Linux Shell Scripting Cookbook

Solve real-world shell scripting problems with over 110 simple but incredibly effective recipes

Sarath Lakshman

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Linux Shell Scripting Cookbook

First published: January 2011

Production Reference: 1200111

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-849513-76-0

www.packtpub.com

Cover Image by Charwak A (charwak86@gmail.com)

Table of Contents

Preface

GNU/Linux is a remarkable operating system that comes with a complete development environment that is stable, reliable, and extremely powerful. The shell, being the native interface to communicate with the operating system, is capable of controlling the entire operating system. An understanding of shell scripting helps you to have better awareness of the operating system and helps you to automate most of the manual tasks with a few lines of script, saving you an enormous amount of time. Shell scripts can work with many external command-line utilities for tasks such as querying information, easy text manipulation, scheduling task running times, preparing reports, sending mails, and so on. There are numerous commands on the GNU/Linux shell, which are documented but hard to understand. This book is a collection of essential command-line script recipes along with detailed descriptions tuned with practical applications. It covers most of the important commands in Linux with a variety of use cases, accompanied by plenty of examples. This book helps you to perform complex data manipulations involving tasks such as text processing, file management, backups, and more with the combination of few commands.

Do you want to become the command-line wizard who performs any complex text manipulation task in a single line of code? Have you wanted to write shell scripts and reporting tools for fun or serious system administration? This cookbook is for you. Start reading!.

What this book covers

Chapter 1, *Shell Something Out*, has a collection of recipes that covers the basic tasks such as printing in the terminal, performing mathematical operations, arrays, operators, functions, aliases, file redirection, and so on by using Bash scripting. This chapter is an introductory chapter for understanding the basic concepts and features in Bash.

Chapter 2, Have a Good Command, shows various commands that are available with GNU/Linux that come under practical usages in different circumstances. It introduces various essential commands such as cat, md5sum, find, tr, sort, uniq, split, rename, look, and so on. This chapter travels through different practical usage examples that users may come across and that they could make use of.

Chapter 3, *File In, File Out,* contains a collection of task recipes related to files and file systems. This chapter explains how to generate large size files, installing a file system on files and mounting files, finding and removing duplicate files, counting lines in a file, creating ISO images, collecting details about files, symbolic link manipulation, file permissions and file attributes, and so on.

Chapter 4, *Texting and Driving,* has a collection of recipes that explains most of the command-line text processing tools well under GNU/Linux with a number of task examples. It also has supplementary recipes for giving a detailed overview of regular expressions and commands such as sed and awk. This chapter goes through solutions to most of the frequently used text processing tasks in a variety of recipes.

Chapter 5, *Tangled Web? Not At All!*, has a collection of shell-scripting recipes that are adherent to the Internet and Web. This chapter is intended to help readers understand how to interact with the web using shell scripts to automate tasks such as collecting and parsing data from web pages, POST and GET to web pages, writing clients to web services, downloading web pages, and so on.

Chapter 6, *The Backup Plan*, shows several commands used for performing data backup, archiving, compression, and so on, and their usages with practical script examples. It introduces commands such as tar, gzip, bunzip, cpio, lzma, dd, rsync, git, squashfs, and much more. This chapter also walks through essential encryption techniques.

Chapter 7, *The Old-boy Network,* has a collection of recipes that talks about networking on Linux and several commands useful to write network-based scripts. The chapter starts with an introductory basic networking primer. Important tasks explained in the chapter include password-less login with SSH, transferring files through network, listing alive machines on a network, multi-cast messaging, and so on.

Chapter 8, *Put on the Monitor's Cap*, walks through several recipes related to monitoring activities on the Linux system and tasks used for logging and reporting. The chapter explains tasks such as calculating disk usage, monitoring user access, CPU usage, syslog, frequently used commands, and much more.

Chapter 9, *Administration Calls,* has a collection of recipes for system administration. This chapter explains different commands to collect details about the system, user management using scripting, sending messages to users, bulk image resizing, accessing MySQL databases from shell, and so on.

What you need for this book

Basic user experience with any GNU/Linux platform will help you easily follow the book. We have tried to keep all the recipes in the book precise and as simple to follow as possible. Your curiosity for learning with the Linux platform is the only prerequisite for the book. Step-by-step explanations are provided for solving the scripting problems explained in the book. In order to run and test the examples in the book, an Ubuntu Linux installation is recommended, however, any other Linux distribution is enough for most of the tasks. You will find the book to be a straightforward reference to essential shell scripting tasks as well as a learning aid to code real-world efficient scripts.

Who this book is for

If you are a beginner or an intermediate user who wants to master the skill of quickly writing scripts to perform various tasks without reading entire manpages, this book is for you. You can start writing scripts and one-liners by simply looking at a similar recipe and its descriptions without any working knowledge of shell scripting or Linux. Intermediate or advanced users as well as system administrators or developers and programmers can use this book as a reference when they face problems while coding.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can use formatted strings with `printf`."

A block of code is set as follows:

```
#!/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
```

Any command-line input or output is written as follows:

```
$ chmod +s executable_file

# chown root.root executable_file
# chmod +s executable_file
$ ./executable_file
```

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the SUGGEST A TITLE form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for this book

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the errata submission form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Shell Something Out

In this chapter, we will cover:

- Printing in the terminal
- Playing with variables and environment variables
- Doing Math calculations with the shell
- Playing with file descriptors and redirection
- Arrays and associative arrays
- Visiting aliases
- Grabbing information about the terminal
- Getting, setting dates, and delays
- Debugging the script
- Functions and arguments
- Reading output of a sequence of commands in a variable
- Reading "n" characters without pressing Return
- Field separators and iterators
- Comparisons and tests

Introduction

UNIX-like systems are amazing operating system designs. Even after many decades, the UNIX-style architecture for operating systems serves as one of the best designs. One of the most important features of this architecture is the command-line interface or 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 the script file is executed using the shell utility.

In this book 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 based on a UNIX-style architecture, most of the examples and discussions are written by keeping Linux systems in mind.

The primary purpose of this chapter is to give readers an insight about the shell environment and become familiar with the basic features that come around the shell. Commands are typed and executed in a shell terminal. When opened, in a terminal, a prompt is available. It is usually in 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.

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

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

For any scripting language in a Linux environment, a script starts with a special line called shebang. Shebang is a line for 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 for `sh` or run a self executable with execution permission.

The script can be run with the filename as a command-line argument as follows:

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

Or:

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

If a script is run as a command-line argument for `sh`, the shebang in the script is of no use.

In order to self execute a shell script, it requires executable permission. While running as a self executable, it makes use of the shebang. It runs the script using the interpreter path that is appended to `#!` in shebang. The execution permission for the script 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 shell program will read the first line and see that the shebang is `#!/bin/bash`. It will identify the `/bin/bash` and execute the script internally as:

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

When a terminal is opened it initially executes a set of commands to define various settings like prompt text, colors, and many more. This set of commands (run commands) are read from a shell script called `.bashrc`, which is located in the home directory of the user (`~/.bashrc`). The bash shell also maintains a history of commands run by the user. It is available in the file `~/.bash_history`. `~` is the shorthand for the user home directory path.

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. A comment section starts with # and proceeds up to the end of that line. The comment lines are most often used to provide comments about the code in the file or to stop a line of code from being executed.

Now let's move on to the basic recipes in this chapter.

Printing in the terminal

The terminal is an interactive utility by which a user interacts with the shell environment. Printing text in the terminal is a basic task that most shell scripts and utilities need to perform regularly. Printing can be performed via various methods and in different formats.

How to do it...

`echo` is the basic command for printing in the terminal.

`echo` puts a newline at the end of every 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 quote'
```

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

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

This will return the following:

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

Hence, if you want to print `!`, do not use within double-quotes or you may escape the `!` with a special escape character (`\`) prefixed with it.

```
$ echo Hello world !
```

Or:

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

Or:

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

When using `echo` with double-quotes, you should add `set +H` before issuing `echo` so that you can use `!`.

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.
- When using `echo` with single quotes, the variables (for example, `$var` will not be expanded) inside the quotes will not be interpreted by Bash, but will be displayed as is.

This means:

`$ echo '$var'` will return `$var`

whereas

`$ echo $var` will return the value of the variable `$var` if defined or nothing at all if it is not defined.

Another command for printing in the terminal is the `printf` command. `printf` 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
```

We will receive the formatted output:

```
No    Name       Mark
1     Sarath     80.35
2     James      91.00
3     Jeff       77.56
```

`%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 `\n` newline is issued.

There's more...

It should be always noted that flags (such as -e, -n, and so on) for `echo` and `printf` should appear before any strings in the command, else Bash will consider the flags as another string.

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 argument. For using escape sequences, use `echo` as `echo -e "string containing escape sequences"`. For example:

```
echo -e "1\t2\t3"
123
```

Printing colored output

Producing colored output on the terminal is very interesting stuff. We produce colored output using escape sequences.

Color codes are used to represent each color. For example, reset=0, black=30, red=31, green=32, yellow=33, blue=34, magenta=35, cyan=36, and white=37.

In order to print colored text, enter the following:

```
echo -e "\e[1;31m This is red text \e[0m"
```

Here `\e[1;31` is the escape string that sets the color to red and `\e[0m` resets the color back. Replace `31` with the required color code.

For a colored background, reset = 0, black = 40, red = 41, green = 42, yellow = 43, blue = 44, magenta = 45, cyan = 46, and white=47, are the color code that are commonly used.

In order to print a colored background, enter the following:

```
echo -e "\e[1;42m Green Background \e[0m"
```

Playing with variables and environment variables

Variables are essential components of every programming language and are used to hold varying data. Scripting languages usually do not require variable type declaration before its use. It can be assigned directly. In Bash, the value for every variable is string. If we assign variables with quotes or without quotes, they are stored as string. There are special variables used by the shell environment and the operating system environment to store special values, which are called environment variables.

Let's look at the recipes.

Getting ready

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

```
cat /proc/$PID/environ
```

Set the `PID` with the process ID of the relevant process (PID is always an integer).

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
```

Note that many environment variables are stripped off for convenience. The actual output may contain numerous variables.

The above mentioned command returns a list of environment variables and their values. Each variable is represented as a `name=value` pair and are separated by a null character (`\0`). 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'
```

Now, let's see how to assign and manipulate variables and environment variables.

How to do it...

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 white space characters (like a space), it need not be enclosed in quotes, else it must be enclosed in single or double quotes.

Note that `var = value` and `var=value` are different. It is a common mistake to write `var =value` instead of `var=value`. The later is the assignment operation, whereas the former is an equality operation.

Printing the 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}
```

The output is 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 is 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=http://192.168.0.2:3128
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, 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.

There's more...

Let's see some more tips associated with regular and environment variables.

Finding length of string

Get the length of a variable value as follows:

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

For example:

```
$ var=12345678901234567890
$ echo ${#var}
20
```

`length` is the number of characters in the string.

Identifying the current shell

Display the currently used shell as follows:

```
echo $SHELL
```

Or, you can also use:

```
echo $0
```

For example:

```
$ echo $SHELL
/bin/bash

$ echo $0
bash
```

Check for super user

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

Modifying the Bash prompt string (username@hostname:~$)

When we open a terminal or run a shell, we see a prompt string like `user@hostname: /home/$`. Different GNU/Linux distributions have slightly different prompts and different colors. We can customize the prompt text using the `PS1` environment variable. The default prompt text for the shell is set using a line in the `~/.bashrc` file.

- We can list the line used to set the `PS1` variable as follows:

  ```
  $ cat ~/.bashrc | grep PS1
  PS1='${debian_chroot:+($debian_chroot)}\u@\h:\w\$ '
  ```

- In order to set a custom prompt string, enter:

  ```
  slynux@localhost: ~$ PS1="PROMPT>"
  PROMPT> Type commands here # Prompt string changed.
  ```

- We can use colored text by using the special escape sequences like `\e[1;31` (refer to the *Printing in the terminal* recipe of this chapter).

There are also certain special characters that expand to system parameters. For example, `\u` expands to username, `\h` expands to hostname, and `\w` expands to the current working directory.

Doing math calculations with the shell

Arithmetic operations are an essential requirement for every programming language. The Bash shell comes with a variety of methods for arithmetic operations.

Getting ready

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.

How to do it...

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

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

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
  ```

  These are equal to `let no=no+6` and `let no=no-6` respectively.

- Alternate methods:

  The `[]` operator can be used similar to the `let` command as follows:

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

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

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

`(( ))` can also be used. `$` prefixed with a variable name is used when the `(( ))` 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 above methods do not support floating point numbers, and operate on integers only.

`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`.

- **Specifying decimal precision (scale):** 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's convert from decimal to binary, and binary to octal:

  ```
  #!/bin/bash
  Description: 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
```

Playing with file descriptors and redirection

File descriptors are integers that are associated with file input and output. They keep track of opened files. The best-known file descriptors are `stdin`, `stdout`, and `stderr`. We can redirect the contents of one file descriptor to another. The following recipe will give examples on how to manipulate and redirect with file descriptors.

Getting ready

While writing scripts we use standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`) frequently. Redirection of output to a file by filtering the contents is one of the essential things we need to perform. While a command outputs some text, it can be either an error or an output (non-error) message. We cannot distinguish whether it is output text or an error text by just looking at it. However, we can handle them with file descriptors. We can extract text that is attached to a specific descriptor.

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)

How to do it...

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.

Next, consider the following example:

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

This would append the text into the file.

`>` and `>>` operators are different. Both of them redirect text to a file, but the first one empties the file and then writes to it, whereas the later one adds the output to the end of the existing file.

View the contents of the file as follows:

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

When we use a redirection operator, it won't print in the terminal but it is directed to a file. When redirection operators are used, by default, it takes standard output. In order to explicitly take a specific file descriptor, you must prefix the descriptor number to the operator.

`>` is equivalent to `1>` and similarly it applies for `>>` (equivalent to `1>>`).

Let's 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
```

Here + is an invalid argument and hence an error is returned.

Successful and unsuccessful command

When a command returns after error, it returns a non-zero exit status. The command returns zero when it terminates after successful completion. 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:

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

However, in the following command the `stdout` output is empty, so an empty file `out.txt` is generated:

```
$ 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 an alternate approach:

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

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 `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 read-write-execute permission for the user. When you need to print the contents of files starting with `a`, you can 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 `stderr` data. We can redirect `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:

```
$ some_command 2> /dev/null
```

In this case, the `stderr` output is dumped to the `/dev/null` file. `/dev/null` is a special device file where any data received by the file is discarded. The null device is often called the bit bucket or black hole.

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 sequence through `stdin`.

However, there is a tricky 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 the `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, `stdin` data is received by the `tee` command. It writes a copy of `stdout` to the file `out.txt` 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 `stdin`. `tee` 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`.

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`.

There's more...

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 | cmd2
```

Redirection from file to 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 `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
```

Custom file descriptors

A file descriptor is an abstract indicator for accessing a file. Each file access is associated with a special number called a file descriptor. 0, 1, and 2 are reserved descriptor numbers for `stdin`, `stdout`, and `stderr`.

We can create our own custom file descriptors using the `exec` command. If you are already familiar with file programming with any other programming languages, you might have noticed modes for opening files. Usually, three modes are used:

- Read mode
- Write with truncate mode
- Write with append mode

`<` is an operator used to read from the file to `stdin`. `>` is the operator used to write to a file with truncation (data is written to the target file after truncating the contents). `>>` is an operator used to write to a file with append (data is appended to the existing file contents and the contents of the target file will not be lost). File descriptors can be created with one of the three modes.

Create a file descriptor for reading a file, as follows:

```
$ exec 3<input.txt # open for reading with descriptor number 3
```

We could use it as follows:

```
$ echo this is a test line > input.txt
$ exec 3<input.txt
```

Now you can use file descriptor `3` with commands. For example, `cat <&3` as follows:

```
$ cat <&3
this is a test line
```

If a second read is required, we cannot reuse file descriptor `3`. It is needed to reassign file descriptor `3` for read using `exec` for making a second read.

Create a file descriptor for writing (truncate mode) as follows:

```
$ exec 4>output.txt # open for writing
```

For example:

```
$ exec 4>output.txt
$ echo newline >&4
$ cat output.txt
newline
```

Create a file descriptor for writing (append mode) as follows:

```
$ exec 5>>input.txt
```

For example:

```
$ exec 5>>input.txt
$ echo appended line >&5
$ cat input.txt
newline
appended line
```

Arrays and associative arrays

Arrays are a very important component for storing a collection of data as separate entities using indexes.

Getting ready

Bash supports regular arrays as well as associative arrays. Regular arrays are arrays which can use only integers as its array index. But associative arrays are arrays which can take a string as its array index.

Associative arrays are very useful in many types of manipulations. Associative array support came with version 4.0 of Bash. Therefore, older versions of Bash will not support associative arrays.

How to do it...

An array can be defined in many ways. Define an array using a list of values in a line, as follows:

```
array_var=(1 2 3 4 5 6)
#Values will be stored in consecutive locations starting from index 0.
```

Alternately, define an array as a set of index-value pairs as follows:

```
array_var[0]="test1"
array_var[1]="test2"
array_var[2]="test3"
array_var[3]="test4"
array_var[4]="test5"
array_var[5]="test6"
```

Print the contents of an array at a given index using:

```
$ echo ${array_var[0]}
test1

index=5
$ echo ${array_var[$index]}

test6
```

Print all of the values in an array as a list using:

```
$ echo ${array_var[*]}
test1 test2 test3 test4 test5 test6
```

Alternately, you can use:

```
$ echo ${array_var[@]}
test1 test2 test3 test4 test5 test6
```

Print the length of an array (the number of elements in an array), as follows:

```
$ echo ${#array_var[*]}
6
```

There's more...

Associative arrays have been introduced to Bash from version 4.0. They are useful entities to solve many problems using the hashing technique. Let's go into more details.

Defining associative arrays

In an associative array, we can use any text data as an array index. However, ordinary arrays can only use integers for array indexing.

Initially, a declaration statement is required to declare a variable name as an associative array. A declaration can be made as follows:

```
$ declare -A ass_array
```

After the declaration, elements can be added to the associative array using two methods, as follows:

1. By using inline index-value list method, we can provide a list of index-value pairs:

   ```
   $ ass_array=([index1]=val1 [index2]=val2)
   ```

2. Alternately, you could use separate index-value assignments:

```
$ ass_array[index1]=val1
$ ass_array[index2]=val2
```

For example, consider the assignment of prices for fruits using an associative array:

```
$ declare -A fruits_value
$ fruits_value=([apple]='100dollars' [orange]='150 dollars')
```

Display the content of an array as follows:

```
$ echo "Apple costs ${fruits_value[apple]}"
Apple costs 100 dollars
```

Listing of array indexes

Arrays have indexes for indexing each of the elements. Ordinary and associative arrays differ in terms of index type. We can obtain the list of indexes in an array as follows:

```
$ echo ${!array_var[*]}
```

Or, we can also use:

```
$ echo ${!array_var[@]}
```

In the previous `fruits_value` array example, consider the following:

```
$ echo ${!fruits_value[*]}
orange apple
```

This will work for ordinary arrays too.

Visiting aliases

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

Getting ready

Aliases can be implemented in multiple ways, either by using functions or by using the `alias` command.

How to do it...

An alias can be implemented 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`.

The `alias` command is temporary; aliasing exists until we close the current terminal only. In order 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
```

To remove an alias, remove its entry from `~/.bashrc` or use the `unalias` command. Another method is to define a function with a new command name and write it in `~/.bashrc`.

We can alias `rm` so that it will delete the original and keep a copy in a backup directory:

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

When you create an alias, if the item being aliased already exists, it will be replaced by this newly aliased command for that user.

There's more...

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 a good security practise to ignore aliases by prefixing the command with `\`. The attacker might have aliased the privileged command with his own custom command to steal the critical information that is provided to the command by the user.

Grabbing information about terminal

While writing command-line shell scripts, we will often need to heavily manipulate information about the current terminal, such as number of columns, rows, cursor positions, masked password fields, and so on. This recipe helps to learn about collecting and manipulating terminal settings.

Getting ready

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

How to do it...

Get number of columns and rows in a terminal as follows:

```
tput cols
tput lines
```

In order to print the current terminal name, use:

```
tput longname
```

For moving the cursor to a position 100,100 you can enter:

```
tput cup 100 100
```

Set the background color for terminal as follows:

```
tput setb no
```

`no` can be a value in the range of 0 to 7.

Set the foreground color for text as follows:

```
tput setf no
```

`no` can be a value in the range of 0 to 7.

In order to make the text bold use:

```
tput bold
```

Start and end underlining by using:

```
tput smul
tput rmul
```

In order to delete from cursor to end of the line use:

```
tput ed
```

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 above disables output to the terminal, whereas `echo` enables output.

Getting, setting dates, and delays

Many applications require printing dates in different formats, setting the date and time, and performing manipulations based on the date and time. Delays are commonly used to provide a wait time (for example, 1 second) during the program's execution. Scripting contexts, such as performing a monitoring task every five seconds, demand the understanding of writing delays in a program. This recipe will show you how to work with dates and time delays.

Getting ready

Dates can be printed in a variety of formats. We can also set dates from the command line. In UNIX-like systems, dates are stored as an integer in seconds since 1970-01-01 00:00:00 UTC. This is called epoch or UNIX time. Let's see how to read dates and set them.

How to do it...

You can read the date as follows:

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

The epoch time can be printed as follows:

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

Epoch is defined as the number of seconds that have elapsed since midnight proleptic Coordinated Universal Time (UTC) of January 1, 1970, not counting leap seconds. Epoch time is useful when you need to calculate the difference between two dates or time. You may find out the epoch times for two given timestamps and take the difference between the epoch values. Therefore, you can find out the total number of seconds between two dates.

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. You can convert a 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 output. Feeding 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
```

The date format strings are listed in 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)

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
```

We can set the date and time as follows:

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

For example:

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

Sometimes we need to check the time taken by a set of commands. We can display it as follows:

```
#!/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.
```

An alternate method would be to use `timescriptpath` to get the time that it took to execute the script.

There's more...

Producing time intervals are essential when writing monitoring scripts that execute in a loop. Let's see how to generate time delays.

Producing delays in a script

In order to delay execution in a script for some period of time, use `sleep`:
`$ sleep no_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 [ $x -lt 40 ];
```

```
then let count++;
sleep 1;
tput rc
tput ed
echo -n $count;
else exit 0;
fi
done
```

In the above 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`. `tput ed` 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 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. There are also some other tricky ways to debug.

Getting ready

No special utilities are required to debug shell scripts. Bash comes with certain flags that can print arguments and inputs taken by the scripts. Let's see how to do it.

How to do it...

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 current status. Note that you can also use `sh -x script`.

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`: Displays arguments and commands upon their execution
- `set +x`: Disables debugging

- `set –v`: Displays input when they are read
- `set +v`: Disables printing input

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

The above 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 script, debug information will not be printed. In Bash the command ':' tells the shell to do nothing.

There's more...

We can also use other convenient ways to debug scripts. We can make use of shebang in a trickier way to debug scripts.

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

Like any other scripting languages, Bash also supports functions. Let's see how to define and use functions.

How to do it...

A function can be defined as follows:

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

Or alternately,

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

A function can be invoked just by using its name:

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

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 the most used one. `"$*"` is used rarely since it gives all arguments as a single string.

There's more...

Let's explore more tips on Bash functions.

Recursive function

Functions in Bash also support recursion (the function that can call itself). For example, `F() { echo $1; F hello; sleep 1; }`.

Fork bomb

:(){ :|:& };:

This recursive function is a function that calls itself. It infinitely spawns processes and ends up in a denial of service attack. `&` is postfixed with the function call to bring the subprocess into the background. This is a dangerous code as it forks processes and, therefore, it is called a fork bomb.

You may find it difficult to interpret the above code. See Wikipedia page `http://en.wikipedia.org/wiki/Fork_bomb` for more details and interpretation of the fork bomb.

It can be prevented by restricting the maximum number of processes that can be spawned from the config file `/etc/security/limits.conf`.

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 command return value (status)

We can get the return value of a command or function as follows:

```
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, else it will be non-zero.

We can check whether a command terminated successfully or not as follows:

```
#!/bin/bash
#Filename: success_test.sh
CMD="command" #Substitute with command for which you need to test 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 NO` is another option that takes a number. Also the command takes a filename as argument. It can be executed in multiple ways as follows:

```
$ command -p -v -k 1 file
```

Or:

```
$ command -pv -k 1 file
```

Or:

```
$ command -vpk 1 file
```

Or:

```
$ command file -pvk 1
```

Reading the output of a sequence 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. This recipe illustrates how to combine multiple commands and how its output can be read.

Getting ready

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` for output.

Commands are called as filters. We connect each filter using pipes. The piping operator is "|". 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.

How to do it...

Have a look at the following code:

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

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

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

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

This is called the subshell method. For example:

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

Another method, called back-quotes 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.

There's more...

There are multiple ways of grouping commands. Let's go through few of them.

Spawning a separate process with subshell

Subshells are separate processes. A subshell can be defined using the `( )` operators as follows:

```
pwd;
(cd /bin; ls);
pwd;
```

When some commands are executed in a subshell none of the changes occur in the current shell; changes are restricted to the subshell. For example, when the current directory in a subshell is changed using the `cd` command, the directory change is not reflected in the main shell environment.

The `pwd` command prints the path of the working directory.

The `cd` command changes the current directory to the given directory path.

Subshell quoting to preserve spacing and newline character

Suppose we are reading the output of a command to a variable using a subshell or the back-quotes method, we always quote them in double-quotes to preserve the spacing and newline character (\n). For example:

```
$ cat text.txt
1
2
3

$ out=$(cat text.txt)
$ echo $out
1 2 3 # Lost \n spacing in 1,2,3
```

```
$ out="$(cat tex.txt)"
$ echo $out
1
2
3
```

Reading "n" characters without pressing Return

`read` is an important Bash command that can be used to read text from keyboard or standard input. We can use `read` to interactively read an input from the user, but `read` is capable of much more. Let's look at a new recipe to illustrate some of the most important options available with the `read` command.

Getting ready

Most of the input libraries in any programming language read the input from the keyboard; but string input termination is done when *Return* is pressed. There are certain critical situations when *Return* cannot be pressed, but the termination is done based on number of characters or a single character. For example, in a game a ball is moved up when up + is pressed. Pressing + and then pressing *Return* everytime to acknowledge the + press is not efficient. The read command provides a way to accomplish this task without having to press *Return*.

How to do it...

The following statement will read "n" characters from input into the variable `variable_name`:

```
read -n number_of_chars variable_name
```

For example:

```
$ read -n 2 var
$ echo $var
```

Many other options are possible with `read`. Let's see take a look at these.

Read a password in non-echoed mode as follows:

```
read -s var
```

Display a message with `read` using:

```
read -p "Enter input:"  var
```

Read the input after a timeout as follows:

```
read -t timeout var
```

For example:

```
$ read -t 2 var
#Read the string that is typed within 2 seconds into variable var.
```

Use a delimiter character to end the input line as follows:

```
read -d delim_charvar
```

For example:

```
$ read -d ":" var
hello:#var is set to hello
```

Field separators and iterators

The Internal Field Separator 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** (**IFS**) 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's see how to do it.

Getting ready

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 "," 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 were not set as ",", then it would print the entire data as a single string.

How to do it...

Let's go through another example usage of IFS by taking `/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. In order to print users and their default shells, we can use the IFS hack as follows:

```
#!/bin/bash
#Description: 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's see how to use them.

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 we can generate partial list using `{a..h}`. Similarly, 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
}
```

While loop:

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

For an infinite loop, use `true` as the condition.

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.

Getting ready

We can use `if`, `if else`, and logical operators to perform tests and certain comparison operators to compare data items. There is also a command called `test` available to perform tests. Let's see how to use those commands.

How to do it...

If condition:

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

else if and else:

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

Nesting is also possible with if and else. `if` conditions can be lengthy. We can use logical operators to make them shorter as follows:

```
[ condition ] && action; # action executes if condition is true.
[ condition ] || action; # action executes if condition is false.
```

`&&` is the logical AND operation and `||` is the logical OR operation. This is a very helpful trick while writing Bash scripts. Now let's go into conditions and comparisons operations.

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 not equals 0
```

Other important operators are:

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

Multiple test conditions can be combined as follows:

```
[ $var1 -ne 0 -a $var2 -gt 2 ]  # using AND -a
[ $var -ne 0 -o var2 -gt 2 ] # OR -o
```

Filesystem related tests:

We can test different filesystem related attributes using different condition flags as follows:

- `[ -f $file_var ]`: Returns true if the given variable holds a regular filepath or filename.
- `[ -x $var ]`: Returns true if the given variable holds a file path or filename which is executable.
- `[ -d $var ]`: Returns true if the given variable holds a directory path or directory name.
- `[ -e $var ]`: Returns true if the given variable holds an existing file.
- `[ -c $var ]`: Returns true if the given variable holds path of a character device file.
- `[ -b $var ]`: Returns true if the given variable holds path of a block device file.
- `[ -w $var ]`: Returns true if the given variable holds path of a file which is writable.
- `[ -r $var ]`: Returns true if the given variable holds path of a file which is readable.
- `[ -L $var ]`: Returns true if the given variable holds 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 use of single brackets can sometimes lead to errors. Usage of single brackets sometimes lead to error. So it is better to avoid them.

Two strings can be compared to check whether they are the same as follows;

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

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

- `[[ $str1 != $str2 ]]`: Returns true when str1 and str2 mismatches

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

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

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

- `[[ -z $str1 ]]`: Returns true if str1 holds an empty string
- `[[ -n $str1 ]]`: Returns true if str1 holds a non-empty string

It is easier to combine multiple conditions using the logical operators `&&` and `||` as follows:

```
if [[ -n $str1 ]] && [[ -z $str2 ]] ;
then
commands;
fi
```

For example:

```
str1="Not empty "
str2=""
if [[ -n $str1 ]] && [[ -z $str2 ]];
then
echo str1 is non-empty and str2 is empty string.
fi
```

The output is as follows:

```
str1 is non-empty and str2 is empty string.
```

The `test` command can be used for performing condition checks. It helps to avoid usage of many braces. The same set of test conditions enclosed within `[]` can be used for the test command.

For example:

```
if  [ $var -eq 0 ]; then echo "True"; fi
can be written as
if  test $var -eq 0 ; then echo "True"; fi
```

2
Have a Good Command

In this chapter, we will cover:

- Concatenating with cat
- Recording and playback of terminal sessions
- Finding files and file listing
- Command output as argument to a command (xargs)
- Translating with tr
- Checksum and verification
- Sorting, unique and duplicates
- Temporary file naming and random numbers
- Splitting files and data
- Slicing filenames based on extension
- Renaming files in bulk with rename and mv
- Spell check and dictionary manipulation
- Automating interactive input

Introduction

Commands are beautiful components of UNIX-like systems. They help us achieve many tasks making our work easier. When you practise the use of commands everywhere, you will love it. Many circumstances make you say "wow!". Once you've had a chance to try some of the commands that Linux offers you to make your life easier and more productive, you'll wonder how you did without using them before. Some of my personal favorite commands are `grep`, `awk`, `sed`, and `find`.

Using the UNIX/Linux command line is an art. You will get better at using it as you practice and gain experience. This chapter will introduce you to some of the most interesting and useful commands.

Concatenating with cat

`cat` is one of the first commands which a command line warrior must learn. `cat` is a beautiful and simple command. It is usually used to read, display, or concatenate the contents of a file, but `cat` is capable of more than just that.

Getting ready

We 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 `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.

How to do it...

The `cat` command is a very simple command that is used very frequently in daily life. `cat` stands for concatenate.

The general syntax of `cat` for reading a file's contents is:

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

This command outputs concatenated data from the files with file names provided as command-line arguments. For example:

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

How it works...

There are a lot of features that come along with `cat`. Let's walk through several usage techniques that are possible with `cat`.

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

In order 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 code `-` acts as filename for `stdin` text.

There's more...

The `cat` command has few other options for viewing files. Let's go through them.

Squeezing blank lines

Sometimes many empty lines in text need to be squeezed into one to make it readable or for some other purpose. Squeeze adjacent blank lines in a text file by using the following syntax:

```
$ cat -s file
```

For example:

```
$ cat multi_blanks.txt
line 1


line2

line3


line4

$ cat -s multi_blanks.txt # Squeeze adjacent blank lines
line 1

line2

line3
```

```
line4
```

Alternately, we can remove all blank lines by using `tr` as follows:

```
$ cat multi_blanks.txt | tr -s '\n'
line 1
line2
line3
line4
```

In the above usage of `tr`, it squeezes adjacent `'\n'` characters into a single `'\n'` (newline character).

Displaying tabs as ^I

It is hard to distinguish tabs and repeated space characters. While writing programs in languages like Python, it keeps special meaning for tabs and spaces for indentation purposes. They are treated differently. Therefore, the use of tab instead of spaces causes problems in indentation. It may become difficult to track where the misplacement of the tab or space occurred by looking through a text editor. `cat` has a feature that can highlight tabs. This is very helpful in debugging indentation errors. Use the `-T` option with `cat` to highlight tab characters as ^I. An example is as follows:

```
$ cat file.py
def function():
    var = 5
        next = 6
    third = 7

$ cat -T file.py
def function():
^Ivar = 5
        next = 6
^Ithird = 7^I
```

Line numbers

Using the `-n` flag for the `cat` command will output each line with a line number prefixed. It is to be noted that the `cat` command never changes a file; instead it produces an output on `stdout` with modifications to input according to the options provided. For example:

```
$ cat lines.txt
line
```

```
line
line

$ cat -n lines.txt
     1 line
     2 line
     3 line
```

Recording and playback of terminal sessions

When you need to show somebody how to do something in the terminal or you need to prepare a tutorial on how to do something through command line, you would normally type the commands manually and show them. Or you could record a screencast video and playback the video to them. What if we can record the order and timing of the commands that we typed before and replay these commands again so that others can watch as if they were typing? The output of the commands gets displayed on the terminal until the playback is complete. Sounds interesting? It can be done using the commands `script` and `scriptreplay`.

Getting ready

`script` and `scriptreplay` commands are available in most of the GNU/Linux distributions. Recording the terminal sessions to a file will be interesting. You can create tutorials of command-line hacks and tricks to achieve some task by recording the terminal sessions. You can also share the recorded files for others to playback and see how to perform a particular task using the command line.

How to do it...

We can start recording the terminal session as follows:

```
$ script -t 2> timing.log -a output.session
type commands;
…
..
exit
```

Two configuration files are passed to the `script` command as arguments. One file is for storing timing information (`timing.log`) at which each of the commands are run, whereas the other file (`output.session`) is used for storing command output. The `-t` flag is used to dump timing data to `stderr`. `2>` is used to redirect `stderr` to `timing.log`.

By using the two files, `timing.log` (stores timing information) and `output.session` (stores command output information), we can replay the sequence of command execution as follows:

```
$ scriptreplay timing.log output.session
# Plays the sequence of commands and output
```

How it works...

Usually, we record the desktop video to prepare tutorials. However, videos require good amount of storage. But a terminal script file is just a text file. Therefore, it always has a file size only in the order of Kilobytes.

You can share the files `timing.log` and `output.session` with anyone who wants to replay a terminal session in their terminal.

The `script` command can also be used to set up a terminal session that can be broadcasted to multiple users. It is a very interesting experience. Let's see how to do it.

Open two terminals, Terminal1 and Terminal2.

1. In Terminal1 enter the following command:

   ```
   $ mkfifo scriptfifo
   ```

2. In Terminal2 enter the following command:

   ```
   $ cat scriptfifo
   ```

3. Go back to Terminal1 and enter the following command:

   ```
   $ script -f scriptfifo
   $ commands;
   ```

When you need to end the session, type `exit` and press *Return*. It will show the message "Script done, file is scriptfifo".

Now Terminal1 is the broadcaster and Terminal2 is the receiver.

When you type anything in real-time on Terminal1, it will be played on Terminal2 or any terminal that supplies the following command:

```
cat scriptfifo
```

This method can be used when handling a tutorial session for many users in a computer lab or over the Internet. It will save bandwidth as well as provide a real-time experience.

Finding files and file listing

`find` is one of the great utilities in the UNIX/Linux command-line toolbox. It is a very useful command for shell scripts, but most people do not use it effectively due to the lack of understanding. This recipe deals with most of the use cases of `find` and how it can be used to solve problems of different criterions.

Getting ready

The `find` command uses the following strategy: `find` descends through a hierarchy of files, matches the files that meet specified criteria, and performs some actions. Let's go through different use cases of find and the basic usages.

How to do it...

In order 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 the `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 file system.

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.

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

There's more...

In this recipe we have learned the usage of the most commonly-used `find` command with an example. The `find` command is a powerful command-line tool and it is armed with a variety of interesting options. Let's walk through some of these different options of the `find` command.

Search based on file name or regular expression match

The `-name` argument specifies a matching string for the filename. We can pass wildcards as its argument text. `*.txt` 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`. `–iname` matches the name 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 criterions, we can use OR conditions as shown below:

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

The previous code will print all of the `.txt` and `.pdf` files, since the `find` command matches both `.txt` and `.pdf` files. `\(` and `\)` is 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
```

The `-regex` argument is similar to `-path`, but `-regex` matches the file paths based on regular expressions.

Regular expressions are an advanced form of wildcard matching. It enables to specify a text with patterns. By using the patterns, we can make matches to the text and print them. A typical example of text matching using regular expressions is: parsing all e-mail addresses from a given pool of text. An e-mail address takes the form `name@host.root`. So, it can be generalized as `[a-z0-9]+@[a-z0-9]+.[a-z0-9]+`. The `+` signifies that the previous class of characters can occur one or more times, repeatedly, in the characters that follow.

The following command matches `.py` or `.sh` files:

```
$ ls
new.PY  next.jpg  test.py
$ find . -regex ".*\(\.py\|\.sh\)$"
./test.py
```

Similarly, using `-iregex` ignores the case for the regular expressions that are available. For example:

```
$ find . -iregex ".*\(\.py\|\.sh\)$"
./test.py
./new.PY
```

Negating arguments

`find` can also take negation of arguments using "!". For example:

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

The above `find` construct matches all the file names, as long as the name does not end with `.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
```

Search based on the directory depth

When the `find` command is used it recursively walks through all the subdirectories as much as possible until it reaches the leaf of the subdirectory tree. We can restrict the depth to which the `find` command traverses using some depth parameters given to the find. `-maxdepth` and `-mindepth` are the parameters.

In most of the cases, we need to search only in the current directory. It should not further descend into the subdirectories from the current directory. In such cases, we can restrict the depth to which the `find` command should descend using depth parameters. In order to restrict `find` from descending into the subdirectories from the current directory, the depth can be set as 1. When we need to descend to two levels, the depth is set as 2, and so on for the rest of the levels.

For specifying the maximum depth we use the `–maxdepth` level parameter. Similarly, we can also specify the minimum level at which the descending should start. If we want to start searching from the second level onwards, we can set the minimum depth using the `–mindepth` level parameter. Restrict the `find` command to descend to a maximum depth of 1, by using the following command:

```
$ find . -maxdepth 1 -type f -print
```

This command lists all the regular files only from the current directory. If there are subdirectories, they are not printed or traversed. Similarly, `-maxdepth 2` traverses up to at most two descending levels of subdirectories.

`-mindepth` is similar to `–maxdepth`, but it sets the least depth level for the `find` traversal. It can be used to find and print the files that are located with a minimum level of depth from the base path. For example, to print all the files that are at least two subdirectories distant from the current directory use the following command:

```
$ find . -mindepth 2 -type f -print
./dir1/dir2/file1
./dir3/dir4/f2
```

Even if there are files in the current directory or `dir1` and `dir3`, it will not be printed.

`-maxdepth` and `–mindepth` should be specified as the third argument to the find. If they are specified as the fourth or further arguments, it may affect the efficiency of the find as it has to do unnecessary checks (for example, if `–maxdepth` is specified as the fourth argument and `–type` as the third argument, the `find` command first finds out all the files having the specified `–type` and then finds all of the matched files having the specified depth. However, if the depth were specified as the third argument and `–type` as the fourth, `find` could collect all the files having at most the specified depth and then check for the file type, which is the most efficient way of searching.

Search based on file type

UNIX-like operating systems consider every object as a file. There are different kinds of files such as regular file, directory, character devices, block devices, symlinks, hardlinks, sockets, FIFO, and so on.

The file search can be filtered out using the `-type` option. By using `–type`, we can specify to the `find` command that it should only match files having a specified type.

List only directories including descendants as follows:

```
$ find . -type d -print
```

It is hard to list directories and files separately. But `find` helps to do it. List only regular files as follows:

```
$ find . -type f -print
```

List only symbolic links as follows:

```
$ find . -type l -print
```

You can use the `type` arguments from the following table to properly match the required file type:

File type	Type argument
Regular file	`f`
Symbolic link	`l`
Directory	`d`
Character special device	`c`
Block device	`b`
Socket	`s`
Fifo	`p`

Search on up file times

UNIX/Linux file systems have three types of timestamp on each file. They are as follows:

- **Access time** (`-atime`): This is the last timestamp of when the file was accessed by some user
- **Modification time** (`-mtime`): This is the last timestamp of when the file content was modified
- **Change time** (`-ctime`): This is the last timestamp of when the metadata for a file (such as permissions or ownership) was modified

There is nothing called creation time in UNIX.

`-atime`, `-mtime`, `-ctime` are the time parameter options available with `find`. They can be specified with integer values in "number of days". These integer values are often attached with `-` or + signs. The `-` sign implies less than whereas the `+` implies greater than. For example:

- Print all the files that were accessed within the last 7 days as follows:

  ```
  $ find . -type f -atime -7 -print
  ```

- Print all the files that are having access time exactly 7 days old as follows:

  ```
  $ find . -type f -atime 7 -print
  ```

- Print all the files that are having access time older than 7 days as follows:

  ```
  $ find . -type f -atime +7 -print
  ```

Similarly, we can use the `-mtime` parameter for search files based on modification time and `-ctime` for search based on change time.

`-atime`, `-mtime`, and `-ctime` are time-based parameters that use the time metric in days. There are some other time-based parameters that use the time metric in minutes. These are as follows:

- `-amin` (access time)
- `-mmin` (modification time)
- `-cmin` (change time)

For example:

In order to print all the files that are having access time older than seven minutes, use the following command:

```
$ find . -type f -amin +7 -print
```

Another nice feature available with `find` is the `-newer` parameter. By using `-newer`, we can specify a reference file to compare with the timestamp. We can find all the files that are newer (older modification time) than the specified file with the `-newer` parameter.

For example, find all the files that are having a modification time greater than that of the modification time of a given `file.txt` file as follows:

```
$ find . -type f -newer file.txt -print
```

Timestamp manipulation flags for the `find` command are very useful for writing system backup and maintenance scripts.

Search based on file size

Based on the file sizes of the files, a search can be performed as follows:

```
$ find . -type f -size +2k
# Files having size greater than 2 kilobytes

$ find . -type f -size -2k
# Files having size less than 2 kilobytes

$ find . -type f -size 2k
# Files having size 2 kilobytes
```

Instead of `k` we can use different size units as the following:

- `b` – 512 byte blocks
- `c` – bytes
- `w` – two byte words
- `k` – Kilobyte
- `M` – Megabyte
- `G` – Gigabyte

Deleting based on the file matches

The `-delete` flag can be used to remove files that are matched by `find`.

Remove all the `.swp` files from the current directory as follows:

```
$ find . -type f -name "*.swp" -delete
```

Match based on the file permissions and ownership

It is possible to match files based on the file permissions. We can list out the files having specified file permission as follows:

```
$ find . -type f -perm 644 -print
# Print files having permission 644
```

As an example usage case, we can consider the case of Apache web server. The PHP files in the web server require proper permissions to execute. We can find out the PHP files that are not having proper execute permissions as follows:

```
$ find . –type f –name "*.php" ! -perm 644 –print
```

We can also search files based on ownership of the files. The files owned by a specific user can be found out using the `-user USER` option.

The `USER` argument can be a username or UID.

For example, to print the list of all files owned by the user slynux, you can use the following command:

```
$ find . -type f -user slynux -print
```

Executing commands or actions with find

The `find` command can be coupled with many of the other commands using the `-exec` option. `-exec` is one of the most powerful features that comes with `find`.

Let's see how to use the `–exec` option.

Consider the example in the previous section. We used `-perm` to find out the files that do not have proper permissions. Similarly, in the case where we need to change the ownership of all files owned by a certain user (for example, `root`) to another user (for example, `www-data` the default Apache user in the web server), we can find all the files owned by root by using the `-user` option and using `-exec` to perform ownership change operation.

You must run the `find` command as root for performing ownership change.

Let's have a look at the following example:

```
# find . -type f -user root -exec chown slynux {} \;
```

In this command, {} is a special string used with the `-exec` option. For each file match, {} will be replaced with the file name in place for `-exec`. For example, if the `find` command finds two files `test1.txt` and `test2.txt` with owner slynux, the find command will perform:

```
chown slynux {}
```

This gets resolved to `chown slynux test1.txt` and `chown slynux test2.txt`.

Another usage example is to concatenate all the C program files in a given directory and write it to a single file `all_c_files.txt`. We can use `find` to match all the C files recursively and use the `cat` command with the `-exec` flag as follows:

```
$ find . -type f -name "*.c" -exec cat {} \;>all_c_files.txt
```

`-exec` is followed with any command. {} is a match. For every matched filename, {} is replaced with filename.

To redirect the data from `find` to the `all_c_files.txt` file, we used the > operator instead of >> (append) because the entire output from the `find` command is a single data stream (`stdin`). >> is necessary only when multiple data streams are to be appended to a single file.

For example, to copy all the `.txt` files that are older than 10 days to a directory `OLD`, use the following command:

```
$ find . -type f -mtime +10 -name "*.txt" -exec cp {} OLD  \;
```

Similarly, the `find` command can be coupled with many other commands.

-exec with multiple commands

We cannot use multiple commands along with the `-exec` parameter. It accepts only a single command, but we can use a trick. Write multiple commands in a shell script (for example, `commands.sh`) and use it with `-exec` as follows:

```
-exec ./commands.sh {} \;
```

`-exec` can be coupled with `printf` to produce a very useful output. For example:

```
$ find . -type f -name "*.txt" -exec printf "Text file: %s\n" {} \;
```

Skip specified directories from the find

Skipping certain subdirectories for a performance improvement is sometimes required while doing a directory search and performing some action. For example, when programmers look for particular files on a development source tree, which is under a version control system such as Git, the source hierarchy will always contain the `.git` directory in each of the subdirectories (`.git` stores version control related information for every directory). Since version control related directories do not produce useful output, they should be excluded from the search. The technique of excluding files and directories from the search is known as pruning. It can be performed as follows:

```
$ find devel/source_path  \( -name ".git" -prune \) -o \( -type f -print \)

# Instead of \( -type -print \), use required filter.
```

The above command prints the name (path) of all the files that are not from the `.git` directories.

Here, `\( -name ".git" -prune \)` is the exclude portion, which specifies that the `.git` directory should be excluded and `\( -type f -print \)` specifies the action to be performed. The actions to be performed are placed in the second block `-type f –print` (the action specified here is to print the names and path of all the files).

Playing with 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"
```

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

All the Bash hackers love one-line commands. One-liners are command sequences that are joined by using the pipe operator, but do not use the semi colon terminator (;) between the commands used. Crafting one-line commands makes tasks efficient and simpler to solve. It requires proper understanding and practise to formulate one-liners for solving text processing problems. `xargs` is one of the important components for building one-liner commands.

