can highlight the word in the line by using the --color option as follows:

```
$ grep word filename --color=auto
this is the line containing word
```

5. Usually, the grep command only interprets some of the special characters in match_text. To use the full set of regular expressions as input arguments, the –E option should be added, which means an extended regular expression. Or, we can use an extended regular expression enabled grep command, egrep. For example:

```
$ grep -E "[a-z]+" filename
```

Or:

```
$ egrep "[a-z]+" filename
```

6. In order to output only the matching portion of a text in a file, use the -o option as follows:

```
$ echo this is a line. | egrep -o "[a-z]+\."
line.
```

7. In order to print all of the lines, except the line containing match_pattern, use:

```
$ grep -v match_pattern file
The -v option added to grep inverts the match results.
```

8. Count the number of lines in which a matching string or regex match appears in a file or text, as follows:

```
$ grep -c "text" filename
10
```

It should be noted that -c counts only the number of matching lines, not the number of times a match is made. For example:

```
$ echo -e "1 2 3 4\nhello\n5 6" | egrep -c "[0-9]"
2
```

Even though there are six matching items, it prints 2, since there are only two matching lines. Multiple matches in a single line are counted only once.

9. To count the number of matching items in a file, use the following trick:

```
$ echo -e "1 2 3 4\nhello\n5 6" | egrep -o "[0-9]" | wc -l
6
```

10. Print the line number of the match string as follows:

```
$ cat sample1.txt
gnu is not unix
linux is fun
bash is art

$ cat sample2.txt
planetlinux

$ grep linux -n sample1.txt
2:linux is fun
```

or

```
$ cat sample1.txt | grep linux -n
```

If multiple files are used, it will also print the filename with the result as follows:

```
$ grep linux -n sample1.txt sample2.txt
sample1.txt:2:linux is fun
sample2.txt:2:planetlinux
```

11. Print the character or byte offset at which a pattern matches, as follows:

```
$ echo gnu is not unix | grep -b -o "not"
7:not
```

The character offset for a string in a line is a counter from 0, starting with the first character. In the preceding example, not is at the seventh offset position (that is, not starts from the seventh character in the line; that is, gnu is not unix).

The -b option is always used with -o.

12. To search over multiple files, and list which files contain the pattern, we use the following:

```
$ grep -l linux sample1.txt sample2.txt
sample1.txt
sample2.txt
```

The inverse of the -l argument is -L. The -L argument returns a list of non-matching files.

Using sed to perform text replacement

sed stands for stream editor. It is a very essential tool for text processing, and a marvelous utility to play around with regular expressions. A well-known usage of the sed command is for text replacement.

1. It can be matched using regular expressions.

```
$ sed 's/pattern/replace_string/' file
```

Or:

```
$ cat file | sed 's/pattern/replace_string/'
```

This command reads from stdin.

2. By default, sed only prints the substituted text. To save the changes along with the substitutions to the same file, use the -i option. Most of the users follow multiple redirections to save the file after making a replacement as follows:

```
$ sed 's/text/replace/' file >newfile
$ mv newfile file
```

However, it can be done in just one line; for example:

```
$ sed -i 's/text/replace/' file
```

3. These usages of the sed command will replace the first occurrence of the pattern in each line. If we want to replace every occurrence, we need to add the g parameter at the end, as follows:

```
$ sed 's/pattern/replace_string/g' file
```

The /g suffix means that it will substitute every occurrence. However, we sometimes need to replace only the Nth occurrence onwards. For this, we can use the /Ng form of the option.

Have a look at the following commands:

```
$ echo thisthisthisthis | sed 's/this/THIS/2g'
thisTHISTHISTHIS
```

```
$ echo thisthisthisthis | sed 's/this/THIS/3g'
thisthisTHISTHIS
```

```
$ echo thisthisthisthis | sed 's/this/THIS/4g'
thisthisthisTHIS
```

We have used / in sed as a delimiter character. We can use any delimiter characters as follows:

```
sed 's:text:replace:g'
sed 's|text|replace|g'
```

When the delimiter character appears inside the pattern, we have to escape it using the \ prefix, as follows:

```
sed 's|te\|xt|replace|g'
```

\| is a delimiter appearing in the pattern replaced with escape.

Removing a sentence in a file containing a word

Let's create a file with some text to carry out the substitutions. For example:

```
$ cat sentence.txt

Linux refers to the family of Unix-like computer operating systems that use the Linux kernel. Linux can be installed on a wide variety of computer hardware, ranging from mobile phones, tablet computers and video game consoles, to mainframes and supercomputers. Linux is predominantly known for its use in servers.
```

We will remove the sentence containing the words mobile phones. Use the following sed expression for this task:

```
$ sed 's/ [^.]*mobile phones[^.]*\.//g' sentence.txt

Linux refers to the family of Unix-like computer operating systems that use the Linux kernel. Linux is predominantly known for its use in servers.
```

Parsing e-mail addresses and URLs from a text

Parsing a required text from a given file is a common task that we encounter in text processing. Items such as, e-mails and URLs can be found out with the help of correct regex sequences. Mostly, we need to parse e-mail addresses from a contact list of an e-mail client, which is composed of many unwanted characters and words, or from an HTML web page.

