INTRODUCTION TO LINUX

MARCEL JAR

CONTENTS

I INTRODUCTION TO COMMAND LINE INTERFACES 1

1 WHAT IS AN OPERATING SYSTEM 2
- 1.1 Operating System Components . . . 2
- 1.2 Computer Architecture Basics . . . 4
  - 1.2.1 CPU . . . 4
  - 1.2.2 RAM . . . 4
  - 1.2.3 HD . . . 5
  - 1.2.4 Peripherals . . . 5
- Exercises . . . 5

2 A BRIEF HISTORY OF LINUX 6
- 2.1 Unix . . . 6
- 2.2 GNU . . . 7
- 2.3 Linux . . . 8

3 LINUX DISTRIBUTIONS 10
- 3.1 Distribution Components . . . 10
  - 3.1.1 Desktop Environment . . . 10
  - 3.1.2 Software Management . . . 12
  - 3.1.3 Default Applications . . . 12
- 3.2 Popular Distributions . . . 12
  - 3.2.1 Ubuntu, Mint, Kubuntu . . . 13
  - 3.2.2 Debian . . . 13
  - 3.2.3 SUSE, OpenSUSE . . . 13
  - 3.2.4 Red Hat, Fedora, CentOS . . . 13
- Exercises . . . 14

II INTRODUCTION TO COMMAND LINE INTERFACES 15

4 COMMAND LINE INTERFACE, TERMINAL, AND SHELL 16
- 4.1 Terminology . . . 17
- 4.2 Shells . . . 18
- Exercises . . . 18

5 BASIC SHELL COMMANDS I 19
- 5.1 Accessing a Terminal Emulator . . . 19
- 5.2 Terminal Basics . . . 19
- 5.3 **`date`** . . . 20
- 5.4 **`whoami`** . . . 20
- 5.5 **`pwd`** . . . 21
- 5.6 **`ls`** . . . 21
- 5.7 **`tree`** . . . 22
- 5.8 **`cd`** . . . 23
- 5.9 Relative and Absolute Paths . . . 23
  - 5.9.1 Relative Path . . . 24
  - 5.9.2 Absolute Path . . . 25

5.10 **clear** . . . 25
5.11 Command History . . . 26
Exercises . . . 26
6 BASIC SHELL COMMANDS II 28
6.1 **mkdir** . . . 28
6.2 **rmdir** . . . 29
6.3 **touch** . . . 29
6.4 **rm** . . . 30
6.5 **mv** . . . 31
6.5.1 Moving files and folders accross directories . . . 31
6.5.2 Renaming files and folders . . . 32
6.6 **cp** . . . 32
6.6.1 Copying files within the working directory . . . 33
6.6.2 Copying files to other directories . . . 33
Exercises . . . 34
7 GETTING HELP 36
7.1 **help** . . . 36
7.2 **man** . . . 37
7.2.1 Sections . . . 39
7.3 **whatis** . . . 40
7.4 **apropos** . . . 40
7.5 **info** . . . 40
Exercises . . . 41
8 READING AND EDITING TEXT FILES FROM THE SHELL 43
8.1 Reading text files . . . 43
8.1.1 **more** . . . 43
8.1.2 **Less** . . . 44
8.2 Editing Text Files . . . 46
8.2.1 **nano** . . . 46
8.2.2 **vi** and **vim** . . . 47
Exercises . . . 52

III ADVANCED COMMAND LINE INTERFACES 53
9 LINUX FILE SYSTEMS 54
9.1 File System: Data Storage Format . . . 54
9.1.1 Linux File System Evolution . . . 54
9.1.2 Linux File System Format . . . 55
9.1.3 **stat** . . . 55
9.1.4 Directories in Linux . . . 56
9.1.5 Accessing data and metadata . . . 57
9.1.6 Performing actions on files, directories, and inodes . . . 57
9.1.7 **shred** . . . 58
9.2 Directories Hierarchy . . . 59
Exercises . . . 61
10 FILE LINKS 63

10.1 Hard Links . . . 63
10.1.1 **ln** . . . 63
10.1.2 Hard Link Properties . . . 66
10.1.3 Hard link limitations . . . 66
10.1.4 Hard Link usage . . . 66
10.2 Soft Links . . . 67
10.2.1 **ln -s** . . . 67
10.2.2 Soft link properties . . . 70
Exercises . . . 70
11 FILE GLOBBING: USING WILDCARDS 72
11.1 star - * . . . 72
11.2 Question Mark - **?** . . . 74
11.3 Square Brackets - **[]** . . . 75
11.4 Escaping special characters . . . 75
Exercises . . . 76
Exercises . . . 76
12 GREP AND REGULAR EXPRESSIONS 77
12.1 **grep** . . . 77
12.1.1 grep options . . . 78
12.2 Regular Expressions . . . 78
12.2.1 Examples of **grep** commands using regex . . . 79
Exercises . . . 81
13 BASIC FILTERS 82
13.1 **cat** . . . 82
13.2 **head** . . . 83
13.3 **tail** . . . 84
13.4 **tac** . . . 84
13.5 **sort** . . . 85
13.6 **cut** . . . 87
13.7 **sed** . . . 88
Exercises . . . 89
14 AWK, RENAME, AND FIND 91
14.1 **awk** . . . 91
14.2 **rename** . . . 93
14.3 **find** . . . 94
Exercises . . . 96
15 REDIRECTION AND PIPING 97
15.1 Redirection . . . 97
15.1.1 Redirection Syntax . . . 98
15.2 Piping . . . 99
15.2.1 Piping Application Scenario . . . 100
15.2.2 Piping Syntax . . . 101
15.2.3 Piping examples . . . 101
15.2.4 Piping and Redirection . . . 102
15.2.5 **tee** . . . 102
Exercises . . . 103

IV SCRIPTING 104
16 INTRODUCTION TO SCRIPTING 105
16.1 Hello World Script . . . 105
16.1.1 **shebang** line . . . 105
16.1.2 Comments . . . 106
16.1.3 **echo** . . . 106
16.2 Using the terminal to run scripts . . . 107
16.2.1 **PATH** variable . . . 107
16.2.2 Granting permission to execute a Script . . . 108
16.3 Variables . . . 109
16.3.1 Mathematical Operations . . . 109
16.4 Environments . . . 110
16.4.1 **source** . . . 110
16.5 Gathering user's input . . . 111
16.5.1 **read** . . . 111
16.5.2 Passing arguments to a script . . . 112
Exercises . . . 112
17 LOGICAL EXPRESSIONS 114
17.1 Basic **if/else** syntax . . . 114
17.2 Logical Expressions . . . 115
17.2.1 String Expressions . . . 115
17.2.2 Integer Expressions . . . 116
17.2.3 File Expressions . . . 117
17.3 Combining expressions using && and || . . . 118
17.4 Basic **if/elif/else** syntax . . . 119
17.5 Nested logical expressions . . . 121
Exercises . . . 123
18 LOOPS 125
18.1 **for** Loops . . . 125
18.1.1 Using wildcards . . . 126
18.1.2 Using brace expansion . . . 126
18.1.3 Using c-style **for** loops . . . 127
18.2 **while** Loops . . . 128
18.2.1 Reading files line by line . . . 129
18.2.2 **until** Loops . . . 130
18.3 **break** . . . 130
18.4 **continue** . . . 131
18.5 Nested Loops . . . 132
Exercises . . . 132
19 FUNCTIONS 134
19.1 Escope of Variables . . . 137
19.2 Passing arguments to Functions . . . 137
19.3 Returning an Exit Status from a Function . . . 138
Exercises . . . 140
20 ARRAYS 142
20.1 Creating Arrays . . . 142

20.2 Accessing elements of an Array . . . 142
20.3 Editing the contents of an array . . . 144
20.4 Deleting an array . . . 145
Exercises . . . 145
21 PERMISSIONS, USERS, AND GROUPS 146
21.1 File and Folder Permissions . . . 146
21.1.1 File permissions . . . 147
21.1.2 Directory Permissions . . . 149
21.2 Users . . . 149
21.2.1 Adding new users . . . 150
21.2.2 Changing users . . . 152
21.2.3 Editing existing users . . . 153
21.2.4 Removing existng users from the system . . . 153
21.3 Groups . . . 154
21.3.1 Primary Groups . . . 154
21.3.2 Creating, configuring, and deleting groups . . . 155
21.3.3 Adding and removing users from groups . . . 156
21.3.4 Granting group ownership to regular users . . . 156
Exercises . . . 156

LIST OF FIGURES

Figure 1.1 Basic computer architecture. 4
Figure 4.1 Notepad application for Windows 7. 16
Figure 5.1 Directory tree. 24
Figure 5.2 Directory tree for questions 9, 10, and 11. . . . 27
Figure 6.1 Directory tree for questions 4 and 5. 35
Figure 8.1 Vi operational modes and the keys required to change modes. 49
Figure 9.1 Sequence of steps to access a text file. 58
Figure 10.3 Directory tree for questions 2 and 3. 70
Figure 15.1 Visual representation of data streams. 98
Figure 21.1 File Permissions. 147

LIST OF TABLES

Table 1.1 Windows OS Components. 3
Table 4.1 List of Linux Shells. 18
Table 5.1 Long list information for the **`seneca.pdf`** file. . 22
Table 7.1 Manual Page Sections 39
Table 7.2 Shortcuts to navigate info pages 41
Table 8.1 Less navigation keys. 45
Table 8.2 Information displayed in **`vim`**. 48
Table 8.3 Keys to switch to insert mode. 49
Table 8.4 List of some **command mode** important commands. 50
Table 8.5 List of some extended mode important commands. 51
Table 9.1 Contents of the **`/home/marcel`** directory file. . . 56
Table 9.2 Basic directory contents according to the FHS. 61
Table 12.1 **`grep`** options. 78
Table 12.2 Special characters for regex. 79
Table 13.1 **`sort`** options. 85
Table 14.1 **`awk`** patterns. 92
Table 14.2 **`find`** search criteria types. 95
Table 15.1 Bash data streams. 97
Table 17.1 String Expressions. 115
Table 17.2 Integer Expressions. 116
Table 17.3 File Expressions. 117
Table 21.1 Types of file permissions. 147

Table 21.2 File types. 148
Table 21.3 Types of dirctory permissions. 149
Table 21.4 Description of the fields in the **passwd** file. . . 151
Table 21.5 Description of the fields in the **group** file. . . . 154

LISTINGS

Listing 7.1 Manual page for the **mkdir** command. 38
Listing 8.1 less user interface. 45
Listing 8.2 Nano's user interface. 47
Listing 8.3 **vim**'s user interface. 48
Listing 13.1 Poem1 text file 83
Listing 13.2 Poem2 text file 83
Listing 13.3 Countries file. 85
Listing 14.1 Car dearlership text file. 91
Listing 15.1 Computing Science students information (cs.info). 100
Listing 15.2 Fine Arts students information (fa.info). 100
Listing 16.1 Script for a Hello World example. 105
Listing 16.2 Script for changing the **USER** variable. 110
Listing 16.3 Script that uses the **read** keyword. 112
Listing 16.4 Script that takes arguments from the user's script call. 112
Listing 17.1 Script containing a simple **if/else** block. . . . 114
Listing 17.2 Script containing two string expressions. . . . 116
Listing 17.3 Script containing an integer expression. 117
Listing 17.4 Script containing two file expressions. 118
Listing 17.5 Script using a **&&** operator. 118
Listing 17.6 Script using a **||** operator. 119
Listing 17.7 Script using an **if/elif/else** syntax. 120
Listing 17.8 Script using nested **if/elif/else** blocks. . . . 122
Listing 18.1 Script containing a simple **for** loop. 125
Listing 18.2 Script using a **for** loop with wildcards. 126
Listing 18.3 Script using a **for** loop with brace expansion. . 126
Listing 18.4 Script using a **for** loop with **wrong** brace expansion. 127
Listing 18.5 Script using a **for** loop with user input and **wrong** brace expansion. 127
Listing 18.6 Script using a **c-style for** loop with user input. 128
Listing 18.7 Script using a **while** loop. 129
Listing 18.8 Reading file contents using a **while** loop. . . . 129
Listing 18.9 Script using an **until** loop. 130
Listing 18.10 Script using a **break**. 131
Listing 18.11 Script using a **continue**. 131

Listing 18.12 Script using nested **for** loops. 132
Listing 18.13 File to be used with question 4. 132
Listing 19.1 Script for a calculator program. 135
Listing 19.2 Script for a calculator program using a function. 136
Listing 19.3 Script containing a local variable. 137
Listing 19.4 Script containing a function that takes arguments. 138
Listing 19.5 Script function that takes redirected arguments. 138
Listing 19.6 Script containing a function that returns an exit status. 139
Listing 19.7 Script showing how to save values echoed from functions into variables. 139
Listing 20.1 Script creating an array with numebrs from 1 to 100. 143
Listing 20.2 Script that displays all elements from an array. 143
Listing 20.3 Script that allows users to change elements from an array. 144
Listing 20.4 Simplified script to create an array with numbers from 1 to 100. 144
Listing 21.1 ls -l. 147
Listing 21.2 Last five lines of a **passwd** file. 150
Listing 21.3 Adding a user to a Linux system. 150
Listing 21.4 Providing or changing a password for a given user. 152
Listing 21.5 Switching users. 152
Listing 21.6 Switching users. 153
Listing 21.7 Contents of the **/etc/group** file. 154
Listing 21.8 Adding a group to a Linux system. 155
Listing 21.9 Changing group properties with **groupmod**. . . 155
Listing 21.10 Making changes to groups using **gpasswd**. . . . 156

ACRONYMS

CLI Command Line Interface

GUI Graphical User Interface

UI User Interface

OS Operating System

FHS File system Hierarchy Standard

PC Personal Computer

GNU	GNU is Not Unix
FSF	Free Software Foundation
CPU	Central Processing Unit
RAM	Random Access Memory
HD	Hard Disk
SSD	Solid State Device
regex	Regular Expression
CSV	Comma-separated values

Part I

INTRODUCTION TO COMMAND LINE INTERFACES

In the following chapters, we explain what an operating system is, as well as its relationship with the computer hardware and software applications. Following, we present a brief overview of Linux's history. Finally, we discuss what Linux distributions are, and we comment on some of the defining characteristics of some of the most widely used distributions.

1 WHAT IS AN OPERATING SYSTEM

To understand Linux Operating Systems, it is necessary first to understand what an Operating System (OS) is. In a nutshell, an operating system is a fundamental piece software that manages both hardware and software resources, while also providing users with a set of core utilities.

1.1 OPERATING SYSTEM COMPONENTS

An OS is normally divided in many components in order to make the different parts of a computer work together. Amongst those components, it is worth to cite:

KERNEL The kernel is the most crucial part of any OS. It serves as an intermediate between software applications and the hardware, having complete control over everything that happens in the system. It controls which software processes are running at any given time, how much memory each process is assigned to, and also provide a gateway between these processes and hardware such as printers, network cards, keyboards, etc.

USER INTERFACE In order to be useful, an OS needs to provide users with a way to access the data stored in it, as well as run different types of applications. In the past, most OSs had only command line interfaces. However, most modern systems nowadays provide users with increasingly easier to use graphical user interfaces.

FILE SYSTEM The responsibility to store and retrieve data into/from memory devices, as well as ensure that it hasn't been corrupted, belongs to the OS.

SECURITY Any respectable OS must provide methods for users to control access to the system itself, as well as to control access to individual files and processes.

CORE UTILITIES The OS is also expected to provide the user with basic utilities to perform things such as: find files, edit text files, control system settings, control processes, etc.

Probably, the most famous OS is Microsoft Windows, whose version 10 now comes installed by default in a large number of laptops and desktops. It is easy to see that Windows has all the components described above, as seen in Table 1.1.

Kernel	The Windows kernel contains a large set of device drivers that allows it to interact with a very wide array of hardware components, from external keyboards to virtual reality glasses. It is able to run multiple applications at once, and allow users to control which applications get more priority.
User Interface	The graphical interface for Windows is classicaly composed of a Start Buttom, a Taskbar, as well as "Windows," where each "Window" normally correspond to an specific application.
File System	Windows is normally installed using an NTFS file system that ensures the reliability of the stored data.
Security	Windows provides control access to the system, as you normally need to enter a password to access it. Also, some versions of Windows allow users to set specif access rights to individual files.
Core Utilities	Windows comes with multiple core utilities such as: File Explorer to retrieve any file in its memory, Notepad to edit text files, Control Panel to modify systems settings, etc.