Getting ready

The `xargs` command should always appear immediately after a pipe operator. `xargs` uses standard input as the primary data stream source. It uses `stdin` and executes another command by providing command-line arguments for that executing command using the stdin data source. For example:

```
command | xargs
```

How to do it...

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 new line character and replacing with the " " (space) character. '`\n`' is interpreted as a newline, which is the delimiter for the lines. By using `xargs`, we can ignore all the newlines with spaces 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 maximum no 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 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
```

How it works...

The `xargs` command is appropriate to be applied to many problem scenarios with its rich and simple options. Let's see how these options can be used wisely to solve problems.

We can also use our own delimiter towards separating arguments. In order to specify a custom delimiter for input, use the `-d` option as follows:

```
$ echo "splitXsplitXsplitXsplit" | xargs -d X
split split split split
```

In the above code, `stdin` contains a string consisting of multiple 'X' characters. We can use 'X' as the input delimiter by using it with `-d`. Here we have explicitly specified X as the input delimiter, whereas in the default case `xargs` takes Internal Field Separator (space) as the input delimiter.

By using `-n` along with the above command, we can split the input into multiple lines having two words each as follows:

```
$ echo "splitXsplitXsplitXsplit" | xargs -d X -n 2
split split
split split
```

There's more...

We have learned how to format `stdin` to different output as arguments from the above examples. Now let's learn how to supply these formatted output as arguments to commands.

Passing formatted arguments to a command by reading stdin

Write a small custom echo for better understanding of example usages with xargs to provide command arguments.

```
#!/bin/bash
#Filename: cecho.sh
echo $*'#'
```

When arguments are passed to the `cecho.sh`, it will print the arguments terminated by the # character. For example:

```
$ ./cecho.sh arg1 arg2
arg1 arg2 #
```

Let's have a look at a problem:

- I have a list of arguments in a file (one argument in each line) to be provided to a command (say `cecho.sh`). I need to provide arguments in two methods. In the first method, I need to provide one argument each for the command as follows:

  ```
  ./cecho.sh arg1
  ./cecho.sh arg2
  ./cecho.sh arg3
  ```

  Or, alternately, I need to provide two or three arguments each for each execution of command. For two arguments each, it would be similar to the following:

  ```
  ./cecho.sh arg1 arg2
  ./cecho.sh arg3
  ```

- In the second method, I need to provide all arguments at once to the command as follows:

  ```
  ./cecho.sh arg1 arg2 arg3
  ```

Run the above commands and note down the output before going through the following section.

The above problems can be solved using `xargs`. We have the list of arguments in a file called `args.txt`. The contents are as follows:

```
$ cat args.txt
arg1
arg2
arg3
```

For the first problem, we can execute the command multiple times with one argument per execution, by using:

```
$ cat args.txt | xargs -n 1 ./cecho.sh
arg1 #
arg2 #
arg3 #
```

For executing a command with X arguments per each execution, use:

```
INPUT | xargs –n X
```

For example:

```
$ cat args.txt | xargs -n 2 ./cecho.sh
arg1 arg2 #
arg3 #
```

For the second problem, we can execute the command at once with all the arguments, by using:

```
$ cat args.txt | xargs ./ccat.sh
arg1 arg2 arg3 #
```

In the above examples, we have supplied command-line arguments directly to a specific command (for example, `cecho.sh`). We could only supply the arguments from the `args.txt` file. However, in realtime, we may also need to add some constant parameter with the command (for example, `cecho.sh`) along with the arguments taken from `args.txt`. Consider the following example with the format:

```
./cecho.sh –p arg1 –l
```

In the above command execution `arg1` is the only variable text. All others should remain constant. We should read arguments from a file (`args.txt`) and supply it as:

```
./cecho.sh –p arg1 –l
./cecho.sh –p arg2 –l
./cecho.sh –p arg3 –l
```

To provide a command execution sequence as shown, `xargs` has an option `–I`. By using `–I` we can specify a replacement string that will be replaced while `xargs` expands. When `–I` is used with `xargs`, it will execute as one command execution per argument.

Let's do it as follows:

```
$ cat args.txt | xargs -I {} ./cecho.sh -p {} -l
-p arg1 -l #
-p arg2 -l #
-p arg3 -l #
```

`-I {}` specifies the replacement string. For each of the arguments supplied for the command, the `{}` string will be replaced with arguments read through `stdin`. When used with `-I`, the command is executed like in a loop. When there are three arguments the command is executed three times along with the command `{}`. Each time `{}` is replaced with arguments one by one.

Using xargs with find

`xargs` and `find` are best friends. They can be combined to perform tasks easily. Usually, people combine them in a wrong way. For example:

```
$ find . -type f -name "*.txt"  -print | xargs rm -f
```

This is dangerous. It may sometimes cause removal of unnecessary files. Here, we cannot predict the delimiting character (whether it is `'\n'` or `' '`) for the output of the `find` command. Many of the filenames may contain a space character (' ') and hence `xargs` may misinterpret it as a delimiter (for example, "hell text.txt" is misinterpreted by `xargs` as "hell" and "text.txt").

Hence we must use `-print0` along with `find` to produce an output with delimited character null (`'\0'`) whenever we use the `find` output as the `xargs` input.

Let's use `find` to match and list of all the `.txt` files and remove them using `xargs`:

```
$ find . -type f -name "*.txt" -print0 | xargs -0 rm -f
```

This removes all `.txt` files. `xargs -0` interprets that the delimiting character is `\0`.

Counting number of lines of C code in a source code directory over many C files.

This is a task most programmers do, that is, counting all C program files for LOC (Lines of Code). The code for this task is as follows:

```
$ find source_code_dir_path -type f -name "*.c" -print0 | xargs -0 wc -l
```

While and subshell trick with stdin

`xargs` is restricted to provide arguments in limited ways to supply arguments. Also, xargs cannot supply arguments to multiple set of commands. For executing commands with collected arguments from standard input, we have a very flexible method. I call it a subshell hack. A subshell with a `while` loop can be used to read arguments and execute commands in a trickier way as follows:

```
$ cat files.txt  | ( while read arg; do cat $arg; done )
# Equivalent to cat files.txt | xargs -I {} cat {}
```

Here, by replacing `cat $arg` with any number of commands using a `while` loop, we can perform many command actions with same arguments. We can also pass the output to other commands without using pipes. Subshell `( )` tricks can be used in a variety of problem environments. When enclosed within subshell operators, it acts as a single unit with multiple commands inside.

```
$ cmd0 | ( cmd1;cmd2;cmd3) | cmd4
```

If `cmd1` is `cd /`, within the subshell, the path of the working directory changes. However, this change resides inside the subshell only. `cmd4` will not see the directory change.

Translating with tr

`tr` is a small and beautiful command in the UNIX command-warrior toolkit. It is one of the important commands frequently used to craft beautiful one-liner commands.

`tr` can be used to perform substitution of characters, deletion of the characters, and squeezing of repeated characters from the standard input. It is often called translate, since it can translate a set of characters to another set.

Getting ready

`tr` accepts input only through `stdin` (standard input). It cannot accept input through command-line arguments. It has the following invocation format:

```
tr [options] set1 set2
```

Input characters from `stdin` are mapped from `set1` to `set2` and the output is written to `stdout` (standard output). `set1` and `set2` are character classes or a set of characters. If the length of sets is unequal, `set2` is extended to the length of `set1` by repeating the last character, or else, if the length of `set2` is greater than that of `set1`, all the characters exceeding the length of `set1` are ignored from `set2`.

How to do it...

In order to perform translation of characters in the input from uppercase to lowercase, use the following command:

```
$ echo "HELLO WHO IS THIS" | tr 'A-Z' 'a-z'
```

`'A-Z'` and `'a-z'` are the sets. We can specify custom sets as needed by appending characters or character classes.

`'ABD-}'`, `'aA.,'`, `'a-ce-x'`, `'a-c0-9'`, and so on are valid sets. We can define sets easily. Instead of writing continuous character sequences, we can use the `'startchar-endchar'` format. It can also be combined with any other characters or character classes. If `startchar-endchar` are not a valid continuous character sequence, then they are taken as a set of three characters (for example, `startchar`, `-`, and `endchar`). You can also use special characters such as `'\t'`, `'\n'`, or any ASCII characters.

How it works...

By using `tr` with the concept of sets, we can map characters from one set to another set easily. Let's go through an example on how to use `tr` for encrypting and decrypting numeric characters:

```
$ echo 12345 | tr '0-9' '9876543210'
87654 #Encrypted

$ echo 87654 | tr '9876543210' '0-9'
12345 #Decrypted
```

Let's try another interesting example.

ROT13 is a well known encryption algorithm. In the ROT13 scheme, the same function is used to encrypt and decrypt text. The ROT13 scheme performs alphabetic rotation of characters for 13 characters. Let's perform ROT13 using `tr` as follows:

```
$ echo "tr came, tr saw, tr conquered." | tr
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
'NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm'
```

The output will be:

```
ge pnzr, ge fnj, ge pbadhrerq.
```

By sending the encrypted text again to the same ROT13 function, we get:

```
$ echo ge pnzr, ge fnj, ge pbadhrerq. | tr
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
'NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm'
```

The output will be:

```
tr came, tr saw, tr conquered.
```

`tr` can be used to convert tab characters into space as follows:

```
$ cat text | tr '\t' ' '
```

There's more...

Deleting characters using tr

`tr` has an option `-d` to delete a set of characters that appear on `stdin` by using the specified set of characters to be deleted as follows:

```
$ cat file.txt | tr -d  '[set1]'
#Only set1 is used, not set2
```

For example:

```
$ echo "Hello 123 world 456" | tr -d '0-9'
Hello  world
# Removes the numbers from stdin and print
```

Complementing character set

We can use a set complement to `set1` by using the `-c` flag. `-c [set]` is equivalent to specifying a set (complement-set) which contains complement characters of `[set]`:

```
tr -c [set1] [set2]
```

The complement of `set1` means that it is the set having all the characters except for characters in `set1`.

The best usage example is to delete all the characters from the input text except the ones specified in the complement set. For example:

```
$ echo hello 1 char 2 next 4 | tr -d -c '0-9 \n'
 1  2  4
```

Here, the complement set is the set containing all numerals, the space character, and newline. All other characters are removed since `–d` is used with `tr`.

Squeezing characters with tr

The `tr` command is very helpful in many text processing contexts. Repeated continuous characters should be squeezed to a single character in many circumstances. Squeezing of whitespace is a frequently occurring task.

`tr` provides the `–s` option to squeeze repeating characters from the input. It can be performed as follows:

```
$ echo "GNU is       not     UNIX. Recursive   right ?" | tr -s ' '
GNU is not UNIX. Recursive right ?
# tr -s '[set]'
```

Let's use `tr` in a tricky way to add a given list of numbers from a file as follows:

```
$ cat sum.txt
1
2
3
4
5
$ cat sum.txt | echo $[ $(tr '\n' '+' ) 0 ]
15
```

How does this hack work?

Here, the `tr` command is used to replace `'\n'` with the `'+'` character, hence we form the string `"1+2+3+..5+"`, but at the end of the string we have an extra `+` operator. In order to nullify the effect of the `+` operator, `0` is appended.

`$[ operation ]` performs a numeric operation. Hence it forms the string as follows:

```
echo $[ 1+2+3+4+5+0 ]
```

If we use a loop to perform addition by reading numbers from a file, it would take few lines of code. Here a one-liner does the trick. The skill of crafting one-liners is attained by practice.

Character classes

`tr` can use different character classes as sets. The different classes are as follows:

- `alnum`: Alphanumeric characters
- `alpha`: Alphabetic characters
- `cntrl`: Control (non-printing) characters
- `digit`: Numeric characters
- `graph`: Graphic characters
- `lower`: Lower-case alphabetic characters
- `print`: Printable characters
- `punct`: Punctuation characters
- `space`: Whitespace characters
- `upper`: Upper-case characters
- `xdigit`: Hexadecimal characters

We can select the required classes and use them with as follows:

```
tr [:class:] [:class:]
```

For example:

```
tr '[:lower:]' '[:upper:]'
```

Checksum and verification

Checksum programs are used to generate checksum key strings from the files and verify the integrity of the files later by using that checksum string. A file might be distributed over the network or any storage media to different destinations. Due to many reasons, there are chances for the file being corrupted due to a few bits missing during the data transfer. These errors happen most often while downloading the files from the Internet, transferring through the network, CD ROM damage, and so on.

Hence, we need to know whether the received file is the correct one or not by applying some kind of test. The special key string that is used for this file integrity test is known as **checksum**.

We calculate the checksum for the original file as well as the received file. By comparing both of the checksums, we can verify whether the received file is the correct one or not. If the checksums (calculated from original file at the source location and the one calculated from destination) are equal, it means that we have received the correct file without causing any erroneous data loss during the data transfer, or else, the user has to repeat the data transfer and try the checksum comparison again.

Checksums are crucial while writing backup scripts or maintenance scripts that consist of transfer of files through the network. By using checksum verification, files corrupted during the data transfer over the network can be identified and those files can be resend again from the source to the destination. Thus the integrity of the data received can always be ensured.

Getting ready

The most famous and widely-used checksum techniques are `md5sum` and `sha1sum`. They generate checksum strings by applying the corresponding algorithm to the file content. Let's see how can we generate a checksum and from a file and verify the integrity of a file.

How to do it...

In order to compute the `md5sum`, use the following command:

```
$ md5sum filename
68b329da9893e34099c7d8ad5cb9c940 filename
```

An `md5sum` is a 32 character hexadecimal string, as given above.

We redirect the checksum output into a file and use that MD5 file for verification as follows:

```
$ md5sum filename > file_sum.md5
```

How it works...

The syntax for `md5sum` checksum calculation is as follows:

```
$ md5sum file1 file2 file3 ..
```

When multiple files are used, the output will contain a checksum for each of the file having one checksum string per line, as follows:

```
[checksum1]   file1
[checksum1]   file2
[checksum1]   file3
```

The integrity of a file can be verified by using the generated file as follows:

```
$ md5sum -c file_sum.md5
# It will output message whether checksum matches or not
```

Or, alternately, if we need to check all the files using all `.md5` info available, use:

```
$ md5sum *.md5
```

SHA1 is another commonly used checksum algorithm like md5sum. It generates a 40-character hex code from a given input file. The command used for calculating a SHA1 string is `sha1sum`. Its usage is very similar to that of `md5sum`. Replace `md5sum` with `sha1sum` in all the commands mentioned previously in this recipe. Instead of `file_sum.md5`, change the output filename to `file_sum.sha1`.

Checksum verification is much useful to verify the integrity of files that we download from the Internet. The ISO images that we download from the Internet are usually much more prone to erroneous bits. Therefore, to check whether we received the file correctly, checksums are widely used. For the same file data the checksum program will always produce the same checksum string.

There's more...

Checksum are also useful when used with a number of files. Let's see how to apply checksum to many files and verify correctness.

Checksum for directories

Checksums are calculated for files. Calculating the checksum for a directory would mean that we will need to calculate the checksums for all the files in the directory, recursively.

It can be achieved by the command `md5deep` or `sha1deep`. Install the package `md5deep` to make these commands available. An example of this command is as follows:

```
$ md5deep -rl directory_path > directory.md5
# -r  for enable recursive.
# -l for using relative path. By default it writes absolute file path in
output
```

Alternately, combine it with `find` to calculate checksums recursively:

```
$ find directory_path -type f -print0 | xargs -0 md5sum >> directory.md5
```

To verify, use the following command:

```
$ md5sum -c directory.md5
```

Sorting, unique and duplicates

Sorting is a common task that we always encounter with text files. Hence, in text processing tasks, sort is very useful. `sort` commands help us to perform sort operations over text files and `stdin`. Most often, it can also be coupled with many other commands to produce the required output. `uniq` is another command that is often used along with a `sort` command. It helps to extract unique lines from a text or `stdin`. `sort` and `uniq` can be coupled to find duplicates. This recipe illustrates most of the use cases with `sort` and `uniq` commands.

Getting ready

The `sort` command accepts input as filenames as well as from `stdin` (Standard input) and outputs the result by writing into `stdout`. The `uniq` command follows the same sequence of operation.

How to do it...

We can easily sort a given set of files (for example, `file1.txt` and `file2.txt`) as follows:

```
$ sort file1.txt file2.txt .. > sorted.txt
```

Or:

```
$ sort file1.txt file2.txt .. -o sorted.txt
```

In order to find the unique lines from a sorted file, use:

```
$ cat sorted_file.txt | uniq> uniq_lines.txt
```

How it works...

There are numerous scenarios where `sort` and `uniq` commands can be used. Let's go through various options and usage techniques.

For numerical sort use:

```
$ sort -n file.txt
```

To sort in reverse order use:

```
$ sort -r file.txt
```

For sorting by months (in the order Jan, Feb, March) use:

```
$ sort -M months.txt
```

A file can be tested whether sorted or not as follows:

```
#!/bin/bash
#Desc: Sort
sort -C file ;
if [ $? -eq 0 ]; then
   echo Sorted;
else
   echo Unsorted;
fi
# If we are checking numerical sort, it should be sort -nC
```

In order to merge two sorted files without sorting again, use:

```
$ sort -m sorted1 sorted2
```

There's more...

Sort according to the keys or columns

We use sort by column if we need to sort a text as follows:

```
$ cat data.txt
1  mac      2000
2  winxp    4000
3  bsd      1000
4  linux    1000
```

We can sort this in many ways; currently it is numeric sorted by serial number (the first column). We can also sort by second column and third column.

`-k` specifies the key by which the sort is to be performed. Key is the column number by which sort is to be done. `-r` specifies the sort command to sort in the reverse order. For example:

```
# Sort reverse by column1
$ sort -nrk 1  data.txt
4      linux         1000
3      bsd           1000
2      winxp         4000
1      mac           2000
# -nr means numeric and reverse

# Sort by column 2
$ sort -k 2  data.txt
```

```
3       bsd             1000
4       linux           1000
1       mac             2000
2       winxp           4000
```

Always be careful about the `-n` option for numeric sort. The sort command treats alphabetical sort and numeric sort differently. Hence, in order to specify numeric sort the `-n` option should be provided.

Usually, by default, keys are columns in the text file. Columns are separated by space characters. But in certain circumstances, we will need to specify keys as a group of characters in the given character number range (for example, key1= character4-character8). In such cases where keys are to be specified explicitly as a range of characters, we can specify the keys as ranges with the character position at key starts and key ends as follows:

```
$ cat data.txt
1010hellothis
2189ababbba
7464dfddfdfd
$ sort -nk 2,3 data.txt
```

The highlighted characters are to be used as numeric keys. In order to extract them, use their start-pos and end-pos as the key format.

In order to use the first character as the key, use:

```
$ sort -nk 1,1 data.txt
```

Make the sort's output xargs compatible with `\0` terminator, by using the following command:

```
$ sort -z data.txt | xargs -0
#Zero terminator is used to make safe use with xargs
```

Sometimes the text may contain unnecessary extraneous characters like spaces. To sort by ignoring them in dictionary order by ignoring punctuations and folds, use:

```
$ sort -bd unsorted.txt
```

The option `-b` is used to ignore leading blanks from the file and the `-d` option is used to specify sort in the dictionary order.

uniq

`uniq` is a command used to find out the unique lines from the given input (`stdin` or from filename as command argument) by eliminating the duplicates. It can also be used to find out the duplicate lines from the input. `uniq` can be applied only for sorted data input. Hence, `uniq` is to be used always along with the `sort` command using pipe or using a sorted file as input.

You can produce the unique lines (unique lines means that all lines in the input are printed, but the duplicate lines are printed only once) from the given input data as follows:

```
$ cat sorted.txt
bash
foss
hack
hack

$ uniq sorted.txt
bash
foss
hack
```

Or:

```
$ sort unsorted.txt | uniq
```

Or:

```
$ sort -u unsorted.txt
```

Display only unique lines (the lines which are not repeated or duplicate in input file) as follows:

```
$ uniq -u sorted.txt
bash
foss
```

Or:

```
$ sort unsorted.txt | uniq -u
```

In order to count how many times each of the line appears in the file, use the following command:

```
$ sort unsorted.txt | uniq -c
      1 bash
      1 foss
      2 hack
```

Find duplicate lines in the file as follows:

```
$ sort unsorted.txt  | uniq -d
hack
```

To specify keys, we can use the combination of `-s` and `-w` arguments.

- `-s` specifies the number for the first `N` characters to be skipped
- `-w` specifies the maximum number of characters to be compared

This comparison key is used as the index for the `uniq` operation as follows:

```
$ cat data.txt
u:01:gnu
d:04:linux
u:01:bash
u:01:hack
```

We need to use the highlighted characters as the uniqueness key. This is used to ignore the first 2 characters (`-s 2`) and the max number of comparison characters is specified using the `–w` option `(-w 2)`:

```
$ sort data.txt | uniq -s 2 -w 2
d:04:linux
u:01:bash
```

While we use output from one command as input to the xargs command, it is always preferable to use a zero byte terminator for each of the lines of the output, which acts as source for `xargs`. While using the `uniq` commands output as the source for `xargs`, we should use a zero terminated output. If a zero byte terminator is not used, space characters are by default taken as delimiter to split the arguments in the `xargs` command. For example, a line with text "this is a line" from `stdin` will be taken as four separate arguments by the `xargs`. But, actually, it is a single line. When a zero byte terminator is used, `\0` is used as the delimiter character and hence, a single line including space is interpreted as a single argument.

Zero byte terminated output can be generated from the `uniq` command as follows:

```
$ uniq -z file.txt
```

The following command removes all the files, with filenames read from `files.txt`:

```
$ uniq –z file.txt | xargs -0 rm
```

If multiple line entries of filenames exist in the file, the `uniq` command writes the filename only once to `stdout`.

String pattern generation with uniq

Here is an interesting question for you: We have a string containing repeated characters. How can we find the number of times each of the character appears in the string and output a string in the following format?

Input: ahebhaaa

Output: 4a1b1e2h

Each of the characters is repeated once, and each of them is prefixed with the number of times they appear in the string. We can solve this using `uniq` and `sort` as follows:

```
INPUT= "ahebhaaa"
OUTPUT=` echo $INPUT | sed 's/[^\n]/&\n/g' | sed '/^$/d' | sort | uniq
-c | tr -d ' \n'`
echo $OUTPUT
```

In the above code, we can split each of the piped commands as follows:

```
echo $INPUT  # Print the input to stdout
sed 's/./&\n/g'
```

Append a newline character to each of the characters so that only one character appears in one line. This is done to make the characters sortable by using the `sort` command. The `sort` command can take only items delimited by newline.

- `sed '/^$/d'`: Here the last character is replaced as character `+\n`. Hence an extra newline is formed and it will form a blank line at the end. This command removes the blank line from the end.
- `sort`: Since each character appears in each line, it can be sorted so that it can serve as input to uniq.
- `uniq –c`: This command prints each of the line with how many times they got repeated(count).
- `tr –d ' \n'`: This removes the space characters and newline characters from the input so that output can be produced in the given format.

Temporary file naming and random numbers

While writing shell scripts, we use often require to store temporary data. The most suitable location to store temporary data is `/tmp` (which will be cleaned out by the system on reboot). We can use two methods to generate standard filenames for temporary data.

How to do it...

`tempfile` is not seen in non-Debian Linux distributions. The `tempfile` command comes shipped with Debian-based distributions, such as Ubuntu, Debian, and many more.

The following code will assign a temporary filename to the variable `temp_file`:

```
temp_file=$(tempfile)
```

Use `echo $temp_file` to print the temporary file name in the terminal.

The output will look similar to `/tmp/fileaZWm8Y`.

Sometimes we may use a filename with a random number attached to it as a temporary filename. This can be done as follows:

```
temp_file="/tmp/file-$RANDOM"
```

The `$RANDOM` environment variable always returns a random number.

How it works...

Instead of using the `tempfile` command, we can also use our own temporary. Most experienced UNIX programmers use the following conventions:

```
temp_file="/tmp/var.$$"
```

The `.$$` suffix is attached. `$$` is expanded as the process ID of the current script upon execution.

Splitting files and data

Splitting of files into many smaller pieces becomes essential in certain situations. Earlier, when memory was limited with devices like floppy disks, it was crucial to split files into smaller file sizes to transfer data in many disks. However, nowadays we split files for other purposes, such as readability, for generating logs, and so on.

How to do it...

Generate a test file (`data.file`) of 100kb as follows:

```
$ dd if=/dev/zero bs=100k count=1 of=data.file
```

The above command creates a file filled with zeros with the size of 100kb.

You can split files into smaller files by specifying the split size as follows:

```
$ split -b 10k data.file
$ ls
data.file  xaa  xab  xac  xad  xae  xaf  xag  xah  xai  xaj
```

It will split `data.file` into many files, each of a 10k chunk. The chunks will be named in the manner `xab`, `xac`, `xad`, and so on. This means it will have alphabetic suffixes. To use the numeric suffixes, use an additional `-d` argument. It is also possible to specify a suffix length using `-a length` as follows:

```
$ split -b 10k data.file -d  -a 4
$ ls
data.file x0009  x0019  x0029  x0039  x0049  x0059  x0069  x0079
```

Instead of the `k` (kilobyte) suffix we can use `M` for MB, `G` for GB, `c` for byte, `w` for word, and so on.

There's more...

The `split` command has more options. Let's go through them.

Specifying filename prefix for the split files

The above split files have a filename prefix "x". We can also use our own filename prefix by providing a prefix filename. The last command argument for the split command is `PREFIX`. It is in the format:

```
$ split [COMMAND_ARGS] PREFIX
```

Let's run the previous command with the prefix filename for split files:

```
$ split -b 10k data.file -d  -a 4 split_file
$ ls
data.file       split_file0002  split_file0005  split_file0008  strtok.c
split_file0000  split_file0003  split_file0006  split_file0009
split_file0001  split_file0004  split_file0007
```

In order to split files based on number of lines in each split rather than chunk size, use `-l no_of_lines` as follows:

```
$ split -l 10 data.file
# Splits into files of 10 lines each.
```

There is another interesting utility called `csplit`. It can be used to split log file-based specified conditions and string match options. Let's see how to work with it.

`csplit` is a variant of the `split` utility. The `split` utility can only split files based on chunk size or based on the number of lines. `csplit` makes the split based on context based split. It can be used to split files based on existence of a certain word or text content.

Look at the example log:

```
$ cat server.log
SERVER-1
[connection] 192.168.0.1 success
[connection] 192.168.0.2 failed
[disconnect] 192.168.0.3 pending
[connection] 192.168.0.4 success
SERVER-2
[connection] 192.168.0.1 failed
[connection] 192.168.0.2 failed
[disconnect] 192.168.0.3 success
[connection] 192.168.0.4 failed
SERVER-3
[connection] 192.168.0.1 pending
[connection] 192.168.0.2 pending
[disconnect] 192.168.0.3 pending
[connection] 192.168.0.4 failed
```

We may need to split the files into `server1.log`, `server2.log`, and `server3.log` from the contents for each `SERVER` in each file. This can be done as follows:

```
 $ csplit server.log /SERVER/ -n 2 -s {*}  -f server -b "%02d.log"  ; rm
server00.log

$ ls
server01.log  server02.log  server03.log  server.log
```

The details of the command are as follows:

- `/SERVER/` is the line used to match a line by which the split is to be carried out.
- `/[REGEX]/` is the format. It copies from current line (first line) upto the matching line that contains `"SERVER"` excluding match line.
- `{*}` is used to specify to repeat splitting based on match upto the end of the file. By using `{integer}`, we can specify no of times it is to be continued.
- `-s` is the flag to make the command silent rather than printing other messages.
- `-n` is used to specify the number of digits to be used as suffix. 01, 02, 03, and so on.

- `-f` is used for specifying the filename prefix for split files ("server" is the prefix in the previous example).
- `-b` is used to specify the suffix format. `"%02d.log"` is similar to the `printf` argument format in C. Here the filename = prefix + suffix = `"server" + "%02d.log"`.

We remove `server00.log` since the first split file is an empty file (the match word is the first line of the file).

Slicing filenames based on extension

Several custom shell scripts perform manipulations based on file names. We may need to perform actions like renaming the files by preserving extension, converting files from one format to another (change the extension by preserving the name), extracting a portion of the file name, and so on. The shell comes with inbuilt functionalities for slicing filenames based on different conditions. Let's see how to do it.

How to do it...

The name from `name.extension` can be easily extracted by using the `%` operator. You can extract the name from `"sample.jpg"` as follows:

```
file_jpg="sample.jpg"
name=${file_jpg%.*}
echo File name is: $name
```

The output is:

```
File name is: sample
```

The next task is to extract the extension of a file from its filename. The extension can be extracted using the # operator.

Extract `.jpg` from the a filename stored in variable `file_jpg` as follows:

```
extension=${file_jpg#*.}
echo Extension is: jpg
```

The output is:

```
Extension is: jpg
```

How it works..

In the first task, in order to extract the name from the file name in the format `name.extension` we have used the `%` operator.

`${VAR%.*}` can be interpreted as:

- Remove the string match from the `$VARIABLE` for the wildcard pattern that appears to the right-hand side of % (.* in the previous example). Evaluating from the right to the left direction should make the wildcard match.
- Let `VAR=sample.jpg`. Therefore, the wildcard match for `.*` from right to left is `.jpg`. Thus it is removed from the `$VAR` string and the output will be "sample".

`%` is a non-greedy operation. It finds the minimal match for the wildcard from the right to left. There is an operator `%%`, which is similar to `%`. But it is greedy in nature. That means it matches the maximal string for the wildcard.

For example, we have:

```
VAR=hack.fun.book.txt
```

By using the % operator, we have:

```
$ echo ${VAR%.*}
```

The output will be: `hack.fun.book`.

The operator `%` performs a non-greedy match for `.*` from right to left (`.txt`).

By using the %% operator, we have:

```
$ echo ${VAR%%.*}
```

The output will be: `hack`

The `%%` operator matches greedy match for `.*` from right to left (`.fun.book.txt`).

In the second task, we have used the # operator to extract the extension from the filename. It is similar to `%`. But it evaluates from left to right.

`${VAR#*.}` can be interpreted as:

Remove the string match from the `$VARIABLE` for the wildcard pattern match appears right side to the # (`*.` in the above example). Evaluating from the left to right direction should make the wildcard match.

Similarly, as in the case of `%%`, we have another greedy operator for #, which is ##.

It makes greedy matches by evaluating from left to right and removes the match string from the specified variable.

Let's use this example:

```
VAR=hack.fun.book.txt
```

By using the # operator, we have:

```
$ echo ${VAR#*.}
```

The output will be: `fun.book.txt`.

The operator # performs a non-greedy match for `*.` from left to right (`hack.`).

By using the ## operator, we have:

```
$ echo ${VAR##*.}
```

The output will be: `txt`.

The operator ## matches greedy match for *. from left to right (txt).

The ## operator is more preferred over the # operator to extract an extension from a filename since the filename may contain multiple '.' characters. Since ## makes greedy match, it always extract extensions only.

Here is practical example that can be used to extract different portions of a domain name, given URL="`www.google.com`":

```
$ echo ${URL%.*} # Remove rightmost .*
www.google
$ echo ${URL%%.*} # Remove right to leftmost  .* (Greedy operator)
www
$ echo ${URL#*.} # Remove leftmost  part before *.
google.com
$ echo ${URL##*.} # Remove left to rightmost  part before *. (Greedy
operator)
com
```

Renaming and moving files in bulk

Renaming a number of files is one of the tasks we frequently come across. A simple example is, when you download photos from your digital camera to the computer you may delete unnecessary files and it causes discontinuous numbering of image files. Sometimes you many need to rename them with custom prefix and continuous numbering for filenames. We sometimes use third-party tools for performing rename operations. We can use Bash commands to perform a rename operation in a couple of seconds.

Moving all the files having a particular substring in the filename (for example, same prefix for filenames) or with a specific file type to a given directory is another use case we frequently perform. Let's see how to write scripts to perform these kinds of operations.

Getting ready

The `rename` command helps to change file names using Perl regular expressions. By combining the commands `find`, `rename`, and `mv`, we can perform a lot of things.

How to do it...

The easiest way of renaming image files in the current directory to our own filename with a specific format is by using the following script:

```
#!/bin/bash
#Filename: rename.sh
#Description: Rename jpg and png files

count=1;
for img in *.jpg *.png
do
new=image-$count.${img##*.}

mv "$img" "$new" 2> /dev/null

if [ $? -eq  0 ];
then

echo "Renaming $img to $new"
let count++

fi

done
```

The output is as follows:

```
$ ./rename.sh
Renaming hack.jpg to image-1.jpg
Renaming new.jpg to image-2.jpg
Renaming next.jpg to image-3.jpg
```

The script renames all the `.jpg` and `.png` files in the current directory to new filenames in the format `image-1.jpg`, `image-2.jpg`, `image-3.jpg` , `image-4.png`, and so on.

How it works...

In the above rename script, we have used a `for` loop to iterate through the names of all files ending with a `.jpg` extension. The wildcard `*.jpg` and `*.png` are used to match all the JPEG and PNG files. We can do a small improvisation over the extension match. The `.jpg` wildcard matches only the extension in lowercase. However, we can make it case insensitive by replacing `.jpg` with `.[jJ][pP][gG]`. Hence it can match files like `file.jpg` as well as `file.JPG` or `file.Jpg`. In Bash, when characters are enclosed in `[]`, it means to match one character from the set of characters enclosed in `[]`.

`for img in *.jpg  *.png` in the above code will be expanded as follows:

```
for img in hack.jpg new.jpg next.jpg
```

We have initialized a variable `count=1` in order to keep track of the image number. The next step is to rename the file using the `mv` command. The new name of the file should be formulated for renaming. `${img##*.}` in the script parses the extension of the filename currently in the loop (see the *Slicing file names based on extension* recipe for interpretation of `${img##*.}`).

`let count++` is used to increment the file number for each execution of loop.

You can see that error redirection (`stderr`) to `/dev/null` is done for the `mv` command using the `2>` operator. This is to stop the error messages being printed into the terminal.

Since we use `*.png` and `*.jpg`, if atleast one image for a wildcard match is not present, the shell will interpret the wildcard itself as a string. In the above output, you can see that `.png` files are not present. Hence it will take `*.png` as yet another filename and execute `mv *.png image-X.png`, which will cause an error. An `if` statement with `[ $? -eq 0 ]` is used to check the exit status (`$?`). The value of `$?` will be 0 if the last executed command is successful, else it returns non-zero. When the `mv` command fails, it returns non-zero and, therefore, the message "Renaming file" will not be shown to the user, as well as the count will not be incremented.

There are a variety of other ways to perform rename operations. Let's walk through a few of them.

Renaming `*.JPG` to `*.jpg`:

```
$ rename *.JPG *.jpg
```

Replace space in the filenames with the `"_"` character as follows:

```
$ rename 's/ /_/g' *
```

`# 's/ /_/g'` is the replacement part in the filename and `*` is the wildcard for the target files. It can be `*.txt` or any other wildcard pattern.

You can convert any filename of files from uppercase to lowercase and vice versa as follows:

```
$ rename 'y/A-Z/a-z/' *
$ rename 'y/a-z/A-Z/' *
```

In order to recursively move all the `.mp3` files to a given directory, use:

```
$ find path -type f -name "*.mp3" -exec mv {} target_dir \;
```

Recursively rename all the files by replacing space with `"_"` character as follows:

```
$ find path -type f -exec rename 's/ /_/g' {} \;
```

Spell checking and dictionary manipulation

Most Linux distributions come with a dictionary file. However, I find few people are aware of the dictionary file and hence many people fail to make use of them. There is a command-line utility called `aspell` that functions as a spell checker. Let's go through few scripts that make use of the dictionary file and the spell checker.

How to do it...

The `/usr/share/dict/` directory contains some of the dictionary files. Dictionary files are text files that contain a list of dictionary words. We can use this list to check whether a word is a dictionary word or not.

```
$ ls /usr/share/dict/
american-english  british-english
```

In order to check whether the given word is a dictionary word, use the following script:

```
#!/bin/bash
#Filename: checkword.sh
word=$1
grep "^$1$" /usr/share/dict/british-english -q
if [ $? -eq 0 ]; then
  echo $word is a dictionary word;
else
  echo $word is not a dictionary word;
fi
```

The usage is as follows:

```
$ ./checkword.sh ful
ful is not a dictionary word

$ ./checkword.sh fool
fool is a dictionary word
```

How it works...

In `grep`, `^` is the word start marker character and the character `$` is the word end marker.

`-q` is used to suppress any output and to be silent.

Or, alternately, we can use the spell check, `aspell`, to check whether a word is in a dictionary or not as follows:

```
#!/bin/bash
#Filename: aspellcheck.sh
word=$1
output=`echo \"$word\" | aspell list`
if [ -z $output ]; then
        echo $word is a dictionary word;
else
        echo $word is not a dictionary word;
fi
```

The `aspell list` command returns output text when the given input is not a dictionary word, and does not output anything when a dictionary word is the input. A `-z` check ensures whether `$output` is an empty string or not.

List all words in a file starting with a given word as follows:

```
$ look word filepath
```

Or alternately, use:

```
$ grep "^word" filepath
```

By default, if the filename argument is not given to the `look` command, it uses the default dictionary (`/usr/share/dict/words`) and returns an output.

```
$ look word
# When used like this it takes default dictionary as file
```

For example:

```
$ look android
android
android's
androids
```

Automating interactive input

Automating interactive input for command-line utilities are extremely useful for writing automation tools or testing tools. There will be many situations when we deal with commands that read inputs interactively. Interactive input is the input typed by the user only when the command asks for some input. An example for execution of a command and supply of interactive input is as follows:

```
$ command
Enter a number: 1
Enter name : hello
You have entered 1,hello
```

Getting ready

Automating utilities which can automate the acceptance of input as in the above mentioned manner are useful to supply input to local commands as well as for remote applications. Let's see how to automate them.

How to do it...

Think about the sequence of an interactive input. From the previous code we can formulate the steps of the sequence as follows:

`1 [Return] hello [Return]`

Converting the above steps `1, Return, hello, Return` by observing the characters that are actually typed in the keyboard, we can formulate the following string.

`"1\nhello\n"`

The `\n` character is sent when we press *Return*. By appending return (`\n`) characters, we get the actual string that is passed to the `stdin` (standard input).

Hence by sending the equivalent string for the characters typed by the user, we can automate the passing of input in the interactive processes.

How it works...

Let's write a script that reads input interactively and uses this script for automation examples:

```
#!/bin/bash
#Filename: interactive.sh
read -p "Enter number:" no ;
read -p "Enter name:" name
echo You have entered $no, $name;
```

Let's automate the sending of input to the command as follows:

```
$ echo -e "1\nhello\n" | ./interactive.sh
You have entered 1, hello
```

Thus crafting inputs with `\n` works.

We have used `echo -e` to produce the input sequence. If the input is large we can use an input file and redirection operator to supply input.

```
$ echo -e "1\nhello\n"  > input.data
$ cat input.data
1
hello
```

You can also manually craft the input file without `echo` commands by hand typing. For example:

```
$ ./interactive.sh < input.data
```

This redirects interactive input data from a file.

If you are a reverse engineer, you may have played with buffer overflow exploits. To exploit them we need to redirect shellcode like `"\xeb\x1a\x5e\x31\xc0\x88\x46"`, which is written in hex. These characters cannot be typed directly through keyboard since, keys for these characters are not present in the keyboard. Therefore we should use:

```
echo -e "\xeb\x1a\x5e\x31\xc0\x88\x46"
```

This will redirect shellcode to a vulnerable executable.

We have described a method to automate interactive input programs by redirecting expected input text through `stdin` (standard input). We are sending the input without checking the input the program asks for. We are sending the input by expecting the program to ask input in a specific (static) order. If the program asks input randomly or in a changing order, or sometimes certain inputs are never asked, the above method fails. It will send wrong inputs to different input prompts by the program. In order to handle dynamic input supply and provide input by checking the input requirements by the program on runtime, we have a great utility called `expect`. The `expect` command supplies correct input for the correct input prompt by the program. Let's see how to use `expect`.

There's more...

Automation of interactive input can also be done using other methods. Expect scripting is another method for automation. Let's go through it.

Automating with expect

The `expect` utility does not come by default with most of the common Linux distributions. You have to install the expect package manually using package manager.

`expect` expects for a particular input prompt and sends data by checking message in the input prompt.