The regular expression pattern to match an e-mail address is as follows:

```
[A-Za-z0-9._]+@[A-Za-z0-9.]+\.[a-zA-Z]{2,4}
```

For example:

```
$ cat url_email.txt
this is a line of text contains,<email> #slynux@slynux.com. </email> and email address, blog
"http://www.google.com", test@yahoo.com dfdfdfdddfdf; cool.hacks@gmail.com <br /> <a
href="http://code.google.com"><h1>Heading</h1>
```

As we are using extended regular expressions (+, for instance), we should use egrep.

```
$ egrep -o '[A-Za-z0-9._]+@[A-Za-z0-9.]+\.[a-zA-Z]{2,4}' url_email.txt
slynux@slynux.com
test@yahoo.com
cool.hacks@gmail.com
```

The egrep regex pattern for an HTTP URL is as follows:

```
http://[a-zA-Z0-9\-\.]+\.[a-zA-Z]{2,4}
```

For example:

```
$ egrep -o "http://[a-zA-Z0-9.]+\.[a-zA-Z]{2,3}" url_email.txt
http://www.google.com
http://code.google.com
```

Chapter 5: Compressing, Archiving and Backup

Archiving and compressing with tar

The tar command can be used to archive files, originally designed for storing data on Tape archives. It allows you to store multiple files and directories as a single file while retaining all the file attributes, such as owner, permissions, and so on. The file created by the tar command is often referred to as a tarball.

1. To archive files with tar, use the following syntax:

```
$ tar -cf output.tar [SOURCES]
```

For example:

```
$ tar -cf output.tar file1 file2 file3 folder1 ..
```

2. To list files in an archive, use the -t option:

```
$ tar -tf archive.tar
file1
file2
```

Appending files to an archive

In order to append a file into an already existing archive use:

```
$ tar -rvf original.tar new_file
```

Let's create an archive with one text file in it:

```
$ tar -cf archive.tar hello.txt
```

To list the files present in the archive, use:

```
$ tar -tf archive.tar
hello.txt
```

Now add another file to the archive and list its contents again:

```
$ tar -rf archive.tar world.txt
$ tar -tf archive.tar
hello.txt
world.txt
```

The archive now contains both the files.

Extracting files and folders from an archive

The following command extracts the contents of the archive to the current directory:

```
$ tar -xf archive.tar
```

The -x option stands for extract.

When -x is used, the tar command extracts the contents of the archive to the current directory. We can also specify the directory where the files need to be extracted by using the -C flag, as follows:

```
$ tar -xf archive.tar -C /path/to/extraction_directory
```

The command extracts the contents of an archive to a specified directory. It extracts the entire contents of the archive. We can also extract only a few files by specifying them as command arguments:

```
$ tar -xvf file.tar file1 file4
```

The command above extracts only file1 and file4, and ignores other files in the archive.

Concatenating two archives

We can easily merge multiple tar files with the -A option.
Let's pretend we have two tarballs: file1.tar and file2.tar. We can merge the contents of file2.tar to file1.tar as follows:

```
$ tar -Af file1.tar file2.tar
```

Verify it by listing the contents:

```
$ tar -tvf file1.tar
```

Deleting files from the archive

We can remove files from a given archive using the -delete option. For example:

```
$ tar -f archive.tar --delete file1 file2 ..
```

Or,

```
$ tar --delete --file archive.tar [FILE LIST]
```

Let's see an example:

```
$ tar -tf archive.tar
filea
fileb
filec
```

Now let's delete filea:

```
$ tar --delete --file archive.tar filea
$ tar -tf archive.tar
fileb
filec
```

Compression with the tar archive

The tar command only archives files, it does not compress them. For this reason, most people usually add some form of compression when working with tarballs. This can significantly decrease the size of the files. Tarballs are often compressed into one of the following formats:

```
file.tar.gz
file.tar.bz2
file.tar.lzma
```

Different tar flags are used to specify different compression formats:

```
-j for bunzip2
-z for gzip
--lzma for lzma
```

It is possible to use compression formats without explicitly specifying special options as above. tar can compress by looking at the given extension of the output or input file names.

In order for tar to support compression automatically by looking at the extensions, use -a or --auto-compress with tar:

```
$ tar acvf archive.tar.gz filea fileb filec
filea
fileb
filec
```

```
$ tar tf archive.tar.gz
filea
fileb
filec
```

Compressing tarball with Gzip

A gzipped tarball is basically a tar archive compressed using gzip. We can use two methods to create such tarballs:

1. The first method is as follows:

```
$ tar -czvvf archive.tar.gz [FILES]
```

or

```
$ tar -cavvf archive.tar.gz [FILES]
```

The -a option specifies that the compression format should automatically be detected from the extension.

2. Alternatively, here's the second method:

First, create a tarball:

```
$ tar -cvvf archive.tar [FILES]
```

Compress the tarball as follows:

```
$ gzip archive.tar
```

Archiving and compressing with zip

ZIP is a popular compression format used on many platforms. It isn't as commonly used as gzip or bzip2 on Linux platforms, but files from the Internet are often saved in this format.

Let's see how to use various options with zip:

1. In order to archive with ZIP, the following syntax is used:

```
$ zip archive_name.zip [SOURCE FILES/DIRS]
```

For example:

```
$ zip file.zip file
```

Here, the file.zip file will be produced.

2. Archive directories and files recursively as follows:

```
$ zip -r archive.zip folder1 folder2
```

In this command, -r is used for specifying recursive.

3. In order to extract files and folders in a ZIP file, use:

```
$ unzip file.zip
```

It will extract the files without removing filename.zip (unlike unlzma or gunzip).

- In order to update files in the archive with newer files in the filesystem, use the-u flag:

```
$ zip file.zip -u newfile
```

- Delete a file from a zipped archive, by using -d as follows:

```
$ zip -d arc.zip file.txt
```

- In order to list the files in an archive use:

```
$ unzip -l archive.zip
```

Creating entire disk images with fsarchiver

fsarchiver is a tool which can save the contents of a complete filesystem to a compressed archive file. Due to these abilities, it is one of the most complete and easy to use tools for backup.