Table 1.1: Windows OS Components.

1.2 COMPUTER ARCHITECTURE BASICS

In order to better appreciate the role the kernel, it is important to understand the architecture of modern computers. In Figure 1.1, you can see how the kernel fits within a basic computer architecture. In it, it is clear that the kernel acts as a bridge between applications (at the top), and hardware components (at the bottom). Note that the kernel also provides a bridge between different applications, as well as different hardware components.

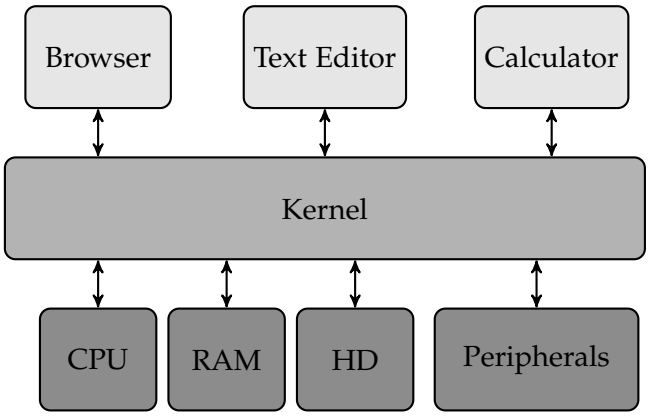


Figure 1.1: Basic computer architecture.

In what follows, we provide a brief introduction of some of the most important hardware components of a modern computer.

1.2.1 *CPU*

The Central Processing Unit, CPU, is the computer's "brain". It is responsible for arithmetic and logical operations, as well as control and input/output (I/O) operations. Whenever an application needs to perform an operation, it requests the kernel to send instructions to the CPU. After the CPU processes the given instructions, it sends outputs back to kernel which, in turn, sends it back to the application.

A single-core computer can only execute one instruction at the time*. When multiple applications are open at the same time, it is the job of the kernel to control access to the CPU. The kernel accomplishes it by having applications taking turns in sending instructions to the CPU. The kernel also ensures that applications with higher priority levels will get more turns to access the CPU than applications with lower priority levels.

*Dual core computers can execute two, and quad-core computers can execute four instructions at time.

1.2.2 *RAM*

Random Access Memory, RAM, is a type of memory which provides very fast access to retrieve and edit the data stored in it. When an

application starts, the kernel normally loads its contents into RAM, together with any data it requires during start-up. Once the application closes, it is the job of the kernel to free RAM in order to be used by other applications.

It is also the job of the kernel to ensure that, when multiple applications a are open, one applications cannot access or overwrite memory addresses used by another applications.

1.2.3 *HD*

The hard disk, HD, is where all data is ultimately stored in a computer. Historically, computers have always used magnetic hard-disks for this purpose. However, some new laptops use solid-state drives SSD*. Both HDs and SSDs can hold data even without a power source. However, they require a power source to retrieve the data.

*Which are normally faster, but more expensive than magnetic devices.

All your personal files, as well as application files, and even system files are stored in the HD. Whenever an user starts an application, the kernel retrives the application data in the HD and loads it into the RAM. This process is the reason why some applications take a few seconds to start*. When an application opens a personal file, the file is retrieved from the HD and stored in the RAM. When an user creates a new file, or edit an existing file, the kernel is responsible for transferring it from the RAM into the HD.

*This is also why computers equipped with SSD have faster application boot times.

1.2.4 *Peripherals*

Modern computers are normally connected to multiple devices that can request data from the computer, or send data to it. As some examples, we have wireless adaptors, keyboards, computer screens, printers, etc. It is the kernel, using device drivers, that provides a bridge that allow applications to interact with these devices.

EXERCISES

1. With your own words, explain what an oepraing system is.
2. With your own words, explain what is the role of a kernel.
3. What is the role of the CPU?
4. Why do most computers have two types of memory? I.e., why do computers have RAM memory as well as HD (or SSD) memory?

2

A BRIEF HISTORY OF LINUX

Most operating systems, such as Microsoft Windows or Apple OS X are well-defined products from a particular company. As a result, all users of one of these systems have exactly the same OS installed on their computers*.

*Some releases of Microsoft Windows may have different editions such as Server, Home, or Professional.

When talking about Linux, quite the opposite is true. There is a myriad of versions of Linux out there, and each one caters for a different type of public or need. There are versions of Linux tailored to run on low-power devices with little amount of memory, such as Raspberry Pis, and versions of Linux running on the world fastest super computers. There are versions of Linux with only a command line interface used to control industrial equipment, and versions of Linux with very friendly graphical user interfaces for phones, tablets, and laptops. There are even versions of Linux powering servers that control most of the traffic of information in the World Wide Web.

In fact, due to its flexibility, and contrary to popular belief, Linux OSs are the most popular operating systems in the world. Even though, in many cases, users are not even aware they are using Linux. For example, Linux was the basis upon which Google's Android OS was developed. They are also the OS of choice for the vast majority of smart devices such as TVs, thermostats, wireless routers, etc. Also, they dominate the server market, with major corporations such as Google and Facebook currently using* Linux servers to run their searche engines and host their social media data, respectively.

*As of 2016.

To understand how Linux became so popular, and why there are so many different versions of it, it is crucial to learn from where it came from. In what follows, we present a brief overview of its history.

2.1 UNIX

The history of Linux, as well as that of most operating systems, starts in 1969 when Ken Thompson, Dennis Ritchie, and other scientists from AT&T Bell Labs released the first version of the Unix OS.

During the late seventies and early eighties, the popularity of Unix grew tremendously amongst academia and corporations. Part of its popularity had to do with its visionary design choices, such as:

- It was written using the C language*, which made it easier to port it to multiple devices.
- It allowed multiple users to use multiple applications concurrently (at the same time). This was very important back then,

*First it was written in Assembly, but later it was rewritten in C.

when many universities and corporations had hundreds of users, but only a handful of mainframe computers.

- It had an easy-to-use modular design. Instead of relying on complex tools to perform complex operations, it relied on simple tools that could easily be combined to perform complex operations. See the Unix philosophy box below.

Another reason behind the surge in Unix's popularity was its price tag. Due to antitrust restrictions, AT&T Bell labs could not sell products that weren't specifically targeted for telecommunications. As a matter of fact, they were required to provide the Unix OS source code, free of charge, to anyone who asked for it. As word of mouth had been quite positive, many Universities and corporations required the Unix OS source code and started using it. Also, given that they had access to the source code, they improved it by removing bugs, porting it to more systems, and adding more features to it. In fact, a lot of tools used in Unix to this day were written by students, professors, and developers, and then made freely available.

Unix Philosophy

The Unix philosophy is not a formal design method nor an academic driven methodoloy. Instead, it is a bottom-up, pragmatic, and grounded in experience way of designing operating systems and software in general.
It was summarized by Peter H. Salus, autor of *A Quarter-Century of Unix*, as a set of three simple rules:

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

2.2 GNU

In the early eighties, the computer industry grew into a multi-billion dollar business. Thus, as an attempt to increase their profits, some companies started to market their own modified versions of Unix, normally called Unix-like OSs, as closed-source OSs*.

All of a sudden, the Unix world had changed. It went from an open community in which everyone had access to use it, change it as they pleased, and share their results, to one where people and companies had to pay thousand of dollars to use an OS which they could only treat as a black box.

**Perhaps the most famous closed-source Unix-like systems today are Apple's OS X (laptops) and iOS (iPhones and iPads).*

Many people, as one would expect, were very displeased by this paradigm shift for several reasons. One of the reasons was that they didn't think it was fair to have to pay to use a tool they had helped to develop. Another reason was that, as different companies were changing their Unix-like OSs in different ways, and the source code was been kept confidential, tools developed in one Unix-like system would often not work properly in other Unix-like system.

One of the more famous voices against this paradigm shift was Richard Stallman. In order to try to re-stablish a culture of collaboration, as it was the case during the early days of Unix, he created the Free Software Foundation (FSF) to support a project called GNU*, with the goal of re-creating Unix using an open-source model. His efforts were quite succesful, as many important UNIX tools that were rewritten for the GNU project are still used to this day. Also, the license models championed by the GNU Project, which allowed anyone to use, modify, and redistribute source code, fostered hundreds of other projects. Nevertheless, there was one glaring missing piece in their quest to create an Unix-like OS built entirely using open source code. Their progress in re-creating the Unix kernel was going very slow.

*A recursive acronym meaning GNU is not Unix.

2.3 LINUX

In 1991, a Finnish student from the University of Helsinki called Linus Torvalds started working in his own re-implementation of the Unix kernel. More specifically, he used an Unix-like system called MINIX as a basis for his own system.

The source code for the resulting kernel, which was dubbed Linux*, was made freely available, together with a call for comments and suggestions for improvement. Because a kernel by itself is not very useful, Torvalds also published a list of GNU tools for Linux users in order to have a functional OS.

*A combination of the words Linus and Unix.

Many computing enthusiasts started using, and contributing to the further development of an open source OS consisting of a Linux kernel and GNU tools. Over time, the name Linux started to be commonly used to denote the entire OS, as opposed to just the kernel*.

*Some people refer to it as GNU/Linux, though.

At first, Linux users had to download and compile the kernel and all required tools separately. Hence, Linux acquired a reputation of being difficult to use. However, over time, different developers started to create packages containing the kernel and sets of tools in order to make it easier for other users to install and use it. This resulted in the rise of a number of different Linux distributions that we discuss in our next chapter.

The Rise of MS Windows

From the middle to late eighties, while Unix was gaining traction in academia and high-tech corporations, a competing OS from Microsoft, called DOS, together with its graphical user interface, called Windows, was making great strides in the households and offices market.
The reasons behind the dominance that Microsoft still holds on these markets are plenty, but it is worth to cite a few:

- A deal with IBM in the eighties guaranteed that Microsoft products would come installed by default in all IBM personal computers PCs.
- These PCs found a sweet spot in the computer market. They were significantly cheaper than the more powerful Apple computers at the time, but they were still powerful enough to run the types of applications most users needed at the time. This led to IBM PCs, which used Microsoft Windows, to dominate the computer market by the early nineties.
- They were considered easier to use, as they were focused on a Graphical User Interface (GUI), as opposed to Command Line Interfaces (CLI) common in Unix-like systems at te time.
- Microsoft cemented their OS dominance with Windows-exclusive applications that became ubiquitous in offices and households such as: Word, Excel, and Power Point, which would later be bundled into Microsoft Office.

3 LINUX DISTRIBUTIONS

As mentioned in the previous chapter, in the beginning of its history, Linux wasn't an OS that catered for unexperienced users. For starters, in order to install it, users needed to download the source code for the Linux kernel, as well the source code for GNU tools, and compile them in their target systems.

In order to remove this technical entrance barrier, some Linux users started to compile the kernel's source code, as well as the source code for a number of GNU tools, for popular computer models. The resulting binaries were then combined into single packages, normally called distributions, which were made available for other users to download. This allowed Linux OSs to be installed without requiring the technical expertise to compile the source code.

Given the open nature of Linux and GNU, anyone could package different sets of tools together in order to create their own distribution. That is exactly what happened. Over time, more and more users packaged their distributions with different sets of components*.

*It is important to note that, over time, many tools that are not part of the GNU project were added to different distributions.

3.1 DISTRIBUTION COMPONENTS

The one piece of source code that defines an operating system as a Linux OS is the Linux Kernel. Apart from it, the other components can vary tremendously from one distribution to another. In what follows, we describe some of the most important components of Linux distributions.

3.1.1 *Desktop Environment*

The most noticiable aspect of a Linux distribution is its Desktop Environment. A Desktop Environment provides a graphical interface for users to access different applications, files, as well as control the system settings. Multiple desktop environments were created over the years, with different design proposals, and tackling different issues. A small list of popular desktop environments is provided in what follows. We also provide a few examples of popular distributions that use the discussed environments by default*.

*Many Linux Distributions allow users to switch their Desktop Environment from its default.

KDE The K Desktop Environment, **KDE**, was the first popular desktop environment created for Linux OSs that is still in use today. It follows a design proposal similar to that of the classical Windows OS. I.e., it provides users with a start button from where

the users can start different applications, a taskbar, and a Desktop where the user can drag and drop files and shortcuts. **KDE** provides many extras bells and whistles that are not available in Windows though, such as multiple workspaces, Desktop Folders, etc. *Ex: OpenSUSE*

GNOME **KDE** required tools that were not open source*. As a result, members of the GNU project started to work towards creating a Desktop Environment composed entirely of open source code. The GNU Network Object Model Environment, **GNOME**, was the result of this initiative. The **GNOME** Desktop Enviroment was built with a focus in productivity and provides a set of accessibility tnools for users with disabilities. It is currently in its third version (**GNOME 3**). *Ex: Debian, Red Hat, Fedora, CentOS*

*More specifically the **QT** framework.

UNITY Unity is a Desktop Environment specifically designed for the Ubuntu distribution. It was designed to make a more efficient use of smaller screens. One of its most noticeable aspects is its laucher bar at the left of the screen. The launcher, as the name suggests can launch different applications, switch between open applications, as well as launch the **dash**, an overlay that allows the user to search quickly for applications, files, folders, etc., both locally and remotely. *Ex: Ubuntu*

OTHERS There are many other desktop environment available that cater for different types of users. Some of these environment were designed for low-power systems, such as **Xfce** and **LXDE**. Other environments are forks of the **GNOME 2*** with added features, for example **Cinnamon** and **MATE**.

*See the ***GNOME 3*** *controversy box below.*

GNOME 3 controversy

The third version of **GNOME**, released in 2011, departured significantly from the previous desktop metaphor used in its second version. In this new version, users were no longer provided with a start button and a taskbar that are always visible. Instead, it requires users to constantly switch to an overview mode in which a series of elements such as a dash, a search bar, a list of current workspaces, etc. are available.
Many users complained about such abrupt change in their desktop environment. This lead some users to create other desktop environments based (forked) on **GNOME 2**, such as **Cinnamon** and **MATE**. These environments aim at keeping **GNOME 2**'s desktop metaphor, while at the same time, keeping it up to date with the most modern Linux technology.

3.1.2 *Software Management*

Virtually all smart phones come with app management tools. These tools are used to download new apps, as well as to keep them up to date. For example, the **App Store** fullfils this role on iOS devices, whereas **Google Play** does the same for Android devices.

Similarly, most Linux distributions come with software management tools which allow users to download and maintain software in their systems. Amongst the most popular package management tools are **`apt`*** (**Debian**, **Ubuntu**, etc.), and **`yum`** (**Fedora**, **Red Hat**, etc).

**We cover Software Management on Chapter XXX.*

Most distributions also maintain their own online databases with a list of certified and up-to-date software packages (applications). These databases, called repositories, allow users to quickly retrieve and install new applications. They also allows the system to check which applications are up to date, by comparing the version of the installed application to the latest version in the repositories.

3.1.3 *Default Applications*

Most Linux Distributions nowadays come with basic terminal GNU tools*, **Mozilla Firefox** installed as its browser, and an office application suite called **LibreOffice**, which presents great alternatives for Microsoft Office programs such as Microsoft Word, Microsoft Excel, etc. However, apart from these two applications, different distributions may come with different sets of applications that are installed by default.

**These tools are extensively covered in this book.*

For example, some distributions come with **Mozilla Thunderbird** pre-installed and set as its default email client, while other distributions may come with **Evolution Email**, instead. Some distributions might not come with any pre-installed email client.

Some specialized distributions come with a set of applications for specific purposes. For example, **Kali Linux** is a distribution designed for security experts. Hence, it comes with a set of penetration testing tools. As another example, **Scientific Linux** is a distribution that comes with a set of tools for mathematical modelling, staristical inferencing, and data analysis.