```
#!/usr/bin/expect
#Filename: automate_expect.sh
spawn ./interactive .sh
expect "Enter number:"
send "1\n"
expect "Enter name:"
send "hello\n"
expect eof
```

Run as:

```
$ ./automate_expect.sh
```

In this script:

- `spawn` parameter specifies which command is to be automated
- `expect` parameter provides the expected message
- `send` is the message to be sent.
- `expect eof` defines the end of command interaction

3
File In, File Out

In this chapter, we will cover:

- Generating files of any size
- Intersection and set difference (A-B) on text files
- Finding and deleting duplicate files
- Making directories for a long path
- File permissions, ownership and sticky bit
- Making files immutable
- Generating blank files in bulk
- Finding symbolic links and its target
- Enumerating file type statistics
- Loopback files and mounting
- Creating ISO files, Hybrid ISO
- Finding difference between files, patching
- head and tail - printing the last or first 10 lines
- Listing only directories - alternative methods
- Fast command line directory navigation using pushd and popd
- Counting the number of lines, words, and characters in a file
- Printing directory tree

Introduction

UNIX treats every object in the operating system as a file. We can find the files associated with every action performed and can make use of them for different system- or process-related manipulations. For example, the command terminal that we use is associated with a device file. We can write to the terminal by writing to the corresponding device file for that specific terminal. Files take different forms such as directories, regular files, block devices, character special devices, symbolic links, sockets, named pipes, and so on. Filename, size, file type, modification time, access time, change time, inode, links associated, and the filesystem the file is on are all attributes and properties that files can have. This chapter deals with recipes that handle any of the operations or properties related to files.

Generating files of any size

For various reasons, you may need to generate a file filled with random data. It may be for creating a test file to perform tests, such as an application efficiency test that uses a large file as input, or to test the splitting of files into many, or to create loopback filesystems (loopback files are files that can contain a filesystem itself and these files can be mounted similar to a physical device using the `mount` command). It is hard to create such files by writing specific programs. So we use general utilities.

How to do it...

The easiest way to create a large sized file with a given size is to use the `dd` command. The `dd` command clones the given input and writes an exact copy to the output. Input can be `stdin`, a device file, a regular file, or so on. Output can be `stdout`, a device file, a regular file, and so on. An example of the `dd` command is as follows:

```
$ dd if=/dev/zero of=junk.data bs=1M count=1
1+0 records in
1+0 records out
1048576 bytes (1.0 MB) copied, 0.00767266 s, 137 MB/s
```

The above command will create a file called `junk.data` that is exactly 1MB in size. Let's go through the parameters: `if` stands for `– input` file, `of` stands for `– output` file, `bs` stands for BYTES for a block, and `count` stands for the number of blocks of `bs` specified to be copied.

Here we are only creating a file 1MB in size by specifying `bs` as 1MB with a count of 1. If `bs` was set to `2M` and a count to 2, the total file size would be 4MB.

We can use various units for **Block Size** (**BS**) as follows. Append any of the following characters to the number to specify the size in bytes:

Unit size	Code
Byte (1B)	`c`
Word (2B)	`w`
Block (512B)	`b`
Kilo Byte (1024B)	`k`
Mega Byte (1024 KB)	`M`
Giga Byte (1024 MB)	`G`

We can generate a file of any size using this. Instead of MB we can use any other unit notations such as the ones mentioned in the previous table.

`/dev/zero` is a character special device, which infinitely returns the zero byte (`\0`).

If the input parameter (`if`) is not specified, it will read the input from `stdin` by default. Similarly, if the output parameter (`of`) is not specified, it will use `stdout` as the default output sink.

The `dd` command can also be used to measure the speed of memory operations by transferring a large quantity of data and checking the command output (for example, `1048576 bytes (1.0 MB) copied, 0.00767266 s, 137 MB/s` as seen the previous example).

Intersection and set difference (A-B) on text files

Intersection and set difference operations are commonly used in mathematical classes on set theory. However, similar operations on text are also very helpful in some scenarios.

Getting ready

The `comm` command is a utility to perform comparison between the two files. It has many nice options to arrange the output in such a way that we can perform intersection, difference, and set difference operations.

- **Intersection**: The intersection operation will print the lines that the specified files have in common with one another.
- **Difference**: The difference operation will print the lines that the specified files contain and that are not the same in all of those files.
- **Set difference**: The set difference operation will print the lines in file "A" that do not match those in all of the set of files specified ("B" plus "C" for example).

How to do it...

Note that `comm` takes sorted files as input. Take a look at the following example:

```
$ cat A.txt
apple
orange
gold
silver
steel
iron

$ cat B.txt
orange
gold
cookies
carrot

$ sort A.txt -o A.txt ; sort B.txt -o B.txt
$ comm A.txt B.txt
apple
        carrot
        cookies
                gold
iron
                orange
silver
steel
```

The first column of the output contains lines that are in `A.txt` excluding common lines in two files. The second column contains lines that are in `B.txt` excluding common lines. The third column contains the common lines from `A.txt` and `B.txt`. Each of the columns are delimited by using the tab (`\t`) character.

Some options are available to format the output as per our requirement. For example:

- `-1` removes first column from output
- `-2` removes the second column
- `-3` removes the third column

In order to print the intersection of two files, we need to remove the first and second columns and print the third column only as follows:

```
$ comm A.txt B.txt -1 -2
gold
orange
```

Print lines that are uncommon in two files as follows:

```
$ comm A.txt B.txt  -3
apple
        carrot
        cookies
iron
silver
steel
```

Using the `-3` argument in the `comm` command removes the third column from the output. But, it writes column-1 and column-2 to the output. The column-1 contains the lines in `A.txt` excluding the lines in `B.txt`. Similarly, column-2 has the lines from `B.txt` excluding the lines in `A.txt`. As the output is a two-column output, it is not that useful. Columns have their fields blank for each of the unique lines. Hence both columns will not have the content on the same line. Either one of the two columns will have the content. In order to make it in a usable output text format, we need to remove the blank fields and make two columns into a single column output as follows:

```
apple
carrot
cookies
iron
silver
steel
```

In order to produce such an output, we need to remove the `\t` character at the beginning of the lines. We can remove the `\t` character from the start of each line and unify the columns into one as follows:

```
$ comm A.txt B.txt  -3 | sed 's/^\t//'
apple
carrot
cookies
iron
silver
steel
```

The `sed` command is piped to the `comm` output. The `sed` removes the `\t` character at the beginning of the lines. The `s` in the `sed` script stands for substitute. `/^\t/` matches the `\t` at the beginning of the lines (`^` is the start of the line marker). `//` (no character) is the replacement string for every `\t` at the beginning of the line. Hence every `\t` at the start of the line gets removed.

A set difference operation on two files can be performed as explained in the following paragraphs.

The set difference operation enables you to compare two files and print all the lines that are in the file `A.txt` or `B.txt` excluding the common lines in `A.txt` and `B.txt`. When `A.txt` and `B.txt` are given as arguments to the `comm` command, the output will contain column-1 with the set difference for `A.txt` with respect to `B.txt` and column-2 will contain the set difference for `B.txt` with respect to `A.txt`.

By removing the unnecessary columns, we can produce the set difference for `A.txt` and `B.txt` as follows:

- **Set difference for A.txt**:

  ```
  $ comm A.txt B.txt -2 -3
  ```

  `-2 -3` removes the second and third columns.

- **Set difference for B.txt**:

  ```
  $ comm A.txt B.txt -1 -3
  ```

  `-2 -3` removes the second and third columns.

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. This recipe deals with finding out duplicate files and performing operations based on the result.

Getting ready

Duplicate files are files with different names but same data. We can identify the duplicate files by comparing the file content. Checksums are calculated by looking at the file contents. Since files with exactly the same content will produce duplicate checksum values, we can use this to remove duplicate lines.

How to do it...

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
```

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 | 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.
```

Run it as:

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

How it works...

The commands above will find the copies of same file in a directory and remove all except one copy of the file. Let's go through the code and see how it works. `ls -lS` will list the details of the files sorted by file size in the current directory. `awk` will read the output of `ls -lS` and perform comparisons on columns and rows of the input text to find out the duplicate files.

The logic behind the previous code is as follows:

- We list the files sorted by file size so that the similarly sized files will be grouped together. The files having the same file size are identified as a first step to finding files that are the same. Next, we calculate the checksum of the files. If the checksums match, then the files are duplicates and one set of the duplicates are removed.
- The `BEGIN{}` block of `awk` is executed first before lines are read from the file. Reading of lines takes place in the `{}` block and after the end of reading and processing all lines, the `END{}` block statements are executed. The output of `ls -lS` is:

```
total 16
4 -rw-r--r-- 1 slynux slynux 5 2010-06-29 11:50 other
4 -rw-r--r-- 1 slynux slynux 6 2010-06-29 11:50 test
4 -rw-r--r-- 1 slynux slynux 6 2010-06-29 11:50 test_copy1
4 -rw-r--r-- 1 slynux slynux 6 2010-06-29 11:50 test_copy2
```

- The output of the first line tells us the total number of files, which in this case is not useful. We use `getline` to read the first line and then dump it. We need to compare each of the lines and the next line for sizes. For that we read the first line explicitly using `getline` and store name and size (which are the eighth and fifth columns). Hence a line is read ahead using `getline`. Now, when `awk` enters the `{}` block (in which the rest of the lines are read) that block is executed for every read offline. It compares size obtained from the current line and the previously stored size kept in the `size` variable. If they are equal, it means two files are duplicates by size. Hence they are to be further checked by `md5sum`.

We have played some tricky ways to reach the solution.

The external command output can be read inside `awk` as:

```
"cmd"| getline
```

Then we receive the output in line `$0` and each column output can be received in `$1,$2,..$n`, and so on. Here we read the md5sum of files in the `csum1` and `csum2` variables. Variables `name1` and `name2` are used to store consecutive file names. If the checksums of two files are the same, they are confirmed to be duplicates and are printed.

We need to find a file each from the group of duplicates so that we can remove all other duplicates except one. We calculate the `md5sum` of the duplicates and print one file from each group of duplicates by finding unique lines by comparing `md5sum` only from each line using `-w 32` (the first 32 characters in the `md5sum` output; usually, `md5sum` output consists of a 32 character hash followed by the filename). Therefore, one sample from each group of duplicates is written in `duplicate_sample`.

Now, we need to remove all the files listed in `duplicate_files`, excluding the files listed in `duplicate_sample`. The `comm` command prints files in `duplicate_files` but not in `duplicate_sample`.

For that, we use a set difference operation (refer to the intersection, difference, and set difference recipes).

`comm` always accepts files that are sorted. Therefore, `sort -u` is used as a filter before redirecting to `duplicate_files` and `duplicate_sample`.

Here the `tee` command is used to perform a trick so that it can pass filenames to the `rm` command as well as `print`. `tee` writes lines that appear as `stdin` to a file and sends them to `stdout`. We can also print text to the terminal by redirecting to `stderr`. `/dev/stderr` is the device corresponding to `stderr` (standard error). By redirecting to a `stderr` device file, text that appears through `stdin` will be printed in the terminal as standard error.

See also

- *Basic awk primer* of *Chapter 4* explains the awk command.
- *Checksum and verification* of *Chapter 2* explains the md5sum command.

Making directories for a long path

There are circumstances when we are required to make a tree of empty directories. If some intermediate directories exist in the given path, it will also have to incorporate checks to see whether the directory exists or not. It will make the code larger and inefficient. Let's see the use case and the recipe to solve the issue.

Getting ready

`mkdir` is the command for creating directories. For example:

```
$ mkdir dirpath
```

If the directory already exists, it will return a "File exists" error message, as follows:

```
mkdir: cannot create directory `dir_name': File exists
```

You are given a directory path (`/home/slynux/test/hello/child`). The directory `/home/slynux` already exist. We need to create rest of the directories (`/home/slynux/test`, `/home/slynux/test/hello`, and `/home/slynux/test/hello`) in the path.

The following code is used to figure out whether each directory in a path exists:

```
if  [ -e /home/slynux ]; then
  # Create next level directory
fi
```

-e is a parameter used in the condition construct `[ ]`, to determine whether a file exists. In UNIX-like systems, directory is also a type of file. `[ -e FILE_PATH ]` returns true if the file exists.

How to do it...

The following sequence of code needs to be executed to create directories in a tree in several levels:

```
$ mkdir /home 2> /dev/null
$ mkdir /home/slynux 2> /dev/null
$ mkdir /home/slynux/test 2> /dev/null
$ mkdir /home/slynux/test/hello 2> /dev/null
$ mkdir /home/slynux/test/hello/child 2> /dev/null
```

If an error, such as "Directory exists", is encountered, it is ignored and the error message is dumped to the `/dev/null` device using the `2>` redirection. But this is lengthy and non-standard. The standard one-liner to perform this action is:

```
$ mkdir -p /home/slynux/test/hello/child
```

This single command takes the place of the five different commands listed above. It ignores if any level of directory exists and creates the missing directories.

File permissions, ownership, and sticky bit

File permissions and ownership are one of the distinguishing features of UNIX/Linux file systems such as extended (ext FS). In many circumstances while working on UNIX/Linux platforms, we come across issues related to permissions and ownership. This recipe is a walk through different use cases of permissions and ownership.

Getting ready

In Linux systems, each file is associated with many types of permissions. Out of these permissions, three set 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) that are permitted some access to the file. Others are any entity 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
-rw-r--r-- 1 slynux slynux  16237 2010-02-06 21:42 c9.php
drwxr-xr-x 2 slynux slynux  4096  2010-05-27 14:31a.py
-rw-r--r-- 1 slynux slynux  539   2010-02-10 09:11 cl.pl
```

The first column of output specifies the following. The first letter corresponds 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

The rest of the portions can be divided into three groups of three letters each (`------`). The first `---` three characters correspond the permissions of the user (owner), the second set of three characters correspond to the permissions of the group, and the third set of three characters correspond to the permissions of others. Each character in the nine character sequence (nine permissions) specifies whether a permission is set or unset. If the permission is set, a character appears in the corresponding position, else a '-' character appears in that position, which means that the corresponding permission is unset (unavailable).

Let's take a look at what each of these three character set means 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. 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 as follows:

```
-rwS------
```

The read, write, and execute permissions are also applied to the directories. However, the interpretation of read, write, and execute permissions are slightly different in the context of directories as follows:

- Read permission (`r`) for the directories enables to read the list of files and sub-directories in the directory
- Write permission (`w`) for a directory enables to create or remove files and directories from a directory
- Execute permission (`x`) specifies whether the access to the files and directories in a directory is possible or not

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 user. Instead of setuid, the group has a setgid (`S`) bit. It enables 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 as follows:

```
----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` (like setuid and setgid).

Directories have a special permission called sticky bit. When a sticky bit is set for a directory, the user who created the directory can only delete the files in the directory even if 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`. `t` 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 is set, character `T` appears in the position of `x`.

For example:

`------`rwt , `------`rwT

A typical example of a directory with sticky bit turned on by default is `/tmp`. The sticky bit is a type of write-protection.

In each of the `ls -l` output line, 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.

How to do it...

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

In order to add additional permissions on the current file, use + to add permission to 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 any permission, use -. For example:

```
$ chmod a-x filename
```

Permissions can also be set using octal numbers. Permissions are denoted by three-digit octal numbers in which each of the digit corresponds to user, group, and other in the 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 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
```

There's more...

Let's go through some additional tasks that can be performed for files and directories.

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.

Setting the sticky bit

The sticky bit is an interesting type of permission applied to directories. By setting the sticky bit, it restricts only the user owning it to delete the files even though group and others have sufficient permissions.

In order to set the sticky bit, `+t` is applied on a directory with `chmod` as follows:

```
$ chmod a+t directory_name
```

Applying permissions recursively to files

Sometimes it may be required to recursively change the permissions of all the files and directories inside the current directory. This can be done as follows:

```
$ chmod 777 . -R
```

The `-R` option specifies to apply change permission recursively.

We have used "." to specify the path as the current working directory. It is equivalent to:

```
$ chmod 777 "$(pwd)" –R.
Sarath Lakshman 7 January 2011 8:41 PM
```

Applying ownership recursively

We can apply the ownership recursively by using the `-R` flag with the `chown` command as follows:

```
$ chown user.group . -R
```

Running an executable as a different user (setuid)

Some executables need to be executed as a different user (other than the current user that initiates the execution of the file), effectively, whenever they are executed, by using the file path, such as `./executable_name`. A special permission attribute for files called `setuid` permission enables to effectively execute as the file owner when any other user runs the program.

First change the ownership to the user to which it needs to be executed every time and login as the owner user. Then, run the following command:

```
$ chmod +s executable_file

# chown root.root executable_file
# chmod +s executable_file
$ ./executable_file
```

Now it executes effectively as the root user every time.

`setuid` is restricted such that `setuid` won't work for scripts, but only for Linux ELF binaries. This is a fix for ensuring security.

Making files immutable

Files on extended type file systems, which are common in Linux (for example, ext2, ext3, ext4, and so on) can be made immutable. Certain type of file attributes help to set the immutable attribute to the file. When a file is made immutable, any user or super user cannot remove the file until the immutable attribute is removed from the file. We can easily find out the file system type of any mounted partition by looking at the `/etc/mtab` file. The first column of the file specifies the partition device path (for example, `/dev/sda5`) and the third column specifies the file system type (for example, ext3). Let's see how to make files immutable.

Getting ready

`chattr` can be used for to make files immutable. However, it is not the only extended attribute that can be changed by chattr.

Making a file immutable is one of the methods for securing files from modification. The best known example is in the case of the `/etc/shadow` file. The shadow file consists of encrypted passwords of every user in the current system. By injecting encrypted passwords, we can login into the system. Users can, usually, change their password by using the `passwd` command. When you execute the `passwd` command, it actually modifies the `/etc/shadow` file. We can make the shadow file immutable so that no user is able to change the password. Let's see how to do it.

How to do it...

A file can be made immutable as follows:

```
chattr +i file
```

Or:

```
$ sudo chattr +i file
```

The file is therefore made immutable. Now try the following command:

```
rm file
rm: cannot remove `file': Operation not permitted
```

In order to make it writable, remove the immutable attribute as follows:

```
chattr -i file
```

Generating blank files in bulk

Sometimes we many need to generate test cases. We may use programs that operate on 1000s of files. But how are test files generated?

Getting ready

`touch` is a command that can create blank files or modify the timestamp of files if they already exist. Let's take a look at how to use them.

How to do it...

A blank file with the name `filename` will be created using the following command:

```
$ touch filename
```

Generate bulk files with a different name pattern as follows:

```
for name in {1..100}.txt
do
touch $name
done
```

In the above code `{1..100}` will be expanded as a string "1, 2, 3, 4, 5, 6, 7...100". Instead of `{1..100}.txt`, we can use various shorthand patterns such as `test{1..200}.c`, `test{a..z}.txt`, and so on.

If a file already exists, then the `touch` command changes all timestamps associated with the file to the current time. However, if we want to specify that only certain stamps are to be modified, we use the following options:

- `touch -a` modifies only the access time
- `touch -m` modifies only the modification time

Instead of using the current time for the timestamp, we can specify the time and date with which to stamp the file as follows:

```
$ touch -d "Fri Jun 25 20:50:14 IST 1999" filename
```

The date string that is used with `-d` need not always be in the same format. It will accept any standard date formats. We can omit time from the string and provide handy date formats like "Jan 20 2010".

Finding a symbolic link and its target

Symbolic links are common with UNIX-like systems. We may come across various manipulations based on symbolic links. This recipe may not be having any practical purpose, but it gives practice of handling symbolic links that may be helpful in writing shell scripts for other purposes.

Getting ready

Symbolic links are just pointers to other files. They are similar in function to aliases in Mac OS X or shortcuts in Windows. When symbolic links are removed, they will not cause any harm to the original file.

How to do it...

We can create a symbolic link as follows:

```
$ ln -s target symbolic_link_name
```

For example:

```
$ ln –l -s /var/www/ ~/web
```

This creates a symbolic link (called "web") in the logged in user's home directory. The link points to `/var/www/`. This is seen in the output of the following command:

```
$ ls web
lrwxrwxrwx 1 slynux slynux 8 2010-06-25 21:34 web -> /var/www
```

`web -> /var/www` specifies that `web` points to `/var/www`.

For every symbolic link, the permission notation block (`lrwxrwxrwx`) starts with letter "l", which represents a symlink.

So, in order to print symbolic links in the current directory, use the following command:

```
$ ls -l | grep "^l" | awk '{ print $8 }'
```

`grep` will filter the lines from the `ls -l` output such that it displays only lines starting with l. ^ is the start marker for the string. `awk` is used to print the eighth column. Hence it prints the eighth column, which is the filename.

Another way to print symbolic links is to use `find` as follows:

```
$ find . -type l -print
```

In the above command, in the `find` argument `type` we have specified "l", which will instruct the `find` command to search only for symbolic link files. The `–print` option is used to print the list of symbolic links to the standard output (`stdout`). The path from which the file search should begin is given as '.', which means it is the current directory.

In order to print the target of a symbolic link use the following command:

```
$ ls -l web | awk '{ print $10 }'
/var/www
```

The `ls –l` command lists many details with each of the line corresponding to the details of a file. `ls –l web` lists the details for the file called `web`, which is a symbolic link. The tenth column in the output of `ls –l` contains the link to which the file points to (if the file is a symbolic link). Hence in order to find the target associated with a symbolic link, we can use `awk` to print the tenth column from the file details listing (the output from `ls –l`).

Or, alternately, we can use the standard way of reading the target path for a given symbolic link using the command `readlink`. It is the most preferred method and can be used as follows:

```
$ readlink web
/var/www
```

Enumerating file type statistics

There are many file types. It will be an interesting exercise to write a script that can enumerate through all the files inside a directory, its descendants, and print a report that provides details on types of files (files with different file types) and the count of each file type present. This recipe is an exercise on how to write scripts that can enumerate through a bulk of files and collecting details.

Getting ready

The file command can be used to find out the type of the file by looking at the contents of the file. In UNIX/Linux systems, file types are not determined based on the extension of the file (like the Microsoft Windows platform does). This recipe aims at collecting file type statistics of a number of files. For storing the count of files of the same type, we can use an associative array and the `file` command can be used to fetch the file type details from each of the files.

How to do it...

In order to print the file type of a file use the following command:

```
$ file filename

$ file /etc/passwd
/etc/passwd: ASCII text
```

Print the file type only by excluding the filename as follows:

```
$ file -b filename
ASCII text
```

The script for files statistics is as follows:

```
#!/bin/bash
# Filename: filestat.sh
if [ $# -ne 1 ];
then
  echo $0 basepath;
  echo
fi
path=$1
declare -A statarray;
while read line;
do
```

```
    ftype=`file -b "$line"`
    let statarray["$ftype"]++;
done< <(find $path -type f -print)
echo ============ File types and counts =============
for ftype in "${!statarray[@]}";
do
  echo $ftype :  ${statarray["$ftype"]}
done
```

The usage is as follows:

```
$ ./filestat.sh /home/slynux/temp
```

A sample output is shown below:

```
$ ./filetype.sh /home/slynux/programs
============ File types and counts =============
Vim swap file : 1
ELF 32-bit LSB executable : 6
ASCII text : 2
ASCII C program text : 10
```

How it works...

Here an associative array named `statarray` is declared so that it can take file type as file indices and store the count of each file type in the array. `let` is used to increment the count each time when a file type is encountered. The `find` command is used to get the list of file paths recursively. A `while` loop is used to iterate line by line through the `find` command's output. The input line `ftype=`file -b "$line"`` in the previous script is used to find out the file type using the `file` command. The `-b` option specifies file command to print only file type (without filename in the output). The file type output consists of more details, such as image encoding used and resolution (in the case of an image file). But we are not interested in more details, we need only the basic information. Details are comma separated as in the following example:

```
$ file a.out -b
ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.15, not stripped
```

We need to extract only the "ELF 32-bit LSB executable" from the above details. Hence we use `cut -d, -f1`, which specifies to use "," as the delimiter and print only the first field.

`done< <(find $path -type f -print);` is an important bit of code. The logic is as follows:

`while` read line;

do something

`done<` filename

Instead of the filename we used the output of `find`.

`<(find $path -type f -print)` is equivalent to a filename. But it substitutes filename with subprocess output. Note that there is an additional `<`.

`${!statarray[@]}` is used to return the list of array indexes.

Loopback files and mounting

Loopback filesystems are very interesting components of Linux like systems. We usually create filesystems on devices (for example, disk drive partitions). These storage devices are available as device files like `/dev/device_name`. In order to use the storage device filesystem, we need to mount it at some directory called a mount point. Loopback filesystems are those that we create in files rather than a physical device. We can mount those files as devices at a mount point. Let's see how to do it.

Getting ready

Loopback filesystems reside on a file. We mount these files by attaching it to a device file. An example of a loopback filesystem is the initial ramdisk file, which you would see at `boot/initrd.img`. It stores an initial filesystem for the kernel in a file.

Let's see how to create an ext4 filesystem on a file of size 1GB.

How to do it...

The following command will create a file that is 1 GB in size.

```
$ dd if=/dev/zero of=loopbackfile.img bs=1G count=1
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 37.3155 s, 28.8 MB/s
```

You can see that the size of the created file exceeds 1GB. This is because the hard disk is a block device and hence storage is allocated by integral multiples of blocks size.

Now format the 1GB file using the `mkfs` command as follows:

```
# mkfs.ext4 loopbackfile.img
```

This command formats it to ext4. Check the file type using the following command:

```
$ sudo file loopbackfile.img
loopbackfile.img: Linux rev 1.0 ext4 filesystem data, UUID=c9d56c42-f8e6-
4cbd-aeab-369d5056660a (extents) (large files) (huge files)
```

Now you can mount the loopback file as follows:

```
$ sudo mkdir /mnt/loopback
# mount -o loop loopback.img /mnt/loopback
```

The `-o loop` additional option is used to mount any loopback file systems.

This is the shortcut method. We do not attach it to any devices. But internally it attaches to a device called `/dev/loop1` or `loop2`.

We can do it manually as follows:

```
# losetup /dev/loop1 loopback.img
# mount /dev/loop1 /mnt/loopback
```

The first method cannot be used in all circumstances. Suppose we want to create a hard disk file, and then want to partition it and mount a sub partition, we cannot use `mount -o loop`. We have to use the second method. Partition a zeros dumped file as follows:

```
# losetup /dev/loop1 loopback.img
# fdisk /dev/loop1
```

Create partitions in `loopback.img` in order to mount the first partition as follows:

```
# losetup -o 32256 /dev/loop2 loopback.img
```

Now `/dev/loop2` represents first partition.

`-o` is the offset flag. `32256` bytes are for a DOS partition scheme. The first partition starts after an offset of 32256 bytes from the start of the hard disk.

We can set up the second partition by specifying the required offset. After mounting we can perform all regular operations as we can on physical devices.

In order to `umount`, use the following syntax:

```
# umount mount_point
```

For example:

```
# umount /mnt/sda1
```

Or, alternately, we can use device file path as an argument to the `umount` command as:

```
# umount /dev/sda1
```

Note that `umount` command should be executed as a root user since it is a privileged command.

There's more...

Let's explore more about additional mount options.

Mounting ISO files as loopback

An ISO file is an archive of any optical media. We can mount ISO files in the same way that we mount physical discs by using loopback mounting.

A mount point is just a directory, which is used as access path to contents of a device through a filesystem. We can even use a non-empty directory as the mount path. Then the mount path will contain data from the devices rather than original contents until the device is unmounted. For example:

```
# mkdir /mnt/iso
# mount -o loop linux.iso /mnt/iso
```

Now perform operations using files from `/mnt/iso`. ISO is a read-only filesystem.

Flush changes immediately with sync

While making changes on a mounted device, they are not immediately written to the physical devices. They are only written when the buffer is full. But we can force writing of changes immediately by using the `sync` command as follows:

```
# sync
```

You should execute the `sync` command as root.

Creating ISO files, Hybrid ISO

An ISO image is an archive format that stores the exact storage images of optical disks like CD ROMs, DVD ROMs, and so on. It is a common use case that we burn ISO images to optical disks. But what if you want to create an image of an optical disk? For that we need to create an ISO image from an optical disk. Many people rely on third-party utilities to create an ISO image from an optical disk. However, using the command line, it's just a single line job.

Also, many people don't distinguish between bootable and non-bootable optical disks. Bootable disks are capable of booting from themselves and also running an operating system or another product. Non-bootable ISOs cannot do that. The practice that people usually follow is to copy files from a bootable CD-ROM and paste it to another location for keeping the copy. After that, they use the copied directory to burn a CD ROM. But then, it will lose its bootable nature. To preserve the bootable nature, it should be copied as a disk image or an ISO file.

Nowadays, most people use devices such as flash drives or hard disks as a replacement for optical disks. When we write a bootable ISO to a flash drive it will no longer be bootable unless we use a special hybrid ISO image designed specifically for the purpose.

This recipe will give you an insight on ISO images and manipulations.

Getting ready

As we described many times in this book, UNIX handles everything as files. Every device is a file. Hence what if we want to copy an exact image of a device? We need to read all data from it and write to another file, right?

As we know, the `cat` command can be used to read any data and redirection can be used to write to a file.

How to do it...

In order to create an ISO image from `/dev/cdrom` use the following command:

```
# cat /dev/cdrom > image.iso
```

This will work, it will read all the bytes from the device and write an ISO image.

Using the `cat` command for creating an ISO image is a tricky way to do it. But the most preferred way to create an ISO image is to use the `dd` utility.

```
# dd if=/dev/cdrom of=image.iso
```

`mkisofs` is a command used to create ISO system. The output file of `mkisofs` can be written to CD ROM or DVD ROM using utilities like `cdrecord`. We can use `mkisofs` to create an ISO file using a directory containing all the required files that should appear as contents of an ISO file as follows:

```
$ mkisofs -V "Label" -o image.iso source_dir/
```

The `-o` option in the `mkisofs` command specifies the ISO file path. The `source_dir` is the path of the directory that should be used as source content for the ISO and the `-V` option specifies the label that should be used for the ISO file.

There's more...

Let's learn more commands and techniques related to ISO files.

Hybrid ISO that boots off flash drive or hard disk

Usually, bootable ISO files cannot be transferred or written to a USB storage device and boot the OS from the USB key. But special type of ISO files called hybrid ISOs can be flashed and they are capable of booting from such devices.

We can convert standard ISO files into hybrid ISOs with the `isohybrid` command. The `isohybrid` command is a new utility and most Linux distros don't include this by default. You can download the syslinux package from: `http://syslinux.zytor.com`.

Have a look at the following command:

```
# isohybrid image.iso
```

Using this command, we will have a hybrid ISO with the file name `image.iso` and it can be written to USB storage devices.

Write the ISO to a USB storage by using the following command:

```
# dd if=image.iso of=/dev/sdb1
```

Use the appropriate device instead of `sdb1`.

Or, you can use `cat` as follows:

```
# cat image.iso > /dev/sdb1
```

Burning an ISO from command line

The `cdrecord` command is used to burn an ISO file into a CD ROM or DVD ROM. It can be used to burn the image to the CD ROM as follows:

```
# cdrecord -v dev=/dev/cdrom image.iso
```

Some extra options are as follows:

- We can specify the burning speed with the `-speed` option as follows:

  ```
  -speed SPEED
  ```

  For example:

  ```
  # cdrecord -v dev=/dev/cdrom image.iso -speed 8
  ```

  The speed is 8x, which is specified as 8.

- A CD ROM can be burned in multisessions such that we can burn data multiple times on a disk. Multisession burning can be performed using the `–multi` option as follows:

  ```
  # cdrecord –v dev=/dev/cdrom image.iso -multi
  ```

Playing with CD Rom tray

Try the following commands and have fun:

- **`$ eject`**

  This command is used to eject the tray.

- **`$ eject -t`**

  This command is used to close the tray.

Try to write a loop that opens the tray and closes the tray for "N" number of times.

Finding difference between files, patching

When multiple versions of a file are available, it is very useful when we can find the differences between files being highlighted rather than comparing two files manually by looking through them. If the files are of 1000s of lines, they are practically very difficult and time consuming to compare. This recipe illustrates how to generate differences between files highlighted with line numbers. When working on large files by multiple developers, when one of them has made changes and these changes need to be shown to the other, sending the entire source code to other developers is costly in consumption of space and time to manually check the changes. Sending a different file is helpful. It consists of only lines that are changed, added, or removed and line numbers are attached with it. This difference file is called a patch file. We can add the changes specified in the patch file to the original source code by using the patch command. We can also revert the changes by patching again. Let's see how to do this.

How to do it...

The `diff` command utility is used to generate difference files.

In order to generate difference information, create the following files:

- File 1: `version1.txt`

  ```
  this is the original text
  line2
  line3
  line4
  happy hacking !
  ```

- File 2: `version2.txt`

```
this is the original text
line2
line4
happy hacking !
GNU is not UNIX
```

Non-unified `diff` output (without the `-u` flag) will be as follows:

```
$ diff version1.txt version2.txt
3d2
<line3
6c5
> GNU is not UNIX
```

The unified `diff` output will be as follows::

```
$ diff -u version1.txt version2.txt
--- version1.txt    2010-06-27 10:26:54.384884455 +0530
+++ version2.txt    2010-06-27 10:27:28.782140889 +0530
@@ -1,5 +1,5 @@
this is the original text
line2
-line3
line4
happy hacking !
-
+GNU is not UNIX
```

The `-u` option is used to produce unified output. Everyone prefers unified output, as the unified output is more readable and because it is easier to interpret the difference that is being made between two files.

In unified `diff`, the lines starting with `+` are the newly added lines and the lines starting with `–` are the removed lines.

A patch file can be generated by redirecting the `diff` output to a file, as follows:

```
$ diff -u version1.txt version2.txt > version.patch
```

Now using the patch command we can apply changes to any of the two files. When applied to `version1.txt`, we get `version2.txt` file. When applied to `version2.txt`, we receive `version1.txt`.

Apply the patch by using the following command:

```
$ patch -p1 version1.txt < version.patch
patching file version1.txt
```

We now have `version1.txt` with the same contents as that of `version2.txt`.

In order to revert the changes back, use the following command:

```
$ patch -p1 version1.txt < version.patch
patching file version1.txt
Reversed (or previously applied) patch detected!  Assume -R? [n] y
#Changes are reverted.
```

Revert the changes without prompting the user with `y/n` by using the `-R` option along with the patch command.

There's more...

Let's go through additional features available with `diff`.

Generating diff against directories

The `diff` command can also act recursively against directories. It will generate a difference output for all the descendant files in the directories.

Use the following command:

```
$ diff -Naur directory1 directory2
```

The interpretation of each of the above options is as follows:

- `-N` is for treating absent files as empty
- `-a` is to consider all files as text files
- `-u` is to produce unified output
- `-r` is to recursively traverse through the files in the directories

head and tail – printing the last or first 10 lines

When looking into a large file, which consists of thousands of lines, we will not use a command like `cat` to print the entire file contents. Instead we look for a sample (for example, the first 10 lines of the file or the last 10 lines of the file). We may also need to print the first n lines or last n lines. Also we may need to print all the lines except the last "n" lines or all lines except first "n" lines.

Another use case is to print lines from n-th to m-th lines.

The commands `head` and `tail` can help us do this.

How to do it...

The `head` command always reads the header portion of the input file.

Print first 10 lines as follows:

```
$ head file
```

Read the data from stdin as follows:

```
$ cat text | head
```

Specify the number of first lines to be printed as follows:

```
$ head -n 4 file
```

This command prints four lines.

Print all lines excluding the last `N` lines as follows:

```
$ head -n -N file
```

Note that it is negative N.

For example, to print all the lines except the last 5 lines use the following code:

```
$ seq 11 | head -n -5
1
2
3
4
5
6
```

The following command will, however, print from 1 to 5:

```
$ seq 100 | head -n 5
```

Printing by excluding the last lines is a very important usage of `head`. But people always look at some other complex methods to do the same.

Print the last 10 lines of a file as follows:

```
$ tail file
```

In order to read from `stdin`, you can use the following code:

```
$ cat text | tail
```

Print the last 5 lines as follows:

```
$ tail -n 5 file
```

In order to print all lines excluding first N lines, use the following code:

```
$ tail -n +(N+1)
```

For example, to print all lines except the first 5 lines, N + 1 = 6, therefore the command will be as follows:

```
$ seq 100 | tail -n +6
```

This will print from 6 to 100.

One of the important usages of `tail` is to read a constantly growing file. Since new lines are constantly appended to the end of the file, `tail` can be used to display all new lines as they are written to the file. When we run `tail` simply, it will read the last 10 lines and exit. However, by that time, new lines would have been appended to the file by some process. In order to constantly monitor the growth of file, `tail` has a special option `-f` or `--follow`, which enables `tail` to follow the appended lines and keep being updated with the data growth:

```
$ tail -f growing_file
```

An example of such growing files are logfiles. The command to monitor the growth of the files would be:

```
# tail -f /var/log/messages
```

or

```
$ dmesg | tail -f
```

We frequently run `dmesg` to look at kernel ring buffer messages either to debug the USB devices or to look at the `sdX` (`X` is the minor number for the `sd` device). The `tail -f` can also add a sleep interval `-s`, so that we can set the interval during which the file updates are monitored.

`tail` has the interesting property that allows it to terminate after a given process ID dies.

Suppose we are reading a growing file, and a process `Foo` is appending data to the file, `tail -f` should be executed until process `Foo` dies.

```
$ PID=$(pidof Foo)
$ tail -f file --pid $PID
```

When the process `Foo` terminates, `tail` also terminates.

Let's work on an example.

Create a new file `file.txt` and open the file in gedit (You can use any text editor).

Add new lines to the file and make frequent file saves in gedit.

Now run:

```
$ PID=$(pidof gedit)
$ tail -f file.txt --pid $PID
```

When you make frequent changes to the file, it will be written to the terminal by the tail command. When you close the `gedit`, the `tail` command will get terminated.

Listing only directories – alternative methods

Though listing only directories seems to be a simple task, many would not be able to do it. I have seen this often, even when asked to people who are good at shell scripting. This recipe is worth knowing since it introduces multiple ways of listing only directories with various tricky techniques.

Getting ready

There are multiple ways of listing directories only. When you ask people about these techniques, the first answer that they would probably give is `dir`. But, it is wrong. The `dir` command is just another command like `ls` with fewer options than `ls`. Let's see how to list directories.

How to do it...

There are four ways in which directories in the current path can be displayed. They are:

- `$ ls -d */`

  Only the above combination with `-d` will print directories.

- `$ ls -F | grep "/$"`

  When the `-F` parameter is used, all entries are appended with some type of file character such as `@`, `*`, `|`, and so on. For directories, entries are appended with the `/` character. We use `grep` to filter only entries ending with the `/$` end of line indicator.

- `$ ls -l | grep "^d"`

  The first character of `ls -d` output lines of each file entries is the type of file character. For directory, the type of file character is `"d"`. Hence we use `grep` to filter lines starting with `"d"`. `^` is the start of line indicator.

- `$ find . -type d -maxdepth 1 -print`

  The `find` command can take the parameter `type` as directory and `maxdepth` is set to `1` since it should not search the directories of descendants.

Fast command-line navigation using pushd and popd

When dealing with multiple locations on a terminal or shell prompt, our common practice is to copy and paste the paths. Copy-paste is only effective when mouse is used. When there is only command-line access without a GUI, it is hard to deal with navigation through multiple paths. For example, if we are dealing with locations `/var/www`, `/home/slynux`, and `/usr/src`, when we need to navigate these locations one by one, it is really difficult to type the path every time when we need to switch between the paths. Hence the command-line interface (CLI) based navigation techniques such as pushd and popd are used. Let's see how to practice them.

Getting ready

`pushd` and `popd` are used to switch between multiple directories without the copy-paste of directory paths. `pushd` and `popd` operate on a stack. We know that stack is a **Last In First Out** (**LIFO**) data structure. It will store the directory paths in a stack and switch between them using push and pop operations.

How to do it...

We omit the use of the `cd` command while using `pushd` and `popd`.

In order to push and change directory to a path use:

```
~ $ pushd /var/www
```

Now the stack contains `/var/www ~` and the current directory is changed to `/var/www`.

Now again push the next directory path as follows:

```
/var/www $ pushd /usr/src
```

Now the stack contains `/usr/src /var/www ~` and the current directory is `/usr/src`.

You can similarly push as many directory paths as needed.

View the stack contents by using the following command:

```
$ dirs
/usr/src /var/www ~ /usr/share /etc
0        1        2 3          4
```

When you want to switch to any path in the list, number each path from `0` to `n`, then use the path number for which we need to switch, for example:

```
$ pushd +3
```

It will rotate the stack and switch to the directory `/usr/share`.

`pushd` will always add paths to the stack, to remove paths from the stack use `popd`.

Remove a last pushed path and change directory to the next directory by using:

```
$ popd
```

Suppose the stack is `/usr/src /var/www ~ /usr/share /etc` such that the current directory is `/usr/src`, `popd` will change the stack to `/var/www ~ /usr/share /etc` and change the directory to `/var/www`.

In order to remove a specific path from the list, use `popd +no`.

The `no` is counted as `0` to `n` from left to right.

There's more...

Let's go through essential directory navigation practices.

Most frequently used directory switching

`pushd` and `popd` can be used when there are more than three directory paths are used. But when you use only two locations, there is an alternative and easier way. That is `cd -`.

If the current path is `/var/www`, perform the following:

```
/var/www $  cd /usr/src
/usr/src $ # do something
```

Now to switch back to `/var/www`, you don't have to type it out again, but just execute:

```
/usr/src $ cd -
```

Now you can switch to `/usr/src` as follows:

```
/var/www $ cd -
```

Counting number of lines, words, and characters in a file

Counting the number of lines, words, and characters from a text or file are very useful for text manipulations. In several cases, count of words or characters are used in indirect ways to perform some hacks to produce required output patterns and results. This book includes some of such tricky examples in other chapters. **Counting LOC** (**Lines of Code**) is an important application for developers. We may need to count special types of files excluding unnecessary files. A combination of `wc` with other commands help to perform that.

Getting ready

`wc` is the utility used for counting. It stands for **Word Count (wc)**. Let's see how to use `wc` to count lines, words, and characters.

How to do it...

Count number of lines as follows:

```
$ wc -l file
```

In order to use `stdin` as input, use the following command:

```
$ cat file | wc -l
```

Count the number of words as follows:

```
$ wc -w file
$ cat file | wc -w
```

In order to count number of characters, use:

```
$ wc -c file
$ cat file | wc -c
```

For example, we can count the characters in a text as follows:

```
echo -n 1234 | wc -c
4
```

`-n` is used to avoid an extra newline character.

When `wc` is executed without any options as:

```
$ wc file
```

it will print number of lines, words, and characters delimited by tabs.

There's more...

Let's go through additional options available with `wc` command.

Print length of longest length line

`wc` can be also used to print the length of longest line using the `-L` option:

```
$ wc file -L
```

Printing directory tree

Graphically representing directories and filesystem as tree hierarchy is quite useful when preparing tutorials and documents. Also they are sometimes useful in writing certain monitoring scripts that helps to look at the filesystem using easy-to-read tree representations. Let's see how to do it.

Getting ready

The `tree` command is the hero that helps to print graphical trees of files and directories. Usually, `tree` does not come with Linux distributions. You need to install it using the package manager.

How to do it...

The following is a sample UNIX file system 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
```

`The tree` command comes with many interesting options, let us look at few of them.

Highlight only files matched by pattern as follows:

```
$ tree path -P PATTERN # Pattern should be wildcard
```

For example:

```
$ tree PATH -P "*.sh" # Replace PATH with a directory path
|-- home
|   |-- pactpub
|   |   `-- automate.sh
```

Highlight only files excluding the match pattern by using:

```
$ tree path -I PATTERN
```

In order to print size along with files and directories use the `-h` option as follows:

```
$ tree -h
```

There's more...

Let's see an interesting option that is available with the `tree` command.

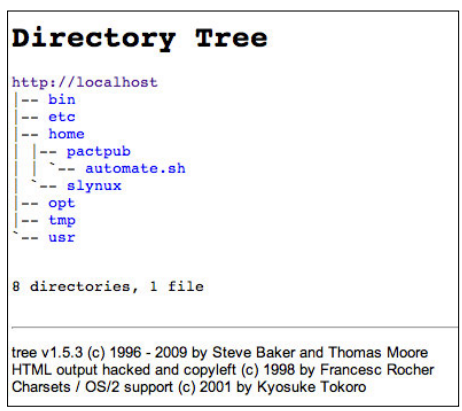
HTML output for tree

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

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

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

The web page generated from the directory listing will look as follows:



4
Texting and Driving

In this chapter, we will cover:

- A basic regular expression primer
- Searching and mining "text" inside a file with grep
- Column-wise cutting of a file with cut
- Determining the frequency of words used in a given file
- A basic sed primer
- A basic awk primer
- Replacing strings from a text or file
- Compressing or decompressing JavaScript
- Iterating through lines, words, and characters in a file
- Merging multiple files as columns
- Printing the nth word or column in a file or line
- Printing text between line numbers or patterns
- Checking palindrome strings with a script
- Printing lines in the reverse order
- Parsing e-mail address and URLs from text
- Printing a set number of lines before or after a pattern in a file
- Removing a sentence in a file containing a word
- Implementing head, tail, and tac with awk
- Text slicing and parameter operations

Introduction

The Shell Scripting language is packed with essential problem-solving components for UNIX/Linux systems. Bash can always provide some quick solutions to the problems in a UNIX environment. Text processing is one of the key areas where shell scripting is used. It comes with beautiful utilities such as sed, awk, grep, cut, and so on, which can be combined to solve text processing related problems. Most of the programming languages are designed to be generic, and hence it takes a lot of effort to write programs that can process text and produce the desired output. Since Bash is a language that is designed by also keeping text processing in mind, it has a lot of functionalities.

Various utilities help to process a file in fine detail as a character, line, word, column, row, and so on. Hence we can manipulate a text file in many ways. Regular expressions are the core of pattern matching techniques. Most of the text processing utilities come with regular expression support. By using suitable regular expression strings, we can produce the desired output such as filtering, stripping, replacing, searching, and much more.

This chapter includes a collection of recipes, which walks through many contexts of problems based on text processing that will be helpful in writing real scripts.

Basic regular expression primer

Regular expressions are the heart of the pattern-matching based text-processing techniques. For fluency in writing text-processing tools, one must have basic understanding of regular expressions. Regular expressions are a form of tiny, highly-specialized programming language used to match text. Using wild card techniques, the scope of matching text with patterns is very limited. This recipe is a walk through of basic regular expressions.

Getting ready

Regular expressions are the language used in most text processing utilities. Hence you will use the techniques learned in this recipe in many other recipes. `[a-z0-9_]+@[a-z0-9]+\.[a-z]+` is an example of regular expression for matching an e-mail address.

Does this seem weird? Don't worry, it is really simple once you understand the concepts.

How to do it...

In this section, we will go through regex, the POSIX character class, and meta characters.

Let's first go through the basic components of regular expressions (regex).

regex	Description	Example
^	The start of the line marker.	^tux matches a string that starts the line with tux.
$	The end of the line marker.	tux$ matches strings of a line that ends with tux.
.	Matches any one character.	Hack. matches Hack1, Hacki but not Hack12, Hackil, only one additional character matches.
[]	Matches any one of the characters enclosed in [chars].	coo[kl] matches cook or cool.
[^]	Matches any one of the characters EXCEPT those that are enclosed in [^chars].	9[^01] matches 92, 93 but not 91 or 90.
[-]	Matches any character within the range specified in [].	[1-5] matches any digits from 1 to 5.
?	The preceding item must match one or zero times.	colou?r matches color or colour but not colouur.
+	The preceding item must match one or more times.	Rollno-9+ matches Rollno-99, Rollno-9 but not Rollno-.
*	The preceding item must match zero or more times.	co*l matches cl, col, coool.
()	Creates a substring from the regex match.	ma(tri)?x matches max or matrix.
{n}	The preceding item must match n times.	[0-9]{3} matches any three-digit number. [0-9]{3} can be expanded as: [0-9][0-9][0-9].
{n,}	Minimum number of times that the preceding item should match.	[0-9]{2,} matches any number, that is, two digits or more.
{n, m}	Specifies the minimum and maximum number of times the preceding item should match.	[0-9]{2,5} matches any number that is having two digits to five digits.
|	Alternation—one of the items on either of sides of | should match.	Oct (1st | 2nd) matches Oct 1st or Oct 2nd.
\	The escape character for escaping any of the special characters mentioned above.	a\.b matches a.b but not ajb. It ignores special meaning of . by prefexing \.

A POSIX character class is a special meta sequence of the form `[:....:]` that can be used to match a range of specified characters. The POSIX classes are as follows:

Regex	Description	Example
`[:alnum:]`	Alphanumeric character	`[[:alnum:]]+`
`[:alpha:]`	Alphabet character (lowercase and uppercase)	`[[:alpha:]]{4}`
`[:blank:]`	Space and tab	`[[:blank:]]*`
`[:digit:]`	Digit	`[[:digit:]]?`
`[:lower:]`	Lowercase alphabet	`[[:lower:]]{5,}`
`[:upper:]`	Uppercase alphabet	`([[:upper:]]+)?`
`[:punct:]`	Punctuation	`[[:punct:]]`
`[:space:]`	All whitespace characters including newline, carriage return, and so on.	`[[:space:]]+`

Meta characters are a type of Perl-style regular expression that is supported by a subset of text processing utilities. Not all of the utilities will support the following notations. But the above character classes and regular expression are universally accepted.

Regex	Description	Example
`\b`	Word boundary	`\bcool\b` matches only `cool` not `coolant`.
`\B`	Non-word boundary	`cool\B` matches `coolant` and not `cool`.
`\d`	Single digit character	`b\db` matches `b2b` not `bcb`.
`\D`	Single non-digit	`b\Db` matches `bcb` not `b2b`.
`\w`	Single word character(alnum and _)	`\w` matches `1` or a not `&`.
`\W`	Single non-word character	`\w` matches `&` not `1` or `a`.
`\n`	Newline	`\n` Matches a new line.
`\s`	Single whitespace	`x\sx` matches `xx` not `xx`.
`\S`	Single non-space	`x\Sx` matches `xkx` not `xx`.
`\r`	Carriage return	`\r` matches carriage return.

How it works...

The tables seen in the previous section are the key element tables for regular expressions. By using the suitable keys from the tables, we can construct any suitable regular expression string to match text according to the context. regex is a generic language to match text. Therefore, we are not introducing any tools in this recipe. However, it follows in the other recipes in this chapter.

Let's see a few examples of text matching:

- In order to match all words in a given text, we can write the regex as:

  `( ?[a-zA-Z]+ ?)`

  "?" is the notation for optional space that precedes and follows a word. The `[a-zA-Z]+` notation represents one or more alphabet characters (a-z and A-Z).

- To match an IP address, we can write the regex as:

  `[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}`

  or

  `[[:digit:]]{1,3}\.[[:digit:]]{1,3}\.[[:digit:]]{1,3}\.[[:digit:]]{1,3}`

  We know that an IP address is in the form 192.168.0.2. It is in the form of four integers (each from 0-255) separated by dots (for example, 192.168.0.2).

  `[0-9]` or `[:digit:]` represents a match for digits 0-9. `{1,3}` matches one to three digits and `\.` matches ".".

There's more...

Let's see how the special meanings of certain characters are specified in the regular expressions.

Treatment of special characters

Regular expressions use some characters such as `$`, `^`, `.`, `*`, `+`, `{`, and `}` as special characters. But what if we want to use these characters as non-special characters (a normal text character)? Let's see an example.

regex: `[a-z]*.[0-9]`

How is this interpreted?

It can be zero or more [a-z] `([a-z]*)`, then any one character (`.`), and then one character in the set `[0-9]` such that it matches `abcdeO9`.

It can also be interpreted as one of `[a-z]`, then a character `*`, then a character `.` (period), and a digit such that it matches `x*.8`.

In order to overcome this problem, we precede the character with a forward slash "\" (doing this is called "escaping the character"). Characters such as * that have multiple meanings are prefixed with "\" to make them into a special meaning or to make them non special. Whether special characters or non-special characters are to be escaped varies depending on the tool that you are using.

Searching and mining "text" inside a file with grep

Searching inside a file is an important use case in text processing. We may need to search through thousands of lines in a file to find out some required data by using certain specifications. This recipe will help you learn how to locate data items of a given specification from a pool of data.

Getting ready

The `grep` command is the master UNIX utility for searching in the text. It accepts regular expressions and wild cards. We can produce output in various formats using the numerous interesting options that come with `grep`. Let's see how to do it.

How to do it...

Search in a file for a word as follows:

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

Or:

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

It will return lines of text that contain the given `match_pattern`.

We can also read from `stdin` as follows:

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

Perform a search in multiple files using a single `grep` invocation as follows:

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

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
```

Usually, the `grep` command considers `match_text` as a wildcard. To use regular expressions as input arguments, the `-E` option should be added—which means extended regular expression. Or we can a use regular expression enabled `grep` command, `egrep`. For example:

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

Or:

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

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

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

Or:

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

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.