1. Creating a backup of a filesystem/partition
Use the savefs option of fsarchiver like this:

```
fsarchiver savefs backup.fsa /dev/sda1
```

where backup.fsa is the final backup file and /dev/sda1 is the partition to backup

2. Backup more than one partition at the same time
Use the savefs option as earlier and pass the partitions as the last parameters to fsarchiver:

```
fsarchiver savefs backup.fsa /dev/sda1 /dev/sda2
```

3. Restore a partition from a backup archive
Use the restfs option of fsarchiver like this:

```
fsarchiver restfs backup.fsa id=0,dest=/dev/sda1
```

id=0 denotes that we want to pick the first partition from the archive to the partition specified as dest=/dev/sda1

4. Restore multiple partitions from a backup archive
As earlier, use the restfs option as follows:

```
fsarchiver restfs backup.fsa id=0,dest=/dev/sda1 id=1,dest=/dev/sdb1
```

Here, we use two sets of the id,dest parameter to tell fsarchiver to restore the first two partitions from the backup to two physical partitions.

Very similar to the way tar works, fsarchiver goes through the filesystem to create a list of files and then saves them to a compressed archive file. The advantage here is that unlike tar which only saves information about the files, fsarchiver performs a backup of the filesystem as well. This means that it is easier to restore the backup on a fresh system as it is not necessary to recreate the filesystem.

If you are seeing the /dev/sda1 notation for partitions for the first time, this deserves some explanation. /dev in Linux holds special files called device files which refer to a physical device. The sd in sda1 refers to SATA disk, the next letter can be a, b, c and so on, followed by the partition number.

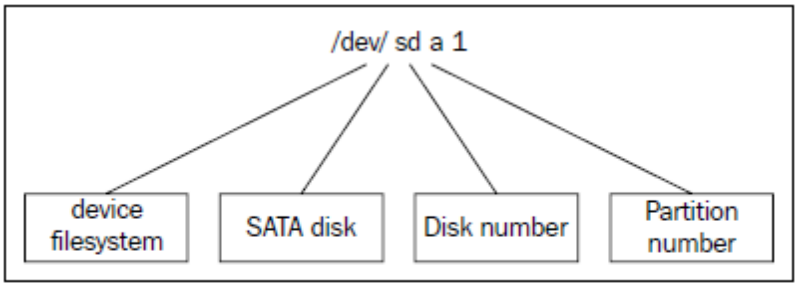

Diagram showing the various parts of a disk device's filename in Linux

Chapter 6: Shell Scripting for Network

Setting up the network

A network interface is used to connect a machine to a network. Usually, Linux denotes network interfaces using names like eth0, eth1 (referring to Ethernet interfaces). Other interfaces, such as usb0, wlan0, and so on are available for USB network interfaces, wireless LAN respectively.

We will use these commands: ifconfig, route, nslookup, and host.

ifconfig is the command that is used to configure and display details about network interfaces, subnet mask, and so on. On a typical system, it should be available at /sbin/ifconfig.

1. List the current network interface configuration:
$ ifconfig

2. In order to manually set the IP address for a network interface, use:

```
# ifconfig wlan0 192.168.0.80
```

You will need to run the preceding command .as root. 192.168.0.80 is the address to be set, Set the subnet mask along with the IP address as follows:

```
# ifconfig wlan0 192.168.0.80 netmask 255.255.252.0
```

3. Automatically configure network interfaces. If you are connecting to, let's say a wired network which supports automatically assigning IPs, just use this to configure the network interface:

```
# dhclient eth0
```

Printing the list of network interfaces

Here is a one-line command sequence to print the list of network interfaces available on a system:

```
$ ifconfig | cut -c-10 | tr -d ' ' | tr -s '\n'
lo
wlan0
```

The first 10 characters of each line in ifconfig output is reserved for writing names of network interfaces. Hence, we use cut to extract the first 10 characters of each line. tr –d ' ' deletes every space character in each line. Now, the \n newline character is squeezed using tr -s '\n' to produce a list of interface names.

Displaying IP addresses

The ifconfig command displays details of every active network interface available on the system. However, we can restrict it to a specific interface using:

```
$ ifconfig iface_name
```

For example:

```
$ ifconfig wlan0
```

In several scripting contexts, we may need to extract any of these addresses from the script for further manipulations. Extracting the IP address is a frequently needed task. In order to extract the IP address from the ifconfig output use:

```
$ ifconfig wlan0 | egrep -o "inet addr:[^ ]*" | grep -o "[0-9.]*"
192.168.0.82
```

Here ,the first command egrep -o "inet addr:[^]*" will print inet addr:192.168.0.82. The pattern starts with inet addr: and ends with some non-space character sequence (specified by [^]*). Now in the next pipe, it prints the character combination of digits and '.'.

Name server and DNS (Domain Name Service)

Name servers assigned to the current system can be viewed by reading /etc/resolv.conf, for example:

```
$ cat /etc/resolv.conf
nameserver 8.8.8.8
```

We can add name servers manually as follows:

```
# echo nameserver IP_ADDRESS >> /etc/resolv.conf
```

PING tests

```
$ ping ADDRESS
```

The ADDRESS can be a hostname, domain name, or an IP address itself.

Limiting the number of packets to be sent

```
$ ping 192.168.0.1 -c 2
```

Return status of the ping command

The ping command returns exit status 0 when it succeeds and returns non-zero when it fails. Successful means destination host is reachable, whereas Failure is when the destination host is unreachable.