3.2 POPULAR DISTRIBUTIONS

There are, literaly, hundreds of Linux distributions available today. This fragmentation comes as a direct result of the open nature of Linux development. Anyone can modify an existing open source distribution to create a new one. This process is normally called forking.

While it is impossible to list all available distributions, it is worth to take a look at some of the most popular ones

3.2.1 *Ubuntu, Mint, Kubuntu*

Ubuntu is, nowadays, the most popular Linux distribution for desktops by a large margin. It has a clear focus on usability and comes with a large number of applications to allow users to start being productive without having to make any changes to the system.

Due to its popularity, **Ubuntu** has been forked into a number of new distributions, such as **Linux Mint**, and **Kubuntu**. These distributions mostly differ from **Ubuntu** by presenting different Desktop Environments by default*.

MATE and Cinammon for **Linux Mint, and KDE for **Kubuntu**.*

All the examples in the remaining chapters of this book were performed on an Ubuntu OS. However, unless specifically stated otherwise, they should work on any other Linux distribution.

3.2.2 *Debian*

Debian is one of the oldest and most successful Linux Distributions. Its first release dates back to 1993, and so far it has undergone eight major releases*.

Each **Debian release is named after a character from the Toy Story movies.*

The reason behind **Debian** only having had eight major releases is due to its focus on stability. **Debian** development is conducted using three branches: *unstable*, *testing*, and *stable*. Any changes are first tried at the *unstable* branch. Then, after extensive testing by multiple users at the *testing* branch, the change finally finds its way to the *stable* branch.

Due to its focus on stability, **Debian** is one of the favorite distributions for servers. Also, many other Linux distributions are derived from one of the three Debian branches, including **Ubuntu**.

3.2.3 *SUSE, OpenSUSE*

SUSE is a Linux distribution aimed at enterprises. In contrast to most other Linux Distributions, it cannot be downloaded for free. It needs to be purchased together with an online support plan.

SUSE is based on the **OpenSUSE** distribution, which as the name suggests, can be downloaded and used for free. Novell, the company the owns **SUSE**, refines and enhances **OpenSUSE** to create **SUSE** OSs for specific enterprise goals, such as data center deployments, server deployments, business desktops, etc.

3.2.4 *Red Hat, Fedora, CentOS*

Like **SUSE**, **Red Hat** is an enterprise Linux distribution* that needs to be purchased together with an online support plan. It is mantained by Red Hat, Inc.

**Its full name is Red Hat Enterprise Linux.*

Also, just like **SUSE** is based upon **OpenSUSE**, **Red Hat** is based on **Fedora** OS, a popular Linux distribution sponsored by Red Hat, Inc., but made publicly available for free*. **Fedora** OS is known for its fast development pace, where multiple releases can happen in a year.

*Its source code is also publicly available.

Due to its leading position in enterprise Linux, many **Red Hat** clones have appeared through the years. The most popular of these clones, **CentOS**, is sponsored by no one lese than Red Hat, Inc itself. The main difference between these two distributions is that **CentOS** is an open source project. Hence, it can be obtained and modified free of charge and it comes with no official support.

EXERCISES

1. With your own words, explain what is a Linux distribution.
2. With your own words, explain what is a Desktop Environment.
3. List at leat two differences between a Fedora Distribution, and an Ubuntu Distribution.

Part II

INTRODUCTION TO COMMAND LINE INTERFACES

In the following chapters, we introduce the concept of command line interfaces, and explain how they can be accessed, as well as their relationship with the **shell**. Following, we present some basic **shell** commands together with methods to get help about these commmands. Finally, we end this part by introducing a series of tools that can be used to read and edit text files using a command line interface.

4

COMMAND LINE INTERFACE, TERMINAL, AND SHELL

Most User Interfaces (UI) rely on visual cues to guide the user. For example, on Microsoft Windows, the user can use the mouse (or the finger in a touchscreen device) to click on an icon representing a folder to see its contents. Another example would be that, by clicking on a x symbol at the top right edge* of an application, the application is closed. As yet another example, drop-down menus with items such as File, Edit, and About, are commonplace, as seen in Figure 4.1.

*In Apple and Linux systems, this symbol might be on the top left edge.

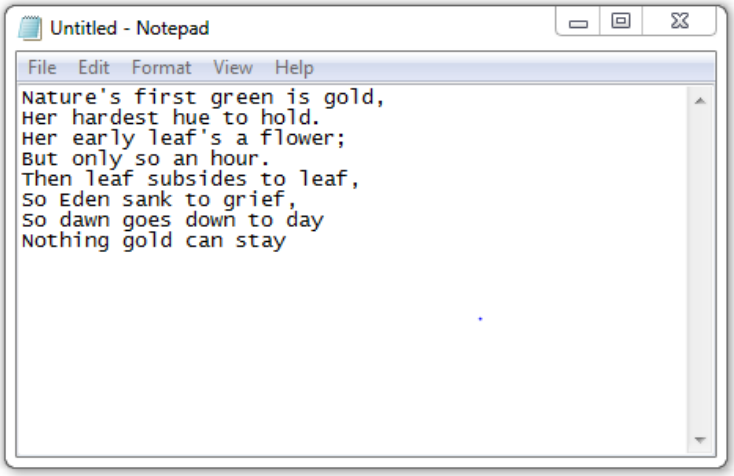


Figure 4.1: Notepad application for Windows 7.

A UI that relies of visual cues, as the ones described above, is called a Graphical User Interface (GUI). The invention of such interfaces played a major role in the popularization of computers in the 80s. This is due to the fact that most users are more comfortable using such interfaces, as opposed to Command Line Interfaces (CLI) that have been around since the late 60s.

In a CLI, as opposed to an intuitive graphical interface, the user is presented with a **shell*** prompt in which it can write commands to be interpreted by the **shell**. You can see an example below of a CLI in which a user entered the command **`ls`** to list all files and folders in the working directory*.

*Shells are explained in Sections 4.1 and 4.2.

*In this book, the words ***folder*** and ***directory*** are used interchengeably.

```
marcel@dell:~$ ls
Documents      Downloads      Pictures
Music          Video          seneca.pdf
marcel@dell:~$
```

In order to properly use a CLI, the user needs to learn a series of commands that the shell understands, as well as the syntax of these commands. Hence, it should come to no surprise that most regular users prefer to use GUIs which do not incur in such steep learning curves.

If you are reading this book, however, I expect you not to be a regular user. In fact, I expect you to be starting a career as systems analyst, a software developer, or other positions in the IT industry. For people like you, having a clear understanding of how to use a CLI presents a number of advantages:

- There are tasks that can only be achieved by using a CLI because no GUI has been created to perform them.
- Some tasks can be performed, using a CLI, in a fraction of the time it would take using a GUI tool.
- CLIs can be accessed remotely in a fast and straightforward fashion, allowing you to control multiple systems from a single computer
- Users can quickly create scripts that run on CLIs to automate repetitive tasks such as: adding new users to the system, backing up data, updating the system, etc.
- The steps required to perform some actions using GUIs can vary a lot depending on which OS, or Linux distribution, you are using. On the other hand, they are normally identical using a CLI. In fact, the same commands* can be used in all Linux distributions, Apple computers, and since 2016, even in Windows systems.

*With a few notable exceptions.

Most Linux distributions have a GUI that allow regular users to achieve daily tasks such as web browsing, email, or writing documents. However, all Linux distributions also come with a terminal emulator that allow super users to do much more using a CLI.

4.1 TERMINOLOGY

Before moving forward to learn how to use a Linux CLI, It is important to notice the difference between three concepts that are often used interchangeably:

CLI The CLI is just the technical name of the method by which we interact with a computer by means of written instructions.

TERMINAL All major Linux distributions have a GUI. However, they normally also provide you with a terminal, or more precisely a terminal emulator. A terminal emulator is nothing else than a program that provides us with a CLI to interact with the **shell**.

SHELL The **shell** is a software that reads keyboard commands and passes them to the OS to carry out. Such commands could be used to perform simple tasks like listing the files in a particular folder, adding users to the system, creating folders, or complex tasks such as upgrading the OS, installing applications, etc.

4.2 SHELLS

As mentioned before, a **shell** is a software that interprets commands passed by the user, and passes them to the OS. However, just like there are multiple languages such as English, Spanish, or Mandarim, there are multiple shells available. Table 4.1 below presents a description of a few such shells*.

*Many other shells exist, such as dash, ksh, etc.

sh	The Bourne shell, which has been distributed as the default **shell** for Unix since 1979.
bash	Bourne Again Shell, an upgraded version of **sh**, written by the GNU Project, which has become the *de facto* standard Linux shell.
csh or **tcsh**	A Unix shell using a syntax similar to the **C** programming language. **tcsh** is identical to **csh**, with the addition of some extra features such as command-line completion.

Table 4.1: List of Linux Shells.

In this book, we will cover exclusively the **bash shell**. This is due to the fact that **bash** is the default shell for the most widely used Linux distributions. In fact, it is hard to find a Linux distribution that does no come with a **bash shell**. That said, after learning how to use one shell, it is much easier to learn other shells, as they share many concepts and syntax.

EXERCISES

1. Explain, with your own words, what is a Command Line Interface (CLI).
2. Explain, with your own words, what is a Graphical User Interface (GUI).
3. What is the relationship between a terminal emulator, and a shell?
4. Which commands are used to list all files in the working directory using the following shells: sh, bash, csh, dash, and ksh?

5 BASIC SHELL COMMANDS I

In this chapter we will cover how to get access to a terminal emulator, as well as some very basic **bash** commands.

5.1 ACCESSING A TERMINAL EMULATOR

Once a user logs into a Linux OS with a GUI, there are multiple ways to get access to a **terminal emulator**. Some methods work only on a specific distribution, whereas some methods work on most distributions. In what follows, a list of methods to access terminal emulators are presented for some of the most popular desktop environments*. The most commom Linux distribution for each desktop environment is presented in the margin notes.

*Desktop environments are covered in Appendix XXX.

GNOME3 Press the super (windows) button to call the list of applications, and then type **`terminal`** in the search field, or search for the terminal in the provided applications list.

Debian, Red Hat, Fedora, CentOS, Kali

KDE Click on the start button at the lower left, and then click on accessories, and finally on **`terminal`** .

OpenSUSE, Kubuntu, Slackware

UNITY Press the super (windows) button to call the dash (or conversely click on the dash button at the upper left), and then type **`terminal`**.

Ubuntu

In all of these desktop environments, the terminal emualtor can also be started by pressing **`crtl+alt+T`**.

5.2 TERMINAL BASICS

Once the terminal emulator starts, the user is presented with an application displaying a blinking cursor at a **`shell prompt`**, just like shown bellow.

```
marcel@dell:~$
```

This prompt contain important information about the current parameters of the shell, namely: the username (**`marcel`**), system name (**`dell`**), and working directory (represented by a tilde [**`~`**])*. A brief description of these parameters follows:

*The **`$`** separates command prompts from these current parameters.

USERNAME Linux systems, as any Unix-like system, was designed to allow multiple users to access the system. When a terminal emulator starts, it assumes the user to be the same one that logged

in into the GUI system. in Chapter XX, we will show how the **su** command can be used to switch users. In the example above, the username is **marcel**.

SYSTEM NAME During the installation of Linux systems, users are required to give your system (local machine) a name. When a terminal emulator starts, it assumes that you are using the local machine. In chapter XXX we will see how to log into remote machines using the **ssh** command. In the example above, my local machine is called is **dell**.

WORKING DIRECTORY Imagine that you issue a command to create a file. In which directory will this file be created? The answer is: in the current **working directory**. When a terminal emulator starts, it assumes by default that you are at the home folder (directory) of the user currently logged in. Hence, whichever actions you perform will, by default, affect this directory. In Section 5.8 , we will show how to use the **cd** command to change directories. In the example above, this home folder, which is located at **/home/marcel** is represented by a tilde (**~**).

In what follows, a number of basic **bash*** commands are presented.

*You can easily switch shells by typing the name of the shell you want to switch to. For example, you can type ***dash*** to start using a ***dash shell***.

5.3 **date**

To ask what day is today, you can type **date** in the terminal emulator and press enter. The shell will verify the date with the OS, display it in the terminal, and start a new prompt line ready to receive more commands, as shown below*. Note that the terminal also shows the current time and the time zone (Eastern Daylight Time - EDT).

*For the following commands, we will ommit this new prompt line.

```
marcel@dell:~$ date
Wed May 18 19:20:55 EDT 2016
marcel@dell:~$
```

Note that shell commands are **case-sensitive**. I.e., the shell diferentiates between upper-case and lower-case letters. For example, while **date** is a valid command, **Date** and **DATE** are not.

5.4 **whoami**

Not all terminal emulators display the username at each command prompt. Therefore, the shell needs to provide a method to verify the username of the current user. This is accomplished by the **whoami** (who am I) command, as shown below.

```
marcel@dell:~$ whoami
marcel
```

5.5 `pwd`

As mentioned before, when a terminal emulator is open, it needs a directory to act as a starting point. In most Linux system, this starting point is the user's home folder, located at **`/home/username`**, represented by a tilde (**`~`**).

As we will show later, users can move to different directories using the shell. The shell's current location is called **working directory**. This is due to the fact that this is the directory that the shell will, by default, act (work) upon. In order to see in which directory you are currently at, you can type **`pwd`*** in the terminal emulator and press enter. The shell will then print the working directory in the terminal, as shown below.

**short for print working directory.*

```
marcel@dell:~$ pwd
/home/marcel
```

5.6 `ls`

To obtain a list of all files and folders present in the working directory, as demonstrated in Chapter 4, you can use the **`ls`*** command. The shell will then print a list of all files and folders, as shown below*.

**short for list.*

**Some terminal emulators use colours to distinguish between files, folders, scripts, etc.*

```
marcel@dell:~$ ls
Documents      Downloads      Pictures
Music          Video          seneca.pdf
```

The list command, by default, does not show hidden files*. In order to do so, you must add the option **`-a`** or **`-all`** to the ls command, as shown below:

Hidden files are files whose names starts with a **`.` (dot).*

```
marcel@dell:~$ ls
Documents      Downloads      Pictures
Music          Video          seneca.pdf
marcel@dell:~$ ls -a
.              ..             Documents
Downloads      Pictures       Music
Video          seneca.pdf     .System.conf
```

Note that in the command above, the file **`.System.conf`** only appears when the **`ls`** command is issued with the **`-a`** option. Also, note that two extra directories appear: a folder (**`.`**) and a folder (**`..`**). These are not real directories. They are simply hard links to the self (**`.`**) and parent (**`..`**) directories.

The list comand can also used to gather information about the files and folders in the current working directory. To do so, the **long list** option **`-l`** must be invoked, as shown below:

```
marcel@dell:~$ ls -a -l
drwxrwxr-x  3 marcel marcel  4096 Jun 21 23:06 .
```

```
drwxr-xr-x 54 marcel marcel  4096 Jun 21 18:59 ..
drwxrwxr-x  2 marcel marcel  4096 Jun 21 23:06 Documents
drwxrwxr-x  2 marcel marcel  4096 Jun 21 23:06 Downloads
drwxrwxr-x  2 marcel marcel  4096 Jun 21 23:06 Pictures
drwxrwxr-x  2 marcel marcel  4096 Jun 21 23:06 Music
drwxrwxr-x  2 marcel marcel  4096 Jun 21 23:06 Video
-rw-rw-r--  1 marcel marcel 12238 Jun 29 22:54 seneca.pdf
-rw-rw-r--  1 marcel marcel   126 Jun 28 20:52 .System.conf
```

This long list provides a number fo columns containing information about each file. In Table 5.1, we explain what the information in each column represents using the file **seneca.pdf** as an example.

column	example	meaning
1	**-rw-rw-r-**	File type and permission sets. These topics are covered in Chapter XX.
2	**1**	Number of hard links to this file. Links are covered in Chapter XX.
3	**marcel**	Name of the user owner of the file. The concepts of users and ownership are covered in Chapter XX.
4	**marcel**	Name of the group owner of the file. The concepts of groups and ownership are covered in Chapter XX.
5	**12238**	Size of the file in bytes.
6	**Jun 29 22:54**	Timestamp indicating when the file was edited for the last time.
7	**seneca.pdf**	File name.