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 6 matching items, it prints `2` since there are only `2` matching lines. Multiple matches in a single line are counted only once.

In order to count the number of matching items in a file, use the following hack:

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

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
```

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 above example, "not" is at the seventh offset position (that is, `not` starts from the seventh character in the line (`gnu is not unix`).

The `-b` option is always used with `-o`.

To search over many files and find out in which of the files a certain text matches use:

```
$ 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.

There's more...

We have used the basic usage examples for the `grep` command. But the `grep` command comes with rich features. Let's go through the different options available along with `grep`.

Recursively search many files

To recursively search for a text over many directories of descendants use:

```
$ grep "text" . -R -n
```

In this command `"."` specifies the current directory.

For example:

```
$ cd src_dir
$ grep "test_function()" . -R -n
./miscutils/test.c:16:test_function();
```

`test_function()` exists in line number 16 of `miscutils/test.c`.

This is one of the most frequently used commands by developers. It is used to find the file of source code in which a certain text exists.

Ignoring case of pattern

The `-i` argument helps match patterns to be evaluated without considering if the characters are uppercase or lowercase. For example:

```
$ echo hello world | grep -i "HELLO"
hello
```

grep by matching multiple patterns

Usually, we can specify single pattern for matching. However, we can use an argument `-e` to specify multiple patterns for matching as follows:

```
$ grep -e "pattern1" -e "pattern"
```

For example:

```
$ echo this is a line of text | grep -e "this" -e "line" -o
this
line
```

There is also another way to specify multiple patterns. We can use a pattern file for reading patterns. Write patterns to match line by line and execute `grep` with a `-f` argument as follows:

```
$ grep -f pattern_file source_filename
```

For example:

```
$ cat pat_file
hello
cool

$ echo hello this is cool | grep -f pat_file
hello this is cool
```

Include and exclude files (wild card pattern) in grep search

`grep` can include or exclude files in which to search. We can specify include files or exclude files using wild card patterns.

To search only `.c` and `.cpp` files recursively in a directory by excluding all other file types, use:

```
$ grep "main()" . -r  --include *.{c,cpp}
```

Note that `some{string1,string2,string3}` expands as `somestring1 somestring2 somestring3`.

Exclude all README files in the search as follows:

```
$ grep "main()" . -r --exclude "README"
```

To exclude directories use the `--exclude-dir` option.

To read a list of files to exclude from a file use `--exclude-from FILE`.

Using grep with xargs with zero-byte suffix

The `xargs` command is often used to provide a list of file names as a command-line argument to another command. When filenames are used as command-line arguments, it is recommended to use a zero-byte terminator for the file names instead of a space terminator. Some of the file names can contain a space character and it will be misinterpreted as a terminator and a single file name may be broken into two file names (for example, `New file.txt` can be interpreted as two filenames `New` and `file.txt`). This problem can be avoided by using a zero-byte suffix. We use `xargs` so as to accept `stdin` text from commands like `grep`, `find`, and so on. Such commands can output text to the `stdout` with a zero-byte suffix. In order to specify that the input terminator for filenames is zero byte (`\0`), we should use `–0` with `xargs`.

Create some test files as follows:

```
$ echo "test" > file1
$ echo "cool" > file2
$ echo "test" > file3
```

In the following command sequence, `grep` outputs filenames with a zero byte terminator (`\0`). It is specified by using the `–Z` option with `grep`. `xargs -0` reads the input and separates file names with a zero byte terminator:

```
$ grep "test" file* -lZ | xargs -0 rm
```

Usually, `-Z` is used along with `-l`.

Silent output for grep

The previously mentioned usages of `grep` return output in different formats. There are some cases when we need to know whether a file contains the specified text or not. We have to perform a test condition that returns true or false. It can be performed using the quiet condition (`-q`). In quiet mode, the `grep` command does not write any output to the standard output. Instead it runs the command and returns exit status based on success or failure.

We know that a command returns 0 if success and non-zero if failure.

Let's go through a script that makes uses of `grep` in quiet mode for testing whether a match text appears in a file or not.

```
#!/bin/bash
#Filename: silent_grep.sh
#Description: Testing whether a file contain a text or not

if [ $# -ne 2 ];
then
echo "$0 match_text filename"
fi

match_text=$1
filename=$2

grep -q $match_text $filename

if [ $? -eq 0 ];
then
echo "The text exists in the file"
else
echo "Text does not exist in the file"
fi
```

The `silent_grep.sh` script can be run as follows by providing a match word (`Student`) and a filename (`student_data.txt`) as the command argument:

```
$ ./silent_grep.sh Student student_data.txt
The text exists in the file
```

Print lines before and after text matches

Context-based printing is a one of the nice features of `grep`. Suppose a matching line for a given match text is found, `grep` usually prints only the matching lines. But we may need "n" lines after the matching lines or "n" lines before the matching line or both. It can be performed using context line control in `grep`. Let's see how to do it.

In order to print three lines after a match, use the `-A` option:

```
$ seq 10 | grep 5 -A 3
5
6
7
8
```

In order to print three lines before the match, use the `-B` option:

```
$ seq 10 | grep 5 -B 3
2
3
4
5
```

Print three lines after and before the match, use the `-C` option as follows:

```
$ seq 10 | grep 5 -C 3
2
3
4
5
6
7
8
```

If there are multiple matches, each section is delimited by a line "--":

```
$ echo -e "a\nb\nc\na\nb\nc" | grep a -A 1
a
b
--
a
b
```

Column-wise cutting of a file with cut

We may need to cut text by column rather than row. Let's assume that we have a text file containing student reports with columns, such as `No`, `Name`, `Mark`, and `Percentage`. We need to extract only the name of students to another file or any n-th column in the file or extract two or more columns. This recipe will illustrate how to perform this task.

Getting ready

`cut` is a small utility that often comes to our help for cutting in column fashion. It can also specify the delimiter that separates each column. In `cut` terminology, each column is known as a field.

How to do it...

In order to extract the first field or column, use the following syntax:

```
cut -f FIELD_LIST filename
```

`FIELD_LIST` is a list of columns that are to be displayed. The list consists of column numbers delimited by commas. For example:

```
$ cut -f 2,3 filename
```

Here, the second and the third columns are displayed.

`cut` can also read input text from `stdin`.

Tab is the default delimiter for fields or columns. If lines without delimiters are found, they are also printed. To avoid printing lines that do not have delimiter characters, attach the `-s` option along with `cut`. An example of using the `cut` command for columns is as follows:

```
$ cat student_data.txt
No  Name      Mark   Percent
1   Sarath     45      90
2   Alex       49      98
3   Anu        45      90

$ cut -f1 student_data.txt
No
1
2
3
```

Extract multiple fields as follows:

```
$ cut -f2,4 student_data.txt
Name      Percent
Sarath    90
Alex      98
Anu       90
```

To print multiple columns, provide a list of column numbers separated by commas as argument to `-f`.

We can also complement the extracted fields using the `--complement` option. Suppose you have many fields and you want to print all the columns except the third column, use:

```
$ cut -f3 --complement student_data.txt
No  Name    Percent
1   Sarath  90
2   Alex    98
3   Anu     90
```

To specify the delimiter character for the fields, use the `-d` option as follows:

```
$ cat delimited_data.txt
No;Name;Mark;Percent
1;Sarath;45;90
2;Alex;49;98
3;Anu;45;90

$ cut -f2 -d";" delimited_data.txt
Name
Sarath
Alex
Anu
```

There's more...

The `cut` command has more options to specify the character sequences to be displayed as columns. Let's go through the additional options available with `cut`.

Specifying range of characters or bytes as fields

Suppose that we don't rely on delimiters, but we need to extract fields such that we need to define a range of characters (counting from 0 as start of line) as a field, such extractions are possible with `cut`.

Let's see what notations are possible:

N-	from N-th byte, character or field, to end of line
N-M	from N-th to M-th (included) byte, character or field
-M	from first to M-th (included) byte, character or field

We use the above notations to specify fields as range of bytes or characters with the following options:

- `-b` for bytes
- `-c` for characters
- `-f` for defining fields

For example:

```
$ cat range_fields.txt
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxy
```

You can print the first to fifth characters as follows:

```
$ cut -c1-5 range_fields.txt
abcde
abcde
abcde
abcde
```

The first two characters can be printed as follows:

```
$ cut range_fields.txt -c-2
ab
ab
ab
ab
```

Replace `-c` with `-b` to count in bytes.

We can specify output delimiter while using with `-c`, `-f` and `-b` as:

```
    --output-delimiter "delimiter string"
```

When multiple fields are extracted with `-b` or `-c`, `--output-delimiter` is a must. Else, you cannot distinguish between fields if it is not provided. For example:

```
$ cut range_fields.txt -c1-3,6-9 --output-delimiter ","
abc,fghi
abc,fghi
abc,fghi
abc,fghi
```

Frequency of words used in a given file

Finding the frequency of words used in a file is an interesting exercise to apply the text processing skills. It can be done in many different ways. Let's see how to do it.

Getting ready

We can use associative arrays, awk, sed, grep, and so on to solve this problem in different ways.

How to do it...

Words are alphabetic characters delimited by space and dot. First we should parse all the words in the given file. Hence the count of each word needs to be found out. Words can be parsed by using regex with any of the tools such as sed, awk, or grep.

To find out the count of each word, we can have a different approach. One way of doing it is to loop through each word, and then use another loop to go through the words and check if they are equal. If they are equal, increment a count and print it at the end of file. This is an inefficient method. In an associative array, we use the word as the array index and count as the array value. We will only need one loop to achieve this by looping through each word. `array[word] = array[word] + 1` while initially its value is set `0`. Hence we can get an array containing the counts for each word.

Now let's do it. Create the shell script as follows:

```
#!/bin/bash
#Name: word_freq.sh
#Description: Find out frequency of words in a file

if [ $# -ne 1 ];
then
echo "Usage: $0 filename";
exit -1
fi

filename=$1

egrep -o "\b[[:alpha:]]+\b" $filename | \

awk '{ count[$0]++ }
END{ printf("%-14s%s\n","Word","Count") ;
for(ind in count)
{  printf("%-14s%d\n",ind,count[ind]);  }

}'
```

A sample output is as follows:

```
$ ./word_freq.sh words.txt
Word           Count
used            1
this            2
counting        1
```

How it works...

Here `egrep -o "\b[[:alpha:]]+\b" $filename` is used to output only words. The `-o` option will print the matching character sequence delimited by a newline character. Hence we receive words in each line.

`\b` is the word boundary character. `[:alpha:]` is a character class for alphabets.

The `awk` command is used to avoid the iteration through each word. Since `awk`, by default, executes the statements in the { } block for each row, we don't need a specific loop for doing that. Hence the count is incremented as `count[$0]++` using the associative array. Finally, in the `END{}` block, we print the words and their count by iterating through the words.

See also

- *Arrays and associative arrays* of *Chapter 1*, explains the arrays in Bash
- *Basic awk primer*, explains the awk command

Basic sed primer

sed stands for stream editor. It is a very essential tool for text processing. It is a marvelous utility that can play around regular expressions. A well-known usage of the `sed` command is for text replacement. This recipe will cover most of the frequently used `sed` techniques.

How to do it...

`sed` can be used to replace occurrences of a string with another string in a given text. It can be matched using regular expressions.

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

or

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

This command reads from `stdin`.

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
```

The previously seen `sed` commands will replace the first occurrence of the pattern in each line. But in order 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, sometimes we need not replace the first "N" occurrences, but only the rest of them. There is a built-in option to ignore the first "N" occurrences and replace from the"N+1th"occurrence onwards.

Have a look at the following commands:

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

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

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

Place `/Ng` when it needs to start the replacement from the N-th occurrence.

`/` in `sed` is 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 `\` prefix as:

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

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

There's more...

The `sed` command comes with numerous options for text manipulation. By combining the options available with `sed` in logical sequences, many complex problems can be solved in one line. Let's see some different options available with `sed`.

Removing blank lines

Removing blank lines is a simple technique using `sed` to remove blank lines. Blanks can be matched with regular expression `^$`:

```
$ sed '/^$/d' file
```

`/pattern/d` will remove lines matching the pattern.

For blank lines, the line end marker appears next to the line start marker.

Matched string notation (&)

In `sed` we can use `&` as the matched string for the substitution pattern such that we can use the matched string in replacement string.

For example:

```
$ echo this is an example | sed 's/\w\+/[&]/g'
[this] [is] [an] [example]
```

Here the regex `\w\+` matches every word. Then we replace it with `[&]`. `&` corresponds to the word that is matched.

Substring match notation (\1)

& is a string which corresponds to match string for the given pattern. But we can also match the substrings of the given pattern. Let's see how to do it.

```
$ echo this is digit 7 in a number | sed 's/digit \([0-9]\)/\1/'
this is 7 in a number
```

It replaces `digit 7` with `7`. `\(pattern\)` is used to match the substring. The pattern is enclosed in `()` and is escaped with slashes. For the first substring match, the corresponding notation is `\1`, for the second it is `\2`, and so on. Go through the following example with multiple matches:

```
$ echo seven EIGHT | sed 's/\([a-z]\+\) \([A-Z]\+\)/\2 \1/'
EIGHT seven
```

`([a-z]\+\)` matches the first word and `\([A-Z]\+\)` matches the second word. `\1` and `\2` are used for referencing them. This type of referencing is called back referencing. In the replacement part, their order is changed as `\2 \1` and hence it appears in reverse order.

Combination of multiple expressions

The combination of multiple `sed` using a pipe can be replaced as follows:

```
sed 'expression' | sed 'expression'
```

Which is equivalent to:

```
$ sed 'expression; expression'
```

Quoting

Usually, it is seen that the `sed` expression is quoted using single quotes. But double-quotes can also be used. Double-quotes expand the expression by evaluating it. Using double-quotes is useful when we want to use some variable string in a `sed` expression.

For example:

```
$ text=hello
$ echo hello world | sed "s/$text/HELLO/"
HELLO world
```

`$text` is evaluated as "hello".

Basic awk primer

`awk` is a tool designed to work with data streams. It is very interesting as it can operate on columns and rows. It supports many inbuilt functionalities such as arrays, functions, and so on, as in the C programming language. Flexibility is the greatest advantage of it.

How to do it...

The structure of an `awk` script looks like this:

```
awk ' BEGIN{  print "start" } pattern { commands } END{ print "end" }
file
```

The `awk` command can read from `stdin` also.

An `awk` script usually consists of three parts: `BEGIN`, `END`, and a common statements block with the pattern match option. The three of them are optional and any of them can be absent in the script. The script is usually enclosed in single-quotes or double-quotes as follows:

```
awk 'BEGIN { statements } { statements } END { end statements }'
```

Or, alternately, use:

```
awk "BEGIN { statements } { statements } END { end statements }"
```

For example:

```
$ awk 'BEGIN { i=0 } { i++ } END{ print i}' filename
```

Or:

```
$ awk "BEGIN { i=0 } { i++ } END{ print i }" filename
```

How it works...

The `awk` command works in the following manner:

1. Execute the statements in the `BEGIN { commands }` block.
2. Read one line from the file or `stdin`, and execute `pattern { commands }`. Repeat this step until the end of the file is reached.
3. When the end of the input stream is reached, execute the `END { commands }` block.

The `BEGIN` block is executed before `awk` starts reading lines from the input stream. It is an optional block. The statements such as variable initialization, printing the output header for an output table, and so on are common statements that are written in the `BEGIN` block.

The `END` block is similar to the `BEGIN` block. The `END` block gets executed when `awk` has completed reading all the lines from the input stream. The statements like printing results after analyzing all the values calculated for all the lines or printing the conclusion are the commonly-used statements in the `END` block (for example, after comparing all the lines, print the maximum number from a file). This is an optional block.

The most important block is the common commands with the pattern block. This block is also optional. If this block is not provided, by default `{ print }` gets executed so as to print each of the lines read. This block gets executed for each line read by `awk`.

It is like a while loop for line read with provided statements inside the body of the loop.

When a line is read, it checks whether the provided pattern matches the line. The pattern can be a regular expression match, conditions, range of lines match, and so on. If the current read line matches with the pattern, it executes the statements enclosed in `{ }`.

The pattern is optional. If pattern is not used, all the lines are matched and statements inside `{ }` are executed.

Let's go through the following example:

```
$ echo -e "line1\nline2" | awk 'BEGIN{ print "Start" } { print } END{
print "End" } '
Start
line1
line2
End
```

When `print` is used without an argument, it will print the current line. There are two important things to be kept in mind about `print`. When the arguments of the print are separated by commas, they are printed with a space delimiter. Double-quotes are used as the concatenation operator in the context of `print` in `awk`.

For example:

```
$ echo | awk '{ var1="v1"; var2="v2"; var3="v3"; \
print var1,var2,var3 ; }'
```

The above statement will print the values of the variables as follows:

```
v1 v2 v3
```

The `echo` command writes a single line into the standard output. Hence the statements in the { } block of `awk` are executed once. If standard input to `awk` contains multiple lines, the commands in `awk` will be executed multiple times.

Concatenation can be used as follows:

```
$ echo | awk '{ var1="v1"; var2="v2"; var3="v3"; \
print var1"-"var2"-"var3 ; }'
```

The output will be:

```
v1-v2-v3
```

{ } is like a block in a loop iterating through each line of a file.

Usually, we place initial variable assignments, such as `var=0;` and statements to print the file header in the `BEGIN` block. In the `END{}` block, we place statements such as printing results and so on.

There's more...

The `awk` command comes with lot of rich features. In order to master the art of awk programming you should be familiar with the important `awk` options and functionalities. Let's go through the essential functionalities of `awk`.

Special variables

Some special variables that can be used with `awk` are as follows:

- `NR`: It stands for number of records and corresponds to current line number under execution.
- `NF`: It stands for number of fields and corresponds to number of fields in the current line under execution (Fields are delimited by space).
- `$0`: It is a variable that contain the text content of current line under execution.
- `$1`: It is a variable that holds the text of the first field.
- `$2`: It is the variable that holds the test of the second field text.

For example:

```
$ echo -e "line1 f2 f3\nline2 f4 f5\nline3 f6 f7" | \

awk '{
print "Line no:"NR",No of fields:"NF, "$0="$0, "$1="$1,"$2="$2,"$3="$3
}'
Line no:1,No of fields:3 $0=line1 f2 f3 $1=line1 $2=f2 $3=f3
Line no:2,No of fields:3 $0=line2 f4 f5 $1=line2 $2=f4 $3=f5
Line no:3,No of fields:3 $0=line3 f6 f7 $1=line3 $2=f6 $3=f7
```

We can print last field of a line as `print $NF`, last but second as `$(NF-1)` and so on.

`awk` provides the `printf()` function with same syntax as in C. We can also use that instead of print.

Let's see some basic `awk` usage examples.

Print the second and third field of every line as follows:

```
$awk '{ print $3,$2 }'  file
```

In order to count the number of lines in a file, use the following command:

```
$ awk 'END{ print NR }' file
```

Here we only use the `END` block. `NR` will be updated on entering each line by `awk` with its line number. When it reaches the end line it will have the value of last line number. Hence, in the `END` block `NR` will have the value of last line number.

You can sum up all the numbers from each line of field 1 as follows:

```
$ seq 5 | awk 'BEGIN{ sum=0; print "Summation:" }
{ print $1"+"; sum+=$1 } END { print "=="; print sum }'
Summation:
1+
2+
3+
4+
5+
==
15
```

Passing a variable value from outside to awk

By using the `-v` argument, we can pass external values (other than from `stdin`) to `awk` as follows:

```
$ VAR=10000
$ echo | awk -v VARIABLE=$VAR'{ print VARIABLE }'
1
```

There is a flexible alternate method to pass many variable values from outside `awk`. For example:

```
$ var1="Variable1" ; var2="Variable2"
$ echo | awk '{ print v1,v2 }' v1=$var1 v2=$var2
Variable1 Variable2
```

When input is given through a file rather than standard input, use:

```
$ awk '{ print v1,v2 }' v1=$var1 v2=$var2 filename
```

In the above method, variables are specified as key-value pairs separated by space (`v1=$var1 v2=$var2`) as command arguments to awk soon after the BEGIN, { } and END blocks.

Reading a line explicitly using getline

Usually, `grep` reads all lines in a file by default. If you want to read one specific line, you can use the `getline` function. Sometimes we may need to read the first line from the `BEGIN` block.

The syntax is: `getline var`

The variable `var` will contain the content for the line.

If the `getline` is called without an argument, we can access the content of the line by using `$0`, `$1`, and `$2`.

For example:

```
$ seq 5 | awk 'BEGIN { getline; print "Read ahead first line", $0 } {
print $0 }'
Read ahead first line 1
2
3
4
5
```

Filtering lines processed by awk with filter patterns

We can specify some conditions for lines to be processed. For example:

```
$ awk 'NR < 5' # Line number less than 5
$ awk 'NR==1,NR==4' #Line numbers from 1-5
$ awk '/linux/' # Lines containing the pattern linux (we can specify
regex)
$ awk '!/linux/' # Lines not containing the pattern linux
```

Setting delimiter for fields

By default, the delimiter for fields is space. We can explicitly specify a delimiter using `-F "delimiter"`:

```
$ awk -F: '{ print $NF }' /etc/passwd
```

Or:

```
awk 'BEGIN { FS=":" } { print $NF }' /etc/passwd
```

We can set the output fields separator by setting `OFS="delimiter"` in the `BEGIN` block.

Reading command output from awk

In the following code, `echo` will produces a single blank line. The `cmdout` variable will contain output of command `grep root /etc/passwd` and it will print the line containing `root`:

The syntax for reading out of the 'command' in a variable 'output' is as follows:

```
"command" | getline output ;
```

For example:

```
$ echo | awk '{ "grep root /etc/passwd" | getline cmdout ; print cmdout
}'
root:x:0:0:root:/root:/bin/bash
```

By using `getline` we can read the output of external shell commands in a variable called `cmdout`.

`awk` supports associative arrays, which can use text as the index.

Using loop inside awk

A `for` loop is available in `awk`. It has the format:

```
for(i=0;i<10;i++) { print $i ; }
```

Or:

```
for(i in array) { print array[i]; }
```

`awk` comes with many built-in string manipulation functions. Let's have a look at a few of them:

- `length(string)`: It returns the string length.
- `index(string, search_string)`: It returns the position at which the `search_string` is found in the string.
- `split(string, array, delimiter)`: It stores the list of strings generated by using the delimiter in the array.
- `substr(string, start-position, end-position)`: It returns the substring created from the string by using start and end character offets.
- `sub(regex, replacement_str, string)`: It replaces the first occurring regular expression match from the string with `replacment_str`.
- `gsub(regex, replacment_str, string`: It is similar to `sub()`. But it replaces every regular expression match.
- `match(regex, string)`: It returns the result of whether a regular expression (regex) match is found in the string or not. It returns non-zero if match is found, else it returns zero. Two special variables are associated with `match()`. They are `RSTART` and `RLENGTH`. The `RSTART` variable contains the position at which the regular expression match starts. The `RLENGTH` variable contains the length of the string matched by the regular expression.

Replacing strings from a text or file

String replacement is a frequently-used text-processing task. It can be done easily with regular expressions by matching the required text.

Getting ready

When we hear the term 'replace', every system admin will recall sed. `sed` is the universal tool under UNIX-like systems to make replacements in text or in a file. Let's see how to do it.

How to do it...

The `sed` primer recipe contains most of the usages of `sed`. You can replace a string or pattern as follows:

```
$ sed 's/PATTERN/replace_text/g' filename
```

Or:

```
$ stdin | sed 's/PATTERN/replace_text/g'
```

We can also use double quote (") instead of single quote ('). When double quote (") is used, we can specify variables inside the `sed` pattern and replacement strings. For example:

```
$ p=pattern
$ r=replaced
$ echo "line containing apattern" | sed "s/$p/$r/g"
line containing a replaced
```

We can also use it without `g` in `sed`.

```
$ sed 's/PATTEN/replace_text/' filename
```

Then it will replace the occurrence of `PATTERN` first time it appears only. `/g` stands for global. That means, it will replace every occurrence of `PATTERN` in the file.

There's more...

We have seen basic text replacement with `sed`. Let's see how to save the replaced text in the source file itself.

Making replacement saved in the file

When a filename is passed to `sed`, it's output will be available to `stdout`. Instead of sending the output stream into `stdout`, to make changes saved in the file, use the `-i` option as follows:

```
$ sed 's/PATTERN/replacement/' -i filename
```

For example, replace all three-digit numbers with another specified number in a file as follows:

```
$ cat sed_data.txt
11 abc 111 this 9 file contains 111 11 88 numbers 0000

$ cat sed_data.txt  | sed 's/\b[0-9]\{3\}\b/NUMBER/g'
11 abc NUMBER this 9 file contains NUMBER 11 88 numbers 0000
```

The above one-liner replaces three-digit numbers only.`\b[0-9]\{3\}\b` is the regular expression used to match three-digit numbers. [0-9] is the range of digits, that is, from 0 to 9. {3} is used for matching the preceding character thrice. `\` in `\{3\}` is used to give a special meaning for { and }. `\b` is the word boundary marker.

See also

- *Basic sed primer*, explains the sed command

Compressing or decompressing JavaScript

JavaScript is widely used in designing websites. While writing JavaScript code, we use several white spaces, comments, and tabs for readability and maintenance of code. But the use of a lot of white spaces and tabs in JavaScript causes the file size to increase. As the file size increases, it increases page load times. Hence most of the professional websites use compressed JavaScripts for fast loading. Compression is mostly squeezing white spaces and newline characters. Once JavaScript is compressed, it can be decompressed by adding enough white space and newline characters, which makes it readable. Usually, obfuscated code also can be made readable by inserting white space and newlines. This recipe is an attempt to hack similar capabilities in the shell.

Getting ready

We are going to write a JavaScript compressor or obfuscation tool. Also a decompressing tool can be designed. We are going to get our hands dirty using text and character replacement tools `tr` and `sed`. Let's see how to do it.

How to do it...

Let's go through the logical sequences and the code required for compressing and decompressing the JavaScript.

```
$ cat sample.js
functionsign_out()
{

$("#loading").show();
$.get("log_in",{logout:"True"},

function(){

window.location="";

});


}
```

The following are the tasks we need to perform for compressing the JavaScript:

1. Remove newline and tab characters.
2. Squeeze spaces.
3. Replace comments /* content */.

4. Replace the following with substitutions:
    - "{ " with "{"
    - " }" with "}"
    - " (" with "("
    - ") " with ")"
    - ", " with ","
    - " ; " with ";" (we need to remove all extra spaces)

To decompress or to make the JavaScript more readable, we can use the following tasks:

1. Replace ";" with ";\n".
2. Replace "{" with "{\n" and "}" with "\n}".

How it works...

Let's compress the JavaScript by performing these tasks:

1. Remove the'\n' and '\t' characters:

```
tr -d '\n\t'
```

2. Remove extra spaces:

```
tr -s ' ' or sed 's/[ ]\+/ /g'
```

3. Remove comments:

```
sed 's:/\*.*\*/::g'
```

    - `:` is used as a sed delimiter to avoid the need of escaping `/` since we need to use `/*` and `*/`
    - `*` in the sed is escaped as `\*`
    - `.*` is used to match all text in between `/*` and `*/`

4. Remove all spaces preceding and suffixing the `{,},(,),;,:`, and comma.

```
sed 's/ \?\([{}();,:]\) \?/\1/g'
```

The above `sed` statement can be parsed as follows:

- `/ \?\([{}();,:]\) \?/` in the `sed` code is the match part and `/\1 /g` is the replacement part.

- `\([{}();,:]\)` is used to match any one character in the set `[ { }( ) ; , : ]` (inserted spaces for readability). `\(` and `\)` are group operators used to memorize the match and back reference in the replacement part. `(` and `)` are escaped to give them a special meaning as a group operator. `\?` precedes and follows the group operators. It is to match the space character that may precede or follow any of the characters in the set.
- In the replacement part, the match string (that is, the combination of `:` a space (optional), a character from the set, and again optional space) is replaced with the character matched. It uses a back reference to the character matched and memorized using the group operator `()`. Back-referenced characters refer to a group match by using the `\1` symbol.

Combine the above tasks using a pipe as follows:

```
$ catsample.js |  \
tr -d '\n\t' |  tr -s ' ' \
| sed 's:/\*.*\*/::g' \
| sed 's/ \?\([{}();,:]\) \?/\1/g'
```

The output is as follows:

```
functionsign_out(){$("#loading").show();$.get("log_
in",{logout:"True"},function(){window.location="";});}
```

Let's write a decompression script for making obfuscated code readable as follows:

```
$ cat obfuscated.txt | sed 's/;/;\n/g; s/{/{\n\n/g; s/}/\n\n}/g'
```

Or:

```
$ cat obfuscated.txt | sed 's/;/;\n/g' | sed 's/{/{\n\n/g' | sed 's/}/\n\
n}/g'
```

In the previous command:

- `s/;/;\n/g` replaces `;` with `\n;`
- `s/{/{\n\n/g` replaces `{` with `{\n\n`
- `s/}/\n\n}/g` replaces `}` with `\n\n}`

See also

- *Translating with tr* of *Chapter 2*, explains the tr command
- *Basic sed primer*, explains the sed command

Iterating through lines, words, and characters in a file

Iterating through character, word, and lines in a file is a frequently required script element while writing different text processing and file operation scripts. Even though it is simple to perform, we make simple mistakes and it gets erroneous without getting the expected output. This recipe will help you out to learn how to do it.

Getting ready

Iteration with a simple loop and redirection from `stdin` or file are basic components of performing the mentioned tasks.

How to do it...

In this recipe we discuss about performing three tasks of iterating through line, word, and characters. Let's see how each of these tasks can be performed.

1. **Iterate through each line in a file:**

   We can use a `while` loop to read from standard input. Hence it will read a line in each iteration.

   Use file redirection to `stdin` as follows:

   ```
   while read line;
   do
   echo $line;
   done < file.txt
   ```

   Use subshell as follows:

   ```
   cat file.txt | (  while read line; do echo $line; done )
   ```

   Here `cat file.txt` can be replaced with the output of any command sequence.

2. **Iterate through each word in a line**

   We can use a `while` loop to iterate through words in a line as follows:

   ```
   for word in $line;
   do
   echo $word;
   done
   ```

3. **Iterate through each character in a word**

   We can use a `for` loop to iterate a variable `i` from `0` to the length of string. A character can be extracted from the string in each iteration using the special notation `${string:start_position:No_of_characters}`.

```
for((i=0;i<${#word};i++))
do
echo ${word:i:1} ;
done
```

How it works...

Reading lines of a file and reading words in a line are direct ways. But reading a character of a word is a little hack. We use the substring extraction technique.

`${word:start_position:no_of_characters}` returns a substring of a string held in variable `word`.

`${#word}` returns the length of the variable `word`.

See also

- *Field separators and iterators* of *Chapter 1*, explains different loops in Bash.
- *Text slicing and parameter operations*, explains extracting characters from a string.

Merging multiple files as columns

There are different cases when we require to concatenate files in columns. We may need each file's content to appear in separate columns. Usually, the `cat` command concatenates in a line- or row-wise fashion.

How to do it...

`paste` is the command that can be used for column-wise concatenation. The `paste` command can be used with the following syntax:

```
$ paste file1 file2 file3 …
```

Let's try an example as follows:

```
$ cat paste1.txt
1
2
```

```
3
4
5
$ cat paste2.txt
slynux
gnu
bash
hack
$ paste paste1.txt paste2.txt
1slynux
2gnu
3bash
4hack
5
```

The default delimiter is Tab. We can also explicitly specify the delimiter using `-d`. For example:

```
$ paste paste1.txt paste2.txt -d ","
1,slynux
2,gnu
3,bash
4,hack
5,
```

See also

- *Column-wise cutting of a file with cut*, explains extracting data from text files

Printing the nth word or column in a file or line

We may get a file having a number of columns and only a few will actually be useful. In order to print only relevant columns or fields, we filter it.

Getting ready

The most widely-used method is to use `awk` for doing this task. It can be also done using `cut`.

How to do it...

To print the fifth column use the following command:

```
$ awk '{ print $5 }' filename
```

We can also print multiple columns and we can insert our custom string in between columns.

For example, to print the permission and filename of each file in the current directory, use:

```
$ ls -l | awk '{ print $1" :  " $8 }'
-rw-r--r-- :  delimited_data.txt
-rw-r--r-- :  obfuscated.txt
-rw-r--r-- :  paste1.txt
-rw-r--r-- :  paste2.txt
```

See also

- *Basic awk primer*, explains the awk command
- *Column-wise cutting of a file with cut*, explains extracting data from text files

Printing text between line numbers or patterns

We may require to print certain section of text lines based on conditions such as a range of line numbers, range matched by start and end pattern and so on. Let's see how to do it.

Getting ready

We can use utilities such as awk, grep, and sed to perform the printing of a section based on conditions. Still I found `awk` to be the simplest one to understand. Let's do it using `awk`.

How to do it...

In order to print lines of text in a range of line numbers, M to N, use the following syntax:

```
$ awk 'NR==M, NR==N' filename
```

Or, it can take `stdin` input as follows:

```
$ cat filename | awk 'NR==M, NR==N'
```

Replace `M` and `N` with numbers as follows:

```
$ seq 100 | awk 'NR==4,NR==6'
4
5
6
```

To print lines of text in a section with `start_pattern` and `end_pattern`, use the following syntax:

```
$ awk '/start_pattern/, /end _pattern/' filename
```

For example:

```
$ cat section.txt
line with pattern1
line with pattern2
line with pattern3
line end with pattern4
line with pattern5

$ awk '/pa.*3/, /end/' section.txt
line with pattern3
line end with pattern4
```

The patterns used in `awk` are regular expressions.

See also

- *Basic awk primer*, explains the awk command

Checking palindrome strings with a script

Checking whether a string is palindrome is one of the first lab exercises in a C programming course. However, here we have included this recipe to give you an idea of how to solve similar problems in which pattern matching can be extended in a way that previously occurring patterns repeat in the text.

Getting ready

The `sed` command has the capability to remember a previously-matched sub pattern. It is called back referencing. We can solve palindrome problems by using back referencing. We can solve this using multiple ways in Bash.

How to do it...

`sed` can remember previously matched regular expression patterns, thereby we can identify whether duplicates of a character exists in a string. This capability to remember and reference previously matched patterns is called back-reference.

Let's see how we can apply back-referencing in a simpler manner to solve the problem. For example:

```
$ sed -n '/\(.\)\1/p' filename
```

`\(.\)` corresponds to memorize the one sub string inside ( ). Here it is . (period) which is also sed's single character wildcard character.

`\1` corresponds to the memory of the first match inside (). `\2` corresponds to the second match. Hence we can memorize many blocks enclosed in `().` `()` appears as `\( \)` to give `(` and `)` special meaning rather than just a character.

The previous `sed` statement will print any pattern matching two exactly the same.

The structure of all palindrome words is as follows:

- Even number of characters and a sequence of characters concatenated with same characters in reverse order
- Odd number of characters with a sequence of characters concatenated with reverse of same characters, but a common character in between the first sequence and its reverse

Therefore, for matching both, we can keep an optional character in between while writing the regular expression.

A `sed` regex matching a three-letter palindrome word will look like the following:

```
'/\(.\).\1/p'
```

We can place an extra character (`.`) in between the character sequence and its reverse sequence.

Let's write a script that can match a palindrome string of any length as follows:

```
#!/bin/bash
#Filename: match_palindrome.sh
#Description: Find out palindrome strings from a given file
if [ $# -ne 2 ];
then
echo "Usage: $0 filename string_length"
exit -1
fi
```

```
filename=$1 ;
basepattern='/^\(.\)'
count=$(( $2 / 2 ))
for((i=1;i<$count;i++))
do
basepattern=$basepattern'\(.\)' ;
done
if [ $(( $2 % 2 )) -ne 0 ];
then
basepattern=$basepattern'.' ;
fi
for((count;count>0;count--))
do
basepattern=$basepattern'\'"$count" ;
done
basepattern=$basepattern'$/p'
sed -n "$basepattern" $filename
```

Use the dictionary file as the input file to get a list of palindrome words of a given string length. For example:

```
$ ./match_palindrome.sh /usr/share/dict/british-english 4
noon
peep
poop
sees
```

How it works...

The working of the above script is simple. Most of the work is done to generate the `sed` script for a regular expression and a back-reference string generation.

Let's go through its working with the help of some worked out examples.

- If you want to match the character and back-reference it, we use `\(.\)` to match one character and `\1` to reference it. Hence, in order match a two letter palindrome and print it, we use:

  ```
  sed '/\(.\)\1/p'
  ```

  Now, to specify that match string from the beginning of the line, we add line-begin market ^ so that it will become `sed'/^\(.\)\1/p'`. `/p` is used to print the match.

- If we want to match four character palindrome, we use:

  ```
  sed '/^\(.\)\(.\)\2\1/p'
  ```

  We have used two `\(.\)` to match two characters and remember them. Anything enclosed within `\(` and `\)` will be remembered by `sed` and can be back-referenced. `\2\1` is used to back-reference in the reverse order of the matched characters.

In the above script, we have a variable called `basepattern`, which contains the `sed` script.

The pattern is generated using a `for` loop based on the number of characters in the palindrome string.

Initially, basepattern is initialized as `basepattern='/^\(.\)'`, which corresponds to a one-character match. A `for` loop is used to concatenate `\(.\)` with `basepattern` for half the number of times of the length of palindrome string. Again a `for` loop is used to concatenate back-references in the reverse order (like `'\4\3\2\1'`) half the number of times the length of palindrome string. Finally, in order to support palindrome strings with odd length an optional character (`.`) is enclosed between match regex and back-references.

Thus the `sed` palindrome match pattern is crafted. This crafted string is used to find out the palindrome strings from the dictionary file.

In the above script, we have used `sed` pattern generation using `for` loops. Actually there is no need to generate pattern separately. The `sed` command has its own loop implementation using labels and goto. `sed` is a vast language. Palindrome check can be done in a single line using a complex `sed` script. It is hard to explain it from scratch. Just try out the following script:

```
$ word="malayalam"
$ echo $word | sed ':loop ; s/^\(.\)\(.*\)\1/\2/; t loop; /^.\?$/{ s/.*/
PALINDROME/ ; q; };  s/.*/NOT PALINDROME/ '
PALINDROME
```

If you are interested in deep scripting with `sed`, refer to the complete `sed` and `awk` reference book: *sed & awk*, Second Edition by Dale Dougherty and Arnold Robbins.

Try to parse the above one-line `sed` script to test the palindrome using the book.

There's more...

Now let's see some other options, or possibly some pieces of general information that are relevant to this task.

Simplest and direct method

The simplest method to check whether a string is a palindrome is by using the rev command.

The `rev` command takes a file or `stdin` as input and prints the reversed string of every line.

Let's do it:

```
string="malayalam"
if [[ "$string" == "$(echo $string | rev )" ]];
then
echo "Palindrome"
else
echo "Not palindrome"
fi
```

The `rev` command can be used along with other commands to solve different problems. Let's look at an interesting example to reverse the words in a sentence:

```
sentence='this is line from sentence'
echo $sentence | rev | tr ' ' '\n' | tac | tr '\n' ' ' | rev
```

The output is as follows:

```
sentence from line is this
```

In the above one-liner, the characters are reversed first using the `rev` command. Then the words are separated into a word per line by replacing space with the `\n` character by using the `tr` command. Now the lines are reversed in order using the `tac` command. Again, lines are merged into a line using `tr`. Now `rev` is again applied so that a line with words is in the reverse order.

See also

- *Basic sed primer*, explains the sed command
- *Comparisons and tests* of *Chapter 1*, explains the string comparison operators

Printing lines in the reverse order

This is a simple recipe. It may not seem very useful but it can be used to emulate the stack data structure in Bash. This is something interesting. Let's print the lines of text in a file in reverse order.

Getting ready

A little hack with `awk` can do the task. However, there is a direct command `tac` to do the same as well. `tac` is the reverse of `cat`.

How to do it...

Let's do it with `tac` first. The syntax is as follows:

```
tac file1 file2 …
```

It can also read from `stdin` as follows:

```
$ seq 5 | tac
5
4
3
2
1
```

In `tac`, `\n` is the line separator. But we can also specify our own separator by using the `-s` "separator" option.

Let's do it in `awk` as follows:

```
$ seq 9 | \
awk '{ lifo[NR]=$0; lno=NR }
END{ for(;lno>-1;lno--){ print lifo[lno]; }
}'
```

\ in the shell script is used to conveniently break a single line command sequence into multiple lines.

How it works...

The `awk` script is very simple. We store each of the lines into an associative array with the line number as array index (NR gives line number). In the end, `awk` executes the `END` block. In order to get last line number `lno=NR` is used in the { } block. Hence it iterates from the last line number to `0` and prints the lines stored in the array in reverse order.

See also

- *Implementing head, tail, and tac with awk*, explains writing tac using awk

Parsing e-mail addresses and URLs from text

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

Getting ready

This problem can be solved with utilities egrep.

How to do it...

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

egrep regex: `[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 />
<ahref="http://code.google.com"><h1>Heading</h1>

$ 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:

```
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
```

How it works...

The regular expressions are really easy to design part by part. In the e-mail regex, we all know that an e-mail address takes the form `name@domain.some_2-4_letter`. Here the same is written in regex language as follows:

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

`[A-Za-z0-9.]+` means that some combination of characters in the `[]` block should appear one or more times (that is the meaning of +) before a literal `@` character appears. Then `[A-Za-z0-9.]` also should appear one or more times (+). The pattern `\.` means that a literal period should appear and finally the last part should be of length 2 to 4 alphabetic characters.

The case of an HTTP URL is similar to that of an e-mail address but without the `name@` match part of e-mail regex.

```
http://[a-zA-Z0-9.]+\.[a-zA-Z]{2,3}
```

See also

- *Basic sed primer*, explains the sed command
- *Basic regular expression primer*, explains how to use regular expressions

Printing n lines before or after a pattern in a file

Printing a section of text by pattern matching is frequently used in text processing. Sometimes we may need the lines of text before a pattern or after a pattern appears in a text. For example, consider that there is a file containing the rating of film actors where each line corresponds to a film actor's details, and we need to find out the rating of an actor along with the details of actors who are nearest to them in rating. Let's see how to do it.

Getting ready

`grep` is the best tool for searching and finding text in a file. Usually, `grep` prints a matching line or matching text for a given pattern. But the context line control options in `grep` enables it to print before, after, and before-after lines around the line of pattern match.

How to do it...

This technique can be better explained with a film actor list. For example:

```
$ cat actress_rankings.txt | head -n 20
1 Keira Knightley
2 Natalie Portman
3 Monica Bellucci
4 Bonnie Hunt
5 Cameron Diaz
6 Annie Potts
7 Liv Tyler
8 Julie Andrews
9 Lindsay Lohan
10 Catherine Zeta-Jones
11 CateBlanchett
12 Sarah Michelle Gellar
13 Carrie Fisher
14 Shannon Elizabeth
15 Julia Roberts
16 Sally Field
17 TéaLeoni
18 Kirsten Dunst
19 Rene Russo
20 JadaPinkett
```

In order to print three lines after the match "Cameron Diaz" along with the matching line, use the following command:

```
$ grep -A 3 "Cameron Diaz" actress_rankings.txt
5 Cameron Diaz
6 Annie Potts
7 Liv Tyler
8 Julie Andrews
```

In order to print the matched line and the preceding three lines, use the following command:

```
$ grep -B 3 "Cameron Diaz" actress_rankings.txt
2 Natalie Portman
3 Monica Bellucci
4 Bonnie Hunt
5 Cameron Diaz
```

Print the matched line and the two lines before and after the matched line as follows:

```
$ grep -C 2 "Cameron Diaz" actress_rankings.txt
3 Monica Bellucci
4 Bonnie Hunt
```

```
5 Cameron Diaz
6 Annie Potts
7 Liv Tyler
```

Are you wondering where I got this ranking from?

I parsed a website having full of images and HTML content just using basic sed, awk, and grep commands. See the chapter: *Tangled Web? Not at all*.

See also

- *Searching and mining "text" inside a file with grep*, explains the grep command.

Removing a sentence in a file containing a word

Removing a sentence containing a word is a simple task when a correct regular expression is identified. This is just an exercise on solving similar problems.

Getting ready

`sed` is the best utility for making substitutions. Hence let's use `sed` to replace the matched sentence with a blank.

How to do it...

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. It has a server market share ranging
between 20-40%. Most desktop computers run either Microsoft Windows or
Mac OS X, with Linux having anywhere from a low of an estimated 1-2% of
the desktop market to a high of an estimated 4.8%. However, desktop use
of Linux has become increasingly popular in recent years, partly owing
to the popular Ubuntu, Fedora, Mint, and openSUSE distributions and the
emergence of netbooks and smart phones running an embedded Linux.
```

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. It has a server market share ranging between 20-40%. Most
desktop computers run either Microsoft Windows or Mac OS X, with Linux
having anywhere from a low of an estimated 1-2% of the desktop market to
a high of an estimated 4.8%. However, desktop use of Linux has become
increasingly popular in recent years, partly owing to the popular Ubuntu,
Fedora, Mint, and openSUSE distributions and the emergence of netbooks
and smart phones running an embedded Linux.
```

How it works...

Let's evaluate the sed regex `'s/ [^.]*mobile phones[^.]*\.//g'`.

It has the format `'s/substitution_pattern/replacement_string/g`.

It replaces every occurrence of `substitution_pattern` with the replacement string.

Here the substitution pattern is the regex for a sentence. Every sentence is delimited by "." and the first character is a space. Therefore, we need to match the text that is in the format "space" some text MATCH_STRING some text "dot". A sentence may contain any characters except a "dot", which is the delimiter. Hence we have used [^.]. [^.]* matches a combination of any characters except dot. In between the text match string "mobile phones" is placed. Every match sentence is replaced by `//` (nothing).

See also

- *Basic sed primer*, explains the sed command
- *Basic regular expression primer*, explains how to use regular expressions

Implementing head, tail, and tac with awk

Mastering text-processing operations comes with practice. This recipe will help us practice incorporating some of the commands that we have just learned with some that we already know.

Getting ready

The commands `head`, `tail`, `uniq`, and `tac` operate line by line. Whenever we need line by line processing, we can always use `awk`. Let's emulate these commands with `awk`.

How to do it...

Let's see how different commands can be emulated with different basic text processing commands, such as head, tail, and tac.

The `head` command reads the first ten lines of a file and prints them out:

```
$ awk 'NR <=10' filename
```

The `tail` command prints the last ten lines of a file:

```
$ awk '{ buffer[NR % 10] = $0; } END { for(i=1;i<11;i++) { print
buffer[i%10] } }' filename
```

The `tac` command prints the lines of input file in reverse order:

```
$ awk '{ buffer[NR] = $0; } END { for(i=NR; i>0; i--) { print buffer[i] }
}' filename
```

How it works...

In the implementation of `head` using `awk`, we print the lines in the input stream having a line number less than or equal to `10`. The line number is available using the special variable `NR`.

In the implementation of the `tail` command a hashing technique is used. The buffer array index is determined by a hashing function `NR % 10`, where `NR` is the variable that contains the Linux number of current execution. `$0` is the line in the text variable. Hence `%` maps all the lines having the same remainder in the hash function to a particular index of an array. In the `END{}` block, it can iterate through ten index values of an array and print the lines stored in a buffer.

In the `tac` command emulation, it simply stores all the lines in an array. When it appears in the `END{}` block, `NR` will be holding the line number of the last line. Then it is decremented in a `for` loop until it reaches `1` and it prints the lines stored in each iteration statement.

See also

- *Basic awk primer*, explains the awk command
- *head and tail - printing the last or first 10 lines* of *Chapter 3*, explains the commands head and tail
- *Sorting, unique and duplicates* of *Chapter 2*, explains the uniq command
- *Printing lines in reverse order*, explains the tac command

Text slicing and parameter operations

This recipe walks through some of the simple text replacement techniques and parameter expansion short hands available in Bash. A few simple techniques can often help us avoid having to write multiple lines of code.

How to do it...

Let's get into the tasks.

Replacing some text from a variable can be done as follows:

```
$ var="This is a line of text"
$ echo ${var/line/REPLACED}
This is a REPLACED of text"
```

`line` is replaced with `REPLACED`.

We can produce a sub-string by specifying the start position and string length, by using the following syntax:

```
${variable_name:start_position:length}
```

To print from the fifth character onward use the following command:

```
$ string=abcdefghijklmnopqrstuvwxyz
$ echo ${string:4}
efghijklmnopqrstuvwxyz
```

To print eight characters starting from the fifth character, use:

```
$ echo ${string:4:8}
efghijkl
```

The index is specified by counting the start letter as `0`. We can also specify counting from last letter as `-1`. It is but used inside a parenthesis. `(-1)` is the index for the last letter.