The return status can be easily obtained as follows:

```
$ ping domain -c2
if [ $? -eq 0 ];
then
echo Successful ;
else
echo Failure
fi
```

Listing all the machines alive on a network

When we deal with a large local area network, we may need to check the availability of other machines in the network. A machine may not be alive in two conditions: either it is not powered on, or due to a problem in the network. By using shell scripting, we can easily find out and report which machines are alive on the network.

1. The first method is as follows:
We can write our own script using the ping command to query a list of IP addresses and check whether they are alive or not as follows:

```
#!/bin/bash
#Filename: ping.sh
# Change base address 192.168.0 according to your network.
for ip in 192.168.0.{1..255} ;
do
ping $ip -c 2 &> /dev/null ;
if [ $? -eq 0 ];
then
echo $ip is alive
fi
done
```

The output is as follows:

```
$ ./ping.sh
192.168.0.1 is alive
192.168.0.90 is alive
```

2. Using fping, the second method is as follows:
We can use an existing command-line utility to query the status of machines on a network as follows:

```
$ fping -a 192.160.1/24 -g 2> /dev/null
192.168.0.1
192.168.0.90
```

Or, use:

```
$ fping -a 192.168.0.1 192.168.0.255 -g
```

Using fping

The second method uses a different command called fping. It can ping a list of IP addresses simultaneously and respond very quickly. The options available with fping are as follows:

- The -a option with fping specifies to print all alive machine's IP addresses
- The -u option with fping specifies to print all unreachable machines
- The -g option specifies to generate a range of IP addresses from slash-subnet mask notation specified as IP/mask or start and end IP addresses as:

```
$ fping -a 192.160.1/24 -g
```

Or

```
$ fping -a 192.160.1 192.168.0.255 -g
```

- 2>/dev/null is used to dump error messages printed due to an unreachable host to null device

It is also possible to manually specify a list of IP addresses as command-line arguments or as a list through stdin. For example:

```
$ fping -a 192.168.0.1 192.168.0.5 192.168.0.6
```

Passes IP address as arguments

```
$ fping -a < ip.list
```

Passes a list of IP addresses from a file

Transferring files through the network

The major driver for networking of computers is resource sharing, and file sharing is the most prominent shared resource. There are different methods by which we can transfer files between different nodes on a network. This recipe discusses how to transfer files using commonly used protocols FTP, SFTP, RSYNC, and SCP.

File Transfer Protocol (FTP) is a very old file transfer protocol for transferring files between machines on a network. We can use the command lftp for accessing FTP-enabled servers for file transfer. FTP can only be used if the FTP server is installed on the remote machine.

FTP is used in many public websites to share files and the service usually runs on port 21. To connect to an FTP server and transfer files in between, use:

```
$ lftp username@ftphost
```

It will prompt for a password and then display a logged in prompt as follows:

```
lftp username@ftphost:~>
```

You can type commands in this prompt. For example:

- To change to a directory, use cd directory
- To change the directory of a local machine, use lcd
- To create a directory use mkdir
- To list files in the current directory on the remote machine, use ls
- To download a file, use get filename as follows:

```
lftp username@ftphost:~> get filename
```

- To upload a file from the current directory, use put filename as follows:

```
lftp username@ftphost:~> put filename
```

- An lftp session can be terminated by using the quit command

Chapter 7: Shell Scripting and System Monitoring

Monitoring disk usage

Disk space is a limited resource. We frequently perform disk usage calculation on storage media (such as hard disks) to find out the free space available on them. When free space becomes scarce, we find out large files to be deleted or moved in order to create free space. In addition to this, disk usage manipulations are also used in shell scripting contexts.

- To find the disk space used by a file (or files), use:

```
$ du FILENAME1 FILENAME2 ..
```

For example:

```
$ du file.txt
4
```

- To obtain the disk usage for all files inside a directory along with the individual disk usage for each file showed in each line, use:

```
$ du -a DIRECTORY
```

-a outputs results for all files in the specified directory or directories recursively.

For example:

```
$ du -a test
4 test/output.txt
4 test/process_log.sh
4 test/pcpu.sh
16 test
```

An example of using du DIRECTORY is as follows:

```
$ du test
16 test
```

Displaying disk usage in KB, MB, or Blocks

By default, the disk usage command displays the total bytes used by a file. A more human-readable format is expressed in units such as KB, MB, or GB. In order to print the disk usage in a display-friendly format, use -h as follows:

```
du -h FILENAME
```

For example:

```
$ du -h test/pcpu.sh
4.0K test/pcpu.sh
# Multiple file arguments are accepted
```

Or

```
# du -h DIRECTORY
$ du -h hack/
16K hack/
```

Displaying the grand total sum of disk usage

If we need to calculate the total size taken by all the files or directories, displaying individual file sizes won't help. du has an option -c such that it will output the total disk usage of all files and directories given as an argument. It appends a line SIZE total with the result. The syntax is as follows:

```
$ du -c FILENAME1 FILENAME2..
```

For example:

```
du -c process_log.shpcpu.sh
4 process_log.sh
4 pcpu.sh
8 total
```

Or

```
$ du -c DIRECTORY
```

For example:

```
$ du -c test/
16 test/
16 total
```

Or

```
$ du -c *.txt
```

Wildcards

-c can be used along with other options like -a and -h, in which case they will produce their usual output with an extra line containing the total size.