Table 5.1: Long list information for the **seneca.pdf** file.

5.7 tree

The **tree** command is somewhat similar to the **ls** command. They both display a list of files and folders in the working directory. However, contrary to **ls**, in which any subfolder is simply listed alongside files, the **tree** command goes inside each subfolder and displays the files contained in them, creating a tree-like structure. See the example below:

```
marcel@dell:Seneca$tree
.
|-- academic_honesty.pdf
|-- calendar.pdf
|-- OPS105
```

```
|   |-- students_list
|   |-- grades.xls
|-- SRT311
|   |-- students_list
|   |-- grades.xls

2 directories, 6 files
marcel@dell:Seneca$
```

The **tree** command is not available by default in some Linux distributions. To install it, you must write the command **sudo apt-get install tree**, and enter your user password.

5.8 cd

To switch working directories, you need only to use the **cd*** command, followed by the name of the directory you want to go. For example, assuming that the folders shown in the previous command's output do exist, typing **cd Documents** results in:

*short for change directory.

```
marcel@dell:~$ pwd
/home/marcel
marcel@dell:~$ cd Documents
marcel@dell:Documents$ pwd
/home/marcel/Documents
```

The command **cd** by itself always sends the user back to the user's home folder. Also, the command **cd ..** sends the user to the parent folder of the working directory, and the command **cd -** sends the user back to the previous working directory. The parent folder is the folder hierarchically above the current folder. For example, the folder **/home/marcel** is the parent folder of **/home/marcel/Documents**.

Finally, note that if you enter the name of a directory that does not exist, the shell will return an error message and then it will start a new prompt line ready to receive more commands, as shown next.

```
marcel@dell:~$ cd DOCUMENTS
bash: cd: DOCUMENTS: No such file or directory
marcel@dell:~$
```

5.9 Relative and Absolute Paths

In our previous example using the **cd** command, we switched working directories from one folder, **/home/marcel**, to one of its immediate subfolders, **/home/marcel/Documents**. However, you might face more complex situations in which you have a directory tree as shown in Figure 5.1, and you want to switch the current working directory from any folder directly to any other folder.

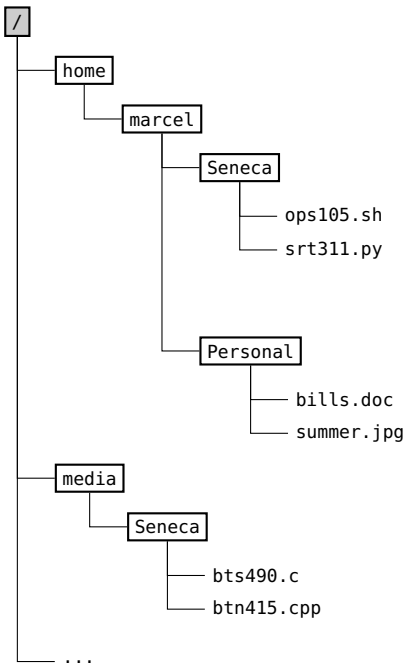


Figure 5.1: Directory tree.

For example, given the directory tree shown in Figure 5.1, how can we directly switch working directories for the following cases:

- from **/home** directly to **/home/marcel/Personal**
- from **/home/marcel/Seneca** directly to **/media**
- from **/home/marcel/Seneca** to **/media/Seneca**

The solution to all these cases is: "by providing either a **relative path** or an **absolute path** to the desired folders." In what follows, we define these two concepts*.

Note that the concepts of relative and absolute paths also apply to files.

5.9.1 *Relative Path*

When a relative path is provided, the shell assumes that the starting point of the path is the current working directory. For example, given that you are currently in **/home**, you can switch your working directory to **/home/marcel/Personal** by issuing the command: **cd marcel/Personal**, as shown in what follows.

```
marcel@dell:home$ pwd
/home
marcel@dell:home$ cd marcel/Personal
marcel@dell:Personal$ pwd
/home/marcel/Personal
```

5.9.2 *Absolute Path*

Absolute paths always start with **/*** and provide the complete path for the desired folder or file. For example, given that your working directory is **/home/marcel/Seneca**, you can switch to **/media** by issuing the command: **cd /media**, or you can switch to **/media/Seneca** by issuing the command: **cd /media/Seneca** , as shown in what follows.

** / denotes the root, of all directories in a Linux system.*

```
marcel@dell:Seneca$ pwd
/home/marcel/Seneca
marcel@dell:Seneca$ cd /media
marcel@dell:media$ pwd
/media
```

```
marcel@dell:Seneca$ pwd
/home/marcel/Seneca
marcel@dell:Seneca$ cd /media/Seneca
marcel@dell:Seneca$ pwd
/media/Seneca
```

Relative and absolute paths can be compared to addresses in navigation systems. For example, if you live in Toronto, and you want to go to a place within the city, you can type on your navigation device an address such as 200 King Street. The device will assume that this address is in Toronto-ON, Canada. This would be the compared to a relative address, as the starting point is assumed to be the city of Toronto. However, if you are in Toronto and you want to go to 200 King Street in Buffalo-NY, you would need to enter the complete address, including the city, state, and sometimes even the country. This would be compared to an absolute value, as the address is completely defined.

5.10 **clear**

All commands you enter, as well as their outputs, stay in the screen and you are given a new shell prompt at the bottom to keep working with the shell. This can lead to a very polluted screen containing a lot of information that is no longer necessary.

In order to clear the screen from all previous commands and ouputs, you simple have to issue the **clear** command. This command will erase from your terminal all information in it and provide you with a command prompt at the top to keep working with the shell.

It is important to note that this command simply clears the terminal display. Any variables* that have been defined, or any modifications to the state of the shell will not be changed by this command.

**Variables are covered in Chapter XXX.*

5.11 COMMAND HISTORY

You can use the up (↑) and down (↓) arrow keys in order to navigate the commands you have previously entered in your shell. For example, by hitting the up arrow key once, you will get the previously entered command. Hitting the up arrow again, will get you the command you entered prior to that one.

EXERCISES

1. Describe two methods to open a terminal emulator on a Ubuntu Linux OS.
2. Given the command prompt below, who is the user currently logged into the shell? Also, what is the name of the local machine? Finally, what is the current working directory?

   ```
   john@lenovo:~$
   ```

3. What is the relationship between a **terminal emulator**, and a **shell**?
4. Which commands are used to list all files in the working directory using the following shells: bash, csh, dash, and ksh?
5. Which command can be used to show in the terminal what time is it?
6. What is the output of the `ls -l` command? What does the 6th column represents?
7. How can you move back to the latest previously accessed working directory using only one shell command?
8. What happens when you apply the parent folder parameter to the `cd` command twice. In other words, what happens when you issue a `cd ../..` command?

   Given the directory tree shown in Figure 5.2 (on next page), answer the following questions:

9. Write a command to switch your working directory from **marcel** to **Seneca** using only one shell command?
10. Write a command to switch your working directory from **media** to **marcel** using only one shell command.
11. Given that your working directory is **media**, can you use a relative path to move to **Seneca**? If yes, can you also use an absolute path? Which one of these options seem more appropriate in this case?

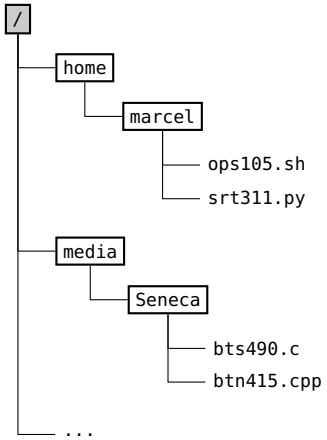


Figure 5.2: Directory tree for questions 9, 10, and 11.

6

BASIC SHELL COMMANDS II

In our previous chapter, we covered a number of bash commands which could be used to: gather information from the system and user (**`date`**, **`whoami`**, **`pwd`**), get information about files and folders (**`ls`**), or to change de current working directory (**`cd`**).

None of the commands could be used to alter files, folders, or change the configurations of the system, though. I.e., they have no lasting effect in the system.

In this chapter, we will cover a series of commands that have lasting effects in files, folders, and possibly the system. The commands we are covering in this chapter can be used to create and remove files and folders, as well as to rename and make copies of them.

6.1 `mkdir`

To create new folders, the **`mkdir`*** command can be used, followed by the name(s) of the folder(s) to be created. In the example below, a folder called **Samples** is created in the current working directory.

*short for make directory.

```
marcel@dell:~$ ls
Documents      Downloads      Pictures
Music          Video          seneca.pdf
marcel@dell:~$ mkdir Samples
marcel@dell:~$ ls
Documents      Downloads      Pictures    Samples
Music          Video          seneca.pdf
```

Note that, by default, the new directory is created as a subfolder of the working directory. To create folders in other locations, without leaving the current working directory, you can use the **`mkdir`** command combined with relative or absolute paths, as shown below:

```
marcel@dell:~$ ls Documents
Seneca      Personal
marcel@dell:~$ mkdir ~/Documents/Samples
marcel@dell:~$ ls Documents
Seneca      Personal    Samples
```

Files and Folders with spaces in their names

In all examples that were given so far is this book, all files and folders do not contain spaces in their names. Issuing a command such as **`mkdir My Documents`** would create two folders,

one called **My** and another called **Documents**. This is because the shell cannot distinguish between an argument with a space, or two arguments without spaces.
Linux can handle files and folder with spaces, though. In order to do so, whenever you are refering to a file or folder with sapces in its name, each space needs to be preceeded by a backslash (\). For example, you can create a folder called **My Documents** by issuing a command **mkdir My\Documents**.

6.2 rmdir

The **rmdir*** command, as the name suggests, deletes existing folders. For example, **rmdir Samples** will remove a folder **Samples** from the current working directory. It is important to note that it can only be used to remove empty folders. In case the folder is not empty, the shell returns an error message.

*short for remove directory.

In the example below, the **rmdir** command is applied to both an empty folder **Samples**, and to a non-empty folder **Pictures**.

```
marcel@dell:~$ ls
Documents      Downloads      Pictures    Samples
Music          Video          seneca.pdf
marcel@dell:~$ rmdir Samples
marcel@dell:~$ ls
Documents      Downloads      Pictures    Music
Video          seneca.pdf
marcel@dell:~$ rmdir Pictures
rmdir: failed to remove 'Pictures': Directory not empty
marcel@dell:~$
```

6.3 touch

As shown in Section 5.6, files and folders in Linux have a timestamp indicating when they were edited for the last time. The **touch** command, when applied to an existing file, updates the timestamp of when it was last edited*. It is equivalent to opening the file, saving it, and then closing it. As an example, see the command prompt below:

*Updated timestamps are useful for some backup programs, among other uses.

```
marcel@dell:~$ ls -l
total 0
-rw-rw-r-- 1 marcel marcel 0 Jul 12 11:31 final_exam.doc
-rw-rw-r-- 1 marcel marcel 0 Jun 21 20:30 introduction.ppt
-rw-rw-r-- 1 marcel marcel 0 May 10 21:41 seneca.pdf
marcel@dell:~$ touch final_exam.doc
marcel@dell:~$ ls -l
```

```
total 0
-rw-rw-r-- 1 marcel marcel 0 Jul 14 22:12 final_exam.doc
-rw-rw-r-- 1 marcel marcel 0 Jun 21 20:30 introduction.ppt
-rw-rw-r-- 1 marcel marcel 0 May 10 21:41 seneca.pdf
```

When applied to non-existing files, the **touch** command creates empty files, as shown below:

```
marcel@dell:~$ ls -l
total 0
-rw-rw-r-- 1 marcel marcel 0 Jul 14 22:12 final_exam.doc
-rw-rw-r-- 1 marcel marcel 0 Jun 21 20:30 introduction.ppt
-rw-rw-r-- 1 marcel marcel 0 May 10 21:41 seneca.pdf
marcel@dell:~$ touch mid_term.doc
marcel@dell:~$ ls -l
total 0
-rw-rw-r-- 1 marcel marcel 0 Jul 14 22:12 final_exam.doc
-rw-rw-r-- 1 marcel marcel 0 Jun 21 20:30 introduction.ppt
-rw-rw-r-- 1 marcel marcel 0 May 10 21:41 seneca.pdf
-rw-rw-r-- 1 marcel marcel 0 Jul 14 22:15 mid_term.doc
```

This second usage of the **touch** command is frequently used for testing purposes*.

*tip: you to use **touch** and **mkdir** in order to recreate the examples provided in this chapter..*

6.4 rm

To delete (remove) files, the **rm*** command can be used, followed by the name(s) of file(s) to be deleted from the working directory. See the example below:

short for remove.

```
marcel@dell:~$ ls
Documents      Downloads      Pictures
Music          Video          seneca.pdf
marcel@dell:~$ rm seneca.pdf
marcel@dell:~$ ls
Documents      Downloads      Pictures
Music          Video
```

The **rm** command can also be used to remove non-empty folders. To do so, we have to apply the **-r** (recursive) option followed by the name of the folder, as shown below.

```
marcel@dell:~$ ls
Documents      Downloads      Pictures
Music          Video          seneca.pdf
marcel@dell:~$ rm -r Pictures
marcel@dell:~$ ls
Documents      Downloads      Music
Video          seneca.pdf
```

Bash Design Goals

It is important to note that the commands described in this chapter don't normally ask for confirmation. For example, when deleting a file using the **rm** command, the **shell** will not ask the user if he/she is sure he/she wants to delete the file. Nor the file will be sent to a recycle bin from which it can be recovered. The file will simply be deleted.
The **shell** performs tasks that might incur in unforeseen consequences, such as accidently deleting a number of files, in a much more direct way because of its design goals. As opposed to GUI interfaces where the goal normally is to provide a platform for inexperienced users to perform basic tasks, a CLI is normally designed with the goal of allowing experienced users to quickly perform complex tasks.

6.5 mv

The **mv*** command can be used in two distinct ways. As the name suggests, it can move a file, a folder, or a number of files and folders from one directory to another. However, it can also be used to rename single files or folders. Both uses of this command are described in what follows.

*short for move.

6.5.1 *Moving files and folders accross directories*

To move a single file from one directory to another, the **mv** command needs two arguments. The first being the name of the file in the working directory to be moved, and the second being the absolute or relative path of the directory we are moving the file to. See the example below:

```
marcel@dell:~$ ls
final_exam.doc  introduction.ppt  seneca.pdf
Folder
marcel@dell:~$ mv final_exam.doc Folder
marcel@dell:~$ ls
introduction.ppt  seneca.pdf  Folder
marcel@dell:~$ ls Folder
final_exam.doc
marcel@dell:~$
```

Note that, in this example, the file **final_exam.doc** disappears from the working directory and appears in the **Folder** directory.

Multiple files can be moved with a single **mv** command. To do so, you first need to provide the names of all files you intend to move as

arguments. Then, you need to provide the relative or absolute path of the directory you want to move them to as the very last argument. This technique is shown in what follows:

```
marcel@dell:~$ ls
final_exam.doc  introduction.ppt  seneca.pdf
Folder
marcel@dell:~$ mv final_exam.doc introduction.ppt Folder
marcel@dell:~$ ls
seneca.pdf  Folder
marcel@dell:~$ ls Folder
final_exam.doc introduction.ppt
marcel@dell:~$
```

Note that, within this context, this command can also be applied to folders (directories). I.e., you can move an entire folder in your working directory into other folders using the exactly same syntax applied for files in this section.

6.5.2 *Renaming files and folders*

Imagine the **mv** command being used with two arguments, the first being a file name in the working directory, and the second being a name that doesn't match any directories. In this scenario, the **mv** command will rename the file indicated by the first argument to the name specified in the second*. See the example below:

*A command called **rename** exists in **bash**. However, it is used to rename multiple files at once using regular expressions. Regular expressions are covered in Chapter XXX.

```
marcel@dell:~$ ls
final_exam.doc  introduction.ppt  seneca.pdf
Folder
marcel@dell:~$ mv final_exam.doc final_exam_fall.doc
marcel@dell:~$ ls
final_exam_fall.doc  introduction.ppt  seneca.pdf
Folder
marcel@dell:~$
```

In this example, you can see that the file **final_exam.doc** was renamed to **final_exam_fall.doc**.