```
echo ${string:(-1)}
z
$ echo ${string:(-2):2}
yz
```

See also

- *Iterating through lines, words, and characters in a file*, explains slicing of a character from a word

5
Tangled Web? Not At All!

In this chapter, we will cover:

- Downloading from a web page
- Downloading a web page as formatted plain text
- A primer on cURL
- Accessing unread Gmail mails from the command line
- Parsing data from a website
- Creating an image crawler and downloader
- Creating a web photo album generator
- Building a Twitter command-line client
- Define utility with Web backend
- Finding broken links in a website
- Tracking changes to a website
- Posting to a web page and reading response

Introduction

The Web is becoming the face of technology. It is the central access point for data processing. Though shell scripting cannot do everything that languages like PHP can do on the Web, there are still many tasks to which shell scripts are ideally suited. In this chapter we will explore some recipes that can be used to parse website content, download and obtain data, send data to forms, and automate website usage tasks and similar activities. We can automate many activities that we perform interactively through a browser with a few lines of scripting. Access to the functionalities provided by the HTTP protocol with command-line utilities enables us to write scripts that are suitable to solve most of the web-automation utilities. Have fun while going through the recipes of this chapter.

Downloading from a web page

Downloading a file or a web page from a given URL is simple. A few command-line download utilities are available to perform this task.

Getting ready

`wget` is a file download command-line utility. It is very flexible and can be configured with many options.

How to do it...

A web page or a remote file can be downloaded using `wget` as follows:

```
$ wget URL
```

For example:

```
$ wget http://slynux.org
--2010-08-01 07:51:20--  http://slynux.org/
Resolving slynux.org... 174.37.207.60
Connecting to slynux.org|174.37.207.60|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 15280 (15K) [text/html]
Saving to: "index.html"

100%[======================================>] 15,280      75.3K/s   in
0.2s

2010-08-01 07:51:21 (75.3 KB/s) - "index.html" saved [15280/15280]
```

It is also possible to specify multiple download URLs as follows:

```
$ wget URL1 URL2 URL3 ..
```

A file can be downloaded using `wget` using the URL as:

```
$ wget ftp://example_domain.com/somefile.img
```

Usually, files are downloaded with the same filename as in the URL and the download log information or progress is written to `stdout`.

You can specify the output file name with the `-O` option. If the file with the specified filename already exists, it will be truncated first and the downloaded file will be written to the specified file.

You can also specify a different logfile path rather than printing logs to `stdout` by using the `-o` option as follows:

```
$ wget ftp://example_domain.com/somefile.img -O dloaded_file.img -o log
```

By using the above command, nothing will be printed on screen. The log or progress will be written to `log` and the output file will be `dloaded_file.img`.

There is a chance that downloads might break due to unstable Internet connections. Then we can use the number of tries as an argument so that once interrupted, the utility will retry the download that many times before giving up.

In order to specify the number of tries, use the `-t` flag as follows:

```
$ wget -t 5 URL
```

There's more...

The `wget` utility has several additional options that can be used under different problem domains. Let's go through a few of them.

Restricted with speed downloads

When we have a limited Internet downlink bandwidth and many applications sharing the internet connection, if a large file is given for download, it will suck all the bandwidth and may cause other process to starve for bandwidth. The `wget` command comes with a built-in option to specify the maximum bandwidth limit the download job can possess. Hence all the applications can simultaneously run smoothly.

We can restrict the speed of `wget` by using the `--limit-rate` argument as follows:

```
$ wget  --limit-rate 20k http://example.com/file.iso
```

In this command `k` (kilobyte) and `m` (megabyte) specify the speed limit.

We can also specify the maximum quota for the download. It will stop when the quota is exceeded. It is useful when downloading multiple files limited by the total download size. This is useful to prevent the download from accidently using too much disk space.

Use `--quota` or `–Q` as follows:

```
$ wget -Q 100m http://example.com/file1 http://example.com/file2
```

Resume downloading and continue

If a download using `wget` gets interrupted before it is completed, we can resume the download where we left off by using the `-c` option as follows:

```
$ wget -c URL
```

Using cURL for download

cURL is another advanced command-line utility. It is much more powerful than `wget`.

cURL can be used to download as follows:

```
$ curl http://slynux.org > index.html
```

Unlike `wget`, `curl` writes the downloaded data into standard output (`stdout`) rather than to a file. Therefore, we have to redirect the data from `stdout` to the file using a redirection operator.

Copying a complete website (mirroring)

`wget` has an option to download the complete website by recursively collecting all the URL links in the web pages and downloading all of them like a crawler. Hence we can completely download all the pages of a website.

In order to download the pages, use the `--mirror` option as follows:

```
$ wget --mirror exampledomain.com
```

Or use:

```
$ wget -r -N -l DEPTH URL
```

`-l` specifies the `DEPTH` of web pages as levels. That means it will traverse only that much number of levels. It is used along with `–r` (recursive). The `-N` argument is used to enable time stamping for the file. `URL` is the base URL for a website for which the download needs to be initiated.

Accessing pages with HTTP or FTP authentication

Some web pages require authentication for HTTP or FTP URLs. This can be provided by using the `--user` and `--password` arguments:

```
$ wget –-user username –-password pass URL
```

It is also possible to ask for a password without specifying the password inline. In order to do that use `--ask-password` instead of the `--password` argument.

Downloading a web page as formatted plain text

Web pages are HTML pages containing a collection of HTML tags along with other elements, such as JavaScript, CSS, and so on. But the HTML tags define the base of a web page. We may need to parse the data in a web page while looking for specific content, and this is something Bash scripting can help us with. When we download a web page, we receive an HTML file. In order to view formatted data, it should be viewed in a web browser. However, in most of the circumstances, parsing a formatted text document will be easier than parsing HTML data. Therefore, if we can get a text file with formatted text similar to the web page seen on the web browser, it is more useful and it saves a lot of effort required to strip off HTML tags. Lynx is an interesting command-line web browser. We can actually get the web page as plain text formatted output from Lynx. Let's see how to do it.

How to do it...

Let's download the webpage view, in ASCII character representation, in a text file using the `–dump` flag with the `lynx` command:

```
$ lynx -dump URL > webpage_as_text.txt
```

This command will also list all the hyper-links (`<a href="link">`) separately under a heading **References** as the footer of the text output. This would help us avoid parsing of links separately using regular expressions.

For example:

```
$ lynx -dump http://google.com > plain_text_page.txt
```

You can see the plain text version of text by using the `cat` command as follows:

```
$ cat plain_text_page.txt
```

A primer on cURL

cURL is a powerful utility that supports many protocols including HTTP, HTTPS, FTP, and much more. It supports many features including POST, cookie, authentication, downloading partial files from a specified offset, referers, user agent strings, extra headers, limit speed, maximum file size, progress bars, and so on. cURL is useful for when we want to play around with automating a web page usage sequence and to retrieve data. This recipe is a list of the most important features of cURL.

Getting ready

cURL doesn't come with any of the main Linux distros by default, so you may have to install it using the package manager. By default, most distributions ship with `wget`.

cURL usually dumps downloaded files to `stdout` and progress information to `stderr`. To avoid progress information from being shown, we always use the `--silent` option.

How to do it...

The `curl` command can be used to perform different activities such as downloading, sending different HTTP requests, specifying HTTP headers, and so on. Let's see how to perform different tasks with cURL.

```
$ curl URL --silent
```

The above command dumps the downloaded file into the terminal (the downloaded data is written to `stdout`).

The `--silent` option is used to prevent the `curl` command from displaying progress information. If progress information is required, remove `--silent`.

```
$ curl URL --silent -O
```

The `-O` option is used to write the downloaded data into a file with the filename parsed from the URL rather than writing into the standard output.

For example:

```
$ curl http://slynux.org/index.html --silent -O
```

`index.html` will be created.

It writes a web page or file to the filename as in the URL instead of writing to `stdout`. If filenames are not there in the URL, it will produce an error. Hence, make sure that the URL is a URL to a remote file. `curl http://slynux.org -O --silent` will display an error since the filename cannot be parsed from the URL.

```
$ curl URL --silent -o new_filename
```

The `-o` option is used to download a file and write to a file with a specified file name.

In order to show the # progress bar while downloading, use `--progress` instead of `--silent`.

```
$ curl http://slynux.org -o index.html --progress
################################## 100.0%
```

There's more...

In the previous sections we have learned how to download files and dump HTML pages to the terminal. There several advanced options that come along with cURL. Let's explore more on cURL.

Continue/Resume downloading

cURL has advanced resume download features to continue at a given offset unlike `wget`. It helps to download portions of files by specifying an offset.

```
$ curl URL/file -C offset
```

The offset is an integer value in bytes.

cURL doesn't require us to know the exact byte offset if we want to resume downloading a file. If you want cURL to figure out the correct resume point, use the `-C -` option, like this:

```
$ curl -C - URL
```

cURL will automatically figure out where to restart the download of the specified file.

Set referer string with cURL

Referer is a string in the HTTP header used to identify the page from which the user reaches the current web page. When a user clicks on a link from web page A and it reaches web page B, the referer header string in the page B will contain a URL of page A.

Some dynamic pages check the referer string before returning HTML data. For example, a web page shows a Google logo attached page when a user navigates to a website by searching on Google, and shows a different page when they navigate to the web page by manually typing the URL.

The web page can write a condition to return a Google page if the referer is `www.google.com` or else return a different page.

You can use `--referer` with the `curl` command to specify the referer string as follows:

```
$ curl --referer Referer_URL target_URL
```

For example:

```
$ curl --referer http://google.com http://slynux.org
```

Cookies with cURL

Using `curl` we can specify as well as store cookies encountered during HTTP operations.

In order to specify cookies, use the `--cookie "COOKIES"` option.

Cookies should be provided as `name=value`. Multiple cookies should be delimited by a semicolon ";". For example:

```
$ curl http://example.com --cookie "user=slynux;pass=hack"
```

In order to specify a file to which cookies encountered are to be stored, use the `--cookie-jar` option. For example:

```
$ curl URL --cookie-jar cookie_file
```

Setting a user agent string with cURL

Some web pages that check the user-agent won't work if there is no user-agent specified. You may have noticed that certain websites work well only in Internet Explorer (IE). If a different browser is used, the website will show a message that it will work only on IE. This is because the website checks for a user agent. You can set the user agent as IE with `curl` and see that it returns a different web page in this case.

Using cURL it can be set using `--user-agent` or `-A` as follows:

```
$ curl URL --user-agent "Mozilla/5.0"
```

Additional headers can be passed with cURL. Use `-H "Header"` to pass multiple additional headers. For example:

```
$ curl -H "Host: www.slynux.org" -H "Accept-language: en" URL
```

Specifying bandwidth limit on cURL

When the available bandwidth is limited and multiple users are sharing the Internet, in order to perform the sharing of bandwidth smoothly, we can limit the download rate to a specified limit from `curl` by using the `--limit-rate` option as follows:

```
$ curl URL --limit-rate 20k
```

In this command `k` (kilobyte) and `m` (megabyte) specify the download rate limit.

Specifying the maximum download size

The maximum download file size for cURL can be specified using the `--max-filesize` option as follows:

```
$ curl URL --max-filesize bytes
```

It will return a non-zero exit code if the file size exceeds. It will return zero if it succeeds.

Authenticating with cURL

HTTP authentication or FTP authentication can be done using cURL with the `-u` argument.

The username and password can be specified using `-u username:password`. It is possible to not provide a password such that it will prompt for password while executing.

If you prefer to be prompted for the password, you can do that by using only `-u username`. For example:

```
$ curl -u user:pass http://test_auth.com
```

In order to be prompted for the password use:

```
$ curl -u user http://test_auth.com
```

Printing response headers excluding data

It is useful to print only response headers to apply many checks or statistics. For example, to check whether a page is reachable or not, we don't need to download the entire page contents. Just reading the HTTP response header can be used to identify if a page is available or not.

An example usage case for checking the HTTP header is to check the file size before downloading. We can check the `Content-Length` parameter in the HTTP header to find out the length of a file before downloading. Also, several useful parameters can be retrieved from the header. The `Last-Modified` parameter enables to know the last modification time for the remote file.

Use the `–I` or `–head` option with `curl` to dump only HTTP headers without downloading the remote file. For example:

```
$ curl -I http://slynux.org
HTTP/1.1 200 OK
Date: Sun, 01 Aug 2010 05:08:09 GMT
Server: Apache/1.3.42 (Unix) mod_gzip/1.3.26.1a mod_log_bytes/1.2
mod_bwlimited/1.4 mod_auth_passthrough/1.8 FrontPage/5.0.2.2635 mod_
ssl/2.8.31 OpenSSL/0.9.7a
Last-Modified: Thu, 19 Jul 2007 09:00:58 GMT
ETag: "17787f3-3bb0-469f284a"
Accept-Ranges: bytes
Content-Length: 15280
Connection: close
Content-Type: text/html
```

See also

- *Posting to a web page and reading response*

Accessing Gmail from the command line

Gmail is a widely-used free e-mail service from Google: `http://mail.google.com/`. Gmail allows you to read your mail via authenticated RSS feeds. We can parse the RSS feeds with the sender's name and an e-mail with subject. It will help to have a look at unread mails in the inbox without opening the web browser.

How to do it...

Let's go through the shell script to parse the RSS feeds for Gmail to display the unread mails:

```
#!/bin/bash
Filename: fetch_gmail.sh
#Description: Fetch gmail tool

username="PUT_USERNAME_HERE"
password="PUT_PASSWORD_HERE"

SHOW_COUNT=5 # No of recent unread mails to be shown

echo

curl  -u $username:$password --silent "https://mail.google.com/mail/
feed/atom" | \
tr -d '\n' | sed 's:</entry>:\n:g' |\
 sed 's/.*<title>\(.*\)<\/title.*<author><name>\([^<]*\)<\/
name><email>
\([^<]*\).*/Author: \2 [\3] \nSubject: \1\n/' | \
head -n $(( $SHOW_COUNT * 3 ))
```

The output will be as follows:

```
$ ./fetch_gmail.sh
Author: SLYNUX [ slynux@slynux.com ]
Subject: Book release - 2

Author: SLYNUX [ slynux@slynux.com ]
Subject: Book release - 1
.
... 5 entries
```

How it works...

The script uses cURL to download the RSS feed by using user authentication. User authentication is provided by the `-u username:password` argument. You can use `-u user` without providing the password. Then while executing cURL it will interactively ask for the password.

Here we can split the piped commands into different blocks to illustrate how they work.

`tr -d '\n'` removes the newline character so that we restructure each mail entry with `\n` as the delimiter. `sed 's:</entry>:\n:g'` replaces every `</entry>` with a newline so that each mail entry is delimited by a newline and hence mails can be parsed one by one. Have a look at the source of `https://mail.google.com/mail/feed/atom` for XML tags used in the RSS feeds. `<entry> TAGS </entry>` corresponds to a single mail entry.

The next block of script is as follows:

```
 sed 's/.*<title>\(.*\)<\/title.*<author><name>\([^<]*\)<\/
name><email>
\([^<]*\).*/Author: \2 [\3] \nSubject: \1\n/'
```

This script matches the substring title using `<title>\(.*\)<\/title`, the sender name using `<author><name>\([^<]*\)<\/name>`, and e-mail using `<email>\([^<]*\)`. Then back referencing is used as follows:

- `Author: \2 [\3] \nSubject: \1\n` is used to replace an entry for a mail with the matched items in an easy-to-read format. `\1` corresponds to the first substring match, `\2` for the second substring match, and so on.
- The `SHOW_COUNT=5` variable is used to take the number of unread mail entries to be printed on terminal.
- `head` is used to display only `SHOW_COUNT*3` lines from the first line. `SHOW_COUNT` is used three times in order to show three lines of the output.

See also

- *A primer on cURL*, explains the curl command
- *Basic sed primer* of *Chapter 4*, explains the sed command

Parsing data from a website

It is often useful to parse data from web pages by eliminating unnecessary details. `sed` and `awk` are the main tools that we will use for this task. You might have come across a list of access rankings in a grep recipe in the previous chapter *Texting and driving*; it was generated by parsing the website page `http://www.johntorres.net/BoxOfficefemaleList.html`.

Let's see how to parse the same data using text-processing tools.

How to do it...

Let's go through the command sequence used to parse details of actresses from the website:

```
$ lynx -dump http://www.johntorres.net/BoxOfficefemaleList.html  | \ grep
-o "Rank-.*" | \
sed 's/Rank-//; s/\[[0-9]\+\]//' | \
sort -nk 1 |\
 awk '
{
 for(i=3;i<=NF;i++){ $2=$2" "$i }
 printf "%-4s %s\n", $1,$2 ;
}' > actresslist.txt
```

The output will be as follows:

```
# Only 3 entries shown. All others omitted due to space limits
1   Keira Knightley
2   Natalie Portman
3   Monica Bellucci
```

How it works...

Lynx is a command-line web browser; it can dump the text version of the website as we would see in a web browser rather than showing us the raw code. Hence it avoids the job of removing the HTML tags. We parse the lines starting with Rank, using `sed` as follows:

```
sed 's/Rank-//; s/\[[0-9]\+\]//'
```

These lines could be then sorted according to the ranks. `awk` is used here to keep the spacing between rank and the name uniform by specifying the width. `%-4s` specifies a four-character width. All the fields except the first field are concatenated to form a single string as `$2`.

See also

- *Basic sed primer* of *Chapter 4*, explains the sed command
- *Basic awk primer* of *Chapter 4*, explains the awk command
- *Downloading a web page as formatted plain text*, explains the lynx command

Image crawler and downloader

Image crawlers are very useful when we need to download all the images that appear in a web page. Instead of going through the HTML sources and picking all the images, we can use a script to parse the image files and download them automatically. Let's see how to do it.

How to do it...

Let's write a Bash script to crawl and download the images from a web page as follows:

```
#!/bin/bash
#Description: Images downloader
#Filename: img_downloader.sh

if [ $# -ne 3 ];
then
  echo "Usage: $0 URL -d DIRECTORY"
  exit -1
fi

for i in {1..4}
do
  case $1 in
  -d) shift; directory=$1; shift ;;
   *) url=${url:-$1}; shift;;
esac
done

mkdir -p $directory;
baseurl=$(echo $url | egrep -o "https?://[a-z.]+")

curl -s $url | egrep -o "<img src=[^>]*>" |
sed 's/<img src=\"\([^"]*\).*/\1/g' > /tmp/$$.list

sed -i "s|^/|$baseurl/|" /tmp/$$.list

cd $directory;

while read filename;
do
  curl -s -O "$filename" --silent

done < /tmp/$$.list
```

An example usage is as follows:

```
$ ./img_downloader.sh http://www.flickr.com/search/?q=linux -d images
```

How it works...

The above image downloader script parses an HTML page, strips out all tags except `<img>`, then parses `src="URL"` from the `<img>` tag and downloads them to the specified directory. This script accepts a web page URL and the destination directory path as command-line arguments. The first part of the script is a tricky way to parse command-line arguments. The `[ $# -ne 3 ]` statement checks whether the total number of arguments to the script is three, else it exits and returns a usage example.

If it is 3 arguments, then parse the URL and the destination directory. In order to do that a tricky hack is used:

```
for i in {1..4}
do
  case $1 in
  -d) shift; directory=$1; shift ;;
   *) url=${url:-$1}; shift;;
esac
done
```

A `for` loop is iterated four times (there is no significance to the number four, it is just to iterate a couple of times to run the case statement).

The `case` statement will evaluate the first argument (`$1`), and matches `-d` or any other string arguments that are checked. We can place the `-d` argument anywhere in the format as follows:

```
$ ./img_downloader.sh -d DIR URL
```

Or:

```
$ ./img_downloader.sh URL -d DIR
```

`shift` is used to shift arguments such that when `shift` is called `$1` will be assigned with `$2`, when again called `$1=$3` and so on as it shifts `$1` to the next arguments. Hence we can evaluate all arguments through `$1` itself.

When `-d` is matched (`-d)`), it is obvious that the next argument is the value for the destination directory. `*)` corresponds to default match. It will match anything other than `-d`. Hence while iteration `$1=""` or `$1=URL` in the default match, we need to take `$1=URL` avoiding `""` to overwrite. Hence we use the `url=${url:-$1}` trick. It will return a URL value if already not `""` else it will assign `$1`.

`egrep -o "<img src=[^>]*>"` will print only the matching strings, which are the `<img>` tags including their attributes. `[^>]*` used to match all characters except the closing `>`, that is, `<img src="image.jpg" …. >`.

`sed 's/<img src=\"\([^"]*\).*/\1/g'` parses `src="url"` so that all image URLs can be parsed from the `<img>` tags already parsed.

There are two types of image source paths: relative and absolute. Absolute paths contain full URLs that start with `http://` or `https://`. Relative URLs starts with `/` or `image_name` itself.

An example of an absolute URL is: `http://example.com/image.jpg`

An example of a relative URL is: `/image.jpg`

For relative URLs the starting `/` should be replaced with the base URL to transform it to `http://example.com/image.jpg`.

For that transformation, we initially find out `baseurl` `sed` by parsing.

Then replace every occurrence of the starting `/` with `baseurl` `sed` as `sed -i "s|^/|$baseurl/|" /tmp/$$.list`.

Then a `while` loop is used to iterate the list line by line and download the URL using `curl`. The `--silent` argument is used with `curl` to avoid other progress messages from being printed on the screen.

See also

- *A primer on cURL*, explains the curl command
- *Basic sed primer* of *Chapter 4*, explains the sed command
- *Searching and mining "text" inside a file with grep* of *Chapter 4*, explains the grep command

Web photo album generator

Web developers commonly design photo album pages for websites that consist of a number of image thumbnails on the page. When thumbnails are clicked, a large version of the picture will be displayed. But when many images are required, copying the `<img>` tag every time, resizing the image to create a thumbnail, placing them in the thumbs directory, testing the links, and so on are real hurdles. It takes a lot of time and repeats the same task. It can be automated easily by writing a simple Bash script. By writing a script, we can create thumbnails, place them in exact directories, and generate the code fragment for `<img>` tags automatically in few seconds. This recipe will teach you how to do it.

Getting ready

We can perform this task with a `for` loop that iterates every image in the current directory. The usual Bash utilities such as `cat` and `convert` (image magick) are used. These will generate an HTML album, using all the images, to `index.html`. In order to use `convert`, make sure you have Imagemagick installed.

How to do it...

Let's write a Bash script to generate a HTML album page:

```
#!/bin/bash
#Filename: generate_album.sh
#Description: Create a photo album using images in current directory

echo "Creating album.."
mkdir -p thumbs
cat <<EOF > index.html
<html>
<head>
<style>

body
{
  width:470px;
  margin:auto;
  border: 1px dashed grey;
  padding:10px;
}

img
{
  margin:5px;
  border: 1px solid black;

}
</style>
</head>
<body>
<center><h1> #Album title </h1></center>
<p>
EOF

for img in *.jpg;
do
  convert "$img" -resize "100x" "thumbs/$img"
  echo "<a href=\"$img\" ><img src=\"thumbs/$img\" title=\"$img\" />
</a>" >> index.html
done

cat <<EOF >> index.html

</p>
</body>
</html>
EOF

echo Album generated to index.html
```

Run the script as follows:

```
$ ./generate_album.sh
Creating album..
Album generated to index.html
```

How it works...

The initial part of the script is to write the header part of the HTML page.

The following script redirects all the contents up to EOF (excluding) to the `index.html`:

```
cat <<EOF > index.html
contents...
EOF
```

The header includes the HTML and stylesheets.

`for img in *.jpg;` will iterate through names of each file and will perform actions.

`convert "$img" -resize "100x" "thumbs/$img"` will create images of 100px width as thumbnails.

The following statement will generate the required `<img>` tag and appends it to the `index.html`:

```
echo "<a href=\"$img\" ><img src=\"thumbs/$img\" title=\"$img\" /></
a>" >> index.html
```

Finally, the footer HTML tags are appended with `cat` again.

See also

- *Playing with file descriptors and redirection* of *Chapter 1*, explains EOF and stdin redirection.

Twitter command-line client

Twitter is the hottest micro blogging platform as well as the latest buzz of online social media. Tweeting and reading tweets is fun. What if we can do both from command line? It is pretty simple to write a command-line Twitter client. Twitter has RSS feeds and hence we can make use of them. Let's see how to do it.

Getting ready

We can use cURL to authenticate and send twitter updates as well as download the RSS feed pages to parse the tweets. Just four lines of code can do it. Let's do it.

How to do it...

Let's write a Bash script using the `curl` command to manipulate twitter APIs:

```
#!/bin/bash
#Filename: tweets.sh
#Description: Basic twitter client

USERNAME="PUT_USERNAME_HERE"
PASSWORD="PUT_PASSWORD_HERE"
COUNT="PUT_NO_OF_TWEETS"

if [[ "$1" != "read" ]] && [[ "$1" != "tweet" ]];
then
  echo -e "Usage: $0 send status_message\n   OR\n      $0 read\n"
  exit -1;
fi

if [[ "$1" = "read" ]];
then
  curl --silent -u $USERNAME:$PASSWORD  http://twitter.com/statuses/
friends_timeline.rss | \
grep title | \
tail -n +2 | \
head -n $COUNT | \
  sed 's:.*<title>\([^<]*\).*:\n\1:'

elif [[ "$1" = "tweet" ]];
then
  status=$( echo $@ | tr -d '"' | sed 's/.*tweet //')
  curl --silent -u $USERNAME:$PASSWORD -d status="$status" http://
twitter.com/statuses/update.xml > /dev/null
  echo 'Tweeted :)'
fi
```

Run the script as follows:

```
$ ./tweets.sh tweet Thinking of writing a X version of wall command
"#bash"
Tweeted :)

$ ./tweets.sh read
bot: A tweet line
t3rm1n4l: Thinking of writing a X version of wall command #bash
```

How it works...

Let's see the working of above script by splitting it into two parts. The first part is about reading tweets. To read tweets the script downloads the RSS information from `http://twitter.com/statuses/friends_timeline.rss` and parses the lines containing the `<title>` tag. Then it strips off the `<title>` and `</title>` tags using `sed` to form the required tweet text. Then a `COUNT` variable is used to remove all other text except the number of recent tweets by using the `head` command. `tail -n +2` is used to remove an unnecessary header text "Twitter: Timeline of friends".

In the sending tweet part, the `-d` status argument of `curl` is used to post data to Twitter using their API: `http://twitter.com/statuses/update.xml`.

`$1` of the script will be the tweet in the case of sending a tweet. Then to obtain the status we take `$@` (list of all arguments of the script) and remove the word "tweet" from it.

See also

- *A primer on cURL*, explains the curl command
- *head and tail - printing the last or first 10 lines* of *Chapter 3*, explains the commands head and tail

define utility with Web backend

Google provides Web definitions for any word by using the search query `define:WORD`. We need a GUI web browser to fetch the definitions. However, we can automate it and parse the required definitions by using a script. Let's see how to do it.

Getting ready

We can use `lynx`, `sed`, `awk`, and `grep` to write the define utility.

How to do it...

Let's go through the code for the define utility script to fetch definitions from Google search:

```
#!/bin/bash
#Filename: define.sh
#Description: A Google define: frontend
limit=0
if  [ ! $# -ge 1 ];
then
  echo -e "Usage: $0 WORD [-n No_of_definitions]\n"
  exit -1;
```

```
fi
if [ "$2" = "-n" ];
then
  limit=$3;
  let limit++
fi
word=$1
lynx -dump http://www.google.co.in/search?q=define:$word | \
awk '/Defini/,/Find defini/' | head -n -1 | sed 's:*:\n*:; s:^[ ]*::'
| \
grep -v "[[0-9]]" | \
awk '{
if ( substr($0,1,1) == "*" )
{ sub("*",++count".") } ;
print
} ' >  /tmp/$$.txt
echo
if [ $limit -ge 1 ];
then
cat /tmp/$$.txt | sed -n "/^1\./, /${limit}/p" | head -n -1
else
cat /tmp/$$.txt;
fi
```

Run the script as follows:

```
$ ./define.sh hack -n 2
1. chop: cut with a hacking tool
2. one who works hard at boring tasks
```

How it works...

We will look into the core part of the definition parser. Lynx is used to obtain the plain text version of the web page. `http://www.google.co.in/search?q=define:$word` is the URL for the web definition web page. Then we reduce the text between "Definitions on web" and "Find definitions". All the definitions are occurring in between these lines of text (`awk '/Defini/,/Find defini/'`).

`'s:*:\n*:'` is used to replace * with * and newline in order to insert a newline in between each definition, and `s:^[ ]*::` is used to remove extra spaces in the start of lines. Hyperlinks are marked as [number] in lynx output. Those lines are removed by `grep -v`, the invert match lines option. Then `awk` is used to replace the * occurring at start of the line with a number so that each definition can assign a serial number. If we have read a `-n` count in the script, it has to output only a few definitions as per count. So `awk` is used to print the definitions with number 1 to count (this makes it easier since we replaced * with the serial number).

See also

- *Basic sed primer* of *Chapter 4*, explains the sed command
- *Basic awk primer* of *Chapter 4*, explains the awk command
- *Searching and mining "text" inside a file with grep* of *Chapter 4*, explains the grep command
- *Downloading a web page as formatted plain text*, explains the lynx command

Finding broken links in a website

I have seen people manually checking each and every page on a site to search for broken links. It is possible only for websites having very few pages. When the number of pages become large, it will become impossible. It becomes really easy if we can automate finding broken links. We can find the broken links by using HTTP manipulation tools. Let's see how to do it.

Getting ready

In order to identify the links and find the broken ones from the links, we can use `lynx` and `curl`. It has an option `-traversal`, which will recursively visit pages in the website and build the list of all hyperlinks in the website. We can use cURL to verify whether each of the links are broken or not.

How to do it...

Let's write a Bash script with the help of the `curl` command to find out the broken links on a web page:

```
#!/bin/bash
#Filename: find_broken.sh
#Description: Find broken links in a website
if [ $# -eq 2 ];
then
  echo -e "$Usage $0 URL\n"
  exit -1;
fi
```

```
echo Broken links:
mkdir /tmp/$$.lynx
cd /tmp/$$.lynx
lynx -traversal $1 > /dev/null
count=0;
sort -u reject.dat > links.txt
while read link;
do
  output=`curl -I $link -s | grep "HTTP/.*OK"`;
  if [[ -z $output ]];
  then
    echo $link;
    let count++
  fi
done < links.txt
[ $count -eq 0 ] && echo No broken links found.
```

How it works...

`lynx -traversal URL` will produce a number of files in the working directory. It includes a file `reject.dat` which will contain all the links in the website. `sort -u` is used to build a list by avoiding duplicates. Then we iterate through each link and check the header response by using `curl -I`. If the header contains first line `HTTP/1.0 200 OK` as the response, it means that the target is not broken. All other responses correspond to broken links and are printed out to `stdout`.

See also

- *Downloading a web page as formatted plain text*, explains the lynx command
- *A primer on cURL*, explains the curl command

Tracking changes to a website

Tracking changes to a website is helpful to web developers and users. Checking a website manually in intervals is really hard and impractical. Hence we can write a change tracker running at repeated intervals. When a change occurs, it can play a sound or send a notification. Let's see how to write a basic tracker for the website changes.

Getting ready

Tracking changes in terms of Bash scripting means fetching websites at different times and taking the difference using the `diff` command. We can use `curl` and `diff` to do this.

How to do it...

Let's write a Bash script by combining different commands to track changes in a web page:

```
#!/bin/bash
#Filename: change_track.sh
#Desc: Script to track changes to webpage
if [ $# -eq 2 ];
then
  echo -e "$Usage $0 URL\n"
  exit -1;
fi

first_time=0
# Not first time
if [ ! -e "last.html" ];
then
  first_time=1
  # Set it is first time run
fi
curl --silent $1 -o recent.html
if [ $first_time -ne 1 ];
then
  changes=$(diff -u last.html recent.html)
  if [ -n "$changes" ];
  then
    echo -e "Changes:\n"
    echo "$changes"
  else
    echo -e "\nWebsite has no changes"
  fi
else
  echo "[First run] Archiving.."
fi

cp recent.html last.html
```

Let's look at the output of the `track_changes.sh` script when changes are made to the web page and when the changes are not made to the page:

- First run:

```
$ ./track_changes.sh http://web.sarathlakshman.info/test.html
[First run] Archiving..
```

- Second Run:

```
$ ./track_changes.sh http://web.sarathlakshman.info/test.html
Website has no changes
```

- Third run after making changes to the web page:

```
$ ./test.sh http://web.sarathlakshman.info/test_change/test.html
Changes:

--- last.html 2010-08-01 07:29:15.000000000 +0200
+++ recent.html     2010-08-01 07:29:43.000000000 +0200
@@ -1,3 +1,4 @@
<html>
+added line :)
<p>data</p>
</html>
```

How it works...

The script checks whether the script is running for the first time using `[ ! -e "last.html" ];`. If `last.html` doesn't exist, that means it is the first time and hence the webpage must be downloaded and copied as `last.html`.

If it is not the first time, it should download the new copy (`recent.html`) and check the difference using the `diff` utility. If changes are there, it should print the changes and finally it should copy `recent.html` to `last.html`.

See also

- *A primer on cURL*, explains the curl command

Posting to a web page and reading response

POST and GET are two types of requests in HTTP to send information to or retrieve information from a website. In a GET request, we send parameters (name-value pairs) through the web page URL itself. In the case of POST, it won't be attached with the URL. POST is used when a form needs to be submitted. For example, a username, the password to be submitted, and the login page to be retrieved.

POSTing to pages comes as frequent use while writing scripts based on web page retrievals. Let's see how to work with POST. Automating the HTTP GET and POST request by sending POST data and retrieving output is a very important task that we practice while writing shell scripts that parse data from websites.

Getting ready

Both cURL and `wget` can handle POST requests by arguments. They are to be passed as name-value pairs.

How to do it...

Let's see how to POST and read HTML response from a real website using `curl`:

```
$ curl URL -d "postvar=postdata2&postvar2=postdata2"
```

We have a website (`http://book.sarathlakshman.com/lsc/mlogs/`) and it is used to submit the current user information such as hostname and username. Assume that, in the home page of the website there are two fields HOSTNAME and USER, and a SUBMIT button. When the user enters a hostname, a user name, and clicks on the SUBMIT button, the details will be stored in the website. This process can be automated using a single line of `curl` command by automating the POST request. If you look at the website source (use the view source option from the web browser), you can see an HTML form defined similar to the following code:

```
<form action="http://book.sarathlakshman.com/lsc/mlogs/submit.php"
method="post" >
<input type="text" name="host" value="HOSTNAME" >
<input type="text" name="user" value="USER" >
<input type="submit" >
</form>
```

Here, `http://book.sarathlakshman.com/lsc/mlogs/submit.php` is the target URL. When the user enters the details and clicks on the Submit button. The host and user inputs are sent to `submit.php` as a POST request and the response page is returned on the browser.

We can automate the POST request as follows:

```
$ curl http://book.sarathlakshman.com/lsc/mlogs/submit.php -d "host=test-
host&user=slynux"
<html>
You have entered :
<p>HOST : test-host</p>
<p>USER : slynux</p>
<html>
```

Now `curl` returns the response page.

`-d` is the argument used for posting. The string argument for `-d` is similar to the GET request semantics. `var=value` pairs are to be delimited by `&`.

The `-d` argument should always be given in quotes. If quotes are not used, `&` is interpreted by the shell to indicate this should be a background process.

There's more

Let's see how to perform POST using cURL and `wget`.

POST in curl

You can POST data in `curl` by using `-d` or `–data` as follows:

```
$ curl --data "name=value" URL -o output.html
```

If multiple variables are to be sent, delimit them with `&`. Note that when `&` is used the name-value pairs should be enclosed in quotes, else the shell will consider `&` as a special character for background process. For example:

```
$ curl -d "name1=val1&name2=val2" URL -o output.html
```

POST data using wget

You can POST data using `wget` by using `--post-data "string"`. For example:

```
$ wget URL -post-data "name=value" -O output.html
```

Use the same format as cURL for name-value pairs.

See also

- *A primer on cURL*, explains the curl command
- *Downloading from a web page* explains the wget command

6
The Backup Plan

In this chapter, we will cover:

- Archiving with tar
- Archiving with cpio
- Compressing with gunzip (gzip)
- Compressing with bunzip (bzip)
- Compressing with lzma
- Archiving and compressing with zip
- Heavy compression squashfs fileystem
- Encrypting files and folders (with standard algorithms)
- Backup snapshots with rsync
- Version controlled backups with git
- Cloning disks with dd

Introduction

Taking snapshots and backups of data are regular tasks we come across. When it comes to a server or large data storage systems, regular backups are important. It is possible to automate backups via shell scripting. Archiving and compression seems to find usage in the everyday life of a system admin or a regular user. There are various compression formats that can be used in various ways so that best results can be obtained. Encryption is another task that comes under frequent usage for protection of data. In order to reduce the size of encrypted data, usually files are archived and compressed before encrypting. Many standard encryption algorithms are available and it can be handled with shell utilities. This chapter walks through different recipes for creating and maintaining files or folder archives, compression formats, and encrypting techniques with shell. Let's go through the recipes.

Archiving with tar

The `tar` command can be used to archive files. It was originally designed for storing data on tape archives (tar). It allows you to store multiple files and directories as a single file. It can retain 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.

Getting ready

The `tar` command comes by default with all UNIX like operating systems. It has a simple syntax and is a portable file format. Let's see how to do it.

`tar` has got a list of arguments: `A`, `c`, `d`, `r`, `t`, `u`, `x`, `f`, and `v`. Each of these letters can be used independently for different purposes corresponding to it.

How to do it...

To archive files with tar, use the following syntax:

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

For example:

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

In this command, `-c` stands for "create file" and `-f` stands for "specify filename".

We can specify folders and filenames as `SOURCES`. We can use a list of file names or wildcards such as `*.txt` to specify the sources.

It will archive the source files into a file called `output.tar`.

The filename must appear immediately after the `-f` and should be the last option in the argument group (for example, `-cvvf filename.tar` and `-tvvf filename.tar`).

We cannot pass hundreds of files or folders as command-line arguments because there is a limit. So it is safer to use the append option if many files are to be archived.

There's more...

Let's go through additional features that are available with the `tar` command.

Appending files to an archive

Sometimes we may need to add files to an archive that already exists (an example usage is when thousands of files are to be archived and when they cannot be specified in one line as command-line arguments).

Append option: `-r`

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

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

List the files in an archive as follows:

```
$ tar -tf archive.tar
yy/lib64/
yy/lib64/libfakeroot/
yy/sbin/
```

In order to print more details while archiving or listing, use the `-v` or the `–vv` flag. These flags are called verbose (`v`), which will enable to print more details on the terminal. For example, by using verbose you could print more details, such as the file permissions, owner group, modification date, and so on.

For example:

```
$ tar -tvvf archive.tar
drwxr-xr-x slynux/slynux     0 2010-08-06 09:31 yy/
drwxr-xr-x slynux/slynux     0 2010-08-06 09:39 yy/usr/
drwxr-xr-x slynux/slynux     0 2010-08-06 09:31 yy/usr/lib64/
```

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 insert image 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.

stdin and stdout with tar

While archiving, we can specify `stdout` as the output file so that another command appearing through a pipe can read it as `stdin` and then do some process or extract the archive.

This is helpful in order to transfer data through a Secure Shell (SSH) connection (while on a network). For example:

```
$ mkdir ~/destination
$ tar -cf - file1 file2 file3 | tar -xvf -  -C ~/destination
```

In the example above, `file1`, `file2`, and `file3` are combined into a tarball and then extracted to `~/destination`. In this command:

- `-f` specifies `stdout` as the file for archiving (when the `-c` option used)
- `-f` specifies `stdin` as the file for extracting (when the `-x` option used)

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
```

Updating files in an archive with timestamp check

The append option appends any given file to the archive. If the same file is inside the archive is given to append, it will append that file and the archive will contain duplicates. We can use the update option `-u` to specify only append files that are newer than the file inside the archive with the same name.

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

This command lists the files in the archive.

In order to append `filea` only if `filea` has newer modification time than `filea` inside `archive.tar`, use:

```
$ tar -uvvf archive.tar filea
```

Nothing happens if the version of `filea` outside the archive and the `filea` inside `archive.tar` have the same timestamp.

Use the `touch` command to modify the file timestamp and then try the `tar` command again:

```
$ tar -uvvf archive.tar filea
-rw-r--r-- slynux/slynux     0 2010-08-14 17:53 filea
```

The file is appended since its timestamp is newer than the one inside the archive.

Comparing files in archive and file system

Sometimes it is useful to know whether a file in the archive and a file with the same filename in the filesystem are the same or contain any differences. The `-d` flag can be used to print the differences:

```
$ tar -df archive.tar filename1 filename2 ...
```

For example:

```
$ tar -df archive.tar afile bfile
afile: Mod time differs
afile: Size differs
```

Deleting files from archive

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

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

Let's see another example:

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

Or, we can also use the following syntax:

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

For example:

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

Compression with 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 significantly decreases the size of the files. Tarballs are often compressed into one of the following formats:

- `file.tar.gz`
- `file.tar.bz2`
- `file.tar.lzma`
- `file.tar.lzo`

Different `tar` flags are used to specify different compression formats.

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

They are explained in the following compression-specific recipes.

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`.

Excluding a set of files from archiving

It is possible to exclude a set of files from archiving by specifying patterns. Use `--exclude [PATTERN]` for excluding files matched by wildcard patterns.

For example, to exclude all `.txt` files from archiving use:

```
$ tar -cf arch.tar * --exclude "*.txt"
```

Note that the pattern should be enclosed in double quotes.

It is also possible to exclude a list of files provided in a list file with the `-X` flag as follows:

```
$ cat list
filea
fileb

$ tar -cf arch.tar * -X list
```

Now it excludes `filea` and `fileb` from archiving.

Excluding version control directories

We usually use tarballs for distributing source code. Most of the source code is maintained using version control systems such as subversion, Git, mercurial, cvs, and so on. Code directories under version control will contain special directories used to manage versions like `.svn` or `.git`. However, these directories aren't needed by the code itself and so should be eliminated from the tarball of the source code.

In order to exclude version control related files and directories while archiving use the `--exclude-vcs` option along with `tar`. For example:

```
$ tar --exclude-vcs -czvvf source_code.tar.gz eye_of_gnome_svn
```

Printing total bytes

It is sometimes useful if we can print total bytes copied to the archive. Print the total bytes copied after archiving by using the `--totals` option as follows:

```
$ tar -cf arc.tar * --exclude "*.txt" --totals
Total bytes written: 20480 (20KiB, 12MiB/s)
```

See also

- *Compressing with gunzip (gzip)*, explains the gzip command
- *Compressing with bunzip (bzip2)*, explains the bzip2 command
- *Compressing with lzma*, explains the lzma command

Archiving with cpio

cpio is another archiving format similar to `tar`. It is used to store files and directories in a file with attributes such as permissions, ownership, and so on. But it is not commonly used as much as `tar`. However, `cpio` seems to be used in RPM package archives, initramfs files for the Linux kernel, and so on. This recipe will give minimal usage examples of `cpio`.

How to do it...

`cpio` takes input filenames through `stdin` and it writes the archive into `stdout`. We have to redirect `stdout` to a file to receive the output `cpio` file as follows:

Create test files:

```
$ touch file1 file2 file3
```

We can archive the test files as follows:

```
$ echo file1 file2 file3 | cpio -ov > archive.cpio
```

In this command:

- `-o` specifies the output
- `-v` is used for printing a list of files archived

By using `cpio`, we can also archive using files as absolute paths. `/usr/somedir` is an absolute path as it contains the full path starting from root (`/`).

A relative path will not start with `/` but it starts the path from the current directory. For example, `test/file` means that there is a directory `test` and the `file` is inside the `test` directory.

While extracting, `cpio` extracts to the absolute path itself. But incase of `tar` it removes the `/` in the absolute path and converts it as relative path.

In order to list files in a `cpio` archive use the following command:

```
$ cpio -it < archive.cpio
```

This command will list all the files in the given `cpio` archive. It reads the files from `stdin`.
In this command:

- `-i` is for specifying the input
- `-t` is for listing

In order to extract files from the `cpio` archive use:

```
$ cpio -id < archive.cpio
```

Here, `-d` is used for extracting.

It overwrites files without prompting. If the absolute path files are present in the archive, it will replace the files at that path. It will not extract files in the current directory like `tar`.

Compressing with gunzip (gzip)

gzip is a commonly used compression format in GNU/Linux platforms. Utilities such as `gzip`, `gunzip`, and `zcat` are available to handle gzip compression file types. `gzip` can be applied on a file only. It cannot archive directories and multiple files. Hence we use a `tar` archive and compress it with `gzip`. When multiple files are given as input it will produce several individually compressed (`.gz`) files. Let's see how to operate with `gzip`.

How to do it...