There is another option -s (summarize), which will print only the grand total as the output. It will print the total sum, and the flag -h can be used along with it to print in human-readable format. This combination has frequent use in practice:

```
$ du -s FILES(s)
$ du -sh DIRECTORY
```

For example:

```
$ du -sh slynux
680K slynux
```

Finding the 10 largest size files from a given directory

Finding large files is a task we come across regularly so that we can delete or move them. We can easily find out such files using du and sort commands like this:

```
$ du -ak SOURCE_DIR | sort -nrk 1 | head
```

Here, -a makes du traverse the SOURCE_DIR and calculates the size of all files and directories. The first column of the output contains the size in kilobytes since -k is specified, and the second column contains the file or folder name.

sort is used to perform a numerical sort with column 1 and reverse it. head is used to parse the first 10 lines from the output. For example:

```
$ du -ak /home/slynux | sort -nrk 1 | head -n 4
50220 /home/slynux
43296 /home/slynux/.mozilla
43284 /home/slynux/.mozilla/firefox
43276 /home/slynux/.mozilla/firefox/8c22khxc.default
```

One of the drawbacks of the preceding one-liner is that it includes directories in the result.

However, when we need to find only the largest files and not directories, we can improve the one-liner to output only the large files as follows:

```
$ find . -type f -exec du -k {} \; | sort -nrk 1 | head
```

Monitoring user logins to find intruders

Let's write the intruder detection script that can generate a report to intruders by using an authentication log file as follows:

```
#!/bin/bash
#Filename: intruder_detect.sh
#Description: Intruder reporting tool with auth.log input
AUTHLOG=/var/log/auth.log
if [[ -n $1 ]];
then
AUTHLOG=$1
echo Using Log file : $AUTHLOG
fi

LOG=/tmp/valid.$$.log
grep -v "invalid" $AUTHLOG > $LOG
users=$(grep "Failed password" $LOG | awk '{ print $(NF-5) }' | sort | uniq)

printf "%-5s|%-10s|%-10s|%-13s|%-33s|%s\n" "Sr#" "User" "Attempts" "IP
address" "Host_Mapping" "Time range"
```

```
ucount=0;
ip_list="$(egrep -o "[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+" $LOG | sort | uniq)"

for ip in $ip_list;
  do
  grep $ip $LOG > /tmp/temp.$$.log
  for user in $users;
    do
    grep $user /tmp/temp.$$.log> /tmp/$$.log
    cut -c-16 /tmp/$$.log > $$.time
    tstart=$(head -1 $$.time);
    start=$(date -d "$tstart" "+%s");
    tend=$(tail -1 $$.time);
    end=$(date -d "$tend" "+%s")
    limit=$(( $end - $start ))
    if [ $limit -gt 120 ];
    then
      let ucount++;
      IP=$(egrep -o "[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+" /tmp/$$.log | head -1 );
      TIME_RANGE="$tstart-->$tend"
      ATTEMPTS=$(cat /tmp/$$.log|wc -l);
      HOST=$(host $IP | awk '{ print $NF }' )
      printf "%-5s|%-10s|%-10s|%-10s|%-33s|%-s\n" "$ucount" "$user"
      "$ATTEMPTS" "$IP" "$HOST" "$TIME_RANGE";
    fi

done
done
rm /tmp/valid.$$.log /tmp/$$.log $$.time /tmp/temp.$$.log 2> /dev/null
```

A sample output is as follows:

```
slynux@slynux-laptop:~$ ./intruder_detect.sh sampleauth.log
Using Log file : sampleauth.log

Sr#  |User   |Attempts|IP address     |Host_Mapping |Time range
1    |alice  |3       |203.110.250.34|attk1.foo.com|Oct 29 05:28:59 -->Oct 29 05:31:59
2    |bob1   |3       |203.110.251.31|attk2.foo.com|Oct 29 05:21:52 -->Oct 29 05:29:52
3    |bob2   |3       |203.110.250.34|attk1.foo.com|Oct 29 05:22:59 -->Oct 29 05:25:52
4    |gvraju |20      |203.110.251.31|attk2.foo.com|Oct 28 04:37:10 -->Oct 29 05:19:09
5    |root   |21      |203.110.253.32|attk3.foo.com|Oct 29 05:18:01 -->Oct 29 05:37:01
```

Checking disks and filesystems for errors

Data is the most important thing in any computer system. Naturally, it is important to monitor the consistency of data stored on physical media.

Let us see how to use fsck with its various options to check filesystems for errors, and optionally fix them.
1. To check for errors on a partition or filesystem, just pass its path to fsck:

```
# fsck /dev/sdb3
fsck from util-linux 2.20.1
e2fsck 1.42.5 (29-Jul-2012)
HDD2 has been mounted 26 times without being checked,
check forced.
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
HDD2: 75540/16138240 files (0.7% non-contiguous),
48756390/64529088 blocks
```

2. To check all the filesystems configured in /etc/fstab, we can use the following syntax:

```
# fsck -A
```

This will go through the /etc/fstab file sequentially, checking each of the filesystems one-by-one. The fstab file basically configures a mapping between disks and mount points which makes it easy to mount filesystems. This also makes it possible to mount certain filesystems during boot.

3. Instruct fsck to automatically attempt fixing errors, instead of interactively asking us whether or not to repair, we can use this form of fsck:

```
# fsck -a /dev/sda2
```

4. To simulate the actions, fsck is going to perform:

```
# fsck -AN
fsck from util-linux 2.20.1
[/sbin/fsck.ext4 (1) -- /] fsck.ext4 /dev/sda8
[/sbin/fsck.ext4 (1) -- /home] fsck.ext4 /dev/sda7
[/sbin/fsck.ext3 (1) -- /media/Data] fsck.ext3 /dev/sda6
```

This will print information on what actions will be performed, which is checking all the filesystems.