You can also rename a folder using the same syntax applied for files in this section.

6.6 **cp**

The **cp*** command can be used to copy files and folders. It can be used in two distinct ways:

*short for copy.

6.6.1 *Copying files within the working directory*

To create a copy of a file in the same working direct, the **cp** command should be used with two arguments. The first argument should be the name of the file to be copied, while the second argument should be the name of the copy, as shown below:

```
marcel@dell:~$ ls
final_exam.doc  introduction.ppt  seneca.pdf
Folder
marcel@dell:~$ cp final_exam.doc final_exam_fall.doc
marcel@dell:~$ ls
final_exam.doc  final_exam_fall.doc  introduction.ppt
seneca.pdf  Folder
marcel@dell:~$
```

To copy an entire folder in your working directory into another folder also in the working directory, you must use the recursive option **-r**, as shown in the example below:

```
marcel@dell:~$ ls
final_exam.doc  introduction.ppt  seneca.pdf
Folder
marcel@dell:~$ cp -r Folder Folder_Copy
marcel@dell:~$ ls
final_exam.doc  final_exam_fall.doc  introduction.ppt
seneca.pdf  Folder  Folder_Copy
marcel@dell:~$
```

6.6.2 *Copying files to other directories*

To create a copy of a file in the working directory to another directory, the second argument of the **cp** command must be the absolute or relative path of the directory in which you want to plae a copy of the file. In this case, the new file will have the same name as the original file. See the example below:

```
marcel@dell:~$ ls
final_exam.doc  introduction.ppt  seneca.pdf
Folder
marcel@dell:~$ cp final_exam.doc Folder
marcel@dell:~$ ls
final_exam.doc  introduction.ppt  seneca.pdf
Folder
marcel@dell:~$ ls Folder
marcel@dell:~$ ls
final_exam.doc
marcel@dell:~$
```

Just as with the previous scenario, to copy an entire folder in the working directory to another directory, you also must use the recursive option **-r**. See the example below:

```
marcel@dell:~$ ls
final_exam.doc  introduction.ppt  seneca.pdf
Folder  Music
marcel@dell:~$ cp Music Folder
cp: omitting directory 'Music'
marcel@dell:~$ ls Folder
final_exam.doc  introduction.ppt
marcel@dell:~$ cp -r Music Folder
marcel@dell:~$ ls Folder
final_exam.doc  introduction.ppt  Music
marcel@dell:~$
```

To copy all contents of a folder to another folder, but not the folder itself, you need to add **/.** to the end of the name of the folder you are copying, as shown below*:

*You can also use this technique with the **mv** command.*

```
marcel@dell:~$ ls
final_exam.doc  introduction.ppt  seneca.pdf
Folder  Music
marcel@dell:~$ cp -r Music/. Folder
marcel@dell:~$ ls Folder
arcade_fire-ready_to_start.mp3  foo_fighters-walk.mp3
marcel@dell:~$
```

EXERCISES

1. What is the main difference between using the copy **cp**, and the move **mv** commands, when applied to files?
2. How can you delete a non-empty folder called **Archive** that exists in your working directory?
3. Which command can be used to move a file called **README.txt** from your working directory to its parent folder
4. Which command can be used to create a folder called **Examples** inside a subfolder called **Documentation** that exists in your working directory. See the diagram in Figure 6.1.
5. Still with regards to the diagram diagram in Figure 6.1, how can you move files **Example1.txt** and **Example2.txt** from the **Documentation** folder to its **Examples** subfolder, without changing your working directory.
6. Is it possible to delete multiple empty folders with only one **rmdir** command? If so, how can you delete two empty folders named **Tests** and **Assignments** from your working directory?

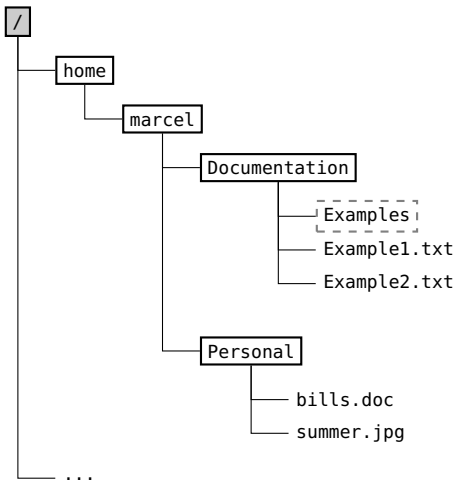


Figure 6.1: Directory tree for questions 4 and 5.

7. What happens when you apply the **touch** command to an existing file? Also, what happens when you apply this command to a non-existing file.

8. How can you make copies of the contents of a file called **exams.txt** from your working directory, to a file called **exams.txt** inside a subfolder **Classes** that exists in your working directory?

9. What happens when the first parameter of a **mv** command is a file, and the second parameter is a folder?

10. Which command can be used to copy the contents of a subfolder named **Folder1**, that exists in the current working directory, to another subfolder named **Folder1Copy** in the same working directory. Note that the subfolder **Folder1Copy** doesn't exist prior to the command you need to enter.

7 GETTING HELP

We have already covered in previous chapters a number of commands which can take multiple options and different numbers of arguments. For example, it was shown in Chapter 5 that the command **`ls`** can work in three different ways:

- By itself, this command prints the names of all files in the working directory.
- When provided with the name of another folder as an argument, this command prints the names of all files in the specified folder.
- If called with the the option **`-l`***, this command gives all the information about files described on Table 5.1, as opposed to just listing their names.

There are many more options for the **`ls` command..*

The bash shell has hundreds of commands like **`ls`** that can take multiple options and different numbers of arguments. Hence, in order to be accessible to non-experts, it needs to provide its users with a way of knowing how to use those commands.

In this chapter we will cover different ways in which users can get help on how to use different commands.

7.1 `help`

By itself, the **`help`** command will list all **built-in*** commands. If one of the built-in commands is provided as an argument, this command provides a quick description of the provided command, and possibly a list of options. You can see below the output of entering **`help pwd`** on bash.

**Built-in commands are explained in the Built-in Commands box that follows.*

```
marcel@dell:~$ help pwd
pwd: pwd [-LP]
    Print the name of the current working directory.

    Options:
      -L  print the value of $PWD if it names the current
    working directory
      -P  print the physical directory, without any symbolic
     links

    By default, 'pwd' behaves as if '-L' were specified.
```

```
   Exit Status:
   Returns 0 unless an invalid option is given or the
   current directory cannot be read.
```

Note that **help** only covers built-in commands. It does not cover commands implemented as binary files in **/bin** or **/usr/bin**, such as **ls**, **rm**, **mv**, and many others.

Built-in Commands

Most commands in bash, such as **ls**, **rm**, and **touch** are implemented by binary files located in the **/bin** and **/usr/bin** folders. These commands are interpreted in the same way as any other application the shell can run. I.e., the shell asks the kernel to execute it, and after it has finished executing, the shell receives its output.

Built-in commands, on the other hand, are an integral part of the shell. They are not implemented in a separated file. The main reason why some commands are implemented directly inside the shell is because they need to change the state of the shell. For example, the **cd** command is a built-in command because it changes the current working directory. Commands implemented as binaries, such as **ls**, **rm**, and **touch**, cannot change the state of the shell.

Another reason for implementing some commands directly into the shell is because it normally enhances their performance.

7.2 man

Since the inception of Unix, it has become standard practice for the authors of scripts that implement shell commands to provide manual pages for them. This practice was continued by the GNU project when rewritting Unix as an open source project*.

All manual pages follow the same structure show in Listing 7.1 (page 38). I.e., they have the following sections:

Name States the name and purpose of the command

Synopsis Briefly describes the command syntax

Description Describes the command as well as its options

Authors Lists the authors of the script that implements the command

Reporting Bugs Provides a link to a page where bugs can be reported

Copyright States that the code is provided as free software

*In fact, the author of the manual pages for many basic commands such as **ls** and **rm** is no one less than Robert Stallman, the programmer that started the GNU project, as discussed in Chapter XXX.

```
MKDIR(1)            User Commands               MKDIR(1)

NAME
       mkdir - make directories
SYNOPSIS
       mkdir [OPTION]... DIRECTORY...
DESCRIPTION
       Create the DIRECTORY(ies), if they do not already
  exist.
       Mandatory arguments to long options are mandatory
  for short options too.

       -m, --mode=MODE
          set file mode (as in chmod), not a=rwx - umask
       -p, --parents
              no error if existing, make parent
  directories as needed
       -v, --verbose
              print a message for each created directory
       -Z     set SELinux security context of each created
   directory to the default type
       --context[=CTX]
              like -Z, or if CTX is specified then set the
   SELinux or SMACK security context to CTX
       --help   display this help and exit
       --version   output version information and exit
AUTHOR
       Written by David MacKenzie.
REPORTING BUGS
       GNU coreutils online help: <http://www.gnu.org/
  software/coreutils/>
       Report mkdir translation bugs to <http://
  translationproject.org/team/>
COPYRIGHT
       Copyright 2014 Free Software Foundation, Inc.
  License GPLv3+: GNU GPL version 3 or later <http://gnu.
  org/licenses/gpl.html>.
       This  is free software: you are free to change and
  redistribute it.  There is NO WARRANTY, to the extent
  permitted by law.
SEE ALSO    mkdir(2)

       Full documentation at: <http://www.gnu.org/software
  /coreutils/mkdir> or available locally via: info '(
  coreutils) mkdir invocation'

GNU coreutils 8.23    November 2015    MKDIR(1)
```

Listing 7.1: Manual page for the **mkdir** command.

See Also Provides a list of related commands

To access manual pages, you need only to use the **man** command, followed by the name of the command you are trying to get information about.*. See the example below:

*Manual pages cover not only commands, but also daemons and config files.

```
marcel@dell:~$ man mkdir
```

7.2.1 *Sections*

Note that the **mkdir** command appears with a **(1)** next to it at the top of Listing 7.1 (see page 38). This number denotes the section of the manual from where the information was retrieved. Some commands have the same name of **deamons** or **config files**. Hence, dividing manual pages in sections makes it possible for users to access the manual page for the right tool. Manual pages are divided in 9 sections, as shown in Table 7.1.

1	Executable programs or shell commands
2	System calls (functions provided by the kernel)
3	Library calls (functions within program libraries)
4	Special files (usually found in **/dev**)
5	File formats and conventions e.g. **/etc/passwd**
6	Games
7	Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8	System administration commands (usually only for root)
9	Kernel routines [Non standard]

Table 7.1: Manual Page Sections

By default, **man COMMAND** retrieves the first occurrence of **COMMAND** in the manual pages. In order to access later occurrences, you need to provide the section as the first argument followed by the name of the command. For example, **passwd** is both a command, as well as a config file. To access the manual for the **passwd** command, one can write:

```
marcel@dell:~$ man passwd
```

or

```
marcel@dell:~$ man 1 passwd
```

However, to access the **passwd** config page manual, one needs to write:

```
marcel@dell:~$ man 5 passwd
```

7.3 whatis

As shown in Listing 7.1, each manual page comes with a description section. The **whatis** command searches the man page of the command provided as an argument and displays its description. See the example below:

```
marcel@dell:~$ whatis ls
ls (1)               - list directory contents
```

If the argument can be found in multiple sections, all descriptions are provided in the output, as shown below:

```
marcel@dell:~$ whatis passwd
passwd (1)           - change user password
passwd (5)           - the password file
```

7.4 apropos

In English, the word apropos means: *concerning, with reference to*. The **apropos** command is called using keywords as arguments. It returns a single line for each man page that contains one or more of the keyword(s) in its **NAME** section.

This command comes handy when you need to search for commands which names you are not certain of. See the example below*:

```
marcel@dell:~$ apropos rename
dpkg-name (1) - rename Debian packages to full package names
mv (1)        - move (rename) files
rename (1)    - renames multiple files
rename (2)    - change the name or location of a file
renameat (2)  - change the name or location of a file
```

*A few extra outputs from this command were ommited for simplicity.

In this example you can see that the **mv** command can be used to rename a single file, while the **rename** command is normally only used to rename multiple files at once.

7.5 info

Man pages were designed in the early seventies. Hence, they can only display simple text and do not take advantage of newer technologies such as hyperlinks. Also, man pages can be quite terse and seldom provide examples.

In order to modernize the system documentation, the GNU project introduced the concept of info pages. Info pages are similar to man pages in which they provide a description of commands, deamons, and config files. However, they differ in two crucial aspects:

1. Info pages provide a much more in depth description of commands and the options they can take. Multiples examples are often provided.

2. Info pages are divided in nodes that can be accessed via hyperlinks*, or with the shortcuts provided in Table 7.2.

*To follow a hyperlink, you need to press enter when you cursor is over a node title.

q	Quits the info page
n	Moves to the next node
p	Moves back to the previous node
u	Go up to the parent node
h	Display a list of shortcuts to navigate info pages (exit it by pressing x)

Table 7.2: Shortcuts to navigate info pages

The choice between using a man page or an info page depends on the user. Some people prefer the terseness of man pages, while other find that it makes it hard to understand. On the other hand, some people like the level of detail present in info pages, as well as the fact that it contain examples, while others find it difficult to find the information they need among multiple nodes.

As a rule of thumb, you may want to read first the man page for a particular command, and only read its info page in case you could not find the information you needed.

EXERCISES

1. Why there is no help page for the **ls** command in bash?
2. Why there is no man page for the **alias** command in bash?
3. What is a built-in command?
4. Why are some commands implemented as built-ins, as opposed to being implemented as binary files?
5. Why are manual pages divided in sections?
6. Explain in which scenarios you would use the **whatis** command.
7. Explain in which scenarios you would use the **apropos** command.
8. what is the difference between a man page and an info page?

9. How can you use the **ls** command to show all files in the working directory (including hidden files), without also showing the implied **.** and **..** directories? *Hint: check its man or info pages.*

10. How can you use the **rm** command to remove an entire directory called **myFolder** in the working directory, while asking for confirmation before deleting each file inside **myFolder**? *Hint: check its man or info pages.*

11. How can you use the **cp** command to make copies of one file called **myFile** in the working directory, saving it into a folder called **myFolder**, only if there are no files called **myFile** inside **myFolder**, or if the file exists, but is older than the new copy you are trying to create. *Hint: check its man or info pages.*

8

READING AND EDITING TEXT FILES FROM THE SHELL

In Chapter 5, we learned how to create empty files with touch, rename them with **mv**, and even delete them using the **rm** command, among other things. However, so far we have not yet covered how to read the information contained in files, nor how to edit them using the terminal.

In this chapter, we will fill this gap by introducing tools that allow us to read text files (**more**, and **less**), as well as to edit text files (**nano**, **vim**).

8.1 READING TEXT FILES

8.1.1 *more*

To read simple text files on your terminal you can use **more**, by calling it with the name of the file you want to read passed as an argument. See the example below:

```
marcel@dell:~$ more poem
Nature's first green is gold,
Her hardest hue to hold.
Her early leafs a flower;
But only so an hour.
Then leaf subsides to leaf.
So Eden sank to grief,
So dawn goes down to day.
Nothing gold can stay.

marcel@dell:~$
```

This tool will display as much of the file as it fits the terminal screen*. To keep reading the file, you need to press the **Space** key. To quite **more** before reaching the end of the file you can press **q**.

more is quite outdated. Its most glaring limitation is that It doesn't allow you to scroll backwards while reading files. You can only scroll forward. So, while we included it here for historical reasons*, even its man page advises users to use the **less** tool instead.

*In the provided example, all text from the **poem** file was able to fit in the screen.

*some embedded systems still use more to take advantage of its small size.

8.1.2 *Less*

The **less** command was created with the main goal of providing backwards scrolling to **more***. It has since became the *de facto* tool to read text files on the terminal. For example, all manual pages are displayed using **less**.

*In fact, its name is a pun on the architecture's minimalist design motto "less is more".