In order to compress a file with `gzip` use the following command:

```
$ gzip filename
```

```
$ ls
filename.gz
```

Then it will remove the file and produce a compressed file called `filename.gz`.

Extract a `gzip` compressed file as follows:

```
$ gunzip filename.gz
```

It will remove `filename.gz` and produce an uncompressed version of `filename.gz`.

In order to list out the properties of a compressed file use:

```
$ gzip -l test.txt.gz
compressed        uncompressed  ratio uncompressed_name
  35                     6        -33.3% test.txt
```

The `gzip` command can read a file from `stdin` and also write a compressed file into `stdout`.

Read from `stdin` and out as `stdout` as follows:

```
$ cat file | gzip -c > file.gz
```

The `-c` option is used to specify output to `stdout`.

We can specify the compression level for `gzip`. Use `--fast` or the `--best` option to provide low and high compression ratios, respectively.

There's more...

The `gzip` command is often used with other commands. It also has advanced options to specify the compression ratio. Let's see how to work with these features.

Gzip with tarball

We usually use `gzip` with tarballs. A tarball can be compressed by using the `–z` option passed to the `tar` command while archiving and extracting.

You can create gzipped tarballs using the following methods:

- **Method - 1**

  ```
  $ 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.

- **Method - 2**

  First, create a tarball:

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

  Compress it after tarballing as follows:

  ```
  $ gzip archive.tar
  ```

If many files (a few hundreds) are to be archived in a tarball and need to be compressed, we use Method - 2 with few changes. The issue with giving many files as command arguments to `tar` is that it can accept only a limited number of files from the command line. In order to solve this issue, we can create a `tar` file by adding files one by one using a loop with an append option (`-r`) as follows:

```
FILE_LIST="file1  file2  file3  file4  file5"
for f in $FILE_LIST;
do
tar -rvf archive.tar $f
done
gzip archive.tar
```

In order to extract a gzipped tarball, use the following:

- `-x` for extraction
- `-z` for gzip specification

Or:

```
$ tar -xavvf archive.tar.gz -C extract_directory
```

In the above command, the `-a` option is used to detect the compression format automatically.

zcat – reading gzipped files without extracting

`zcat` is a command that can be used to dump an extracted file from a `.gz` file to `stdout` without manually extracting it. The `.gz` file remains as before but it will dump the extracted file into `stdout` as follows:

```
$ ls
test.gz

$ zcat test.gz
A test file
# file test contains a line "A test file"

$ ls
test.gz
```

Compression ratio

We can specify compression ratio, which is available in range 1 to 9, where:

- 1 is the lowest, but fastest
- 9 is the best, but slowest

You can also specify the ratios in between as follows:

```
$ gzip -9 test.img
```

This will compress the file to the maximum.

See also

- *Archiving with tar*, explains the tar command

Compressing with bunzip (bzip)

bunzip2 is another compression technique which is very similar to `gzip`. `bzip2` typically produces smaller (more compressed) files than `gzip`. It comes with all Linux distributions. Let's see how to use `bzip2`.

How to do it...

In order to compress with `bzip2` use:

```
$ bzip2 filename
$ ls
filename.bz2
```

Then it will remove the file and produce a compressed file called `filename.bzip2`.

Extract a bzipped file as follows:

```
$ bunzip2 filename.bz2
```

It will remove `filename.bz2` and produce an uncompressed version of `filename`.

`bzip2` can read a file from `stdin` and also write a compressed file into `stdout`.

In order to read from `stdin` and read out as `stdout` use:

```
$ cat file | bzip2 -c > file.tar.bz2
```

`-c` is used to specify output to `stdout`.

We usually use `bzip2` with tarballs. A tarball can be compressed by using the `-j` option passed to the `tar` command while archiving and extracting.

Creating a bzipped tarball can be done by using the following methods:

- **Method - 1**

  ```
  $ tar -cjvvf archive.tar.bz2 [FILES]
  ```

  Or:

  ```
  $ tar -cavvf archive.tar.bz2 [FILES]
  ```

  The `-a` option specifies to automatically detect compression format from the extension.

- **Method - 2**

  First create the tarball:

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

  Compress it after tarballing:

  ```
  $ bzip2 archive.tar
  ```

If we need to add hundreds of files to the archive, the above commands may fail. To fix that issue, use a loop to append files to the archive one by one using the `–r` option. See the similar section from the recipe, *Compressing with gunzip (gzip)*.

Extract a bzipped tarball as follows:

```
$ tar -xjvvf archive.tar.bz2 -C extract_directory
```

In this command:

- `-x` is used for extraction
- `-j` is for `bzip2` specification
- `-C` is for specifying the directory to which the files are to be extracted

Or, you can use the following command:

```
$ tar -xavvf archive.tar.bz2 -C extract_directory
```

`-a` will automatically detect the compression format.

There's more...

bunzip has several additional options to carry out different functions. Let's go through few of them.

Keeping input files without removing them

While using `bzip2` or `bunzip2`, it will remove the input file and produce a compressed output file. But we can prevent it from removing input files by using the `–k` option.

For example:

```
$ bunzip2 test.bz2 -k
$ ls
test test.bz2
```

Compression ratio

We can specify the compression ratio, which is available in the range of 1 to 9 (where 1 is the least compression, but fast, and 9 is the highest possible compression but much slower).

For example:

```
$ bzip2 -9 test.img
```

This command provides maximum compression.

See also

- *Archiving with tar*, explains the tar command

Compressing with lzma

lzma is comparatively new when compared to `gzip` or `bzip2`. `lzma` offers better compression rates than `gzip` or `bzip2`. As `lzma` is not preinstalled on most Linux distros, you may need to install it using the package manager.

How to do it...

In order to compress with `lzma` use the following command:

```
$ lzma filename
$ ls
filename.lzma
```

This will remove the file and produce a compressed file called `filename.lzma`.

To extract an `lzma` file use:

```
$ unlzma filename.lzma
```

This will remove `filename.lzma` and produce an uncompressed version of the file.

The `lzma` command can also read a file from `stdin` and write the compressed file to `stdout`.

In order to read from `stdin` and read out as `stdout` use:

```
$ cat file | lzma -c > file.lzma
```

`-c` is used to specify output to `stdout`.

We usually use `lzma` with tarballs. A tarball can be compressed by using the `--lzma` option passed to the `tar` command while archiving and extracting.

There are two methods to create a `lzma` tarball:

- **Method - 1**

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

  Or:

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

  The `-a` option specifies to automatically detect the compression format from the extension.

- **Method - 2**

  First, create the tarball:

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

  Compress it after tarballing:

  ```
  $ lzma archive.tar
  ```

If we need to add hundreds of files to the archive, the above commands may fail. To fix that issue, use a loop to append files to the archive one by one using the `–r` option. See the similar section from the recipe, *Compressing with gunzip (gzip)*.

There's more...

Let's go through additional options associated with `lzma` utilities

Extracting an lzma tarball

In order to extract a tarball compressed with `lzma` compression to a specified directory, use:

```
$ tar -xvvf --lzma archive.tar.lzma -C extract_directory
```

In this command, `-x` is used for extraction. `--lzma` specifies the use of `lzma` to decompress the resulting file.

Or, we could also use:

```
$ tar -xavvf archive.tar.lzma -C extract_directory
```

The `-a` option specifies to automatically detect the compression format from the extension.

Keeping input files without removing them

While using `lzma` or `unlzma`, it will remove the input file and produce an output file. But we can prevent from removing input files and keep them by using the `-k` option. For example:

```
$ lzma test.bz2 -k
$ ls
test.bz2.lzma
```

Compression ratio

We can specify the compression ratio, which is available in the range of 1 to 9 (where 1 is the least compression, but fast, and 9 is the highest possible compression but much slower).

You can also specify ratios in between as follows:

```
$ lzma -9 test.img
```

This command compresses the file to the maximum.

See also

- *Archiving with tar*, explains the tar command

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.

How to do it...

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.

Archive directories and files recursively as follows:

```
$ zip -r archive.zip folder1 file2
```

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

Unlike `lzma`, `gzip`, or `bzip2`, `zip` won't remove the source file after archiving. `zip` is similar to `tar` in that respect, but `zip` can compress files where `tar` does not. However, `zip` adds compression too.

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
```

squashfs – the heavy compression filesystem

squashfs is a heavy-compression based read-only filesystem that is capable of compressing 2 to 3GB of data onto a 700 MB file. Have you ever thought of how Linux Live CDs work? When a Live CD is booted it loads a complete Linux environment. Linux Live CDs make use of a read-only compressed filesystem called squashfs. It keeps the root filesystem on a compressed filesystem file. It can be loopback mounted and files can be accessed. Thus when some files are required by processes, they are decompressed and loaded onto the RAM and used. Knowledge of squashfs can be useful when building a custom live OS or when required to keep files heavily compressed and to access them without entirely extracting the files. For extracting a large compressed file, it will take a long time. However, if a file is loopback mounted, it will be very fast since the required portion of the compressed files are only decompressed when the request for files appear. In regular decompression, all the data is decompressed first. Let's see how we can use squashfs.

Getting ready

If you have an Ubuntu CD just locate a `.squashfs` file at `CDRom ROOT/casper/filesystem.squashfs`. `squashfs` internally uses compression algorithms such as `gzip` and `lzma`. `squashfs` support is available in all of the latest Linux distros. However, in order to create `squashfs` files, an additional package **squashfs-tools** needs to be installed from package manager.

How to do it...

In order to create a `squashfs` file by adding source directories and files, use:

```
$ mksquashfs SOURCES compressedfs.squashfs
```

Sources can be wildcards, or file, or folder paths.

For example:

```
$ sudo mksquashfs /etc test.squashfs
Parallel mksquashfs: Using 2 processors
Creating 4.0 filesystem on test.squashfs, block size 131072.
[=======================================] 1867/1867 100%

More details will be printed on terminal. They are limited to save space
```

In order to mount the `squashfs` file to a mount point, use loopback mounting as follows:

```
# mkdir /mnt/squash
# mount -o loop compressedfs.squashfs /mnt/squash
```

You can copy contents by accessing `/mnt/squashfs`.

There's more...

The `squashfs` file system can be created by specifying additional parameters. Let's go through the additional options.

Excluding files while creating a squashfs file

While creating a `squashfs` file, we can exclude a list of files or a file pattern specified using wildcards.

Exclude a list of files specified as command-line arguments by using the `-e` option. For example:

```
$ sudo mksquashfs /etc test.squashfs -e /etc/passwd /etc/shadow
```

The `–e` option is used to exclude `passwd` and `shadow` files.

It is also possible to specify a list of exclude files given in a file with `–ef` as follows:

```
$ cat excludelist
/etc/passwd
/etc/shadow

$ sudo mksquashfs /etc test.squashfs -ef excludelist
```

If we want to support wildcards in excludes lists, use `-wildcard` as an argument.

Cryptographic tools and hashes

Encryption techniques are used mainly to protect data from unauthorized access. There are many algorithms available and we use a common set of standard algorithms. There are a few tools available in a Linux environment for performing encryption and decryption. Sometimes we use encryption algorithm hashes for verifying data integrity. This section will introduce a few commonly-used cryptographic tools and a general set of algorithms that these tools can handle.

How to do it...

Let's see how to use the tools such as crypt, gpg, base64, md5sum, sha1sum, and openssl:

- **crypt**

  The `crypt` command is a simple cryptographic utility, which takes a file from `stdin` and a passphrase as input and outputs encrypted data into `stdout`.

  ```
  $ crypt <input_file> output_file
  Enter passphrase:
  ```

  It will interactively ask for a passphrase. We can also provide a passphrase through command-line arguments.

  ```
  $ crypt PASSPHRASE < input_file > encrypted_file
  ```

  In order to decrypt the file use:

  ```
  $ crypt PASSPHRASE -d < encrypted_file > output_file
  ```

- **gpg (GNU privacy guard)**

  gpg (GNU privacy guard) is a widely-used encryption scheme used for protecting files with key signing techniques that enables to access data by authentic destination only. gpg signatures are very famous. The details of gpg are outside the scope of this book. Here we can learn how to encrypt and decrypt a file.

  In order to encrypt a file with gpg use:

  ```
  $ gpg -c filename
  ```

  This command reads the passphrase interactively and generates `filename.gpg`.

  In order to decrypt a gpg file use:

  ```
  $ gpg filename.gpg
  ```

  This command reads a passphrase and decrypts the file.

- **Base64**

  Base64 is a group of similar encoding schemes that represents binary data in an ASCII string format by translating it into a radix-64 representation. The `base64` command can be used to encode and decode the Base64 string.

In order to encode a binary file into Base64 format, use:

```
$ base64 filename > outputfile
```

Or:

```
$ cat file | base64 > outputfile
```

It can read from `stdin`.

Decode Base64 data as follows:

```
$ base64 -d file > outputfile
```

Or:

```
$ cat base64_file | base64 -d > outputfile
```

- **md5sum** and **sha1sum**

  md5sum and **sha1sum** are unidirectional hash algorithms, which cannot be reversed to form the original data. These are usually used to verify the integrity of data or for generating a unique key from a given data. For every file it generates a unique key by analyzing its content.

```
$ md5sum file
8503063d5488c3080d4800ff50850dc9  file

$ sha1sum file
1ba02b66e2e557fede8f61b7df282cd0a27b816b  file
```

  These types of hashes are ideal for storing passwords. Passwords are stored as its hashes. When a user wants to authenticate, the password is read and converted to the hash. Then hash is compared to the one that is stored already. If they are same, the password is authenticated and access is provided, else it is denied. Storing original password strings is risky and poses a security risk of exposing the password.

- **Shadowlike hash (salted hash)**

  Let's see how to generate shadow like salted hash for passwords.

  The user passwords in Linux are stored as its hashes in the `/etc/shadow` file. A typical line in `/etc/shadow` will look like this:

```
test:$6$fG4eWdUi$ohTKOlEUzNk77.4S8MrYe07NTRV4M3LrJnZP9p.qc1bR5c.
EcOruzPXfEu1uloBFUa18ENRH7F70zhodas3cR.:14790:0:99999:7:::
```

  In this line `$6$fG4eWdUi$ohTKOlEUzNk77.4S8MrYe07NTRV4M3LrJnZP9p.qc1bR5c.EcOruzPXfEu1uloBFUa18ENRH7F70zhodas3cR` is the shadow hash corresponding to its password.

  In some situations, we may need to write critical administration scripts that may need to edit passwords or add users manually using a shell script. In that case we have to generate a shadow password string and write a similar line as above to the shadow file. Let's see how to generate a shadow password using `openssl`.

Shadow passwords are usually salted passwords. `SALT` is an extra string used to obfuscate and make the encryption stronger. The salt consists of random bits that are used as one of the inputs to a key derivation function that generates the salted hash for the password.

For more details on salt, see the Wikipedia page `http://en.wikipedia.org/wiki/Salt_(cryptography)`.

```
$ openssl passwd -1 -salt SALT_STRING PASSWORD
$1$SALT_STRING$323VkWkSLHuhbt1zkSsUG.
```

Replace `SALT_STRING` with a random string and `PASSWORD` with the password you want to use.

Backup snapshots with rsync

Backing up data is something that most sysadmins need to do regularly. We may need to backup data in a web server or from remote locations. `rsync` is a command that can be used to synchronize files and directories from one location to another while minimizing data transfer using file difference calculations and compression. The advantage of `rsync` over the `cp` command is that `rsync` uses strong difference algorithms. Also, it supports data transfer across networks. While making copies, it compares the files in the original and destination locations and will only copy the files that are newer. It also supports compression, encryption, and a lot more. Let's see how we can work with `rsync`.

How to do it...

In order to copy a source directory to a destination (to create a mirror) use:

```
$ rsync -av source_path destination_path
```

In this command:

- `-a` stands for archiving
- `-v` (verbose) prints the details or progress on `stdout`

The above command will recursively copy all the files from the source path to the destination path. We can specify paths as remote or localhost paths.

It can be in the format `/home/slynux/data`, `slynux@192.168.0.6:/home/backups/data`, and so on.

`/home/slynux/data` specifies the absolute path in the machine in which the `rsync` command is executed. `slynux@192.168.0.6:/home/backups/data` specifies that the path is `/home/backups/data` in the machine with IP address `192.168.0.6` and is logged in as user `slynux`.

In order to back up data to a remote server or host, use:

```
$ rsync -av source_dir username@host:PATH
```

To keep a mirror at the destination, run the same `rsync` command scheduled at regular intervals. It will copy only changed files to the destination.

Restore the data from remote host to `localhost` as follows:

```
$ rsync -av username@host:PATH destination
```

The `rsync` command uses SSH to connect to another remote machine. Provide the remote machine address in the format `user@host`, where user is the username and host is the IP address or domain name attached to the remote machine. `PATH` is the absolute path address where the data needs to be copied. `rsync` will ask for the user password as usual for SSH logic. This can be automated (avoid user password probing) by using SSH keys.

Make sure that the OpenSSH is installed and running on the remote machine.

Compressing data while transferring through the network can significantly optimize the speed of the transfer. We can use the `rsync` option `-z` to specify to compress data while transferring through a network. For example:

```
$ rsync -avz source destination
```

For the PATH format, if we use / at the end of the source, `rsync` will copy contents of that end directory specified in the `source_path` to the destination.

If / not at the end of the source, `rsync` will copy that end directory itself to the destination.

For example, the following command copies the content of the `test` directory:

```
$ rsync -av /home/test/ /home/backups
```

The following command copies the `test` directory to the destination:

```
$ rsync -av /home/test /home/backups
```

If / is at the end of `destination_path`, `rsync` will copy the source to the destination directory.

If / is not used at the end of the destination path, `rsync` will create a folder, named similar to the source directory, at the end of the destination path and copy the source into that directory.

For example:

```
$ rsync -av /home/test /home/backups/
```

This command copies the source (`/home/test`) to an existing folder called `backups`.

```
$ rsync -av /home/test /home/backups
```

This command copies the source (`/home/test`) to a directory named `backups` by creating that directory.

There's more...

The `rsync` command has several additional functionalities that can be specified using its command-line options. Let's go through them.

Excluding files while archiving with rsync

Some files need not be updated while archiving to a remote location. It is possible to tell `rsync` to exclude certain files from the current operation. Files can be excluded by two options:

```
--exclude PATTERN
```

We can specify a wildcard pattern of files to be excluded. For example:

```
$ rsync -avz /home/code/some_code /mnt/disk/backup/code --exclude "*.txt"
```

This command excludes `.txt` files from backing up.

Or, we can specify a list of files to be excluded by providing a list file.

Use `--exclude-from FILEPATH`.

Deleting non-existent files while updating rsync backup

We archive files as tarball and transfer the tarball to the remote backup location. When we need to update the backup data, we create a TAR file again and transfer the file to the backup location. By default, `rsync` does not remove files from the destination if they no longer exist at the source. In order to remove the files from the destination that do not exist at the source, use the `rsync --delete` option:

```
$ rsync -avz SOURCE DESTINATION --delete
```

Scheduling backups at intervals

You can create a cron job to schedule backups at regular intervals.

A sample is as follows:

```
$ crontab -e
```

Add the following line:

```
0 */10 * * * rsync -avz /home/code user@IP_ADDRESS:/home/backups
```

The above `crontab` entry schedules the `rsync` to be executed every 10 hours.

`*/10` is the hour position of the `crontab` syntax. `/10` specifies to execute the backup every 10 hours. If `*/10` is written in the minutes position, it will execute every 10 minutes.

Have a look at the *Scheduling with cron* recipe in *Chapter 9* to understand how to configure `crontab`.

Version control based backup with Git

People use different strategies in backing up data. Differential backups are more efficient than making copies of the entire source directory to a target the backup directory with the version number using date or time of a day. It causes wastage of space. We only need to copy the changes that occurred to files from the second time that the backups occur. This is called incremental backups. We can manually create incremental backups using tools like `rsync`. But restoring this sort of backup can be difficult. The best way to maintain and restore changes is to use version control systems. They are very much used in software development and maintenance of code, since coding frequently undergoes changes. Git (GNU it) is a very famous and is the most efficient version control systems available. Let's use Git for backup of regular files in non-programming context. Git can be installed by your distro's package manager. It was written by Linus Torvalds.

Getting ready

Here is the problem statement:

We have a directory that contains several files and subdirectories. We need to keep track of changes occurring to the directory contents and back them up. If data becomes corrupted or goes missing, we must be able to restore a previous copy of that data. We need to backup the data at regular intervals to a remote machine. We also need to take the backup at different locations in the same machine (localhost). Let's see how to implement it using Git.

How to do it...

In the directory which is to be backed up use:

```
$ cd /home/data/source
```

Let it be the directory source to be tracked.

Set up and initiate the remote backup directory. In the remote machine, create the backup destination directory:

```
$ mkdir -p /home/backups/backup.git

$ cd /home/backups/backup.git

$ git init --bare
```

The following steps are to be performed in the source host machine:

1. Add user details to Git in the source host machine:

```
$ git config --global user.name  "Sarath Lakshman"
#Set user name to "Sarath Lakshman"

$ git config --global user.email slynux@slynux.com
# Set email to slynux@slynux.com
```

   Initiate the source directory to backup from the host machine. In the source directory in the host machine whose files are to be backed up, execute the following commands:

```
$ git init
Initialized empty Git repository in /home/backups/backup.git/
# Initialize git repository

$ git commit --allow-empty -am "Init"
[master (root-commit) b595488] Init
```

2. In the source directory, execute the following command to add the remote git directory and synchronize backup:

```
$ git remote add origin user@remotehost:/home/backups/backup.git

$ git push origin master
Counting objects: 2, done.
Writing objects: 100% (2/2), 153 bytes, done.
Total 2 (delta 0), reused 0 (delta 0)
To user@remotehost:/home/backups/backup.git
 * [new branch]      master -> master
```

3. Add or remove files for Git tracking.

   The following command adds all files and folders in the current directory to the backup list:

```
$ git add *
```

   We can conditionally add certain files only to the backup list as follows:

```
$ git add *.txt
$ git add *.py
```

   We can remove the files and folders not required to be tracked by using:

```
$ git rm file
```

   It can be a folder or even a wildcard as follows:

```
$ git rm *.txt
```

4. Check-pointing or marking backup points.

   We can mark checkpoints for the backup with a message using the following command:

   ```
   $ git commit -m "Commit Message"
   ```

   We need to update the backup at the remote location at regular intervals. Hence, set up a cron job (for example, backing up every five hours).

   Create a file crontab entry with lines:

   ```
   0 */5 * * *  /home/data/backup.sh
   ```

   Create a script `/home/data/backup.sh` as follows:

   ```
   #!/bin/ bash
   cd /home/data/source
   git add .
   git commit -am "Commit - @ $(date)"
   git push
   ```

   Now we have set up the backup system.

5. Restoring data with Git.

   In order to view all backup versions use:

   ```
   $ git log
   ```

   Update the current directory to the last backup by ignoring any recent changes.

   - To revert back to any previous state or version, look into the commit ID, which is a 32-character hex string. Use the commit ID with `git checkout`.
   - For commit ID 3131f9661ec1739f72c213ec5769bc0abefa85a9 it will be:

     ```
     $ git checkout 3131f9661ec1739f72c213ec5769bc0abefa85a9

     $ git commit -am "Restore @ $(date) commit ID:
     3131f9661ec1739f72c213ec5769bc0abefa85a9"
     $ git push
     ```

   - In order to view the details about versions again, use:

     ```
     $ git log
     ```

   If the working directory is broken due to some issues, we need to fix the directory with the backup at the remote location.

   Then we can recreate the contents from the backup at the remote location as follows:

   ```
   $ git clone user@remotehost:/home/backups/backup.git
   ```

   This will create a directory backup with all contents.

Cloning hard drive and disks with dd

While working with hard drives and partitions, we may need to create copies or make backups of full partitions rather than copying all contents (not only hard disk partitions but also copy an entire hard disk without missing any information, such as boot record, partition table, and so on). In this situation we can use the `dd` command. It can be used to clone any type of disks, such as hard disks, flash drives, CDs, DVDs, floppy disks, and so on.

Getting ready

The `dd` command expands to Data Definition. Since its improper usage leads to loss of data, it is nicknamed as "Data Destroyer". Be careful while using the order of arguments. Wrong arguments can lead to loss of entire data or can become useless. `dd` is basically a bitstream duplicator that writes the entire bit stream from a disk to a file or a file to a disk. Let's see how to use `dd`.

How to do it...

The syntax for `dd` is as follows:

```
$ dd if=SOURCE of=TARGET bs=BLOCK_SIZE count=COUNT
```

In this command:

- `if` stands for input file or input device path
- `of` stands for target file or target device path
- `bs` stands for block size (usually, it is given in the power of 2, for example, 512, 1024, 2048, and so on). `COUNT` is the number of blocks to be copied (an integer).

  Total bytes copied = BLOCK_SIZE * COUNT

  `bs` and `count` are optional.

By specifying `COUNT` we can limit the number of bytes to be copied from input file to target. If `COUNT` is not specified, `dd` will copy from input file until it reaches the end of file (EOF) marker.

In order to copy a partition into a file use:

```
# dd if=/dev/sda1 of=sda1_partition.img
```

Here `/dev/sda1` is the device path for the partition.

Restore the partition using the backup as follows:

```
# dd if=sda1_partition.img of=/dev/sda1
```

You should be careful about the argument `if` and `of`. Improper usage may lead to data loss.

By changing the device path `/dev/sda1` to the appropriate device path, any disk can be copied or restored.

In order to permanently delete all of the data in a partition, we can make `dd` to write zeros into the partition by using the following command:

```
# dd if=/dev/zero of=/dev/sda1
```

`/dev/zero` is a character device. It always returns infinite zero '\0' characters.

Clone one hard disk to another hard disk of the same size as follows:

```
# dd if=/dev/sda  of=/dev/sdb
```

Here `/dev/sdb` is the second hard disk.

In order to take the image of a CD ROM (ISO file) use:

```
# dd if=/dev/cdrom of=cdrom.iso
```

There's more...

When a file system is created in a file which is generated using `dd`, we can mount it to a mount point. Let's see how to work with it.

Mounting image files

Any file image created using `dd` can be mounted using the loopback method. Use the `-o loop` with the `mount` command.

```
# mkdir /mnt/mount_point
# mount -o loop file.img /mnt/mount_point
```

Now we can access the contents of the image files through the location `/mnt/mount_point`.

See also

- *Creating ISO files, Hybrid ISO* of *Chapter 3*, explains how to use dd to create an ISO file from a CD

7
The Old-boy Network

In this chapter, we will cover:

- Basic networking primer
- Let's ping!
- Listing all the machines alive on a network
- Transferring files through network
- Setting up an Ethernet and wireless LAN with script
- Password-less auto-login with SSH
- Running commands on remote host with SSH
- Mounting remote drive at local mount point
- Multi-casting window messages on a network
- Network traffic and port analysis

Introduction

Networking is the act of interconnecting machines through a network and configuring the nodes in the network with different specifications. We use TCP/IP as our networking stack and all operations are based on it. Networks are an important part of every computer system. Each node connected in the network is assigned a unique IP address for identification. There are many parameters in networking, such as subnet mask, route, ports, DNS, and so on, which require a basic understanding to follow.

Several applications that make use of a network operate by opening and connecting to firewall ports. Every application may offer services such as data transfer, remote shell login, and so on. Several interesting management tasks can be performed on a network consisting of many machines. Shell scripts can be used to configure the nodes in a network, test the availability of machines, automate execution of commands at remote hosts, and so on. This chapter focuses on different recipes that introduce interesting tools or commands related to networking and also how they can be used for solving different problems.

Basic networking primer

Before digging through recipes based on networking, it is essential for you to have a basic understanding of setting up a network, the terminology and commands for assigning an IP address, adding routes, and so on. This recipe will give an overview of different commands used in GNU/Linux for networking and their usages from the basics.

Getting ready

Every node in a network requires many parameters to be assigned to work successfully and interconnect with other machines. Some of the different parameters are the IP address, subnet mask, gateway, route, DNS, and so on.

This recipe will introduce commands `ifconfig`, `route`, `nslookup`, and `host`.

How to do it...

Network interfaces are used to connect to a network. Usually, in the context of UNIX-like Operating Systems, network interfaces follow the eth0, eth1 naming convention. Also, other interfaces, such as usb0, wlan0, and so on, are available for USB network interfaces, wireless LAN, and other such networks.

`ifconfig` is the command that is used to display details about network interfaces, subnet mask, and so on.

`ifconfig` is available at `/sbin/ifconfig`. Some GNU/Linux distributions will display an error "command not found" when `ifconfig` is typed. This is because `/sbin` in not included in the user's PATH environment variable. When a command is typed, the Bash looks in the directories specified in PATH variable.

By default, in Debian, `ifconfig` is not available since `/sbin` is not in PATH.

`/sbin/ifconfig` is the absolute path, so try `run ifconfig` with the absolute path (that is, `/sbin/ifconfig`). For every system, there will be a by default interface 'lo' called loopback that points to the current machine. For example:

```
$ ifconfig
lo        Link encap:Local Loopback
```

```
inet addr:127.0.0.1  Mask:255.0.0.0
inet6addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:6078 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6078 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
          RX bytes:634520 (634.5 KB)  TX bytes:634520 (634.5 KB)

wlan0     Link encap:EthernetHWaddr 00:1c:bf:87:25:d2
inet addr:192.168.0.82  Bcast:192.168.3.255  Mask:255.255.252.0
inet6addr: fe80::21c:bfff:fe87:25d2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:420917 errors:0 dropped:0 overruns:0 frame:0
          TX packets:86820 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
          RX bytes:98027420 (98.0 MB)  TX bytes:22602672 (22.6 MB)
```

The left-most column in the `ifconfig` output lists the name of network interfaces and the right-hand columns show the details related to the corresponding network interface.

There's more...

There are several additional commands that frequently come under usage for querying and configuring the network. Let's go through the essential commands and usage.

Printing the list of network interfaces

Here is a one-liner command sequence to print the list of network interface available on a system.

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

The first 10 characters of each line in the `ifconfig` output is reserved for writing the name of the network interface. 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.

Assigning and displaying IP addresses

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

```
$ ifconfig iface_name
```

For example:

```
$ ifconfig wlan0
wlan0     Link encap:Ethernet HWaddr 00:1c:bf:87:25:d2
inet addr:192.168.0.82  Bcast:192.168.3.255
    Mask:255.255.252.0
```

From the outputs of the previously mentioned command, our interests lie in the IP address, broadcast address, hardware address, and subnet mask. They are as follows:

- `HWaddr 00:1c:bf:87:25:d2` is the hardware address (MAC address)
- `inet addr:192.168.0.82` is the IP address
- `Bcast:192.168.3.255` is the broadcast address
- `Mask:255.255.252.0` is the subnet mask

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 common 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 '.'.

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

```
# ifconfig wlan0 192.168.0.80
```

You will need to run the above command as root. `192.168.0.80` is the address to be set.

Set the subnet mask along with IP address as follows:

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

Spoofing Hardware Address (MAC Address)

In certain circumstances where authentication or filtering of computers on a network is provided by using the hardware address, we can use hardware address spoofing. The hardware address appears in `ifconfig` output as `HWaddr 00:1c:bf:87:25:d2`.

We can spoof the hardware address at the software level as follows:

```
# ifconfig eth0 hw ether 00:1c:bf:87:25:d5
```

In the above command, `00:1c:bf:87:25:d5` is the new MAC address to be assigned.

This can be useful when we need to access the Internet through MAC authenticated service providers that provide access to the Internet for a single machine.

Name server and DNS (Domain Name Service)

The elementary addressing scheme for the Internet is IP addresses (dotted decimal form, for example, `202.11.32.75`). However, the resources on the Internet (for example, websites) are accessed through a combination of ASCII characters called URLs or domain names. For example, `google.com` is a domain name. It actually corresponds to an IP address. Typing the IP address in the browser can also access the URL `www.google.com`.

This technique of abstracting IP addresses with symbolic names is called **Domain Name Service** (**DNS**). When we enter `google.com`, the DNS servers configured with our network resolve the domain name into the corresponding IP address. While on a local network, we setup the local DNS for naming local machines on the network symbolically using their hostnames.

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
```

How can we obtain the IP address for a corresponding domain name?

The easiest method to obtain an IP address is by trying to ping the given domain name and looking at the echo reply. For example:

```
$ ping google.com
PING google.com (64.233.181.106) 56(84) bytes of data.
Here 64.233.181.106 is the corresponding IP address.
```

A domain name can have multiple IP addresses assigned. In that case, the DNS server will return one address among the list of IP addresses. To obtain all the addresses assigned to the domain name, we should use a DNS lookup utility.

DNS lookup

There are different DNS lookup utilities available from the command line. These will request a DNS server for an IP address resolution. `host` and `nslookup` are two DNS lookup utilities.

When `host` is executed it will list out all of the IP addressed attached to the domain name. `nslookup` is another command that is similar to `host`, which can be used to query details related to DNS and resolving of names. For example:

```
$ host google.com
google.com has address 64.233.181.105
google.com has address 64.233.181.99
google.com has address 64.233.181.147
google.com has address 64.233.181.106
google.com has address 64.233.181.103
google.com has address 64.233.181.104
```

It may also list out DNS resource records like MX (Mail Exchanger) as follows:

```
$ nslookup google.com
Server:    8.8.8.8
Address:  8.8.8.8#53

Non-authoritative answer:
Name:  google.com
Address: 64.233.181.105
Name:  google.com
Address: 64.233.181.99
Name:  google.com
Address: 64.233.181.147
Name:  google.com
Address: 64.233.181.106
Name:  google.com
Address: 64.233.181.103
Name:  google.com
Address: 64.233.181.104

Server:    8.8.8.8
```

The last line above corresponds to the default nameserver used for DNS resolution.

Without using the DNS server, it is possible to add a symbolic name to IP address resolution just by adding entries into file `/etc/hosts`.

In order to add an entry, use the following syntax:

```
# echo IP_ADDRESS symbolic_name >> /etc/hosts
```

For example:

```
# echo 192.168.0.9 backupserver.com  >> /etc/hosts
```

After adding this entry, whenever a resolution to `backupserver.com` occurs, it will resolve to `192.168.0.9`.

Setting default gateway, showing routing table information

When a local network is connected to another network, it needs to assign some machine or network node through which an interconnection takes place. Hence the IP packets with a destination exterior to the local network should be forwarded to the node machine, which is interconnected to the external network. This special node machine, which is capable of forwarding packets to the external network, is called a gateway. We set the gateway for every node to make it possible to connect to an external network.

The operating system maintains a table called the routing table, which contains information on how packets are to be forwarded and through which machine node in the network. The routing table can be displayed as follows:

```
$ route
Kernel IP routing table
Destination      Gateway   Genmask       Flags  Metric  Ref  UseIface
192.168.0.0         *      255.255.252.0  U     2       0     0wlan0
link-local          *      255.255.0.0    U     1000    0     0wlan0
default          p4.local  0.0.0.0        UG    0       0     0wlan0
```

Or, you can also use:

```
$ route -n
Kernel IP routing table
Destination   Gateway      Genmask        Flags Metric Ref  Use   Iface
192.168.0.0   0.0.0.0      255.255.252.0   U     2      0     0   wlan0
169.254.0.0   0.0.0.0      255.255.0.0     U     1000   0     0   wlan0
0.0.0.0       192.168.0.4  0.0.0.0         UG    0      0     0   wlan0
```

Using `-n` specifies to display the numerical addresses. When `-n` is used it will display every entry with a numerical IP addresses, else it will show symbolic host names instead of IP addresses under the DNS entries for IP addresses that are available.

A default gateway is set as follows:

```
# route add default gw IP_ADDRESS INTERFACE_NAME
```

For example:

```
# route add default gw 192.168.0.1 wlan0
```

Traceroute

When an application requests a service through the Internet, the server may be at a distant location and connected through any number of gateways or device nodes. The packets travel through several gateways and reach the destination. There is an interesting command `traceroute` that displays the address of all intermediate gateways through which the packet travelled to reach the destination. `traceroute` information helps us to understand how many hops each packet should take in order reach the destination. The number of intermediate gateways or routers gives a metric to measure the distance between two nodes connected in a large network. An example of the output from `traceroute` is as follows:

```
$ traceroute google.com
traceroute to google.com (74.125.77.104), 30 hops max, 60 byte packets
1  gw-c6509.lxb.as5577.net (195.26.4.1)  0.313 ms  0.371 ms  0.457 ms
2  40g.lxb-fra.as5577.net (83.243.12.2)  4.684 ms  4.754 ms  4.823 ms
3  de-cix10.net.google.com (80.81.192.108)  5.312 ms  5.348 ms  5.327 ms
4  209.85.255.170 (209.85.255.170)  5.816 ms  5.791 ms 209.85.255.172
(209.85.255.172)  5.678 ms
5  209.85.250.140 (209.85.250.140)  10.126 ms  9.867 ms  10.754 ms
6  64.233.175.246 (64.233.175.246)  12.940 ms 72.14.233.114
(72.14.233.114)  13.736 ms  13.803 ms
7  72.14.239.199 (72.14.239.199)  14.618 ms 209.85.255.166
(209.85.255.166)  12.755 ms 209.85.255.143 (209.85.255.143)  13.803 ms
8  209.85.255.98 (209.85.255.98)  22.625 ms 209.85.255.110
(209.85.255.110)  14.122 ms
*
9  ew-in-f104.1e100.net (74.125.77.104)  13.061 ms  13.256 ms  13.484 ms
```

See also

- *Playing with variables and environment variables* of *Chapter 1*, explains the PATH variable
- *Searching and mining "text" inside a file with grep* of *Chapter 4*, explains the grep command

Let's ping!

`ping` is the most basic network command, and one that every user should first know. It is a universal command that is available on major Operating Systems. It is also a diagnostic tool used for verifying the connectivity between two hosts on a network. It can be used to find out which machines are alive on a network. Let us see how to use ping.

How to do it...

In order to check the connectivity of two hosts on a network, the `ping` command uses **Internet Control Message Protocol** (**ICMP**) echo packets. When these echo packets are sent towards a host, the host responds back with a reply if it is reachable or alive.

Check whether a host is reachable as follows:

```
$ ping ADDRESS
```

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

`ping` will continuously send packets and the reply information is printed on the terminal. Stop the pinging by pressing *Ctrl + C*.

For example:

- When the host is reachable the output will be similar to the following:

```
$ ping 192.168.0.1
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=1.44 ms
^C
--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.440/1.440/1.440/0.000 ms

$ ping google.com
PING google.com (209.85.153.104) 56(84) bytes of data.
64 bytes from bom01s01-in-f104.1e100.net (209.85.153.104): icmp_
seq=1 ttl=53 time=123 ms
^C
--- google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 123.388/123.388/123.388/0.000 ms
```

- When a host is unreachable the output will be similar to:

```
$ ping 192.168.0.99
PING 192.168.0.99 (192.168.0.99) 56(84) bytes of data.
From 192.168.0.82 icmp_seq=1 Destination Host Unreachable
From 192.168.0.82 icmp_seq=2 Destination Host Unreachable
```

Once the host is not reachable, the ping returns a `Destination Host Unreachable` error message.

There's more

In addition to checking the connectivity between two points in a network, the `ping` command can be used with additional options to get useful information. Let's go through the additional options of `ping`.

Round trip time

The `ping` command can be used to find out the **Round Trip Time** (**RTT**) between two hosts on a network. RTT is the time required for the packet to reach the destination host and come back to the source host. The RTT in milliseconds can be obtained from ping. An example is as follows:

```
--- google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4000ms
rtt min/avg/max/mdev = 118.012/206.630/347.186/77.713 ms
```

Here the minimum RTT is 118.012ms, the average RTT is 206.630ms, and the maximum RTT is 347.186ms. The `mdev` (77.713ms) parameter in the `ping` output stands for mean deviation.

Limiting number of packets to be sent

The `ping` command sends echo packets and waits for the reply of `echo` indefinitely until it is stopped by pressing *Ctrl + C*. However, we can limit the count of echo packets to be sent by using the `-c` flag.

The usage is as follows:

```
-c COUNT
```

For example:

```
$ ping 192.168.0.1 -c 2
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=4.02 ms
64 bytes from 192.168.0.1: icmp_seq=2 ttl=64 time=1.03 ms
```

```
--- 192.168.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 1.039/2.533/4.028/1.495 ms
```

In the previous example, the `ping` command sends two echo packets and stops.

This is useful when we need to ping multiple machines from a list of IP addresses through a script and checks its statuses.

Return status of 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, where failure is when destination host is unreachable.

The return status can be easily obtained as follows:

```
$ ping ADDRESS -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, whether alive or not. 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. Let's see how to do it.

Getting ready

In this recipe, we use two methods. The first method uses `ping` and the second method uses `fping`. `fping` doesn't come with a Linux distribution by default. You may have to manually install `fping` using a package manager.

How to do it...

Let's go through the script to find out all the live machines on the network and alternate methods to find out the same.

- **Method 1:**

  We can write our own script using the `ping` command to query 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
```

- **Method 2:**

  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
```

How it works...

In Method 1, we used the `ping` command to find out the alive machines on the network. We used a `for` loop for iterating through the list of IP addresses. The list is generated as `192.168.0.{1..255}`. The `{start..end}` notation will expand and will generate a list of IP addresses, such as `192.168.0.1`, `192.168.0.2`, `192.168.0.3` till `192.168.0.255`.

`ping $ip -c 2 &> /dev/null` will run a `ping` to the corresponding IP address in each execution of loop. `-c 2` is used to restrict the number of echo packets to be sent to two packets. `&> /dev/null` is used to redirect both `stderr` and `stdout` to `/dev/null` so that it won't be printed on the terminal. Using `$?` we evaluate the exit status. If it is successful, the exit status is 0 else non-zero. Hence the successful IP addresses are printed. We can also print the list of unsuccessful IP addresses to give the list of unreachable IP addresses.

Here is an exercise for you. Instead of using a range of IP addresses hard-coded in the script, modify the script to read a list of IP addresses from a file or `stdin`.

In this script, each ping is executed one after the other. Even though all the IP addresses are independent each other, the `ping` command is executed due to a sequential program, it takes a delay of sending two echo packets and receiving them or the timeout for a reply for executing the next `ping` command.

When it comes to 255 addresses, the delay is large. Let's run all the `ping` commands in parallel to make it much faster. The core part of the script is the loop body. To make the `ping` commands in parallel, enclose the loop body in `( )&`. `( )` encloses a block of commands to run as the sub-shell and `&` sends it to the background by leaving the current thread. For example:

```
(
  ping $ip -c2 &> /dev/null ;

  if [ $? -eq 0 ];
  then
    echo $ip is alive
  fi
)&

wait
```

The `for` loop body executes many background process and it comes out of the loop and it terminates the script. In order to present the script to terminate until all its child process end, we have a command called `wait`. Place a `wait` at the end of the script so that it waits for the time until all the child `( )` subshell processes complete.

The `wait` command enables a script to be terminated only after all its child process or background processes terminate or complete.

Have a look at `fast_ping.sh` from the code provided with the book.

Method 2 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 unreachable host to a 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
```

There's more...

The `fping` command can be used for querying DNS data from a network. Let's see how to do it.

DNS lookup with fping

`fping` has an option `-d` that returns host names by using DNS lookup for each echo reply. It will print out host names rather than IP addresses on ping replies.