Chapter 8: System Administration with Shell Scripts

Gathering information about processes

Processes are the running instance of a program. Several processes run on a computer, and each process is assigned a unique identification number called a process ID (PID). Multiple instances of the same program with the same name can be executed at the same time, but they all will have different PIDs. A process consists of several attributes, such as which user owns the process, the amount of memory used by the program, CPU time used by the program, and so on.

ps is an important tool for gathering information about the processes. It provides information on which user owns the process, the time when a process started, the command path used for executing the process, PID, the terminal it is attached with (TTY), the memory used by the process, CPU time used by the process, and so on. For example:

```
$ ps
PID TTY TIME CMD
1220 pts/0 00:00:00 bash
1242 pts/0 00:00:00 ps
```

The ps command is usually used with a set of parameters. When it is run without any parameter, ps will display processes that are running on the current terminal (TTY). The first column shows the process ID (PID), the second column is the TTY (terminal), the third column is how much time has elapsed since the process started, and finally CMD (the command). In order to show more columns consisting of more information, use -f (stands for full) as follows:

```
$ ps -f
UID PID PPID C STIME TTY TIME CMD
slynux 1220 1219 0 18:18 pts/0 00:00:00 -bash
slynux 1587 1220 0 18:59 pts/0 00:00:00 ps -f
```

The preceding ps commands are not useful, as it does not provide any information about processes other than the ones attached to the current terminal. In order to get information about every process running on the system, add the -e (every) option. The -ax (all) option will also produce an identical output.

The -x argument along with -a specifies to remove the default TTY restriction imparted by ps. Usually, using ps without arguments prints processes that are attached to the terminal only.

Run one of these commands: ps -e, ps -ef, ps -ax, or ps -axf.

```
$ ps -e | head
PID TTY TIME CMD
1 ? 00:00:00 init
2 ? 00:00:00 kthreadd
3 ? 00:00:00 migration/0
4 ? 00:00:00 ksoftirqd/0
5 ? 00:00:00 watchdog/0
6 ? 00:00:00 events/0
7 ? 00:00:00 cpuset
8 ? 00:00:00 khelper
9 ? 00:00:00 netns
```

It will be a long list. The example filters the output using head, so we only get the first 10 entries.

An example is as follows. Here, comm stands for COMMAND and pcpu is percent of CPU usage:

```
$ ps -eo comm,pcpu | head

COMMAND     %CPU
init        0.0
kthreadd    0.0
migration/0 0.0
ksoftirqd/0 0.0
watchdog/0  0.0
events/0    0.0
cpuset      0.0
khelper     0.0
netns       0.0
```

The different parameters that can be used with the -o option and their descriptions are as follows:

Parameter	Description
pcpu	Percentage of CPU
pid	Process ID
ppid	Parent Process ID
pmem	Percentage of memory
comm	Executable filename
cmd	Simple command
user	The user who started the process
nice	The priority (niceness)
time	Cumulative CPU time
etime	Elapsed time since the process started
tty	The associated TTY device
euid	The effective user
stat	Process state

top

top is a very important command for system administrators. The top command will, by default, output a list of top CPU consuming processes. The output is updated every few seconds, and is used as follows:

```
$ top
```

Sorting the ps output with respect to a parameter

Output of the ps command can be sorted according to specified columns with the –sort parameter. The ascending or descending order can be specified by using the + (ascending) or - (descending) prefix to the parameter as follows:

```
$ ps [OPTIONS] --sort -paramter1,+parameter2,parameter3..
```

For example, to list the top 10 CPU consuming processes, use:

```
$ ps -eo comm,pcpu --sort -pcpu | head
COMMAND         %CPU
Xorg            0.1
hald-addon-stor 0.0
ata/0           0.0
scsi_eh_0       0.0
gnome-settings- 0.0
```

```
init              0.0
hald              0.0
pulseaudio        0.0
gdm-simple-gree   0.0
```

Here, processes are sorted in the descending order by percentage of CPU usage, and head is applied to extract the top 10 processes.

We can use grep to extract entries in the ps output related to a given process name or another parameter. In order to find out entries about running Bash processes, use:

```
$ ps -eo comm,pid,pcpu,pmem | grep bash
bash 1255 0.0 0.3
bash 1680 5.5 0.3
```

Finding the process ID when given command names

Suppose several instances of a command are being executed, we may need to identify the PID of the processes. This information can be found by using the ps or the pgrep command. We can use ps as follows:

```
$ ps -C COMMAND_NAME
```

Or

```
$ ps -C COMMAND_NAME -o pid=
```

The -o user-defined format specifier was described in the earlier part of the recipe. But here, you can see = appended with pid. This is to remove the header PID in the output of ps. In order to remove headers for each column, append = to the parameter. For example:

```
$ ps -C bash -o pid=
1255
1680
```

This command lists the process IDs of Bash processes.

Alternately, there is a handy command called pgrep. You should use pgrep to get a quick list of process IDs for a particular command. For example:

```
$ pgrep COMMAND
$ pgrep bash
1255
1680
```

Killing processes

Termination of processes is an important task we always come across, including the need to terminate all the instances of a program. The command line provides several options for terminating programs. An important concept regarding processes in Unix-like environments is that of signals. Signals are an inter-process communication mechanism used to interrupt a running process to perform some action. Termination of a program is also performed by using the same technique.

1. In order to list all the signals available, use:

```
$ kill -l
```

It will print signal numbers and corresponding signal names.

2. Terminate a process as follows:

```
$ kill PROCESS_ID_LIST
```

The kill command issues a TERM signal by default. The process ID list is to be specified with space as a delimiter between process IDs.