This command is quite different than all other commands we have learned on chapters 5 and 6. So far, issuing commands would normally result in the steps below:

1. The user calls a command which might take options and arguments
2. The command that was called processes the user's input
3. All command's outputs are displayed on the terminal
4. A new shell prompt is made available just below the previous command's output

When calling **less**, on the other hand, the following sequence of steps happens:

1. The user calls **less** providing the name of a text file as an argument, as shown in the example below:

   ```
   marcel@dell:~$ less poem
   ```

2. **less** starts its own user interface, as shown in Listing 8.1, that takes over the entire terminal screen and displays the beginning of the document
3. The user can use a number of keys to navigate the document
4. The user enters a key to quit **less**' interface
5. A new command prompt is made available below the one in which the user called **less***.

*Note that the contents of the file opened in **less** do not remain in the terminal screen after the user quits **less**.

By creating its own user interface, **less** can allow the user to perform actions that would either not be possible or very cumbersome otherwise, such as backwards scrolling and some advanced navigation methods. These actions can be performed using the keyboard. The most important keys to control **less** are shown in Table 8.1*.

*for a comprehensive list, check **less man** or **info** pages.

Terminal User Interfaces

Many other tools create their own terminal user interface like **less**. Notable examples are the two text editors discussed later in this chapter, **nano** and **vim**, as well as the **top** command covered in Chapter XXX.

```
Nature's first green is gold,
Her hardest hue to hold.
Her early leafs a flower;
But only so an hour.
Then leaf subsides to leaf.
So Eden sank to grief,
So dawn goes down to day.
Nothing gold can stay.

poem (END)
```

Listing 8.1: less user interface.

ENTER or ↓	Moves forward by one line
Page Up or **SPACE**	Moves forward by one screen
y or ↑	Moves backward by one line
Page Down or **b**	Moves backward by one screen
g	Moves to the beginning of the file
G (**Shift + g**)	Moves to the end of the file
q	Quits less
h	Help

Table 8.1: Less navigation keys.

You can think of these tools the same way you think of applications such as MS Word, or your browser in a GUI environment. These tools run on top of your shell, the same way as those applications run on top of your OS. The only difference is that they normally take the whole terminal screen, as opposed to opening in a separate window, and normally can only be controlled via keyboard inputs (as opposed to mouse or touchscreen).
When using tools that create their own interface, it is important to understand that they generally do not understand bash commands. I.e., issuing commands such as **mkdir**, or **ls**, inside these tools will most likely result in an error message.

The ability to search for patterns is also a great feature introduced in **less**. To search for a pattern, you need only to type backslash (\) followed by the desired pattern and press **Enter**. For example, by entering **\folder**, you are shown the first occurrence of the word **folder** starting from the top of the current screen. To keep searching for the next occurrence of the desired pattern, you can type **n**. To go back to a previous occurrence of the desired patter, you can type **N** (**Shift + n**).

8.2 EDITING TEXT FILES

In this section we will cover two widely used text editors. **nano** for small edits, and **vim** for larger projects and scripts. There is another widely used text editor in the Linux world called **emacs**, with some really die-hard fans, that is not covered in this book. There are two reasons why **vim** was chosen over **emacs**. First, **vim** is available by default in more distributions. Second, the author itself is a **vim** enthusiast.

8.2.1 *nano*

The simplest way to edit a text file on terminal is by using the **nano** command. To do so, you need only to call **nano** followed by the name of the file you are trying to edit*, such as in the example below:

*If no file name is provided, nano will open with a new empty file..

```
marcel@dell:~$ nano poem
```

In this example, a file called **poem** is opened, resulting in the user interface shown below:

Once a text file is open in **nano**, you can start editing it using the keyboard. To perform actions such as: saving the file, exiting **nano**, or getting help, you simply need to enter the shortcuts presented at the bottom of the interface*. For example, you can exit **nano** by entering **Crtl + X**, or you can save the file by entering **Crtl + O**.

*The ^ symbol stands for the Crtl button.

```
GNU nano 2.2.6         File: poem

Nature's first green is gold,
Her hardest hue to hold.
Her early leafs a flower;
But only so an hour.
Then leaf subsides to leaf.
So Eden sank to grief,
So dawn goes down to day.
Nothing gold can stay.

^G Get He^O WriteOut^R Read Fi^Y Prev Pag^K Cut Te^C Cur Pos
^X Exit  ^J Justify ^W WhereIs^V Next Pag^U UnCut ^T To Spel
```

Listing 8.2: Nano's user interface.

8.2.2 ***vi*** *and* ***vim***

The **vi*** terminal-based text editor was released for Unix systems more than 40 years ago. It has since being ported to multiple systems and OSs, and many text editors are built upon it. **vi** is, together with text editors derived from it, the most widely used text editor for Linux, and can be found in all Unix and Linux distributions.

*short for visual.

The most famous text editor derived from **vi**, **vim***, augments the capabilities of **vi** by introducing, among other things:

*short for vi improved.

- Syntax highlighting for multiple programming languages.
- Spell check in more than 50 languages.
- Multilevel undo and redo. I.e., you can undo and redo multiple edits to the text, as opposed to only the last one.
- More user-friendly interface.

Over time, **vim** has become so ubiquitous in the Linux world that currently it is made available by default in most Linux distributions. In fact, this ubiquitousness has reached the point that, in some distributions, the command **vi** actually calls **vim** instead of **vi**.

In this section, we will cover how to perform basic tasks in **vim**. However, all methods described here also apply to **vi**, unless stated otherwise.

vim *interface*

Opening **vim** to start editing a file is as simple as opening **nano**. You simply need to call **vim** followed by the name of the file you are trying to edit*. See the example below:

*If no file name is provided, **vim** displays a splash screen with some help information. When you start inserting text, **vim**

```
marcel@dell:~$ vim poem
```

Not that in Listing 8.3, the **vim** user interface indicates empty lines with tildes (~). Also, **vim** provides the user with important information at the bottom of the terminal screen. Table 8.2 presents a list of all information that is displayed, as well as the corresponding values displayed in Listing 8.3.

```
1  Nature's first green is gold,
2  Her hardest hue to hold.
3  Her early leafs a flower;
4  But only so an hour.
5  Then leaf subsides to leaf.
6  So Eden sank to grief,
7  So dawn goes down to day.
8  Nothing gold can stay.
9  ~
10 ~
11 ~
12 "poem" 9L, 210C                          1,1           All
```

Listing 8.3: **vim**'s user interface.

File name	"poem"
Number of lines	9L
Number of characters	210C
Cursor position	1,1 (first line, first column)
Position of the screen	Indicates if you are at the **Top**, **Bot**tom, or which percentage of text has already been read. In the example, **All** text is displayed.

Table 8.2: Information displayed in **vim**.

vim *modes*

Working with **vim** requires understanding its three modes of operation: **command mode**, **insert mode**, and **extended mode**. These modes are described below:

COMMAND MODE In this mode, which is also known as **normal mode**, you can use combinations of keys to enter commands. Commands can be used to copy and paste, delete blocks of text, or undo previous actions, among other things.

INSERT MODE In this mode you can type new text. Note that this is not the mode you need to be to copy or paste blocks of text.

EXTENDED MODE This mode, which is also known as **last-line mode***, can be used to save the current file, open more files, turn the spell check on and off, and quit **vim**, among other uses.

*The name ***extended mode*** is due to the fact that it was based on a previous text editor called **ex**.

Each time you start **vim**, it is in **command mode**. To switch to **insert mode**, you can press any of the following keys: **a**, **A**, **i**, **I**, **o**, or **O**. Each key results in starting the **insert mode** at a different position with regards to where the cursor is, as shown in Table 8.3. To switch back from **insert mode** to **command mode**, you need to press **Esc**

i	Starts insert mode at the cursor
I	Starts insert mode at the beginning of the line the cursor is
a	Starts insert mode after the cursor
A	Starts insert mode at the end of the line the cursor is
o	Opens a new line below the cursor and starts insert mode on it
O	Opens a new line above the cursor and starts insert mode on it

Table 8.3: Keys to switch to insert mode.

To switch from **command mode** to **extended mode**, you need to press the colon key (**:**). To switch back, from the **extended mode** to the **command mode**, you need to press **Enter**. to the In Figure 8.1, you can see how to transition from different nodes. Note that it is not possible to switch from **insert mode** to **extended mode**, or vice versa, directly.

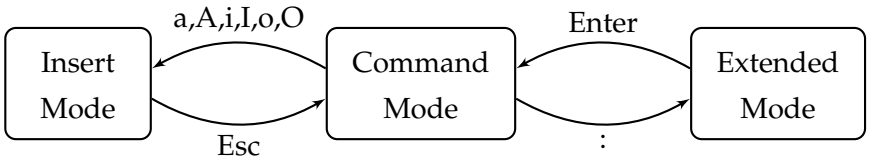


Figure 8.1: Vi operational modes and the keys required to change modes.

Command mode

In **command mode**, **vim** allows the user to perform a multitude of different operations in a fast and direct way by pressing different keys. In fact, **vim** is so powerful that many experienced programers prefer to use it even when editors with sophisticated GUIs are available.

Some commands are quite intuitive such as the arrow keys moving the cursor around character by character, or the **Page Up** and **Page**

Down keys scrolling the text by one screen at a time. However, some commands can be a bit cryptic, such as **y*** standing for copy. Moreover, some commands require a sequence of keys to be pressed in an specific order to attain the desired outcomes.

*It is actually short for yank.

In order to help new users to get familiar with **vim**, Table 8.4 presents some of the most used commands for **vim***. For a comprehensive list of **vim** commands, you can refer to its **man** page.

*This table is introduced here to be used as a reference, not to simply be memorized..

u	Undo the latest command. Note that all edits performed each time you enter the insert mode are considered as just one command.
Ctrl+R	Redo changes that were undone using the **u** command.
dd	Deletes the line the cursor is on. Note that the line is saved on a clipboard which can be later pasted using the **p** command.
#dd	Where # stands for a number, deletes # lines starting at the one where the cursor is on and saves them on the clipboard.
yy	Copies (yanks) the line the cursor is on to the clipboard.
#yy	Where # stands for a number, copies (yanks) # lines starting at the one where the cursor is on to the clipboard.
v	Turns on visual mode that allows you to specify, using the arrow keys, which part of the text you want to act upon with other commands. For example, you can use **v** to select a few paragraphs, and then delete them all by pressing **d**.
p	Pastes the contents of the clipboard after the position the cursor is on.
P	Pastes the contents of the clipboard before the position the cursor is on.

Table 8.4: List of some **command mode** important commands.

Extended Mode

The **extended mode** is mostly used for file operations, such as saving files, opening files, etc. It is also used, however, to configure the behaviour of **vim**. For example, it is in **extended mode** that you can turn spell check on and off. Finally, the **extended mode** is also used to quit **vim**.

On Table 8.5, we show some of the most important operations that can be performed in extended mode. Remember that you need to type colon (**:**), while in command mode, to reach the extended mode.

:q	Quits **vim**. If any open files have non-saved edits, an error message is displayed.
:w	Saves the file under its current name. If the file hasn't been given a name yet, an error message appears.
:w file_name	Saves the file as file_name. If file_name is the same names as the currently open file, the file is overwritten
:e file_name	opens a new file, while placing the current file in a buffer. Having multiple files open whenhe same time can be very convenient. Specially when you need to copy parts of one file, and paste it into another.
:ls	Lists all currently opened files.
:b file_name	Switchs to a file called file_name currently open, i.e., currently in the open buffer. You can use tab to autoconplete the file name. You can also use its buffer number instead of its name.
:b#	Switchs to the previously opened file. This is very convenient when you need to go back and forth between two files.
:set spell spelllang=en_ca	Switchs spellcheck on, and assumes that the current language is Canadian English.
:set nospell	Switchs spellcheck on, and assumes that the current language is Canadian English.

Table 8.5: List of some extended mode important commands.

It is possible to combine commands. for example, you can save and exit **vim** by issuing a **:wq** command, instead of two separate **:w** and **:q** commands..

By default, **vim** tries to prevent users from making mistakes, such as quitting before saving edited files, or overwritting existing files. In case you want to override these security measures, you simply need to add an exclamation point (**!**) to the end of your command. For example, the command **:q** quits **vim** without saving any open files.

Reading non simple-text file

All tools presented in this chapter will assume that the files being opened are simple-text encoded using an **ascii** table. I.e., they will translate sequences of 8 bits into characters. For example, a sequence **00100000** is translated to **Space**, **01100001** to

a (lower case), and so it goes. For a complete **ascii** table, see http://www.asciitable.com/.
In case you try to open a non-simple text file, such as pdf files or binary applications, using the tools presented in this chapter, you will see a seemingly random sequence of ascii characters. This happens because the tool will interpret whichever sequence of bits it finds using an ascii table. Even if these bits were not created to represent a simple text.

EXERCISES

1. What is the relationship between the **more** and the **less** commands?
2. How can you search for a particular word in **less**?
3. Cite two advantages of using **vim** over **nano**.
4. Type **vimtutor** on your shell prompt. It will open an iterative tutorial to help you practice your **vim** skills.
5. Which actions can you perform in **vim**'s **extended mode**?
6. Which actions can you perform in **vim**'s **command mode**?
7. Which command you would enter to delete 20 lines of text, starting with the current line?
8. Which command you would enter to be able to select parts fo your text using the arrow keys, as opposed to entering the number of lines or words you wish to select?
9. Assume that you have two files in your current working directory called **text1** and **text2**. Explain, including all commands you need to enter, how can you copy some lines from one file, and paste it into another.

Part III

ADVANCED COMMAND LINE INTERFACES

In the following chapters, we discuss Linux file systems, covering both its formatting, as well as its directory hierarchy. Also, we introduce file links, discussing both hard links and soft links, as well as their properties. Following, we introduce file globbing techniques and discuss different wildcards that are frequently used. We also provide a gentle introduction to regular expressions, combined with the **`grep`** tool. Finally, we discuss important filters and tools, such as **`cat`**, **`sort`**, **`gawk`**, **`rename`**, and **`find`**. We finish this part by explaining the important concepts of piping and redirection.

9 LINUX FILE SYSTEMS

In Linux, the expression **file system** can refer to two distinct concepts, which can lead to some confusion. One of such concepts has to do with the **format** in which data is stored. The other one has to do with the **hierarchy** of directories and subdirectories that are present in a Linux OS. In what follows, we explain these two concepts separately.

9.1 FILE SYSTEM: DATA STORAGE FORMAT

In order to access data in hard-disks, USB devices, optical devices (CD, DVD, etc), etc., the OS kernel needs to know the format in which this data is written. You can think about this format as the language that has been used to write the data. It specifies how the memory is divided in blocks and how data is stored inside these blocks. Also, it specifies which information about the data, also known as metadata, is stored. Finally, it also specifies in which blocks and what type of metadata needs to be stored.

9.1.1 *Linux File System Evolution*

Originally, the Linux kernel used the MINIX file system*. The first file system format created specifically for the Linux kernel is called the extended file system, or **ext**. Over the years, this file system has been improved upon, leading to three different formats being in use today, as described in what follows:

*MINIX was the Unix-like system Linus Torvalds used as a starting point for creating Linux.

EXT2 This first update on the **ext** file system works on systems with up to 32 Terabytes (compared to only 2 Gigabytes in ext). Also, it keeps track of different timestamps for when files were last accessed, modified, and changed*. This file system is still widely used for SD cards and USB flash drives.

*I.e., had its metadata, such as its permissions, changed.

EXT3 This update improves upon **ext2** by introducing a **journalling system**. Under this system, the kernel keeps a journal* of all modifications it needs to make in order to properly save (or delete) data in the memory. In case there is a crash, the system can avoid data corruption by checking the contents of the journal. Journalling is normally avoided in SD cards and USB flash drives because it requires additional memory writes, which decreases their lifespan.

*The journal is saved as a hidden file.

EXT4 This update improves upon **ext3** by allowing an infinite number of subdirectories, as opposed to "only" 32,000, and by work-

ing with volums of up to 1 Exabyte. Also, it allows some administrative tasks, such as repairing file systems, a lot faster. Finally, **ext4**, as opposed to **ext3**, allows journalling to be turned on and off.

As pointed above, **ext2** is still widely used for smaller data storage devices. Also, some non-Linux OSs still lack full compatibility with **ext4**, making the use of **ext3** a better approach in some situations. In summary, even though **ext4** is the most up to date system format for Linux systems, there are some scenarios in which using **ext2**, or **ext3** are more apropriate.