```
$ cat ip.list
192.168.0.86
192.168.0.9
192.168.0.6

$ fping -a -d 2>/dev/null  <ip.list
www.local
dnss.local
```

See also

- *Playing with file descriptors and redirection* of *Chapter 1*, explains the data redirection
- *Comparisons and tests* of *Chapter 1*, explains numeric comparisons

Transferring files

The major purpose of the networking of computers is for resource sharing. Among resource sharing, the most prominent use is in file sharing. 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.

Getting ready

The commands for performing file transfer over the network are mostly available by default with Linux installations. Files via FTP can be transferred by using the `lftp` command. Files via a SSH connection can be transferred by using `sftp`, RSYNC using SSH with `rsync` command and transfer through SSH using `scp`.

How to do it...

File Transfer Protocol (**FTP**) is an 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. It uses Port 21. FTP can only be used if an FTP server is installed on the remote machine. FTP is used by many public websites to share files.

To connect to an FTP server and transfer files in between, use:

```
$ lftp username@ftphost
```

Now 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 directory of local machine, use `lcd`
- To create a directory use `mkdir`
- 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 exited by using the `quit` command

Auto completion is supported in the `lftp` prompt.

There's more...

Let's go through some additional techniques and commands used for file transfer through a network.

Automated FTP transfer

`ftp` is another command used for FTP-based file transfer. `lftp` is more flexible for usage. `lftp` and the `ftp` command open an interactive session with user (it prompts for user input by displaying messages). What if we want to automate a file transfer instead of using the interactive mode? We can automate FTP file transfers by writing a shell script as follows:

```
#!/bin/bash
#Filename: ftp.sh
#Automated FTP transfer
HOST='domain.com'
USER='foo'
PASSWD='password'
ftp -i -n $HOST <<EOF
user ${USER} ${PASSWD}
binary
cd /home/slynux
puttestfile.jpg
getserverfile.jpg
quit
EOF
```

The above script has the following structure:

```
<<EOF
DATA
EOF
```

This is used to send data through `stdin` to the FTP command. The recipe, *Playing with file descriptors and redirection* in *Chapter 1*, explains various methods for redirection into `stdin`.

The `-i` option of `ftp` turns off the interactive session with user. `user ${USER} ${PASSWD}` sets the username and password. `binary` sets the file mode to binary.

SFTP (Secure FTP)

SFTP is an FTP-like file transfer system that runs on top of an SSH connection. It makes use of an SSH connection to emulate an FTP interface. It doesn't require an FTP server at the remote end to perform file transfer but it requires an OpenSSH server to be installed and running. It is an interactive command, which offers an `sftp` prompt.

The following commands are used to perform the file transfer. All other commands remain same for every automated FTP session with specific HOST, USER, and PASSWD:

```
cd /home/slynux
put testfile.jpg
get serverfile.jpg
```

In order to run `sftp`, use:

```
$ sftp user@domainname
```

Similar to `lftp`, an `sftp` session can be exited by typing the `quit` command.

The SSH server sometimes will not be running at the default Port 22. If it is running at a different port, we can specify the port along with `sftp` as `-oPort=PORTNO`.

For example:

```
$ sftp -oPort=422 user@slynux.org
```

`-oPort` should be the first argument of the `sftp` command.

RSYNC

rsync is an important command-line utility that is widely used for copying files over networks and for taking backup snapshots. This is better explained in separate recipe, *Backup snapshots with rsync*, that explains the usage of `rsync`.

SCP (Secure Copy)

SCP is a file copy technique which is more secure than the traditional remote copy tool called `rcp`. The files are transferred through an encrypted channel. SSH is used as an encryption channel. We can easily transfer files to a remote machine as follows:

```
$ scp filename user@remotehost:/home/path
```

This will prompt for a password. It can be made password less by using autologin SSH technique. The recipe, *Password-less auto-login with SSH*, explains SSH autologin.

Therefore, file transfer using `scp` doesn't require specific scripting. Once SSH login is automated, the `scp` command can be executed without an interactive prompt for the password.

Here `remotehost` can be IP address or domain name. The format of the `scp` command is:

```
$ scp SOURCE DESTINATION
```

`SOURCE` or `DESTINATION` can be in the format `username@localhost:/path` for example:

```
$ scp user@remotehost:/home/path/filename filename
```

The above command copies a file from the remote host to the current directory with the given filename.

If SSH is running at a different port than 22, use `-oPort` with the same syntax as `sftp`.

Recursive copying with SCP

By using `scp` we can recursively copy a directory between two machines on a network as follows with the `-r` parameter:

```
$ scp -r /home/slynux user@remotehost:/home/backups
# Copies the directory /home/slynux recursively to remote location
```

`scp` can also copy files by preserving permissions and mode by using the `-p` parameter.

See also

- *Playing with file descriptors and redirection* of *Chapter 1*, explains the standard input using EOF

Setting up an Ethernet and wireless LAN with script

An Ethernet is simple to configure. Since it uses physical cables, there are no special requirements such as authentication. However, a wireless LAN requires authentication—for example, a WEP key as well as the ESSID of the wireless network to connect. Let's see how to connect to a wireless as well as a wired network by writing a shell script.

Getting ready

To connect to a wired network, we need to assign an IP address and subnet mask by using the `ifconfig` utility. But for a wireless network connection, it will require additional utilities, such as `iwconfig` and `iwlist`, to configure more parameters.

How to do it...

In order to connect to a network from a wired interface, execute the following script:

```
#!/bin/bash
#Filename: etherconnect.sh
#Description: Connect Ethernet

#Modify the parameters below according to your settings
######### PARAMETERS ############

IFACE=eth0
IP_ADDR=192.168.0.5
SUBNET_MASK=255.255.255.0
GW=192.168.0.1
HW_ADDR='00:1c:bf:87:25:d2'
# HW_ADDR is optional
###################################

if [ $UID -ne 0 ];
then
  echo "Run as root"
  exit 1
fi

# Turn the interface down before setting new config
/sbin/ifconfig $IFACE down

if [[ -n $HW_ADDR  ]];
then
  /sbin/ifconfig hw ether $HW_ADDR
 echo Spoofed MAC ADDRESS to $HW_ADDR

fi

/sbin/ifconfig $IFACE $IP_ADDR netmask $SUBNET_MASK

route add default gw $GW $IFACE

echo Successfully configured $IFACE
```

The script for connecting to a wireless LAN with WEP is as follows:

```
#!/bin/bash
#Filename: wlan_connect.sh
#Description: Connect to Wireless LAN

#Modify the parameters below according to your settings
######### PARAMETERS ############
IFACE=wlan0
IP_ADDR=192.168.1.5
SUBNET_MASK=255.255.255.0
```

```
GW=192.168.1.1
HW_ADDR='00:1c:bf:87:25:d2'
#Comment above line if you don't want to spoof mac address
ESSID="homenet"
WEP_KEY=8b140b20e7
FREQ=2.462G
################################
KEY_PART=""
if [[ -n $WEP_KEY ]];
then
  KEY_PART="key $WEP_KEY"
fi
# Turn the interface down before setting new config
/sbin/ifconfig $IFACE down
if [ $UID -ne 0 ];
then
  echo "Run as root"
  exit 1;
fi
if [[ -n $HW_ADDR  ]];
then
  /sbin/ifconfig $IFACE hw ether $HW_ADDR
  echo Spoofed MAC ADDRESS to $HW_ADDR
fi
/sbin/iwconfig $IFACE essid $ESSID $KEY_PART freq $FREQ
/sbin/ifconfig $IFACE $IP_ADDR netmask $SUBNET_MASK
route add default gw $GW $IFACE
echo Successfully configured $IFACE
```

How it works...

The commands `ifconfig`, `iwconfig`, and `route` are to be run as root. Hence a check for the root user is performed at the beginning of the scripts.

The Ethernet connection script is pretty straightforward and it uses the concepts explained in the recipe, *Basic networking primer*. Let's go through the commands used for connecting to the wireless LAN.

A wireless LAN requires some parameters such as the `essid`, `key`, and frequency to connect to the network. The `essid` is the name of the wireless network to which we need to connect. Some **Wired Equivalent Protocol** (**WEP**) networks use a WEP key for authentication, whereas some networks don't. The WEP key is usually a 10-letter hex passphrase. Next comes the frequency assigned to the network. `iwconfig` is the command used to attach the wireless card with the proper wireless network, WEP key, and frequency.

We can scan and list the available wireless network by using the utility `iwlist`. To scan, use the following command:

```
# iwlist scan
wlan0     Scan completed :
          Cell 01 - Address: 00:12:17:7B:1C:65
                    Channel:11
                    Frequency:2.462 GHz (Channel 11)
                    Quality=33/70  Signal level=-77 dBm
                    Encryption key:on
                    ESSID:"model-2"
```

The `Frequency` parameter can be extracted from the scan result, from the line `Frequency:2.462 GHz (Channel 11)`.

See also

- *Comparisons and tests* of *Chapter 1*, explains string comparisons.

Password-less auto-login with SSH

SSH is widely used with automation scripting. By using SSH, it is possible to remotely execute commands at remote hosts and read their output. SSH is authenticated by using username and password. Passwords are prompted during the execution of SSH commands. But in automation scripts, SSH commands may be executed hundreds of times in a loop and hence providing passwords each time is impractical. Hence we need to automate logins. SSH has a built-in feature by which SSH can auto-login using SSH keys. This recipe describes how to create SSH keys and facilitate auto-login.

How to do it...

The SSH uses public key-based and private key-based encryption techniques for automatic authentication. An authentication key has two elements: a public key and a private key pair. We can create an authentication key using the `ssh-keygen` command. For automating the authentication, the public key must be placed at the server (by appending the public key to the `~/.ssh/authorized_keys` file) and its private key file of the pair should be present at the `~/.ssh` directory of the user at client machine, which is the computer you are logging in from. Several configurations (for example, path and name of the `authorized_keys` file) regarding the SSH can be configured by altering the configuration file `/etc/ssh/sshd_config`.

There are two steps towards the setup of automatic authentication with SSH. They are:

1. Creating the SSH key from the machine, which requires a login to a remote machine.
2. Transferring the public key generated to the remote host and appending it to `~/.ssh/authorized_keys` file.

In order to create an SSH key, enter the `ssh-keygen` command with the encryption algorithm type specified as RSA as follows:

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/slynux/.ssh/id_rsa):
Created directory '/home/slynux/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/slynux/.ssh/id_rsa.
Your public key has been saved in /home/slynux/.ssh/id_rsa.pub.
The key fingerprint is:
f7:17:c6:4d:c9:ee:17:00:af:0f:b3:27:a6:9c:0a:05slynux@slynux-laptop
The key's randomart image is:
+--[ RSA 2048]----+
|            .    |
|             o . .|
|     E        o o.|
|      ...oo |
|       .S .+  +o.|
|      .  . .=....|
|     .+.o...|
|      . . + o.  .|
|       ..+       |
+-----------------+
```

You need to enter a passphrase for generating the public-private key pair. It is also possible to generate the key pair without entering a passphrase, but it is insecure. We can write monitoring scripts that use automated login from the script to several machines. In such cases, you should leave the passphrase empty while running the `ssh-keygen` command to prevent the script from asking for a passphrase while running.

Now `~/.ssh/id_rsa.pub` and `~/.ssh/id_rsa` has been generated. `id_dsa.pub` is the generated public key and `id_dsa` is the private key. The public key has to be appended to the `~/.ssh/authorized_keys` file on remote servers where we need to auto-login from the current host.

In order to append a key file, use:

```
$ ssh USER@REMOTE_HOST "cat >> ~/.ssh/authorized_keys" < ~/.ssh/id_rsa.
pub
Password:
```

Provide the login password in the previous command.

The auto-login has been set up. From now on, SSH will not prompt for passwords during execution. You can test this with the following command:

```
$ ssh USER@REMOTE_HOST uname
Linux
```

You will not be prompted for a password.

Running commands on remote host with SSH

SSH is an interesting system administration tool that enables to control remote hosts by login with a shell. SSH stands for Secure Shell. Commands can be executed on the shell received by login to remote host as if we run commands on localhost. It runs the network data transfer over an encrypted tunnel. This recipe will introduce different ways in which commands can be executed on the remote host.

Getting ready

SSH doesn't come by default with all GNU/Linux distributions. Therefore, you may have to install the `openssh-server` and `openssh-client` packages using a package manager. SSH service runs by default on port number 22.

How to do it...

To connect to a remote host with the SSH server running, use:

```
$ ssh username@remote_host
```

In this command:

- `username` is the user that exist at the remote host.
- `remote_host` can be domain name or IP address.

For example:

```
$ ssh mec@192.168.0.1
The authenticity of host '192.168.0.1 (192.168.0.1)' can't be
established.
RSA key fingerprint is 2b:b4:90:79:49:0a:f1:b3:8a:db:9f:73:2d:75:d6:f9.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.1' (RSA) to the list of known
hosts.
Password:

Last login: Fri Sep  3 05:15:21 2010 from 192.168.0.82
mec@proxy-1:~$
```

It will interactively ask for a user password and upon successful authentication it will return the shell for the user.

By default, the SSH server runs at Port 22. But certain servers run the SSH service at different ports. In that case use `-p port_no` with the `ssh` command to specify the port.

In order to connect to an SSH server running at port 422, use:

```
$ ssh user@locahost -p 422
```

You can execute commands in the shell that corresponds to the remote host. Shell is an interactive tool in which a user types and runs commands. However, in shell scripting contexts, we do not need an interactive shell. We need to automate several tasks. We require to execute several commands at the remote shell and display or store its output at localhost. Issuing a password every time is not practical for an automated script, hence autologin for SSH should be configured.

The recipe, *Password-less auto-login with SSH*, explains the SSH commands.

Make sure that auto-login is configured before running automated scripts that use SSH.

To run a command on the remote host and display its output on the localhost shell, use the following syntax:

```
$ ssh user@host 'COMMANDS'
```

For example:

```
$ ssh mec@192.168.0.1 'whoami'
Password:
mec
```

Multiple commands can be given by using semicolon delimiter in between the commands as:

```
$ ssh user@host 'command1 ; command2 ; command3'
```

Commands can be sent through `stdin` and the output of the commands will be available to `stdout`.

The syntax will be as follows:

```
$ ssh user@remote_host  "COMMANDS" > stdout.txt 2> errors.txt
```

The `COMMANDS` string should be quoted in order to prevent a semicolon character to act as delimiter in the localhost shell. We can also pass any command sequence that involves piped statements to the SSH command through `stdin` as follows:

```
$ echo  "COMMANDS" | sshuser@remote_host> stdout.txt 2> errors.txt
```

For example:

```
$ ssh mec@192.168.0.1  "echo user: $(whoami);echo OS: $(uname)"
Password:
user: slynux
OS: Linux
```

In this example, the commands executed on the remote host are:

```
echo user: $(whoami);
echo OS: $(uname)
```

It can be generalized as:

```
COMMANDS="command1; command2; command3"
$ ssh user@hostname  "$COMMANDS"
```

We can also pass a more complex subshell in the command sequence by using the `( )` subshell operator.

Let's write an SSH based shell script that collects the uptime of a list of remote hosts. Uptime is the time for which the system is powered on. The `uptime` command is used to display how long the system has been powered on.

It is assumed that all systems in the `IP_LIST` have a common user `test`.

```
#!/bin/bash
#Filename: uptime.sh
#Description: Uptime monitor

IP_LIST="192.168.0.1 192.168.0.5 192.168.0.9"
USER="test"

for IP in $IP_LIST;
do
  utime=$(ssh $USER@$IP uptime  | awk '{ print $3 }' )
  echo $IP uptime:  $utime
done
```

The expected output is:

```
$ ./uptime.sh
192.168.0.1 uptime: 1:50,
192.168.0.5 uptime: 2:15,
192.168.0.9 uptime: 10:15,
```

There's more...

The `ssh` command can be executed with several additional options. Let's go through them.

SSH with compression

The SSH protocol also supports data transfer with compression, which comes in handy when bandwidth is an issue. Use the `-C` option with the `ssh` command to enable compression as follows:

```
$ ssh -C user@hostname COMMANDS
```

Redirecting data into stdin of remote host shell commands

Sometimes we need to redirect some data into `stdin` of remote shell commands. Let's see how to do it. An example is as follows:

```
$ echo "text" | ssh user@remote_host 'cat >> list'
```

Or:

```
# Redirect data from file as:
$ ssh user@remote_host 'cat >> list'  < file
```

`cat >> list` appends the data received through `stdin` to the file list. Here this command is executed at the remote host. But the data is passed to `stdin` from localhost.

See also

- *Password-less auto-login with SSH*, explains how to configure auto-login to execute commands without prompting for password.

Mounting a remote drive at a local mount point

Having a local mount point to access remote host file-system would be really helpful while carrying out both read and write data transfer operations. SSH is the most common transfer protocol available in a network and hence we can make use of it with `sshfs`. `sshfs` enables you to mount a remote filesystem to a local mount point. Let's see how to do it.

Getting ready

`sshfs` doesn't come by default with GNU/Linux distributions. Install `sshfs` by using a package manager. `sshfs` is an extension to the fuse file system package that allows supported OSes to mount a wide variety of data as if it were a local file system.

How to do it...

In order to mount a filesytem location at a remote host to a local mount point, use:

```
# sshfs user@remotehost:/home/path /mnt/mountpoint
Password:
```

Issue the user password when prompted.

Now data at `/home/path` on the remote host can be accessed via a local mount point `/mnt/mountpoint`.

In order to unmount after completing the work, use:

```
# umount /mnt/mountpoint
```

See also

- *Running commands on remote host with SSH*, explains the ssh command.

Multi-casting window messages on a network

The administrator of a network may often require to send messages to the nodes on the network. Displaying pop-up windows on the user's desktop would be helpful to alert the user with a piece of information. Using a GUI toolkit with shell scripting can achieve this task. This recipe discusses how to send a popup window with custom messages to remote hosts.

Getting ready

For implementing a GUI pop window, zenity can be used. Zenity is a scriptable GUI toolkit for creating windows consisting of textbox, input box, and so on. SSH can be used for connecting to the remote shell on a remote host. Zenity doesn't come installed by default with GNU/Linux distributions. Use a package manager to install zenity.

How to do it...

Zenity is one of the scriptable dialog creation toolkit. There are other toolkits, such as gdialog, kdialog, xdialog, and so on. Zenity seems to be one flexible toolkit that is adherent to the GNOME Desktop Environment.

In order to create an info box with zenity, use:

```
$ zenity --info --text "This is a message"
# It will display a window with "This is a message" as text.
```

Zenity can be used to create windows with input box, combo input, radio button, pushbutton, and more. They are not in the scope of this recipe. Check the man page of zenity for more.

Now, we can use SSH to run these zenity statements on a remote machine. In order to run this statement on the remote host through SSH, run:

```
$ ssh user@remotehost 'zenity --info --text "This is a message"'
```

But this will return an error like:

```
(zenity:3641): Gtk-WARNING **: cannot open display:
```

This is because zenity depends on Xserver. Xsever is a daemon which is responsible for plotting graphical elements on the screen which consists of the GUI. A bare GNU/Linux system consists of only a text terminal or shell prompts.

Xserver uses a special environment variable, `DISPLAY`, to track the Xserver instance that is running on the system.

We can manually set `DISPLAY=:0` to instruct Xserver about the Xserver instance.

The previous SSH command can be rewritten as:

```
$ ssh username@remotehost 'export DISPLAY=:0 ; zenity --info --text "This
is a message"'
```

This statement will display a pop up at `remotehost` if the user with username has been logged in any of the window managers.

In order to multicast the popup window to multiple remote hosts, write a shell script as follows:

```
#!/bin/bash
#Filename: multi_cast_window.sh
# Description: Multi-cast window popups

IP_LIST="192.168.0.5 192.168.0.3 192.168.0.23"
USER="username"

COMMAND='export DISPLAY=:0 ;zenity --info --text "This is a message" '
for host in $IP_LIST;
do
  ssh $USER@$host  "$COMMAND" &
done
```

How it works...

In the above script, we have a list of IP addresses to which the window should be popped up. A loop is used to iterate through IP addresses and execute the SSH command.

In the SSH statement, at the end we have post fixed `&`. `&` will send an SSH statement to the background. It is done to facilitate parallelization in the execution of several SSH statements. If `&` was not used, it will start the SSH session, execute the zenity dialog, and wait for the user to close that pop up window. Unless the user at the remote host closes the window, the next SSH statement in the loop will not be executed. In order to move away from this blocking of the loop from further execution by waiting for the SSH session to terminate, the `&` trick is used.

See also

- *Running commands on remote host with SSH*, explains the ssh command.

Network traffic and port analysis

Network ports are essential parameters of network-based applications. Applications open ports on the host and communicate to a remote host through opened ports at the remote host. Having awareness of opened and closed ports is essential for security context. Malwares and root kits may be running on the system with custom ports and custom services that allow attackers to capture unauthorized access to data and resources. By getting the list of opened ports and services running on the ports, we can analyze and defend the system from being controlled by root kits and the list helps to remove them efficiently. The list of opened ports is not only helpful for malware detection, but also for collecting information about opened ports on the system enables to debug network based applications. It helps to analyse whether certain port connections and port listening functionalities are working fine. This recipe discusses various utilities for port analysis.

Getting ready

Various commands are available for listening to ports and services running on each port (for example, `lsof` and `netstat`). These commands are, by default, available on all GNU/Linux distributions.

How to do it...

In order to list all opened ports on the system along with the details on each service attached to it, use:

```
$ lsof -i
COMMAND    PID   USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
firefox-b 2261 slynux   78u  IPv4  63729      0t0  TCP localhost:47797-
>localhost:42486 (ESTABLISHED)
firefox-b 2261 slynux   80u  IPv4  68270      0t0  TCP slynux-laptop.
local:41204->192.168.0.2:3128 (CLOSE_WAIT)
firefox-b 2261 slynux   82u  IPv4  68195      0t0  TCP slynux-laptop.
local:41197->192.168.0.2:3128 (ESTABLISHED)
ssh       3570 slynux    3u  IPv6  30025      0t0  TCP localhost:39263-
>localhost:ssh (ESTABLISHED)
ssh       3836 slynux    3u  IPv4  43431      0t0  TCP slynux-laptop.
local:40414->boneym.mtveurope.org:422 (ESTABLISHED)
GoogleTal 4022 slynux   12u  IPv4  55370      0t0  TCP localhost:42486
(LISTEN)
GoogleTal 4022 slynux   13u  IPv4  55379      0t0  TCP localhost:42486-
>localhost:32955 (ESTABLISHED)
```

Each entry in the output of `lsof` corresponds to each service that opens a port for communication. The last column of the output consists of lines similar to:

```
slynux-laptop.local:34395->192.168.0.2:3128 (ESTABLISHED)
```

In this output `slynux-laptop.local:34395` corresponds to localhost part and `192.168.0.2:3128` corresponds to remote host.

`34395` is the port opened from current machine, and `3128` is the port to which the service connects at remote host.

In order to list out the opened ports from current machine, use:

```
$ lsof -i | grep ":[0-9]\+->" -o | grep "[0-9]\+" -o  | sort | uniq
```

The `:[0-9]\+->` regex for `grep` is used to extract the host port portion (`:34395->`) from the `lsof` output. The next `grep` is used to extract the port number (which is numeric). Multiple connections may occur through the same port and hence multiple entries of the same port may occur. In order to display each port once, they are sorted and the unique ones are printed.

There's more...

Let's go through additional utilities that can be used for viewing the opened port and network traffic related information.

Opened port and services using netstat

`netstat` is another command for network service analysis. Explaining all the features of `netstat` is not in the scope of this recipe. We will now look at how to list services and port numbers.

Use `netstat -tnp` to list opened ports and services as follows:

```
$ netstat -tnp
(Not all processes could be identified, non-owned process info
will not be shown, you would have to be root to see it all.)
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
PID/Program name
tcp        0      0 192.168.0.82:38163      192.168.0.2:3128
ESTABLISHED 2261/firefox-bin
tcp        0      0 192.168.0.82:38164      192.168.0.2:3128        TIME_
WAIT   -
tcp        0      0 192.168.0.82:40414      193.107.206.24:422
ESTABLISHED 3836/ssh
tcp        0      0 127.0.0.1:42486         127.0.0.1:32955
ESTABLISHED 4022/GoogleTalkPlug
tcp        0      0 192.168.0.82:38152      192.168.0.2:3128
ESTABLISHED 2261/firefox-bin
tcp6       0      0 ::1:22                  ::1:39263
ESTABLISHED -
tcp6       0      0 ::1:39263               ::1:22
ESTABLISHED 3570/ssh
```

8
Put on the Monitor's Cap

In this chapter, we will cover:

- Disk usage hacks
- Calculating the execution time for a command
- Information about logged users, boot logs, failure boots
- Printing the 10 most frequently-used commands
- Listing the top 10 CPU consuming process in 1 hour
- Monitoring command outputs with watch
- Logging access to files and directories
- Logfile management with logrotate
- Logging with syslog
- Monitoring user logins to find intruders
- Remote disk usage health monitoring
- Finding out active user hours on a system

Introduction

An operating system consists of a collection of system software, designed for different purposes, serving different task sets. Each of these programs requires to be monitored by the operating system or the system administrator in order to know whether it is working properly or not. We will also use a technique called logging by which important information is written to a file while the application is running. By reading this file, we can understand the timeline of the operations that are taking place with a particular software or a daemon. If an application or a service crashes, this information helps to debug the issue and enables us to fix any issues. Logging and monitoring also helps to gather information from a pool of data. Logging and monitoring are important tasks for ensuring security in the operating system and for debugging purposes.

This chapter deals with different commands that can be used to monitor different activities. It also goes through logging techniques and their usages.

Disk usage hacks

Disk space is a limited resource. We frequently perform disk usage calculation on hard disks or any storage media to find out the free space available on the disk. When free space becomes scarce, we will need to find out large-sized files that are to be deleted or moved in order to create free space. Disk usage manipulations are commonly used in shell scripting contexts. This recipe will illustrate various commands used for disk manipulations and problems where disk usages can be calculated with a variety of options.

Getting ready

`df` and `du` are the two significant commands that are used for calculating disk usage in Linux. The command `df` stands for disk free and `du` stands for disk usage. Let's see how we can use them to perform various tasks that involve disk usage calculation.

How to do it...

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

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

For example:

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

The result is, by default, shown as size in bytes.

In order 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.

Running `du DIRECTORY` will output a similar result, but it will show only the size consumed by subdirectories. However, they do not show the disk usage for each of the files. For printing the disk usage by files, `-a` is mandatory.

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
```

There's more...

Let's go through additional usage practices for the `du` command.

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 when disk usage is expressed in standard units 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 -sh 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

Suppose 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.sh pcpu.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`. It gives the same output as without using `-c`. The only difference is that it appends 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 flag `-h` can be used along with it to print in human readable format. This command has frequent use in practice. The syntax is as follows:

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

For example:

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

Printing files in specified units

We can force `du` to print the disk usage in specified units. For example:

- Print size in bytes (by default) by using:

  ```
  $ du -b FILE(s)
  ```

- Print the size in kilobytes by using:

  ```
  $ du -k FILE(s)
  ```

- Print the size in megabytes by using:

  ```
  $ du -m FILE(s)
  ```

- Print size in given BLOCK size specified by using:

  ```
  $ du -B BLOCK_SIZE FILE(s)
  ```

  Here, `BLOCK_SIZE` is specified in bytes.

An example consisting of all the commands is as follows:

```
$ du pcpu.sh
4  pcpu.sh
$ du -b pcpu.sh
439    pcpu.sh
$ du -k pcpu.sh
4  pcpu.sh
$ du -m pcpu.sh
1  pcpu.sh
$ du -B 4  pcpu.sh
1024  pcpu.sh
```

Excluding files from disk usage calculation

There are circumstances when we need to exclude certain files from disk usage calculation. Such excluded files can be specified in two ways:

1. **Wildcards**

   We can specify a wildcard as follows:

   ```
   $ du --exclude "WILDCARD" DIRECTORY
   ```

For example:

```
$ du --exclude "*.txt" FILES(s)
# Excludes all .txt files from calculation
```

2. **Exclude list**

   We can specify a list of files to be excluded from a file as follows:

```
$ du --exclude-from EXCLUDE.txt DIRECTORY
# EXCLUDE.txt is the file containing list
```

There are also some other handy options available with `du` to restrict the disk usage calculation. We can specify the maximum depth of the hierarchy that the `du` should traverse as a whole by calculating disk usage with the `--max-depth` parameter. Specifying a depth of `1` calculates the sizes of files in the current directory. Depth `2` will calculate files in the current directory and the next subdirectory and stop traversal at that second subdirectory.

For example:

```
$ du --max-depth 2 DIRECTORY
```

`du` can be restricted to traverse only a single file system by using the `-x` argument. Suppose `du DIRECTORY` is run, it will traverse through every possible subdirectory of `DIRECTORY` recursively. A subdirectory in the directory hierarchy may be a mount point (for example, `/mnt/sda1` is a subdirectory of `/mnt` and it is a mount point for the device `/dev/sda1`). `du` will traverse that mount point and calculate the sum of disk usage for that device filesystem also. In order to prevent `du` from traversing and to calculate from other mount points or filesystems, use the `-x` flag along with other `du` options. `du -x /` will exclude all mount points in `/mnt/` for disk usage calculation.

While using `du` make sure that the directories or files it traverses have the proper read permissions.

Finding the 10 largest size files from a given directory

Finding large-size files is a regular task we come across. We regularly require to delete those huge size files or move them. We can easily find out large-size files using `du` and `sort` commands. The following one-line script can achieve this task:

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

Here `-a` specifies all directories and files. Hence `du` traverses the `SOURCE_DIR` and calculates the size of all files. 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 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 above 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-size files as follows:

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

We used `find` to filter only files to `du` rather than allow `du` to traverse recursively by itself.

Disk free information

The `du` command provides information about the usage, whereas `df` provides information about free disk space. It can be used with and without `-h`. When `-h` is issued with `df` it prints the disk space in human readable format.

For example:

```
$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda1              9611492   2276840   6846412  25% /
none                    508828       240    508588   1% /dev
none                    513048       168    512880   1% /dev/shm
none                    513048        88    512960   1% /var/run
none                    513048         0    513048   0% /var/lock
none                    513048         0    513048   0% /lib/init/rw
none                   9611492   2276840   6846412  25% /var/lib/
ureadahead/debugfs

$ df -h
FilesystemSize  Used Avail Use% Mounted on
/dev/sda1             9.2G  2.2G  6.6G  25% /
none                  497M  240K  497M   1% /dev
none                  502M  168K  501M   1% /dev/shm
```

```
none                  502M   88K  501M   1% /var/run
none                  502M     0  502M   0% /var/lock
none                  502M     0  502M   0% /lib/init/rw
none                  9.2G  2.2G  6.6G  25% /var/lib/ureadahead/debugfs
```

Calculating execution time for a command

While testing an application or comparing different algorithms for a given problem, execution time taken by a program is very critical. A good algorithm should execute in minimum amount of time. There are several situations in which we need to monitor the time taken for execution by a program. For example, while learning about sorting algorithms, how do you practically state which algorithm is faster? The answer to this is to calculate the execution time for the same data set. Let's see how to do it.

How to do it...

`time` is a command that is available with any UNIX-like operating systems. You can prefix `time` with the command you want to calculate execution time, for example:

```
$ time COMMAND
```

The command will execute and its output will be shown. Along with output, the `time` command appends the time taken in `stderr`. An example is as follows:

```
$ time ls
test.txt
next.txt
real    0m0.008s
user    0m0.001s
sys     0m0.003s
```

It will show real, user, and system times for execution. The three different times can be defined as follows:

- **Real** is wall clock time—the time from start to finish of the call. This is all elapsed time including time slices used by other processes and the time that the process spends when blocked (for example, if it is waiting for I/O to complete).
- **User** is the amount of CPU time spent in user-mode code (outside the kernel) within the process. This is only the actual CPU time used in executing the process. Other processes and the time that the process spends when blocked do not count towards this figure.

- **Sys** is the amount of CPU time spent in the kernel within the process. This means executing the CPU time spent in system calls within the kernel, as opposed to library code, which is still running in the user space. Like 'user time', this is only the CPU time used by the process.

An executable binary of the `time` command is available at `/usr/bin/time` as well as a shell built-in named `time` exists. When we run `time`, it calls the shell built-in by default. The shell built-in time has limited options. Hence, we should use an absolute path for the executable (`/usr/bin/time`) for performing additional functionalities.

We can write this time statistics to a file using the `-o filename` option as follows:

```
$ /usr/bin/time -o output.txt COMMAND
```

The filename should always appear after the `–o` flag.

In order to append the time statistics to a file without overwriting, use the `-a` flag along with the `-o` option as follows:

```
$ /usr/bin/time -a -o output.txt COMMAND
```

We can also format the time outputs using format strings with the `-f` option. A format string consists of parameters corresponding to specific options prefixed with `%`. The format strings for real time, user time, and sys time are as follows:

- Real time - `%e`
- User - `%U`
- sys - `%S`

By combining parameter strings, we can create formatted output as follows:

```
$ /usr/bin/time -f "FORMAT STRING" COMMAND
```

For example:

```
$ /usr/bin/time -f "Time: %U" -a -o timing.log uname
Linux
```

Here `%U` is the parameter for user time.

When formatted output is produced, the formatted output of the command is written to the standard output and the output of the `COMMAND`, which is timed, is written to standard error. We can redirect the formatted output using a redirection operator (`>`) and redirect the time information output using the (`2>`) error redirection operator. For example:

```
$ /usr/bin/time -f "Time: %U" uname> command_output.txt 2>time.log
$ cat time.log
Time: 0.00
$ cat command_output.txt
Linux
```

Many details regarding a process can be collected using the `time` command. The important details include, exit status, number of signals received, number of context switches made, and so on. Each parameter can be displayed by using a suitable format string.

The following table shows some of the interesting parameters that can be used:

Parameter	Description
`%C`	Name and command-line arguments of the command being timed.
`%D`	Average size of the process's unshared data area, in kilobytes.
`%E`	Elapsed real (wall clock) time used by the process in [hours:]minutes:seconds.
`%x`	Exit status of the command.
`%k`	Number of signals delivered to the process.
`%W`	Number of times the process was swapped out of the main memory.
`%Z`	System's page size in bytes. This is a per-system constant, but varies between systems.
`%P`	Percentage of the CPU that this job got. This is just user + system times divided by the total running time. It also prints a percentage sign.
`%K`	Average total (data + stack + text) memory usage of the process, in kilobytes.
`%w`	Number of times that the program was context-switched voluntarily, for instance while waiting for an I/O operation to complete.
`%c`	Number of times the process was context-switched involuntarily (because the time slice expired).

For example, the page size can be displayed using the `%Z` parameters as follows:

```
$ /usr/bin/time -f "Page size: %Z bytes" ls> /dev/null
Page size: 4096 bytes
```

Here the output of the timed command is not required and hence the standard output is directed to the `/dev/null` device in order to prevent it from writing to the terminal.

More format strings parameters are available. Read `man time` for more details.

Information about logged users, boot logs, and failure boot

Collecting information about the operating environment, logged in users, the time for which the computer has been powered on, and any boot failures are very helpful. This recipe will go through a few commands used to gather information about a live machine.

Getting ready

This recipe will introduce the commands `who`, `w`, `users`, `uptime`, `last`, and `lastb`.

How to do it...

To obtain information about users currently logged in to the machine use:

```
$ who
slynux   pts/0   2010-09-29 05:24 (slynuxs-macbook-pro.local)
slynux   tty7    2010-09-29 07:08 (:0)
```

Or:

```
$ w
 07:09:05 up  1:45,  2 users,  load average: 0.12, 0.06, 0.02
USER     TTY      FROM    LOGIN@   IDLE  JCPU PCPU WHAT
slynux   pts/0    slynuxs 05:24  0.00s  0.65s 0.11s sshd: slynux
slynux   tty7     :0      07:08  1:45m  3.28s 0.26s gnome-session
```

It will provide information about logged in users, the pseudo TTY used by the users, the command that is currently executing from the pseudo terminal, and the IP address from which the users have logged in. If it is localhost, it will show the hostname. `who` and `w` format outputs with slight difference. The `w` command provides more detail than `who`.

TTY is the device file associated with a text terminal. When a terminal is newly spawned by the user, a corresponding device is created in `/dev/` (for example, `/dev/pts/3`). The device path for the current terminal can be found out by typing and executing the command `tty`.

In order to list the users currently logged in to the machine, use:

```
$ users
Slynux slynux slynux hacker
```

If a user has opened multiple pseudo terminals, it will show that many entries for the same user. In the above output, the user `slynux` has opened three pseudo terminals. The easiest way to print unique users is to use `sort` and `uniq` to filter as follows:

```
$ users | tr ' ' '\n' | sort | uniq
slynux
hacker
```

We have used `tr` to replace `' '` with `'\n'`. Then combination of `sort` and `uniq` will produce unique entries for each user.

In order to see how long the system has been powered on, use:

```
$ uptime
 21:44:33 up  3:17,  8 users,  load average: 0.09, 0.14, 0.09
```

The time that follows the word `up` indicates the time for which the system has been powered on. We can write a simple one-liner to extract the uptime only.

Load average in uptime's output is a parameter that indicates system load. This is explained in more detail in the chapter, *Administration Calls!*. In order to get information about previous boot and user logged sessions, use:

```
$ last
slynux   tty7         :0               Tue Sep 28 18:27   still logged in
reboot   system boot  2.6.32-21-generi Tue Sep 28 18:10 - 21:46  (03:35)
slynux   pts/0        :0.0             Tue Sep 28 05:31 - crash  (12:39)
```

The `last` command will provide information about logged in sessions. It is actually a log of system logins that consists of information such as `tty` from which it has logged in, login time, status, and so on.

The `last` command uses the log file `/var/log/wtmp` for input log data. It is also possible to explicitly specify the log file for the last command using the `-f` option. For example:

```
$ last -f /var/log/wtmp
```

In order to obtain info about login sessions for a single user, use:

```
$ last USER
```

Get information about reboot sessions as follows:

```
$ last reboot
reboot   system boot  2.6.32-21-generi Tue Sep 28 18:10 - 21:48  (03:37)
reboot   system boot  2.6.32-21-generi Tue Sep 28 05:14 - 21:48  (16:33)
```

In order to get information about failed user login sessions use:

```
# lastb
test     tty8         :0               Wed Dec 15 03:56 - 03:56  (00:00)
slynux   tty8         :0               Wed Dec 15 03:55 - 03:55  (00:00)
```

You should run `lastb` as the root user.

Printing the 10 most frequently-used commands

Terminal is the tool used to access the shell prompt where we type and execute commands. Users run many commands in the shell. Many of them are frequently used. A user's nature can be identified easily by looking at the commands he frequently uses. This recipe is a small exercise to find out 10 most frequently-used commands.

Getting ready

Bash keeps track of previously typed commands by the user and stores in the file `~/.bash_history`. But it only keeps a specific number (say 500) of the recently executed commands. The history of commands can be viewed by using the command history or `cat ~/.bash_history`. We will use this for finding out frequently-used commands.

How to do it...

We can get the list of commands from `~/.bash_history`, take only the command excluding the arguments, count the occurrence of each command, and find out the 10 commands with the highest count.

The following script can be used to find out frequently-used commands:

```
#!/bin/bash
#Filename: top10_commands.sh
#Description: Script to list top 10 used commands
printf "COMMAND\tCOUNT\n" ;
cat ~/.bash_history | awk '{ list[$1]++; } \
END{
for(i in list)
{
printf("%s\t%d\n",i,list[i]); }
}'| sort -nrk 2 | head
```

A sample output is as follows:

```
$ ./top10_commands.sh
COMMAND  COUNT
ping    80
ls      56
cat     35
ps      34
sudo    26
du      26
cd      26
ssh     22
sftp    22
clear   21
```

How it works...

In the above script, the history file `~/.bash_history` is the source file used. The source input is passed to `awk` through a pipe. Inside `awk`, we have an associative array list. This array can use command names as index and it stores the count of the commands in array locations. Hence for each arrival or occurrence of a command it will increment by one (`list[$1]++`). `$1` is used as the index. `$1` is the first word of text in a line input. If `$0` were used it would contain all the arguments for the command also. For example, if `ssh 192.168.0.4` is a line from `.bash_history`, `$0` equals to `ssh 192.168.0.4` and `$1` equals to `ssh`.

Once all the lines of the history files are traversed, we will have the array with command names as indexes and their count as the value. Hence command names with maximum count values will be the commands most frequently used. Hence in the `END{}` block of `awk`, we traverse through the indexes of commands and print all command names and their counts. `sort -nrk 2` will perform a numeric sort based on the second column (`COUNT`) and reverse it. Hence we use the `head` command to extract only the first 10 commands from the list. You can customize the top 10 to top 5 or any other number by using the argument `head -n NUMBER`.

Listing the top 10 CPU consuming process in a hour

CPU time is a major resource and sometimes we require to keep track of the processes that consume the most CPU cycles in a period of time. In regular desktops or laptops, it might not be an issue that the CPU is heavily consumed. However, for a server that handles numerous requests, CPU is a critical resource. By monitoring the CPU usage for a certain period we can identify the processes that keep the CPU busy all the time and optimize them to efficiently use the CPU or to debug them due to any other issues. This recipe is a practice with process monitoring and logging.

Getting ready

`ps` is a command used for collecting details about the processes running on the system. It can be used to gather details such as CPU usage, commands under execution, memory usage, status of process, and so on. Processes that consume the CPU for one hour can be logged, and the top 10 can be determined by proper usage of `ps` and text processing. For more details on the `ps` command, see the chapter: *Administration Calls!*.

How to do it...

Let's go through the following shell script for monitoring and calculating CPU usages in one hour:

```
#!/bin/bash
#Name: pcpu_usage.sh
#Description: Script to calculate cpu usage by processes for 1 hour

SECS=3600
UNIT_TIME=60

#Change the SECS to total seconds for which monitoring is to be
performed.
#UNIT_TIME is the interval in seconds between each sampling

STEPS=$(( $SECS / $UNIT_TIME ))

echo Watching CPU usage... ;

for((i=0;i<STEPS;i++))
do
  ps -eo comm,pcpu | tail -n +2 >> /tmp/cpu_usage.$$
  sleep $UNIT_TIME
done

echo
echo CPU eaters :

cat /tmp/cpu_usage.$$ | \
awk '
{ process[$1]+=$2; }
END{
  for(i in process)
  {
    printf("%-20s %s",i, process[i] ;
  }

   }' | sort -nrk 2 | head
rm /tmp/cpu_usage.$$
#Remove the temporary log file
```

A sample output is as follows:

```
$ ./pcpu_usage.sh
Watching CPU usage...
CPU eaters :
Xorg        20
```

```
firefox-bin   15
bash        3
evince      2
pulseaudio    1.0
pcpu.sh         0.3
wpa_supplicant  0
wnck-applet     0
watchdog/0      0
usb-storage     0
```

How it works...

In the above script, the major input source is `ps -eocomm, pcpu`. `comm` stands for command name and `pcpu` stands for the CPU usage in percent. It will output all the process names and the CPU usage in percent. For each process there exists a line in the output. Since we need to monitor the CPU usage for one hour, we repeatedly take usage statistics using `ps -eo comm,pcpu | tail -n +2` and append to a file `/tmp/cpu_usage.$$` running inside a `for` loop with 60 seconds wait in each iteration. This wait is provided by `sleep 60`. It will execute `ps` once in each minute.

`tail -n +2` is used to strip off the header and `COMMAND  %CPU` in the `ps` output.

`$$` in `cpu_usage.$$` signifies that it is the process ID of the current script. Suppose PID is 1345, during execution it will be replaced as `/tmp/cpu_usage.1345`. We place this file in `/tmp` since it is a temporary file.

The statistics file will be ready after one hour and will contain 60 entries corresponding to the process status for each minute. Then `awk` is used to sum the total CPU usage for each process. An associative array process is used for the summation of CPU usages. It uses the process name as an array index. Finally, it sorts the result with a numeric reverse sort according to the total CPU usage and pass through head to obtain top 10 usage entries.

See also

- *Basic awk primer* of *Chapter 4*, explains the awk command
- *head and tail - printing the last or first ten lines* of *Chapter 3*, explains the tail command

Monitoring command outputs with watch

We might need to continuously watch the output of a command for a period of time in equal intervals. For example, for a large file copy, we need to watch the growing file size. In order to do that, newbies repeatedly type commands and press return a number of times. Instead we can use the watch command to view output repeatedly. This recipe explains how to do that.

How to do it...

The `watch` command can be used to monitor the output of a command on the terminal at regular intervals. The syntax of the `watch` command is as follows:

```
$ watch COMMAND
```

For example:

```
$ watch ls
```

Or:

```
$ watch 'COMMANDS'
```

For example:

```
$ watch 'ls -l | grep "^d"'
# list only directories
```

This command will update the output at a default interval of two seconds.

We can also specify the time interval at which the output needs to be updated, by using `-n SECONDS`. For example:

```
$ watch -n 5 'ls -l'
#Monitor the output of ls -l at regular intervals of 5 seconds
```

There's more

Let's explore an additional feature of the `watch` command.

Highlighting the differences in watch output

In `watch`, there is an option for updating the differences that occur during the execution of the command at an update interval to be highlighted using colors. Difference highlighting can be enabled by using the `-d` option as follows:

```
$ watch -d 'COMMANDS'
```

Logging access to files and directories

Logging of file and directory access is very helpful to keep track of changes that are happening to files and folders. This recipe will describe how to log user accesses.

Getting ready

The `inotifywait` command can be used to gather information about file accesses. It doesn't come by default with every Linux distro. You have to install the `inotify-tools` package by using a package manager. It also requires the Linux kernel to be compiled with inotify support. Most of the new GNU/Linux distributions come with inotify enabled in the kernel.

How to do it...

Let's walk through the shell script to monitor the directory access:

```
#/bin/bash
#Filename: watchdir.sh
#Description: Watch directory access
path=$1
#Provide path of directory or file as argument to script

inotifywait -m -r -e create,move,delete $path  -q
```

A sample output is as follows:

```
$ ./watchdir.sh .
./ CREATE new
./ MOVED_FROM new
./ MOVED_TO news
./ DELETE news
```

How it works...

The previous script will log events create, move, and delete files and folders from the given path. The `-m` option is given for monitoring the changes continuously rather than going to exit after an event happens. `-r` is given for enabling a recursive watch the directories. `-e` specifies the list of events to be watched. `-q` is to reduce the verbose messages and print only required ones. This output can be redirected to a log file.

We can add or remove the event list. Important events available are as follows:

Event	Description
`access`	When some read happens to a file.
`modify`	When file contents are modified.
`attrib`	When metadata is changed.
`move`	When a file undergoes move operation.
`create`	When a new file is created.
`open`	When a file undergoes open operation.
`close`	When a file undergoes close operation.
`delete`	When a file is removed.

Logfile management with logrotate

Logfiles are essential components of a Linux system's maintenance. Logfiles help to keep track of events happening on different services on the system. This helps the sysadmin to debug issues and also provides statistics on events happening on the live machine. Management of logfiles is required because as time passes the size of a logfile gets bigger and bigger. Therefore, we use a technique called rotation to limit the size of the logfile and if the logfile reaches a size beyond the limit, it will strip the logfile and store the older entries from the logfile in an archive. Hence older logs can be stored and kept for future reference. Let's see how to rotate logs and store them.

Getting ready

`logrotate` is a command every Linux system admin should know. It helps to restrict the size of logfile to the given SIZE. In a logfile, the logger appends information to the log file. Hence the recent information appears at the bottom of the log file. `logrotate` will scan specific logfiles according to the configuration file. It will keep the last 100 kilobytes (for example, specified SIZE = 100k) from the logfile and move rest of the data (older log data) to a new file `logfile_name.1` with older entries. When more entries occur in the logfile (`logfile_name.1`) and it exceeds the SIZE, it updates the logfile with recent entries and creates `logfile_name.2` with older logs. This process can easily be configured with `logrotate`. `logrotate` can also compress the older logs as `logfile_name.1.gz`, `logfile_name2.gz`, and so on. The option for whether older log files are to be compressed or not is available with the `logrotate` configuration.

How to do it...

`logrotate` has the configuration directory at `/etc/logrotate.d`. If you look at this directory by listing contents, many other logfile configurations can be found.

We can write our custom configuration for our logfile (say `/var/log/program.log`) as follows:

```
$ cat /etc/logrotate.d/program
/var/log/program.log {
missingok
notifempty
size 30k
  compress
weekly
  rotate 5
create 0600 root root
}
```

Now the configuration is complete. `/var/log/program.log` in the configuration specifies the logfile path. It will archive old logs in the same directory path. Let's see what each of these parameters are:

Parameter	Description
`missingok`	Ignore if the logfile is missing and return without rotating the log.
`notifempty`	Only rotate the log if the source logfile is not empty.
`size 30k`	Limit the size of the logfile for which the rotation is to be made. It can be 1M for 1MB.
`compress`	Enable compression with gzip for older logs.
`weekly`	Specify the interval at which the rotation is to be performed. It can be weekly, yearly, or daily.
`rotate 5`	It is the number of older copies of logfile archives to be kept. Since 5 is specified, there will be `program.log.1.gz`, `program.log.2.gz`, and so on till `program.log.5.gz`.
`create 0600 root root`	Specify the mode, user, and the group of the logfile archive to be created.

The options specified in the table are optional; we can specify the required options only in the `logrotate` configuration file. There are numerous options available with `logrotate`. Please refer to the man pages (`http://linux.die.net/man/8/logrotate`) for more information on `logrotate`.

Logging with syslog

Logfiles are an important component of applications that provide services to the users. An applications writes status information to its logfile while it is running. If any crash occurs or we need to enquire some information about the service, we look into the logfile. You can find lots of logfiles related to different daemons and applications in the `/var/log` directory. It is the common directory for storing log files. If you read through a few lines of the logfiles, you can see that lines in the log are in a common format. In Linux, creating and writing log information to logfiles at `/var/log` are handled by a protocol called syslog. It is handled by the `syslogd` daemon. Every standard application makes use of syslog for logging information. In this recipe, we will discuss how to make use of `syslogd` for logging information from a shell script.

Getting ready

Logfiles are useful for helping you deduce what is going wrong with a system. Hence while writing critical applications, it is always a good practice to log the progress of application with messages into a logfile. We will learn the command logger to log into log files with `syslogd`. Before getting to know how to write into logfiles, let's go through a list of important logfiles used in Linux:

Log file	Description
`/var/log/boot.log`	Boot log information.
`/var/log/httpd`	Apache web server log.
`/var/log/messages`	Post boot kernel information.
`/var/log/auth.log`	User authentication log.
`/var/log/dmesg`	System boot up messages.
`/var/log/mail.log`	Mail server log.
`/var/log/Xorg.0.log`	X Server log.

How to do it...

In order to log to the syslog file `/var/log/messages` use:

```
$ logger LOG_MESSAGE
```

For example:

```
$ logger This is a test log line
$ tail -n 1 /var/log/messages
Sep 29 07:47:44 slynux-laptop slynux: This is a test log line
```

The logfile `/var/log/messages` is a general purpose logfile. When the `logger` command is used, it logs to `/var/log/messages` by default. In order to log to the syslog with a specified tag, use:

```
$ logger -t TAG This is a message

$ tail -n 1 /var/log/messages
Sep 29 07:48:42 slynux-laptop TAG: This is a message
```

syslog handles a number of logfiles in `/var/log`. However, while logger sends a message, it uses the tag string to determine in which logfile it needs to be logged. `syslogd` decides to which file the log should be made by using the `TAG` associated with the log. You can see the tag strings and associated logfiles from the configuration files located in the `/etc/rsyslog.d/` directory.

In order to log to the system log with the last line from another logfile use:

```
$ logger -f /var/log/source.log
```

See also

- *head and tail - printing the last or first 10 lines* of *Chapter 3*, explains the head and tail commands

Monitoring user logins to find intruders

Logfiles can be used to gather details about the state of the system. Here is an interesting scripting problem statement:

We have a system connected to the Internet with SSH enabled. Many attackers are trying to log in to the system. We need to design an intrusion detection system by writing a shell script. Intruders are defined as users who are trying to log in with multiple attempts for more than two minutes and whose attempts are all failing. Such users are to be detected and a report should be generated with the following details:

- User account to which a login is attempted
- Number of attempts
- IP address of the attacker
- Host mapping for IP address
- Time range for which login attempts are performed.

Getting started

We can write a shell script that can scan through the logfiles and gather the required information from them. Here, we are dealing with SSH login failures. The user authentication session log is written to the log file `/var/log/auth.log`. The script should scan the log file to detect the failure login attempts and perform different checks on the log to infer the data. We can use the `host` command to find out the host mapping from the IP address.

How to do it...

Let's write an intruder detection script that can generate a report of intruders by using the authentication logfile 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
```

How it works...