3. In order to specify a signal to be sent to a process via the kill command, use:

```
$ kill -s SIGNAL PID
```

The SIGNAL argument is either a signal name or a signal number. Though there are many signals specified for different purposes, we frequently use only a few signals.

They are as follows:

SIGHUP	**1: Hangup detection on death of the controlling process or terminal**
SIGINT	**2: Signal which is emitted when Ctrl + C is pressed**
SIGKILL	**9: Signal used to force kill the process**
SIGTERM	**15: Signal used to terminate a process by default**
SIGTSTP	**20: Signal emitted when Ctrl + Z is pressed**

4. We frequently use force kill for processes. In order to force kill a process, use:

```
$ kill -s SIGKILL PROCESS_ID
```

Or:

```
$ kill -9 PROCESS_ID
```

There are also a few other commands in the kill family that accept the command name as the argument and send a signal to the process.

The killall command terminates the process by name as follows:

```
$ killall process_name
```

In order to send a signal to a process by name, use:

```
$ killall -s SIGNAL process_name
```

In order to force kill a process by name, use:

```
$ killall -9 process_name
```

For example:

```
$ killall -9 gedit
```

Specify the process by name, which is specified by users who own it, by using:

```
$ killall -u USERNAME process_name
```

If you want killall to interactively confirm before killing processes, use the -i argument.

The pkill command is similar to the kill command but it, by default, accepts a process name instead of a process ID. For example:

```
$ pkill process_name
$ pkill -s SIGNAL process_name
```

SIGNAL is the signal number. The SIGNAL name is not supported with pkill. It provides many of the same options that the kill command does. Check the pkill man pages for more details.

Gathering system information

1. In order to print the hostname of the current system, use:

```
$ hostname
```

Or:

```
$ uname -n
```

2. Print long details about the Linux kernel version, hardware architecture, and more by using:

```
$ uname -a
```

3. In order to print the kernel release, use:

```
$ uname -r
```

4. Print the machine type as follows:

```
$ uname -m
```

5. In order to print details about the CPU, use:

```
$ cat /proc/cpuinfo
```

In order to extract the processor name, use:

```
$ cat /proc/cpuinfo | sed -n 5p
```

The fifth line contains the processor name.

6. Print details about the memory or RAM as follows:

```
$ cat /proc/meminfo
```

Print the total memory (RAM) available on the system as follows:

```
$ cat /proc/meminfo | head -1
MemTotal: 1026096 kB
```

7. In order to list out the partitions information available on the system, use:

```
$ cat /proc/partitions
```

Or:

```
$ fdisk -l #If you don't get any output, run as root
```

8. Get the entire details about the system as follows:

```
$ lshw #Recommended to run as root
```

Scheduling with cron

The cron scheduling utility comes with all the GNU/Linux distributions by default. Once we write the cron table entry, the commands will be executed at the time specified for execution. The command crontab is used to add jobs to the cron table. The cron table is a simple text file and each user has a separate copy.

In order to schedule tasks, we should know the format for writing the cron table. A cron job specifies the path of a script or command to be executed and the time at which it is to be executed.

1. cron job to execute the test.sh script at the second minute of all hours on all days:

```
02 * * * * /home/slynux/test.sh
```

2. In order to run the script at the fifth, sixth, and seventh hours on all days, use:

```
00 5,6,7 * * /home/slynux/test.sh
```

3. Execute script.sh at every hour on Sundays as follows:

```
00 */12 * * 0 /home/slynux/script.sh
```

4. Shutdown the computer at 2 A.M. everyday as follows:

```
00 02 * * * /sbin/shutdown -h
```

5. Now, let us see how to schedule a cron job. You can execute the crontab command in multiple ways to schedule the scripts.

Use the -e option to crontab to start editing the cron table:

```
$ crontab -e
02 02 * * * /home/slynux/script.sh
```

When crontab -e is entered, the default text editor (usually vi) is opened up and the user can type the cron jobs and save it. The cron jobs will be scheduled and executed at specified time intervals.

6. There are two other methods we usually use when we invoke the crontab command inside a script for scheduling tasks:

a. Create a text file (for example, task.cron) with the cron job in it, and then run the crontab with this filename as the command argument:

```
$ crontab task.cron
```

b. Or, specify the cron job inline without creating a separate file. For example:

```
crontab<<EOF
02 * * * * /home/slynux/script.sh
EOF
```

The cron job needs to be written in between crontab<<EOF and EOF.

Each cron table consists of six sections in the following order:

```
Minute (0 - 59)
Hour (0 - 23)
Day (1 - 31)
Month (1 - 12)
Weekday (0 - 6)
COMMAND (the script or command to be executed at the specified time)
```

The first five sections specify the time at which an instance of the command is to be executed. There are a few additional options to specify the time schedule.

An asterisk (*) is used to specify that the command should be executed at every instance of time. That is, if * is written in the Hour field in the cron job, the command will be executed for every hour. Similarly, if you would like to execute the command at multiple instances of a particular time period, specify the time period separated by a comma in the corresponding time field (for example, for running the command at the fifth minute and tenth minute, enter 5,10 in the Minute field).

We also have another nice option to run the command at particular divisions of time. Use */5 in the minutes field for running the command at every five minutes. We can apply this to any time field. A cron table can consist of one or more lines of cron jobs and each line in the cron table is a single job.

Cron jobs are executed with privileges with which the crontab command was executed. If you need to execute commands that require higher privileges, such as a command for shutting down the computer, run the crontab command as root.