It is important to note that other OSs have their own file systems such as **FAT32** and **NTFS** for Windows, as well as **HFS+** and **APFS** for macOS and iOS. Android normally uses **ext4**. A Linux OS normally needs to be installed in a partition formatted with **ext2**, **ext3**, or **ext4**. However, most Linux distributions contain drivers that allow users to work with devices such as SD cards and USB flash drives formatted using other file systems.

9.1.2 *Linux File System Format*

File systems in Linux are divided in blocks. These blocks can be divided in three major components*: **superblock**, **inode table**, and regular **data**.

*Strictly speaking, there are other components. However, they are ommitted here for the sake of simplicity.

SUPERBLOCK In Linux, superblocks store information about the file system, such as: its size, the size of each block, which blocks are full and which blocks are empty, which inodes are taken and which inodes are free, etc.

INODE TABLE The inode table stores information about each folder or file, including its permissions, timestamps (access, modify, change), size, location in memory, etc. Note that the inode table does not store the filename*. Each file is assigned to an inode entry (in other words an inode number) in the inode table. Hence, the OS can quickly retrieve information about any group of files by simply consulting the inode table.

*Folder names and file names are stored in directory files, as discussed in the block Secion 9.1.4.

DATA The vast majority of the memory of any device is allocated for the storage of user and system data. To access the data contained in any file, the OS first consults the inode table to check for permissions, as well as for the location of the required data in memory.

9.1.3 ***stat***

To check the data stored in the inode table for a particular file, the **stat** command can be used. It takes as an argument the name of the

file or folder for which the information is required, as shown in the example below:

```
marcel@dell:~$ stat poem
  File:  'poem'
  Size: 210         Blocks: 8          IO Block: 4096
    regular file
Device: 805h/2053d  Inode: 12075464    Links: 1
Access:(0664/-rw-rw-r--)  Uid:(1000/marcel)  Gid:(1000/
    marcel)
Access: 2016-08-01 17:54:19.553690653 -0400
Modify: 2016-07-27 21:49:03.334173611 -0400
Change: 2016-07-27 21:49:03.334173611 -0400
 Birth: -
marcel@dell:~$
```

The **stat** command displays the size of the file in bytes, the number of blocks, the block size, the type of file, a number that identifies the device* in which the file exists (both in hex and in decimal format), its inode number, and how many hard links the file have. Also, the **stat** command displays the access permissions for the file, as well as the name of its user owner and group owner ids. Finally, the **stat** command shows the timestamps for access (file has been opened), modify (contents have been altered), and change (metadata has been altered). Note that it does not display the timestamp for when the file was created, which can be seen by the fact that the **Birth** field is empty. This is a future feature that hasn yet been implemented.

*Hard-disk, USB stick, DVD, etc..

9.1.4 *Directories in Linux*

Directories in Linux are nothing else than special files*. They hold tables containing the following information about its files and subfolders: their names and their corresponding inode numbers. As an example, Table 9.1 below depicts the contents of the directory file for **/home/marcel**.

*As we explain in Section 9.2, everything in Linux is a file.

FILE NAME	INODE
Documents	12583653
Downloads	12583650
Pictures	12583655
Music	12583654
Video	12583652
Seneca.pdf	12583421

Table 9.1: Contents of the **/home/marcel** directory file.

9.1.5 *Accessing data and metadata*

After learning the format in which Linux file systems are implemented, it is important to understand how the OS performs some basic file operations. In what follows, we present a few examples of file operations, and we describe how they are performed, under the hood, by the OS.

For example, when you issue an **`ls`** command, the OS needs only to check the names of all entries in the directory file. However, if you issue an **`ls -l`** command, the OS needs to check the inode table entries for all files in the directory. This is due to the fact that permission access information is stored in inode tables, and not on directory files.

As another example, when a file is deleted, for instance using the **`rm`** command, the OS simply sets the status of its inode number as free in the superblock.

Retrieving data from files is a bit more complex. In this scenario, the OS needs to navigate its way, starting from the root directory (**`/`**) until reaching the desired file. See the example below, where a user is trying to access data from a file **`/home/marcel/script.sh`**

1. The OS starts by checking the inode number 2 in the inode table, which always points to the location of the root directory (**`/`**).
2. The inode number of **`/home`** is retrieved from the root directory file.
3. The inode entry of the **`/home`** directory file is used to retrieve its location in memory.
4. The **`/home`** directory file provides the inode number for the **`/home/marcel`** directory file.
5. The inode entry of the **`/home/marcel`** directory file is used to retrieve its location in memory.
6. The **`/home/marcel`** directory file is analyzed to retrieve the inode number of **`/home/marcel/script.sh`**
7. Finally, the system uses the info located in the inode entry of **`/home/marcel/script.sh`** to retrieve the data in **`script.sh`**.

This sequence of steps is depicted in Figure 9.1.

9.1.6 *Performing actions on files, directories, and inodes*

Even though the sequence of steps presented in Figure 9.1 might appear to be more complex than necessary, the Unix file system* was designed with security, speed, and reliability in mind, as the examples below illustrate:

**Which all Unix-like file system, including Linux, follow.*

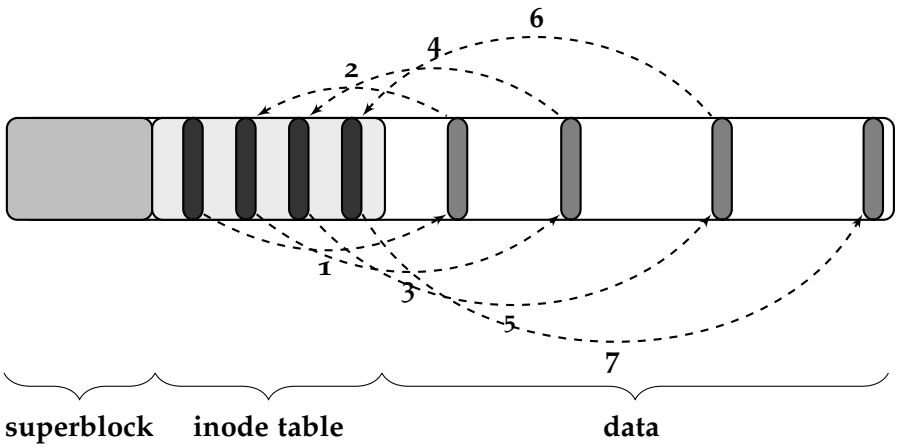


Figure 9.1: Sequence of steps to access a text file.

- When a file moves from one location to another, within the same partition, only the contents of some directory files change. The data itself remains at the same position in memory. Again, this operation is very fast regardless of the size of the files.
- This system keeps metadata, such as access permissions, separated from the data itself. This separation allows, among other things, the data to be stored in an encrypted format, and only decrypted when the user trying to access it has the proper access permissions.
- As stated above, when a file is deleted, the OS simply sets the status of its inode number as free. This way, the memory this file used to occupy becomes free for other files to use in the future. Because only an inode entry needs to have its status changed, deleting files is a very fast process regardless of the size of the file.
- This file system design allows file links to be created in a fast and efficient way, as we will discuss in Chapter XXX.

9.1.7 *shred*

When a file is deleted in Linux, using a command such as **rm**, the OS simply sets the status of its inode to free, as explained above. However, the data itself still exists in the memory device until it gets rewritten by other data. This can pose a security risk, as there are methods to read data in a memory device even if its inode entry is declared free.

In order to really erase sensitive data from a memory device, you need to replace the bits representing the data with random bits. This can be accomplished with the **shred** command. Calling shred with

a file name as an argument, such as in **shred poem**, replaces the bits in the **poem** file with random bits. If the option **-u** is given, such as in **shred -u poem** the **shred** command first replaces the bits from the **poem** file with random ones, and then deletes the file.

9.2 DIRECTORIES HIERARCHY

A crucial aspect of the Linux file system is that it implements just about everything as a file* In other words, a text file is a file, an application binary is a file, a directory is a file, processes are managed as files, and even pieces of hardware such as a printer or a mouse are represented by files.

*This approach, first proposed for Unix, is employed in all Unix-like OSs.

The idea of representing everything as a file greatly simplifies the design of the OS. In fact, following this methodology, the role of the OS is to simply pass text and bytes back and forth between different files. For example, to print a text file using a printer, the OS must pass the information in the text file to the file that represents the printer device.

A clear requirement for an OS to function properly, given this "everything is a file" design methodology, is that it needs to know where all the files necessary for its proper operation are. In order to accomplish this goal, the Linux File system Hierarchy Standard (FHS) established the basic directory hierarchy that Linux distributions should use*. Having all distributions using the same directory hierarchy layout results in a series of advantages, among which it is worth to cite:

*Most distributions deviate slightly from the proper standards. To see your distribution basic directory hierarchy, enter **man hier**.

- It helps different systems to communicate effectively with one another.
- It allows applications to be designed for multiple distributions without requiring any modifications.
- System administrators can easily find system configuration and log files when dealing with multiple distributions.

In what follows, on Table 9.2, we present a list of the most important basic directories ina Linux file system hyerarchy, as specified by the FHS standard. We also explain what types of files these directories should hold.

Directory	Contents
/	The root directory. It gets its name because it is from it that the whole directory tree starts. Note that even USB sticks, external hard disks, and optical devices such as DVDs, are mounted on top of the root direcyory. This is an approach very different than what is done on Windows, where each memory device is mounted as a one letter file system (C:, D:, E:, etc).

/bin Contains executable programs (binaries) which are needed in single user mode and to bring the system up or repair it. Many important commands, such as **ls**, **mv**, and even the **bash** shell itself are implemented as binary files in this directory.

/boot Contains static files for the boot loader. It holds only the files which are needed during the boot process.

/dev Holds files representing physical devices such as a printer, a network card, etc.

/etc This directory is a place for files that did not fit into other previously defined directories. Its name is derived from the latin expression *et cetera*. It mostly contains system configuration files.

/home Holds all user's home directories. For example, the home directory of a user called john will likely be located in **/home/john**.

/lib Contains shared libraries that are necessary to boot the system. Shared libraries are compiled pieces of code that can be used by multiple programs in order to achieve a well-defined goal.

/media Contains mount points for removable media such as CDs, DVDs, SD cards, or USB sticks. I.e., by default these memory devices will be mounted as subfolders inside this directory.

/mnt Countains mount points for temporarily mounted devices. This directory is the point where a sysadmin should use to manually mount devices.

/opt Holds program files for software installed without using the **apt-get** or **yum** tools (covered on Chapter XX). In other words, software installed without using repositories.

/proc holds files with information regarding proccess. It contains runtime system information such as: system memory, hardware configuration, etc.

/root Home directory for the root user (sysadmin)

/sbin Holds executables (binaries) for basic commands that are normally executed by the OS or by users with root access, such as **ifconfig**, **fdisk**, etc.

/srv Holds files containing data that is served by the system, as opposed to directly accessed. It is where files served via **ftp**, **cvs**, or even **http**, should go, for example.

/tmp	Holds temporary files. Temporary files are normally created and deleted without the user's input. They can be used to store information necessary during boot time, to retain important information when some types of software are open, or for providing users with recovery file options after crashes.
/usr	Contains a number of subfolders such as **/bin**, **/sbin**, and even **/lib**. These folders will host binaries and shared libraries deemed non-essential. It also holds binaries and libraries for software installed through the repositories.
/var	This directory contains files which may change in size, such as spool and log files. Some systems use this folder to hold files whose size can vary, containing data that are served by the system. Other systems, use the **/srv** folder.

Table 9.2: Basic directory contents according to the FHS.

EXERCISES

1. Are there good reasons to use the **etx2** or **etx3** formats, as opposed to the newest **etx4** format? If so, provide a few.
2. With regards to file systems, what is a journalling system?
3. Can a Linux OS read memory devices formatted with a FT32 or an NTFS file system?
4. Where in the file system is the information about which inodes are free, and with inodes are taken? In the superblock, in the inode table, in the directory files, or in the data section of the memory device?
5. Where are the names of the files stored? In the superblock, in the inode table, in the directory files, or in the data section of the memory device?
6. What is the difference between the **Modify** and **Change** timestamps?
7. What type of information is stored in directory files?
8. It is said that Linux systems follow a "everything is a file" design. Explain what does this design methodology mean.
9. What happens with the data in a particular file when the file is deleted from the system using a **rm** command.

10. Why is it important for different Linux distributions to follow a standardized directory hierarchy?
11. What is the difference between the **/bin** and **/sbin** directories?
12. What is the difference between the **/media** and **/mnt** directories?
13. What type of information is stored in the **/lib** directory?
14. Where are temporary files stored in a Linux distribution that follows the FHS standard?
15. Assume you have a web server providing/collecting information to/from users using an **http** protocol. Where would you store the contents you are providing to your users?

10 FILE LINKS

Hyperlinks are one of the most common features of pages in the world wide web. As you use your browser* to navigate your way to whatever it is you are trying to find, you click on hyperlinks to go from one page to another

*Edge, Safari, Chrome, Firefox, etc.

File links are not that different. The purpose of a file link is to direct you to another file. Unix-like systems such as Linux have two different types of links: hard links, which were the original way of creating links in Unix, and soft links, also known as symbolic links, which were created to overcome some limitations of hard links*. In what follows, we cover both types of links.

*Soft links are similar to shortcuts in Windows.

10.1 HARD LINKS

Following our discussion in Section 9.1, it is clear that when a file is created, the following steps are taken by the OS:

- A new entry is added to its parent directory file containing its name and inode number.
- The inode table is updated with the proper information.
- The file data is stored in a particular location of the memory device pointed by the inode table.

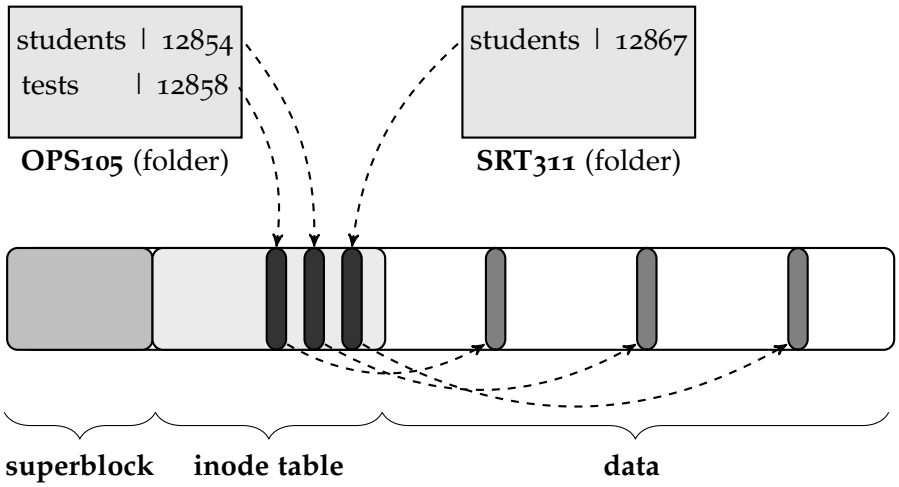
When a hard link is created, the OS simply adds a new entry to the chosen directory file with the same inode number as the original file. Also, this procedure will automatically increment the hard link count associated with the file by one.

In Figure 10.1, you can see a diagram showing how a file system is affected by the addition of a hard link. In this example, a file called **`tests`** initially exists inside a folder **`/OPS105`**, as it can be seen in 10.1a. Then, a hard link to this file, called **`tests_ln`**, is created inside a folder **`/SRT311`**. You can see in 10.1b how both the original file **`tests`** and the hard link **`tests_ln`** point towards the same inode entry.