In the `intruder_detect.sh` script, we use the `auth.log` file as input. We can either provide a log file as input to the script by using a command-line argument to the script or, by default, it reads the `/var/log/auth.log` file. We need to log details about login attempts for valid user names only. When a login attempt for an invalid user occurs, a log similar to `Failed password for invalid user bob from 203.83.248.32 port 7016 ssh2` is logged to `auth.log`. Hence, we need to exclude all lines in the log file having the word "invalid". The `grep` command with the invert option (`-v`) is used to remove all logs corresponding to invalid users. The next step is to find out the list of users for which login attempts occurred and failed. The SSH will log lines similar to `sshd[21197]: Failed password for bob1 from 203.83.248.32 port 50035 ssh2` for a failed password.

Hence we should find all the lines with words "failed password". Now all the unique IP addresses are to be found out for extracting all the log lines corresponding to each IP address. The list of IP address is extracted by using a regular expression for IP address and the `egrep` command. A `for` loop is used to iterate through IP address and the corresponding log lines are found using `grep` and are written to a temporary file. The sixth word from the last word in the log line is the user name (for example, bob1). The `awk` command is used to extract the sixth word from the last word. `NF` returns the column number of the last word. Therefore, `NF-5` gives the column number of the sixth word from the last word. We use `sort` and `uniq` commands to produce a list of users without duplication.

Now we should collect the failed login log lines containing the name of each users. A `for` loop is used for reading the lines corresponding to each user and the lines are written to a temporary file. The first 16 characters in each of the log lines is the timestamp. The `cut` command is used to extract the timestamp. Once we have all the timestamps for failed login attempts for a user, we should check the difference in time between the first attempt and the last attempt. The first log line corresponds to the first attempt and last log line corresponds to last attempt. We have used `head -1` to extract the first line and `tail -1` to extract the last line. Now we have a time stamp for first (`tstart`) and last attempt (`tends`) in string format. Using the `date` command, we can convert the date in string representation to total seconds in UNIX Epoch time (the recipe, *Getting, setting dates, and delays* of *Chapter 1*, explains Epoch time).

The variables start and end have a time in seconds corresponding to the start and end timestamps in the date string. Now, take the difference between them and check whether it exceeds two minutes (120 seconds). Thus, the particular user is termed as an intruder and the corresponding entry with details are to be produced as a log. IP addresses can be extracted from the log by using a regular expression for IP address and the `egrep` command. The number of attempts is the number of log lines for the user. The number of lines can be found out by using the `wc` command. The host name mapping can be extracted from the output of the host command by running with IP address as argument. The time range can be printed using the timestamp we extracted. Finally, the temporary files used in the script are removed.

The above script is aimed only at illustrating a model for scanning the log and producing a report from it. It has tried to make the script smaller and simpler to leave out the complexity. Hence it has few bugs. You can improve the script by using better logic.

Remote disk usage health monitor

A network consists of several machines with different users. The network requires centralized monitoring of disk usage of remote machines. The system administrator of the network needs to log the disk usage of all the machines in the network every day. Each log line should contain details such as the date, IP address of the machine, device, capacity of the device, used space, free space, percentage usage, and health status. If the disk usage of any of the partitions in any remote machine exceeds 80 percent, the health status should be set to ALERT, else it should be set to SAFE. This recipe will illustrate how to write a monitoring script that can collect details from remote machines in a network.

Getting ready

We need to collect the disk usage statistics from each machine on the network, individually, and write a log file in the central machine. A script that collects the details and writes the log can be scheduled to run everyday at a particular time. The SSH can be used to log in to remote systems to collect disk usage data.

How to do it...

First we have to set up a common user account on all the remote machines in the network. It is for the disklog program to log in to the system. We should configure auto-login with SSH for that particular user (the recipe, *Password-less auto-login with SSH* in *Chapter 7*, explains configuration of auto-login). We assume that there is a user called test in all remote machines configured with auto-login. Let's go through the shell script:

```
#!/bin/bash
#Filename: disklog.sh
#Description: Monitor disk usage health for remote systems

logfile="diskusage.log"

if [[ -n $1 ]]
then
  logfile=$1
fi

if [ ! -e $logfile ]
then
  printf "%-8s %-14s %-9s %-8s %-6s %-6s %-6s %s\n" "Date" "IP
address" "Device" "Capacity" "Used" "Free" "Percent" "Status" >
$logfile
fi

IP_LIST="127.0.0.1 0.0.0.0"
#provide the list of remote machine IP addresses

(
for ip in $IP_LIST;
do
  ssh slynux@$ip 'df -H' | grep ^/dev/ > /tmp/$$.df

  while read line;
  do
    cur_date=$(date +%D)
    printf "%-8s %-14s " $cur_date $ip
    echo $line | awk '{ printf("%-9s %-8s %-6s %-6s
%-8s",$1,$2,$3,$4,$5); }'

  pusg=$(echo $line | egrep -o "[0-9]+%")
```

```
    pusg=${pusg/\%/};
    if [ $pusg -lt 80 ];
    then
      echo SAFE
    else
      echo ALERT
    fi
    done< /tmp/$$.df
  done
  ) >> $logfile
```

We can schedule using the cron utility to run the script at regular intervals. For example, to run the script everyday at 10 am, write the following entry in the `crontab`:

```
00 10 * * * /home/path/disklog.sh /home/user/diskusg.log
```

Run the command `crontab -e`. Add the above line and save the text editor.

You can run the script manually as follows:

```
$ ./disklog.sh
```

A sample output log for the above script is as follows:

```
slynux@slynux-laptop:~/book$ cat diskusage.log
Date     IP address     Device    Capacity Used   Free   Percent Status
12/15/10 127.0.0.1      /dev/sda1 9.9G     2.4G   7.0G   26%     SAFE
12/15/10 0.0.0.0        /dev/sda1 9.9G     2.4G   7.0G   26%     SAFE
```

How it works...

In the `disklog.sh` script, we can provide the logfile path as a command-line argument or else it will use the default logfile. If the logfile does not exists, it will write the logfile header text into the new file. `-e $logfile` is used to check whether the file exists or not. The list of IP addresses of remote machines are stored in the variable `IP_LIST` delimited with spaces. It should be made sure that all the remote systems listed in the `IP_LIST` have a common user `test` with auto-login with SSH configured. A `for` loop is used to iterate through each of the IP addresses. A remote command `df -H` is executed to get the disk free usage data using the `ssh` command. It is stored in a temporary file. A `while` loop is used to read the file line by line. Data is extracted using `awk` and is printed. The date is also printed. The percentage usage is extracted using the `egrep` command and `%` is replaced with `none` to get the numeric value of percent. It is checked whether the percentage value exceeds 80. If it is less than 80, the status is set as SAFE and if greater than or equal to 80, the status is set as ALERT. The entire printed data should be redirected to the logfile. Hence the portion of code is enclosed in a subshell `()` and the standard output is redirected to the logfile.

See also

- *Scheduling with cron* of *Chapter 9*, explains the crontab command

Finding out active user hours on a system

Consider a web server with shared hosting. Many users log in to and log out of the server every day. The user activity gets logged in the server's system log. This recipe is a practice task to make use of the system logs and to find out how many hours each of the users have spent on the server and rank them according to the total usage hours. A report should be generated with the details, such as the rank, user, first logged in date, last logged in date, number of times logged in, and total usage hours. Let's see how we can approach this problem.

Getting ready

The `last` command is used to list the details about the login sessions of the users in a system. The log data is stored in the `/var/log/wtmp` file. By individually adding the session hours for each user we can find out the total usage hours.

How to do it...

Let's go through the script to find out active users and generate the report:

```
#!/bin/bash
#Filename: active_users.sh
#Description: Reporting tool to find out active users

log=/var/log/wtmp

if [[ -n $1 ]];
then
  log=$1
fi

printf "%-4s %-10s %-10s %-6s %-8s\n" "Rank" "User" "Start" "Logins"
"Usage hours"

last -f $log | head -n -2   > /tmp/ulog.$$

cat /tmp/ulog.$$ |  cut -d' ' -f1 | sort | uniq> /tmp/users.$$

(
while read user;
do
  grep ^$user /tmp/ulog.$$ > /tmp/user.$$
  seconds=0

while read t
  do
```

```
    s=$(date -d $t +%s 2> /dev/null)
    let seconds=seconds+s
  done< <(cat /tmp/user.$$ | awk '{ print $NF }' | tr -d ')(')
  firstlog=$(tail -n 1 /tmp/user.$$ | awk '{ print $5,$6 }')
  nlogins=$(cat /tmp/user.$$ | wc -l)
  hours=$(echo "$seconds / 60.0" | bc)
  printf "%-10s %-10s %-6s %-8s\n"  $user "$firstlog" $nlogins $hours
done< /tmp/users.$$
) | sort -nrk 4 | awk '{ printf("%-4s %s\n", NR, $0) }'
rm /tmp/users.$$ /tmp/user.$$ /tmp/ulog.$$
```

A sample output is as follows:

```
$ ./active_users.sh
Rank User       Start      Logins Usage hours
1    easyibaa   Dec 11     531    11437311943
2    demoproj   Dec 10     350    7538718253
3    kjayaram   Dec 9      213    4587849555
4    cinenews   Dec 11     85     1830831769
5    thebenga   Dec 10     54     1163118745
6    gateway2   Dec 11     52     1120038550
7    soft132    Dec 12     49     1055420578
8    sarathla   Nov 1      45     969268728
9    gtsminis   Dec 11     41     883107030
10   agentcde   Dec 13     39     840029414
```

How it works...

In the `active_users.sh` script, we can either provide the `wtmp` log file as a command-line argument or it will use the `defaulwtmp` log file. The `last –f` command is used to print the logfile contents. The first column in the logfile is the user name. By using `cut` we extract the first column from the logfile. Then the unique users are found out by using the `sort` and `uniq` commands. Now for each user, the log lines corresponding to their login sessions are found out using `grep` and are written to a temporary file. The last column in the last log is the duration for which the user logged a session. Hence in order to find out the total usage hours for a user, the session durations are to be added. The usage duration is in `(HOUR:SEC)` format and it is to be converted into seconds using the date command.

In order to extract the session hours for the users, we have used the `awk` command. For removing the parenthesis, `tr –d` is used. The list of usage hour string is passed to the standard input for the `while` loop using the `< ( COMMANDS )` operator. It acts as a file input. Each hour string, by using the `date` command, is converted into seconds and added to the variable `seconds`. The first login time for a user is in the last line and it is extracted. The number of login attempts is the number of log lines. In order to calculate the rank of each user according to the total usage hours, the data record is to be sorted in the descending order with usage hours as the key. For specifying the number reverse sort `-nr` option is used along with the `sort` command. `–k4` is used to specify the key column (usage hour). Finally, the output of the sort is passed to `awk`. The `awk` command prefixes a line number to each of the lines, which becomes the rank for each user.

9
Administration Calls

In this chapter, we will cover:

- Gathering information about processes
- Killing processes and send or respond to signals
- Which, whereis, file, whatis, and loadavg explained
- Sending messages to user terminals
- Gathering system information
- Using /proc – gathering information
- Scheduling with cron
- Writing and reading MySQL database from Bash
- User administration script
- Bulk image resizing and format conversion

Introduction

A GNU/Linux ecosystem consists of running programs, services, connected devices, filesystems, users, and a lot more. Having an overview of the entire system and managing the OS as a whole, according to the way we want, is the primary purpose of system administration. One should be armed with the knowledge of commonly-used commands and proper usage practices to gather system information and manage resources to write script and automation tools that perform management tasks. This chapter will introduce several commands and methods for gathering information about your system and make use of these commands to write administration 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. It is an integer. Multiple instances of the same program with the same name can be executed at a time. But they all will have different process IDs. A process consists of several attributes, such as which user owns the process, the amount of memory used by the program, the amount of CPU used by the program, and so on. This recipe will go through how to gather information about processes.

Getting ready

Important commands related to process management are `top`, `ps`, and `pgrep`. Let's see how we can gather information about processes.

How to do it...

`ps` is an important tool for gathering information about the processes. `ps` provides information on a user who owns the process, the time when a process started, command path used for executing the process, process ID (PID), the terminal it is attached with (TTY), the memory used by the process, CPU 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 (TTY) terminal. 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` (this 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 above `ps` commands are not useful since 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 TTY restriction imparted, by default, by `ps`. Usually, using `ps` without arguments prints processes that are attached to terminal only.

Run `ps -e` or `ps -ef` else `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.

The `ps` command supports several information to be displayed along with the process name and process ID. By default, `ps` shows the information as different columns. Most of them are not useful for us. We can actually specify the columns to be displayed using the `-o` flag. Hence we can print only the required columns. Different parameters associated with a process are specified with options for that parameter. The list of parameters and usage of `-o` are discussed next.

In order to display the required columns of output using `ps`, use:

```
$ ps [OTHER OPTIONS] -o parameter1,parameter2,parameter3 ..
```

Parameters for `-o` are delimited by using the comma (,) operator. It should be noted that there is no space in between the comma operator and next parameter. Mostly, the `-o` option is combined with the `-e` (every) option (`-oe`) since it should list every process running in the system. However, when certain filters are used along with `–o`, such as those used for listing the processes owned by specified users, `-e` is not used along with `–o`. Usage of `-e` with a filter will nullify the filter and it will show all process 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 file name
`cmd`	Simple command
`user`	The user who started 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

There's more...

Let's go through additional usage examples of process manipulation commands.

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 command is used as follows:

```
$ top
```

It will display several parameters along with the top CPU consuming processes.

Sorting 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 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 process ID when given command names

Suppose several instances of a command are being executed, we may need to identify the process ID 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
```

`pgrep` requires only a portion of the command name as its input argument to extract a Bash command, for example, `pgrep ash` or `pgrep bas` will also work. But `ps` requires you to type the exact command.

`pgrep` accepts many more output-filtering options. In order to specify a delimiter character for output rather than using a newline as the delimiter use:

```
$ pgrep COMMAND -d DELIMITER_STRING
$ pgrep bash -d ":"
1255:1680
```

Specify a list of owners of the user for the matching processes as follows:

```
$ pgrep -u root,slynux COMMAND
```

In this command, `root` and `slynux` are users.

Return the count of matching processes as follows:

```
$ pgrep -c COMMAND
```

Filters with ps for real user or ID, effective user or ID

With `ps`, it is possible to group processes based on the real and effective user name or ID specified. Specified arguments can be used to filter the `ps` output by checking whether each entry belongs to a specific, effective user or real user from the list of arguments and shows only the entries matching them. This can be done as follows:

- Specify an effective users list by using `-u EUSER1, EUSER2` and so on
- Specify a real users list by using `-U RUSER1, RUSER2` and so on

For example:

```
$ ps -u root -U root -o user,pcpu
```

This command will show all processes running with `root` as the effective user ID and real user ID, and will also show the user and percentage CPU usage columns.

Mostly, we find `-o` along with `-e` as `-eo`. But when filters are applied `-o` should act alone as mentioned above.

TTY filter for ps

The `ps` output can be selected by specifying the TTY to which the process is attached. Use the `-t` option to specify the TTY list as follows:

```
$ ps -t TTY1, TTY2 ..
```

For example:

```
$ ps -t pts/0,pts/1
  PID TTY          TIME CMD
 1238 pts/0    00:00:00 bash
 1835 pts/1    00:00:00 bash
 1864 pts/0    00:00:00 ps
```

Information about process threads

Usually, information about process threads are hidden in the `ps` output. We can show information about threads in the `ps` output by adding the `-L` option. Then it will show two columns NLWP and NLP. NLWP is the thread count for a process and NLP is the thread ID for each entry in PS. For example:

```
$ ps -eLf
```

Or:

```
$ ps -eLf --sort -nlwp | head
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
root       647     1   647  0   64 14:39 ?        00:00:00 /usr/sbin/
console-kit-daemon --no-daemon
root       647     1   654  0   64 14:39 ?        00:00:00 /usr/sbin/
console-kit-daemon --no-daemon
```

```
root       647     1   656  0   64 14:39 ?        00:00:00 /usr/sbin/
console-kit-daemon --no-daemon
root       647     1   657  0   64 14:39 ?        00:00:00 /usr/sbin/
console-kit-daemon --no-daemon
root       647     1   658  0   64 14:39 ?        00:00:00 /usr/sbin/
console-kit-daemon --no-daemon
root       647     1   659  0   64 14:39 ?        00:00:00 /usr/sbin/
console-kit-daemon --no-daemon
root       647     1   660  0   64 14:39 ?        00:00:00 /usr/sbin/
console-kit-daemon --no-daemon
root       647     1   662  0   64 14:39 ?        00:00:00 /usr/sbin/
console-kit-daemon --no-daemon
root       647     1   663  0   64 14:39 ?        00:00:00 /usr/sbin/
console-kit-daemon --no-daemon
```

This command lists 10 processes with maximum number of threads.

Specifying output width and columns to be displayed

We can specify the columns to be displayed in the `ps` output using the user-defined output format specifier `-o`. Another way to specify the output format is with "standard" options. Practice them according to your usage style. Try these options:

- `-f` `ps –ef`
- `u` `ps -e u`
- `ps` `ps -e w` (w stands for wide output)

Showing environment variables for a process

Understanding which environment variables a process is depended on is a very useful bit of information we might need. Whether or not a process works might be heavily dependent on the environmental variables set. We can debug and make use of environment data for fixing several problems related to running of processes.

In order to list environment variables along with `ps` entries use:

```
$ ps -eo cmd e
```

For example:

```
$ ps -eo pid,cmd  e | tail -n 3
 1162 hald-addon-acpi: listening on acpid socket /var/run/acpid.socket
 1172 sshd: slynux [priv]
 1237 sshd: slynux@pts/0
```

```
 1238 -bash USER=slynux LOGNAME=slynux HOME=/home/slynux PATH=/usr/
local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
MAIL=/var/mail/slynux SHELL=/bin/bash SSH_CLIENT=10.211.55.2 49277 22
SSH_CONNECTION=10.211.55.2 49277 10.211.55.4 22 SSH_TTY=/dev/pts/0
TERM=xterm-color LANG=en_IN XDG_SESSION_COOKIE=d1e96f5cc8a7a3bc3a0a73e44c
95121a-1286499339.592429-1573657095
```

An example of where this type of environment tracing can come in handy is in tracing problems with the apt-get package manager. If you use an HTTP proxy to connect to the internet, you may need to set environment variables `http_proxy=host:port`. But sometimes even when it is set, the `apt-get` command will not select the proxy and hence it returns an error. Then you can actually look at an environment variable and track the issue.

We may need some applications to be run automatically with scheduling tools such as `crontab`. But it might be dependent on some environment variables. Suppose we want to open a GUI-windowed application at a given time. We schedule it using `crontab` at a specified time. However, you will notice that the application will not start at a given time if an entry like the following is given:

```
00 10 * * * /usr/bin/windowapp
```

This is because a windowed application always depends on the DISPLAY environment variable. Environment variables need to be passed to the application.

First run `windowapp` manually, and then run `ps -C windowapp -eo cmd e`.

Find out the environment variables. Prefix them before a command name appears in `crontab`. The issue will get resolved.

Modify the entry as follows:

```
00 10 * * * DISPLAY=:0 /usr/bin/windowapp
```

`DISPLAY=:0` can be obtained from the `ps` output.

See also

- *Scheduling with cron*, explains how to schedule tasks

Killing processes and send or respond to signals

Termination of processes is an important task we always come across. Sometimes we may 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 running process to perform some action. Termination of a program is also performed by using the signals technique. This recipe is an introduction to signals and the usage of signals.

Getting ready

Signals are an inter-process mechanism available in Linux. We can interrupt a process by using a specific signal. Each signal is associated with an integer value. When a process receives a signal, it responds by executing a signal handler. In Shell scripting also, it is possible to send and receive signals and respond according to the signals. `KILL` is a signal used to terminate a process. Events such as *Ctrl + C*, *Ctrl + Z* are also types of signals. The `kill` command is used to send signals to processes and the `trap` command is used to handle the received signals.

How to do it...

In order to list all the signals available, use:

```
$ kill -l
```

It will print the signal number and signal names.

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.

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 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

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's more...

Let's walk through additional commands used for terminating and signalling processes.

kill family of commands

The `kill` command takes the process ID as argument. There are also a few other commands in the `kill` family that accept the command name as 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 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
```

In order to ask interactively before killing processes, use the `-i` argument along with `killall`.

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. `SIGNAL` name is not supported with `pkill`.

It provides many of the same options that the `kill` command does. Check the `pkill` manpages for more details.

Capturing and responding to signals

`trap` is a command used to assign signal handler to signals in a script. Once a function is assigned to a signal using the `trap` command, while the script runs and it receives a signal, this function is executed upon reception of a corresponding signal.

The syntax is as follows:

```
trap 'signal_handler_function_name' SIGNAL LIST
```

`SIGNAL LIST` is delimited by space. It can be a signal number or a signal name.

Let's write a shell script that responds to the `SIGINT` signal:

```
#/bin/bash
#Filename: sighandle.sh
#Description: Signal handler

function handler()
{
  echo Hey, received signal : SIGINT
}

echo My process ID is $$
# $$ is a special variable that returns process ID of current process/
script
trap 'handler' SIGINT
#handler is the name of the signal handler function for SIGINT signal

while true;
do
  sleep 1
done
```

Run this script in a terminal. When the script is running, if you press *Ctrl + C* it will show the message by executing the signal handler associated with it. *Ctrl + C* is a `SIGINT` signal.

The `while` loop is used to keep the process running without going to termination by using an infinite loop. Thus the process is kept running infinitely so that it can respond to the signals that are sent to the process asynchronously by another process. The loop that is used to keep the process alive infinitely is often called as the event loop. If an infinite loop is not available, the script will terminate after executing the statements. But for signal handler scripts, it has to wait and respond to the signals.

We can send a signal to the script by using the `kill` command and the process ID of the script:

```
$ kill -s SIGINT PROCESS_ID
```

The `PROCESS_ID` of the above script will be printed when it is executed. Or you can find it out by using the `ps` command

If no signal handlers are specified for signals, it will call the default signal handlers assigned by the operating system. Generally, pressing *Ctrl + C* will terminate a program since the default handler provided by the operating system will terminate the process. But the custom handler defined here specifies a custom action upon receipt of the signal.

We can define signal handlers for any signals available (`kill -l`), by using the `trap` command. It is also possible to set a single signal handler for multiple signals.

which, whereis, file, whatis, and loadavg explained

This recipe aims to explain a few commands we come across. Understanding these commands is helpful for users.

How to do it...

Let's go through each of the commands and their usage examples.

- **which**

  The `which` command is used to find the location of a command. We type commands in the terminal without knowing the location where the executable file is stored.

  When we type a command, the terminal looks for the command in a set of locations and executes the executable file if found at the location. This set of locations is specified using an environment variable `PATH`. For example:

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

  We can export `PATH` and can add our own locations to be searched when command names are typed. For example, to add `/home/slynux/bin` to `PATH` use the following command:

  ```
  $ export PATH=$PATH:/home/slynux/bin
  # /home/slynux/bin is added to PATH
  ```

  The `which` command outputs the location of the command given as argument. For example:

  ```
  $ which ls
  /bin/ls
  ```

- **whereis**

  `whereis` is similar to the `which` command. But it not only returns the path of the command, it will also print the location of the manpage, if available, and also the path of the source code for the command if available. For example:

```
$ whereis ls
ls: /bin/ls /usr/share/man/man1/ls.1.gz
```

- **file**

  The `file` command is an interesting and frequently-used command. It is used for determining the file type:

```
$ file FILENAME
```

  This will print the details of the file regarding its file type.

  An example is as follows:

```
$ file /bin/ls
/bin/ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.15,
stripped
```

- **whatis**

  The `whatis` command outputs a one-line description of the command given as an argument. It parses information from the manpage. For example:

```
$ whatis ls
ls (1)               - list directory contents
```

apropos

Sometimes we need to search if some command related to a word exists. Then we can search the manpages for strings in the command. For this we can use:

```
apropos COMMAND
```

- **Load average**

  Load average is an important parameter for total load on the running system. It specifies the average of the total number of runnable processes on the system. It is specified by three values. The first value indicates the average in one minute, the second indicates average in five minutes, and third indicates the average in 15 minutes.

  It can be obtained by running `uptime`. For example:

```
$ uptime
 12:40:53 up  6:16,  2 users,  load average: 0.00, 0.00, 0.00
```

Sending messages to user terminals

A system administrator may need to send messages to the terminal screen of every user or a specified user on all the machines over a network. This recipe is a guide to perform this task.

Getting ready

`wall` is a command that is used to write messages on the terminals of all logged in users. It can be used to convey messages to all logged in users in a server or multiple access machines. Sending messages to all users may, sometimes, not be useful. We may need to send messages to specific users or a specific terminal. Terminals are treated as devices in a Linux system and hence these opened terminals will have a corresponding device node file at `/dev/pts/`. Writing data to a specific device will display messages on the corresponding terminal.

How to do it...

In order to broadcast a message to all users and all logged in terminals, use:

```
$ cat message | wall
```

Or:

```
$ wall < message
Broadcast Message from slynux@slynux-laptop
        (/dev/pts/1) at 12:54 ...

This is a message
```

The message outline will show who sent the message (which user and which host). The message gets OR is displayed to the current terminal if some other users send a message, only if the "write message" option is enabled. By default, in most distros "write message" is enabled by default. If the sender of the message is root, then the message gets displayed on the screen irrespective of whether the "write message" option is enabled or disabled by the user.

In order to enable write messages use:

```
$ mesg y
```

In order to disable write messages use:

```
$ mesg n
```

Let's write a script for sending messages specifically to a given user's terminal:

```
#/bin/bash
#Filename: message_user.sh
#Description: Script to send message to specified user logged
terminals.
USER=$1
devices=`ls /dev/pts/* -l | awk '{ print $3,$9 }' | grep $USER | awk
'{ print $2 }'`
for dev in $devices;
do
  cat /dev/stdin > $dev
done
```

Run the script as:

```
./message_user.sh USERNAME < message.txt
# Pass message through stdin and username as argument
```

The output will be as follows:

```
$ cat message.txt
A message to slynux. Happy Hacking!
# ./message_user.sh slynux  < message.txt
# Run message_user.sh as root since the message is to be send to a
specifc user.
```

Now, slynux's terminal will receive the message text.

How it works...

The `/dev/pts` directory will contain character devices corresponding to each of the logged in terminals on the system. We can find out who logged into which terminal by looking at the owner of the device files. The `ls -l` output will contain the owner name and the device path. This information is extracted by using `awk`. Then it uses `grep` to extract the lines corresponding to specified user only. The username is accepted as the first argument for the script as stored as variable USER. Then a list of terminals for a given user is made. A `for` loop is used to iterate through each device path. `/dev/stdin` will contain standard input data passed to the current process. Therefore, by reading `/dev/stdin`, data is read and redirected to the corresponding terminal (TTY) devices. Hence the message gets displayed.

Gathering system information

Collecting information about the current system from the command line is very important in logging system data. The different system information data includes hostname, kernel version, Linux distro name, CPU information, memory information, disk partition information, and so on. This recipe will show you different sources in a Linux system to gather information about the system.

How to do it...

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

```
$ hostname
```

Or:

```
$ uname -n
```

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

```
$ uname -a
```

In order to print the kernel release, use:

```
$ uname -r
```

Print the machine type as follows:

```
$ uname -m
```

In order to print details about CPU details, use:

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

In order to extract the processor name, use:

```
$ cat /proc/cpuinfo | head -n 5 | tail -1
```

The fifth line contains the processor name. Therefore, the first five lines are extracted first. Then the last one line is extracted to print processor name.

Print details about 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
```

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

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

Or:

```
$ fdisk -l
```

Get the entire details about the system as follows:

```
$ lshw
```

Using /proc – gathering information

`/proc` is an in-memory pseudo filesystem available on GNU/Linux operating systems. It was introduced to provide an interface to read several system parameters from a user space. It is very interesting and we can gather lots of information from it. Let's see few of the features available with the `proc` filesystem.

How to do it...

If you look at `/proc`, you can see several files and directories. Some of them are already explained in another recipe in this chapter. You can simply `cat` files in `/proc` and the subdirectories to get information. All of them are well-formatted text.

There will be a directory in `/proc` for every process that is running on the system. The directory name for a process in `/proc` is same as that of process ID of that process.

Suppose for Bash, the process ID is 4295 (`pgrep bash`), `/proc/4295` will exist. Each of the directories corresponding to the process will contain a lot of information regarding that process. Few of the important files in `/proc/PID` are as follows.

- `environ`—contains environment variables associated with that process.

  By `cat /proc/4295/environ` we can display all the environment variables passed to that process.
- `cwd`—is a symlink to a working directory of the process.
- `exe`—is a symlink to the running executable for the current process.

  ```
  $ readlink /proc/4295/exe
  /bin/bash
  ```
- `fd`—is the directory consisting of entries on file descriptors used by the process.

Scheduling with cron

It is a common requirement to schedule the execution of scripts at a given time or at given time intervals. The GNU/Linux system comes with different utilities for scheduling tasks. `cron` is such a utility that allows tasks to automatically run in the background of the system at regular intervals by use of the `cron` daemon. The `cron` utility makes use of a file called "cron table" that stores a list of schedule of scripts or commands to be executed and the time at which they are to be executed. It is a very useful utility. A common example usage is to schedule downloads of files from the Internet during the free hours (certain ISPs provide free usage - usually during the night when most people are sleeping). Users are not required to wake up in the night to start the download. Users can write a cron entry and schedule the download. You can also schedule to drop the Internet connection automatically and shut down the system when the free usage hours end.

Getting ready

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 schedule entries to the cron schedule domain. A cron schedule is a simple text file. Each user has his or her own cron schedule. A cron schedule is often called a cron job.

How to do it...

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. 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 hours 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 comma in the corresponding time field (for example, for running the command at the fifth minute and tenth minute, enter `5,10` in the minutes 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 entry can consist of one or more lines of cron jobs. Each line in the cron table entry is a single job. For example:

- Let's write a sample `crontab` entry for illustration:

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

  This cron job will execute the `test.sh` script at the second minute of all hours on all days.

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

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

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

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

- Shut down the computer at 2am everyday as follows:

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

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

When you run the `crontab` manually, use the `-e` option to enter the cron job:

```
$ 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 job and save it. This cron job will be scheduled and executed at specified time intervals.

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

1. Create a text file (for example, `task.cron`) and write the cron job.

   Then run the `crontab` with the filename as the command argument:

   ```
   $ crontab task.cron
   ```

2. By using the next method we can 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`.

Cron jobs are executed with privileges with which the `crontab` command is 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 the cronjob 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. If your command requires certain environment variables to be set for running, you should explicitly set the environment variables.

There's more...

The `crontab` command has more options. Let's see a few of them.

Specifying environment variables

Many of the commands require environment variables to be set properly for execution. We can set environment variables by inserting a line with variable assignment statement in the cron table of the user.

For example, if you are using a proxy server for connecting to the Internet, to schedule a command that uses Internet you have to set the HTTP proxy environment variable `http_proxy`. It can be done as follows:

```
crontab<<EOF
http_proxy=http://192.168.03:3128
00 * * * * /home/slynux/download.sh
EOF
```

Viewing the cron table

We can list the existing cronjobs 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 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 crontable 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.

Writing and reading MySQL database from Bash

MySQL is a widely used Database System. Usually, MySQL databases are used as the storage systems for applications that are written in languages such as PHP, Python, C++, and so on. Accessing and manipulating MySQL databases from shell script will be interesting. We can write scripts to write contents from a text file or CSV (Comma Separated Values) into tables and interact with the MySQL database to read and manipulate data. For example, we can read all the e-mail addresses stored in a guestbook program's database by running a query from the shell script. In this recipe, we will see how to read and write to a MySQL database from Bash. For illustration, here is an example problem:

I have a CSV file containing details of students. I need to insert the contents of the file to a database table. From this data, I need to generate a separate rank list for each department.

Getting ready

In order to handle MySQL databases, you should have mysql-server and mysql-client packages installed on your system. These tools do not come with a Linux distribution by default.
Since MySQL comes with a username and password for authentication, you should have a username and password to run the scripts.

How to do it...

The above problem can be solved using Bash utilities such as `sort`, `awk`, and so on. Alternately, we can solve it by using an SQL database table. We will write three scripts for the purpose of creating a database and table, inserting student data into the table, and reading and displaying processed data from the table.

Create the database and table script as follows:

```
#!/bin/bash
#Filename: create_db.sh
#Description: Create MySQL database and table

USER="user"
PASS="user"

mysql -u $USER -p$PASS <<EOF 2> /dev/null
CREATE DATABASE students;
EOF

[ $? -eq 0 ] && echo Created DB || echo DB already exist

mysql -u $USER -p$PASS students <<EOF 2> /dev/null
CREATE TABLE students(
id int,
name varchar(100),
mark int,
dept varchar(4)
);
EOF

[ $? -eq 0 ] && echo Created table students || echo Table students
already exist

mysql -u $USER -p$PASS students <<EOF
DELETE FROM students;
EOF
```

The script for inserting data into the table is as follows:

```
#!/bin/bash
#Filename: write_to_db.sh
#Description: Read from CSV and write to MySQLdb

USER="user"
PASS="user"

if [ $# -ne 1 ];
then
  echo $0 DATAFILE
  echo
  exit 2
fi

data=$1

while read line;
do

  oldIFS=$IFS
```

```
    IFS=,
    values=($line)
    values[1]="\"`echo ${values[1]} | tr ' ' '#' `\""
    values[3]="\"`echo ${values[3]}`\""

    query=`echo ${values[@]} | tr ' #' ', ' `
    IFS=$oldIFS

    mysql -u $USER -p$PASS students <<EOF
INSERT INTO students VALUES($query);
EOF

done< $data
echo Wrote data into DB
```

The script for the query from the database is as follows:

```
#!/bin/bash
#Filename: read_db.sh
#Description: Read from the database

USER="user"
PASS="user"

depts=`mysql -u $USER -p$PASS students <<EOF | tail -n +2
SELECT DISTINCT dept FROM students;
EOF`

for d in $depts;
do

echo Department : $d

result="`mysql -u $USER -p$PASS students <<EOF
SET @i:=0;
SELECT @i:=@i+1 as rank,name,mark FROM students WHERE dept="$d" ORDER
BY mark DESC;
EOF`"

echo "$result"
echo

done
```

The data for the input CSV file (`studentdata.csv`) is as follows:

```
1,Navin M,98,CS
2,Kavya N,70,CS
3,Nawaz O,80,CS
4,Hari S,80,EC
5,Alex M,50,EC
```

```
6,Neenu J,70,EC
7,Bob A,30,EC
8,Anu M,90,AE
9,Sruthi,89,AE
10,Andrew,89,AE
```

Execute the scripts in the following sequence:

```
$ ./create_db.sh
Created DB
Created table students

$ ./write_to_db.sh studentdat.csv
Wrote data into DB

$ ./read_db.sh
Department : CS
rank  name  mark
1  Navin M  98
2  Nawaz O  80
3  Kavya N  70

Department : EC
rank  name  mark
1  Hari S  80
2  Neenu J 70
3  Alex M  50
4  Bob A   30

Department : AE
rank  name  mark
1  Anu M    90
2  Sruthi   89
3  Andrew   89
```

How it works...

We will now see the explanation of the above scripts one by one. The first script `create_db.sh` is used to create database called `students` and a table named `students` inside it. We need the MySQL username and password to access or modify data in the DBMS. The variables `USER` and `PASS` are used to store the username and password. The `mysql` command is used for MySQL manipulations. The `mysql` command can specify the username by using `-u` and the password by using `-pPASSWORD`. The other command argument for the `mysql` command is the database name. If a database name is specified as an argument to the `mysql` command, it will use that for database operations, else we have to explicitly specify in the SQL query about which database is to be used with the `use database_name` query. The `mysql` command accepts the queries to be executed through standard input (`stdin`). The convenient way of supplying multiple lines through `stdin` is by using the `<<EOF` method. The text that appears in between `<<EOF` and `EOF` is passed to `mysql` as standard input. In the `CREATE DATABASE` query, we have redirected `stderr` to `/dev/null` in order to prevent displaying an error message. Also, in the table creation query, we have redirected `stderr` to `/dev/null` to ignore any errors that occur. Then we check the exit status for the `mysql` command by using the exit status variable `$?` to know if a table or database already exists. If the database or table already exists, a message is displayed to notify that. Else we will create them.

The next script `write_to_db.sh` accepts a filename of the student data CSV file. We read each line of the CSV file by using the `while` loop. So in each iteration a line with comma separated values will be received. We then need to formulate the values in the line to an SQL query. For that, the easiest way to store data items in the comma separated line is by using an array. We know that an array assignment is in the form `array=(val1 val2 val3)`. Here the space character is the **Internal Field Separator** (**IFS**). We have a line with comma separated values, hence by changing the IFS to a comma, we can easily assign values to the array (`IFS=,`). The data items in the comma separated line are `id`, `name`, `mark`, and `department`. `id` and `mark` are integer values whereas `name` and `dept` are strings (strings must be quoted). Also the name can contain space characters. Space can conflict with the Internal Field Separator. Hence we should replace the space in the name with some character (#) and replace it later after formulating the query. In order to quote the strings, the values in the array are prefixed and suffixed with `\"`. The `tr` is used to substitute space in the name to #. Finally, the query is formed by replacing the space character with comma and replacing # with space and this query is executed.

The third script `read_db.sh` is used to find out the department and print the rank list of students for each department. The first query is used to find distinct names of departments. We use a `while` loop to iterate through each department and run the query to display student details in the order of highest marks. `SET @i=0` is an SQL construct used to set the variable `i=0`. On each row it is incremented and is displayed as the rank of the student.

User administration script

GNU/Linux is a multi user operating system. Many users can log in and perform several activities at a time. There are several administration tasks that are handled with user management. The tasks includes 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. This recipe aims at writing a user management tool that can handle all of these tasks.

How to do it...

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
```

How it works...

The `user_adm.sh` script can be used to perform many user management tasks. You can follow the `usage()` text for the proper usage of the script. A function `usage()` is defined to display how to execute the script with different options for the user when any of the parameters given by user gets wrong or has run the `–usage` parameter. A case statement is used to match the command arguments and execute the corresponding commands according to that. The valid command options for the `user_adm.sh` script are: `-adduser`, `-deluser`, `-shell`, `-disable`, `-enable`, `-expiry`, `-passwd`, `-newgroup`, `-delgroup`, `-addgroup`, `-details`, and `-usage`. When the `*)` case is matched, it means its a wrong option and hence `usage()` is invoked. For each match case, we have used `[ $# -ne 3 ] && usage`. It is used for checking number of arguments. If the number of command arguments are not equal to required number, the `usage()` function is invoked and the script will exit without executing further. In order to run the user management commands, the script needs to be run as root. Hence a check for user ID 0 (the root has user ID 0) is performed. If the user has a non-zero user ID, this means it is executing as non-root. Hence a message to run as root is displayed and the script exits.

Let's explain each case one by one:

- `-useradd`:

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

  ```
  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 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 new password.

- `-newgroup` and `addgroup`:

  The `addgroup` command will add a new usergroup to the 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.

Bulk image resizing and format conversion

All of us use digital cameras and download photos from the cameras as well as the Internet. When we need to deal with large number of image files, we can use scripts to easily perform actions on the files in bulk. A regular task we come across with photos is resizing the file. Also, format conversion from one image format to another comes to use (for example, JPEG to PNG conversion). When we download pictures from a camera, the large resolution pictures take a large size. But we may need pictures of lower sizes that are convenient to store and e-mail over the internet. Hence we resize it to lower resolutions. This recipe will discuss how to use scripts for image management.

Getting ready

Imagemagick is an excellent tool for manipulating images that can work across several image formats and different constructs with rich options. Most of the GNU/Linux distributions don't come with Imagemagick installed. You need to manually install the package. `convert` is the command that we will use frequently.

How to do it..

In order to convert from one image format to another image format use:

```
$ convert INPUT_FILE OUTPUT_FILE
```

For example:

```
$ convert file1.png file2.png
```

We can resize an image size to a specified image size either by specifying the scale percentage or by specifying width and height of the output image.

Resize the image by specifying the `WIDTH` or `HEIGHT` as follows:

```
$ convert image.png -resize WIDTHxHEIGHT image.png
```

For example:

```
$ convert image.png -resize 1024x768 image.png
```

It is required to provide either `WIDTH` or `HEIGHT` so that the other will be automatically calculated and resized so as to preserve the image size ratio:

```
$ convert image.png -resize WIDTHx image.png
```

For example:

```
$ convert image.png -resize 1024x image.png
```

Resize the image by specifying the percentage scale factor as follows:

```
$ convert image.png -resize "50%" image.png
```

Let's see a script for image management:

```
#!/bin/bash
#Filename: image_help.sh
#Description: A script for image management

if [ $# -ne 4 -a $# -ne 6 -a $# -ne 8 ];
then
  echo Incorrect number of arguments
  exit 2
fi

while [ $# -ne 0 ];
do
  case $1 in
  -source) shift; source_dir=$1 ; shift ;;
  -scale) shift; scale=$1 ; shift ;;
  -percent) shift; percent=$1 ; shift ;;
  -dest) shift ; dest_dir=$1 ; shift ;;
  -ext) shift ; ext=$1 ; shift ;;
  *) echo Wrong parameters; exit 2 ;;
  esac;
done

for img in `echo $source_dir/*` ;
do
  source_file=$img
  if [[ -n $ext ]];
  then
    dest_file=${img%.*}.$ext
  else
    dest_file=$img
  fi

  if [[ -n $dest_dir ]];
  then
    dest_file=${dest_file##*/}
    dest_file="$dest_dir/$dest_file"
  fi

  if [[ -n $scale ]];
  then
    PARAM="-resize $scale"
```

```
    elif [[ -n $percent ]];
    then
      PARAM="-resize $percent%"
    fi
    echo Processing file : $source_file
    convert $source_file $PARAM $dest_file
  done
```

The following is a sample output, to scale the images in the directory `sample_dir` to `20%` size:

```
$ ./image_help.sh -source sample_dir -percent 20%
Processing file :sample/IMG_4455.JPG
Processing file :sample/IMG_4456.JPG
Processing file :sample/IMG_4457.JPG
Processing file :sample/IMG_4458.JPG
```

In order to scale the images to width 1024 use:

```
$ ./image_help.sh -source sample_dir –scale 1024x
```

Change the files to PNG format by adding `–ext png` along with the above commands.

Scale or convert files with specified destination directory as follows:

```
$ ./image_help.sh -source sample -scale 50% -ext png -dest newdir
# newdir is the new destination directory
```

How it works...

The above `image_help.sh` script can accept several command-line arguments, such as `-source`, `-percent`, `-scale`, `–ext`, and `-dest`. A brief explanation of each is as follows:

- The `-source` parameter is used to specify the source directory for the images.
- The `–percent` parameter is used to specify the scale percent and `–scale` is used to specify scale width and height.
- Either `–percent` or `–scale` is used. Both of them do not appear simultaneously.
- The `–ext` parameter is used to specify the target file format. `–ext` is optional; if it is not specified, format conversion is not performed.
- The `–dest` parameter is used to specify the destination directory for scale or conversion of image files. `–dest` is optional. If `–dest` is not specified, the destination directory will be same as the source directory. As the first step in the script, it checks whether the number of command arguments given to the script are correct. Either 4 or 6 or 8 parameters can appear.

Now, by using a `while` loop and case statement, we will parse the command-line arguments corresponding to variables. `$#` is a special variable that returns the number of arguments. The `shift` command shifts the command arguments one position to left, so that on each execution of shift, we can access command arguments one by one, by using the same `$1` variable rather than using `$1`, `$2`, `$3`, and so on. The case statement matches the value of `$1`. It is like a switch statement in the C programming language. When a case is matched, the corresponding statements are executed. Each match case statement is terminated with `;;`. Once all the parameters are parsed in variables `percent`, `scale`, `source_dir`, `ext`, and `dest_dir`, a `for` loop is used to iterate through path of each file in the source directory and the corresponding action to convert file is performed.

If the variable `ext` is defined (if `-ext` is given in the command argument), the extension of the destination file is changed from `source_file.extension` to `source_file.$ext`. In the next statement it checks whether the `-dest` parameter is provided. If the destination directory is specified, the destination file path is crafted by replacing the directory in source path with destination directory by using file name slicing. In the next statement, it crafts the parameter to the `convert` command for performing resize (`-resize widthx` or `-resize perc%`). After the parameters are crafted, the `convert` command is executed with proper arguments.

See also

- *Slicing filenames based on extension* of *Chapter 2,* explains how to extract portion of file name

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z