The commands specified in a cron job are written with the full path to the command. This is because the environment in which a cron job is executed is different from the one that we execute on a terminal. Hence, the PATH environment variable may not be set.

Running commands at system start up/boot

Running specific commands when the system starts (or, boots) is a common requirement at times. There are a lot of ways to achieve this, and using cron is one of them (the others being adding your commands to /etc/rc.d but that's not guaranteed to be the same across distros).

To run a command at boot, add the following line to your crontab:

```
@reboot command
```

This will run the command as your user at runtime. To run the command as root, edit root's crontab.

Viewing the cron table

We can list these existing cron jobs using the -l option:

```
$ crontab -l
02 05 * * * /home/user/disklog.sh
```

The crontab -l lists the existing entries in the cron table for the current user.

We can also view the cron table for other users by specifying a username with the -u option as follows:

```
$ crontab -l -u slynux
09 10 * * * /home/slynux/test.sh
```

You should run as root when you use the -u option to gain higher privilege.

Removing the cron table

We can remove the cron table for the current user using the -r option:

```
$ crontab -r
```

In order to remove crontab for another user, use:

```
# crontab -u slynux -r
```

Run as root to get higher privilege.

User administration script

GNU/Linux is a multiuser operating system allowing many users to log in and perform several activities at the same time. There are several administration tasks that are handled with user management, which include setting the default shell for the user, disabling a user account, disabling a shell account, adding new users, removing users, setting a password, setting an expiry date for a user account, and so on.

Let's go through the user administration script:

```
#!/bin/bash
#Filename: user_adm.sh
#Description: A user administration tool
function usage()
{
echo Usage:
```

```
echo Add a new user
echo $0 -adduser username password
echo
echo Remove an existing user
echo $0 -deluser username
echo
echo Set the default shell for the user
echo $0 -shell username SHELL_PATH
echo
echo Suspend a user account
echo $0 -disable username
echo
echo Enable a suspended user account
echo $0 -enable username
echo
echo Set expiry date for user account
echo $0 -expiry DATE
echo
echo Change password for user account
echo $0 -passwd username
echo
echo Create a new user group
echo $0 -newgroup groupname
echo
echo Remove an existing user group
echo $0 -delgroup groupname
echo
echo Add a user to a group
echo $0 -addgroup username groupname
echo
echo Show details about a user
echo $0 -details username
echo
echo Show usage
echo $0 -usage
echo
exit
}
if [ $UID -ne 0 ];
then
echo Run $0 as root.
exit 2
fi
case $1 in
-adduser) [ $# -ne 3 ] && usage ; useradd $2 -p $3 -m ;;
-deluser) [ $# -ne 2 ] && usage ; deluser $2 --remove-all-files;;
-shell) [ $# -ne 3 ] && usage ; chsh $2 -s $3 ;;
-disable) [ $# -ne 2 ] && usage ; usermod -L $2 ;;
-enable) [ $# -ne 2 ] && usage ; usermod -U $2 ;;
-expiry) [ $# -ne 3 ] && usage ; chage $2 -E $3 ;;
```

```
-passwd) [ $# -ne 2 ] && usage ; passwd $2 ;;
-newgroup) [ $# -ne 2 ] && usage ; addgroup $2 ;;
-delgroup) [ $# -ne 2 ] && usage ; delgroup $2 ;;
-addgroup) [ $# -ne 3 ] && usage ; addgroup $2 $3 ;;
-details) [ $# -ne 2 ] && usage ; finger $2 ; chage -l $2 ;;
-usage) usage ;;
*) usage ;;
esac
```

A sample output is as follows:

```
# ./user_adm.sh -details test
Login: test Name:
Directory: /home/test Shell: /bin/sh
Last login Tue Dec 21 00:07 (IST) on pts/1 from localhost
No mail.
No Plan.
Last password change : Dec 20, 2010
Password expires : never
Password inactive : never
Account expires : Oct 10, 2010
Minimum number of days between password change : 0
Maximum number of days between password change : 99999
Number of days of warning before password expires : 7
```

Let's explain each case one by one:

- -useradd: The useradd command can be used to create a new user. It has the following syntax:

```
useradd USER -p PASSWORD
```

- The -m option is used to create the home directory. It is also possible to provide the full name of the user by using the -c FULLNAME option.

- -deluser: The deluser command can be used to remove the user. The syntax is as follows:

```
deluser USER
```

- --remove-all-files is used to remove all files associated with the user including the home directory.

- -shell: The chsh command is used to change the default shell for the user.
  The syntax is:

```
chsh USER -s SHELL
```

- -disable and -enable: The usermod command is used to manipulate several attributes related to user accounts.

```
usermod -L USER locks the user account
```

and

```
usermod -U USER unlocks the user account.
```

- -expiry: The chage command is used to manipulate user account expiry information. The syntax is:

```
chage -E DATE
```

There are additional options as follows:

  - -m MIN_DAYS (set the minimum number of days between password changes to MIN_DAYS)
  - -M MAX_DAYS (set the maximum number of days during which a password is valid)
  - -W WARN_DAYS (set the number of days of warning before a password change is required)

- -passwd: The passwd command is used to change passwords for the users.

The syntax is:

```
passwd USER
```

The command will prompt to enter a new password.

- -newgroup and addgroup: The addgroup command will add a new user group to he system. The syntax is:

```
addgroup GROUP
```

In order to add an existing user to a group use:

```
addgroup USER GROUP
-delgroup
```

The delgroup command will remove a user group. The syntax is:

```
delgroup GROUP
```

- -details: The finger USER command will display the user information for the user which includes details such as user home directory path, last login time, default shell, and so on. The chage -l command will display the user account expiry information.