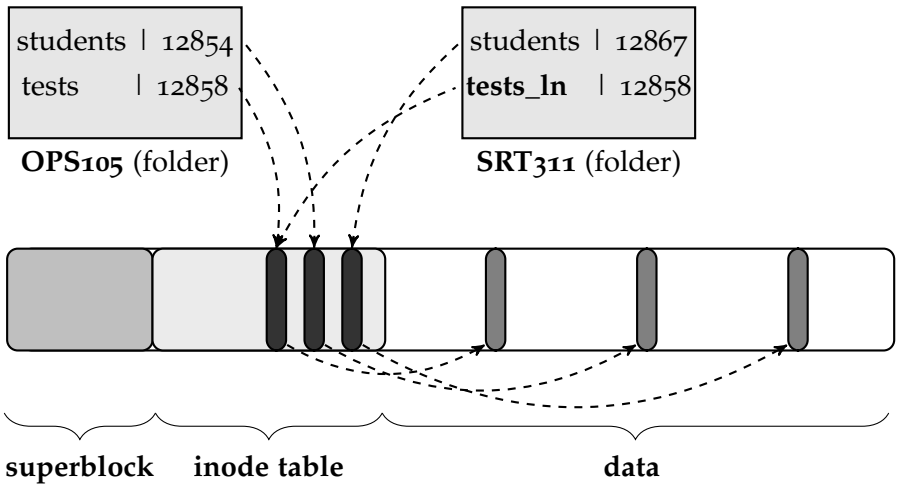
10.1.1 *`ln`*

To create a hard link, you simply need to use the **`ln`*** command, followed by two arguments: the name of the file you are creating a link for, including its absolute path, and the name of the link, including the path to the folder in which you want to place it. See the example in page 65.

*Short for link.



(a) File system before the hard link **tests_ln** was created.



(b) File system after the hard link **tests_ln** was created.

Figure 10.1: Schematics diagram of a file system before and after the addition of a hard link.

```
1  marcel@dell:/$: tree
2  .
3  |-- OPS105
4      |-- students
5      |-- tests
6  |-- SRT311
7      |-- students
8  2 directories, 3 files
9  marcel@dell:/$ln /OPS105/tests /SRT311/tests_ln
10 marcel@dell:/$tree
11 .
12 |-- OPS105
13     |-- students
14     |-- tests
15 |-- SRT311
16     |-- students
17     |-- tests_ln
18 2 directories, 4 files
19 marcel@dell:/$ stat OPS105/tests
20   File: 'OPS105/tests'
21   Size: 128           Blocks: 4          IO Block: 4096
       regular empty file
22 Device: 806h/2054d  Inode: 14680422    Links: 2
23 Access:(0664/-rw-rw-r--) Uid:(1000/marcel) Gid:(1000/marcel)
24 Access: 2016-08-13 14:57:16.437570016 -0400
25 Modify: 2016-08-13 14:57:16.437570016 -0400
26 Change: 2016-08-13 14:59:33.976563502 -0400
27  Birth: -
28 marcel@dell:/$ stat SRT311/tests_ln
29   File: 'SRT311/tests_ln'
30   Size: 128           Blocks: 4          IO Block: 4096
       regular empty file
31 Device: 806h/2054d  Inode: 14680422    Links: 2
32 Access:(0664/-rw-rw-r--) Uid:(1000/marcel) Gid:(1000/marcel)
33 Access: 2016-08-13 14:57:16.437570016 -0400
34 Modify: 2016-08-13 14:57:16.437570016 -0400
35 Change: 2016-08-13 14:59:33.976563502 -0400
36  Birth: -
37 marcel@dell:/$
```

Note that the file **tests** first appears on line 5, under the **OPS105** folder. After the **ln** command, a hard link **tests_ln** appears in the **SRT311** folder (see line 17). Moreover, by using the **stat** command, we can verify that both files have the same inode number and a links count of 2 (see lines 22 and 31).

10.1.2 *Hard Link Properties*

A hard link is indistinguishable from the original file itself. I.e., after the creation of hard links, the OS cannot tell which one was the original file and which ones are its hard links. They are all treated the same way.

An interesting, and somewhat counterintuitive, feature of hard links is that they would allow you to access the contents of the original file, even after the original file is deleted. This is easy to understand by looking at Figure 10.1b. In it, you can see that the hard link provides you with access to the original file's inode entry even if the the original file gets removed. In fact, the inode entry for any file is considered taken until its hard link count goes to zero. A consequence of this behaviour is that, in order to succesfuly delete a file, you need to delete the original file, as well as all of its hard links.

10.1.3 *Hard link limitations*

Hard links have two important limitations:

- A hard link cannot reference a file outside its own file system. This means that a link cannot reference a file that is not on the same disk partition as the link itself. The reason behind this limitation is that each partition has its own inode table. Hence, providing an inode number is not enough for the OS to resolve in which partition is the file located.
- A hard link may not reference a directory. This limitation prevents users from accidently creating link loops while retrieving files. Imagine a scenario where a file **`/var/log/syslog`** exists. Now, let the user create a hard link for the **`/var`** directory using: **`ln /var /var/log`**. In this scenario, the directory **`/var`** would be both a parent, and a child of **`/var/log`**. Hence, the system would find the files: **`/var/log/syslog`**, **`/var/log/var/log/syslog`**, **`/var/log/var/log/var/log/syslog`**, and so it goes.

10.1.4 *Hard Link usage*

Hard links are not as common as soft links, but they do appear in two distinct scenarios:

- To save space while performing automatic backups. Whenever a file to be saved as a backup hasn't changed, you can save memory by simply creating a hard link pointing to the last backup that was modified, as opposed to creating a full copy of it. For example, suppose that your system creates a back up version of a file **`OPS105.zip`** every month. Also, suppose

it hasn't changed since April 2016. Using hard links, you can create links called **OPS105_04_2016.zip**, **OPS105_05_2016.zip**, **OPS105_06_2016.zip**, etc. Without having to waste any memory. Also, you can delete any one of the hard links, and even the original file, without ending up with a broken link*.

*As long as at least one hard link remains.

- To allow different commands to call the same executable. For example, in Ubuntu, **bzcat**, **bunzip2**, and **bzip2**, are hard links to the same compressing tool.

10.2 SOFT LINKS

Soft links, also known as symbolic links, or even as symlinks, were created to overcome the limitations of hard links. They work by creating a special type of file that contains a text pointer to the referenced file or directory. In this regard, they operate in much the same way as a Windows shortcut*.

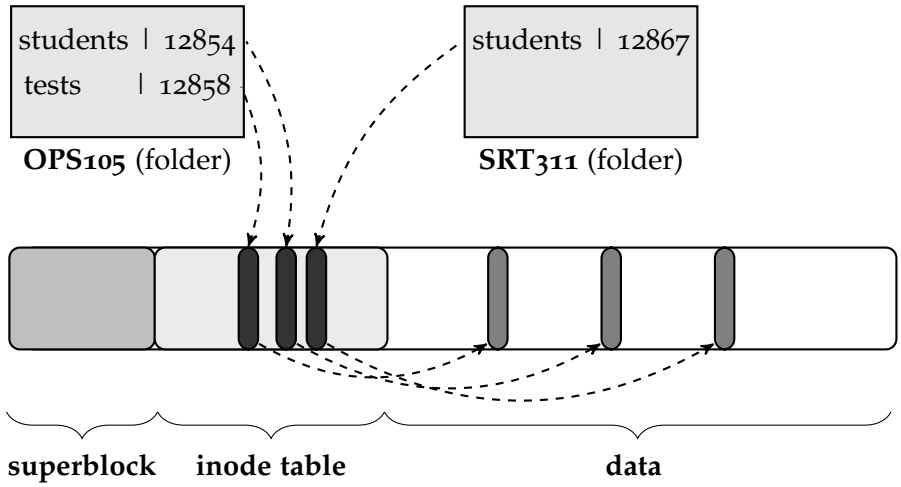
*They predate Windows shortcuts by many years, though.

In Figure 10.2, you can see a diagram showing how a file system is affected by the addition of a soft link. In this example, which is very similiar to the previous one, a file called **tests** initially exists inside a folder **/OPS105**, as it can be seen in 10.2a. Then, a soft link to this file, called **tests_ln**, is created inside a folder **/SRT311**. You can see in 10.2b that the soft link points to a new inode. Also, the file pointed by this new inode, in its turn, points to the address of the original file **/OPS105/tests**.

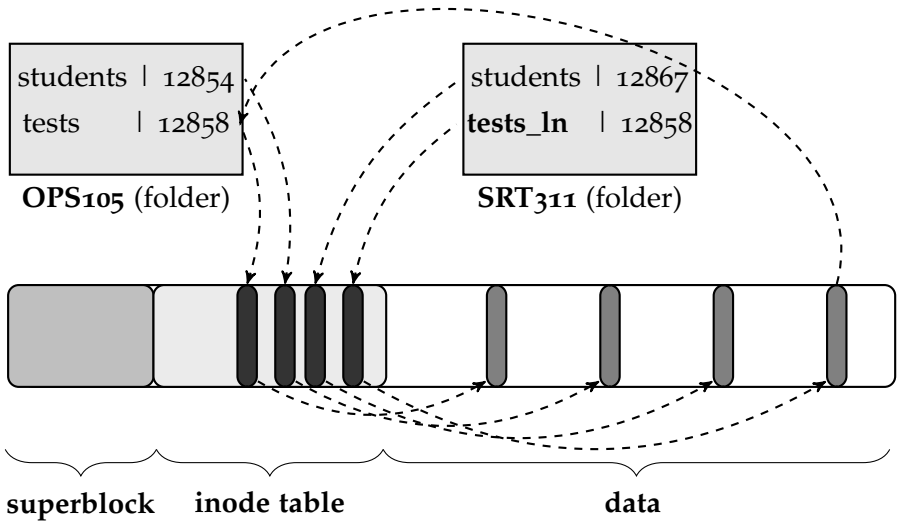
10.2.1 *ln -s*

Creating a soft link is very similar to creating a hard link. You simply need to use the **ln** command with the **-s** option, followed by two arguments: the name of the file you are creating a link for, including its absolute path, and the name of the link, including the path to the folder in which you want to place it. See the example below (which continues in Page 69):

```
1  marcel@dell:/$: tree
2  .
3  |-- OPS105
4      |-- students
5      |-- tests
6  |-- SRT311
7      |-- students
8  2 directories, 3 files
9  marcel@dell:/$ln -s /OPS105/tests /SRT311/tests_ln
10 marcel@dell:/$tree
11 .
12 |-- OPS105
```



(a) File system before the soft link tests_ln was created.



(b) File system after the soft link tests_ln was created.

Figure 10.2: Schematics diagram of a file system before and after the addition of a soft link.

```
13      |-- students
14      |-- tests
15  |-- SRT311
16      |-- students
17      |-- tests_ln -> /OPS105/tests
18  2 directories, 4 files
19  marcel@dell:/$ stat OPS105/tests
20    File: 'OPS105/tests'
21    Size: 128           Blocks: 4          IO Block: 4096
        regular empty file
22  Device: 806h/2054d  Inode: 14680422    Links: 1
23  Access:(0664/-rw-rw-r--) Uid:(1000/marcel) Gid:(1000/marcel)
24  Access: 2016-08-13 14:57:16.437570016 -0400
25  Modify: 2016-08-13 14:57:16.437570016 -0400
26  Change: 2016-08-13 14:59:33.976563502 -0400
27   Birth: -
28  marcel@dell:/$ stat SRT311/tests_ln
29    File: 'SRT311/tests_ln'
30    Size: 128           Blocks: 4          IO Block: 4096
        regular empty file
31  Device: 806h/2054d  Inode: 14680425    Links: 1
32  Access:(0664/-rw-rw-r--) Uid:(1000/marcel) Gid:(1000/marcel)
33  Access: 2016-08-13 14:57:16.437570016 -0400
34  Modify: 2016-08-13 14:57:16.437570016 -0400
35  Change: 2016-08-13 14:59:33.976563502 -0400
36   Birth: -
37  marcel@dell:/$
```

Note that the file **tests** first appears on line 5, under the **OPS105** folder. After the **ln** command, a soft link **tests_ln** appears in the **SRT311** folder (see line 17) pointing towards **/OPS105/tests**. Moreover, by using the **stat** command, we can verify that these files have different inode numbers, and that their links count is only 1 (see lines 22 and 31).

Absolute vs Relative paths for links

It is possible to use a relative path, as opposed to an absolute path, when creating a soft link. However, this practice is not recommended.

The reason for absolute paths being preferred when creating soft links is that the link must contain a path that can lead from it to the original file. I.e., it must be given a relative path with regards to the location of the link, not with regards to the location of the current working directory at the time the link was created. This can be quite confusing and lead to errors.

Another reason for preferring absolute paths, over relative paths, is that soft links created with relative paths become broken when the link is moved to a different directory. Soft links created with an absolute path, on the other hand, still work after moving to different locations.
Note that for hard links, it doesn't matter if you create them with a relative path, or with an absolute path. Once created, they will keep pointing towards the proper inode even if they are moved.

10.2.2 *Soft link properties*

When you try to access the contents of a soft link, the OS redirects you to the original file. For example, if you try to open a soft link in **`vim`**, the original file is opened. If any edits are made to it, the original file changes*.

If the original file is deleted before the symbolic link, the link will continue to exist, but will point to nothing, resulting in a broken link. In many Linux distributions, the ls command displays broken links in a different color, such as red, to indicate that they are broken.

*Note that, commands such as **`mv`**, **`cp`**, or **`rm`** will affect the link itself, not the original file.*

EXERCISES

1. Can you access a file using a soft link, after the soft link is moved to a different folder after being created?

   Given the directory tree shown in Figure 10.3, answer the following questions:

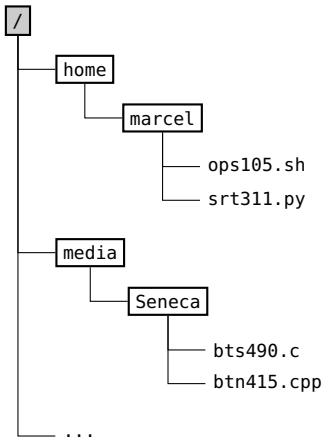


Figure 10.3: Directory tree for questions 2 and 3.

2. Create a hard link for the file **`ops105.sh`**, inside the folder **`Seneca`**. Assume that your current working directory is the root (`/`).

3. Create a soft link for the file **`srt311.py`**, inside the folder **`Seneca`**. Assume that your current working directory is the root (`/`).
4. Can you create a hard link to another hard link? How about creating a soft link that points towards another soft link?
5. What happens to a soft link when the original file it points to is deleted?
6. What happens to a hard link when the original file it points to is deleted?
7. In which scenarios are hard links preferred over soft links? In which scenarios are soft links preferred over hard links?
8. why is it important to use absolute paths while creating soft links?

11

FILE GLOBBING: USING WILDCARDS

System administrators, very often, have to act on multiple files at once. For example, they might want to move all **.pdf** files from one folder to another. As another example, they might want to list all files within a particular directory that have the string* **net** as part of its file name, such as **netcat**, **netstat**, or **networkctl**, in the **/bin** directory.

*A string is a sequence of characters.

Up until now, we have only learned how to run commands with specific file names passed as arguments. For example, **mv grades.xls old_grades.xls** renames a file called **grades.xls** in the current working directory. As another example, **rm midterm.pdf final.pdf** deletes both files **midterm.pdf** and **final.pdf**.

This approach used in the previous paragraph, i.e., specifiying the name of each file we want to work upon in the arguments list, works well when we are dealing with a small number of files. However, it is clearly impractical in scenarios where we need to deal with dozens, hundreds, or even thousands of files at once. For these scenarios, we can use wildcards* that can be used to match all files that follow a particular pattern.

*Wildcards get their name because they work as a Joker card in some card games. They can represent different things, depending on the situation.

In the **Bash** shell, there are three wildcards that can be used for different purposes:

STAR - * : This wildcard can replace any number of characters from file names, including none.

QUESTION MARK - ? : This wildcard can replace one, and only one, character from a file name.

SQUARE BRACKETS - [] : Square brackets can replace one, and only one, character from a file name. However, the space occupied by the square brackets can only be occupied by characters from a specified list.

In what follows, we cover each one of these three wildcards and provide a number of examples.

11.1 STAR - *

The star wildcard is by far the most used one. One particularly common usage of this wildcard is by itself, when it expands to all files in the current working directory. For example, by issuing a **rm *** command, all files in the working directory are deleted.

Another very common use of this wildcard is to allow users to specify all files with a specific file name ending. See the following example:

```
marcel@dell:~$ tree
.
|-- filea.pdf
|-- fileb.pdf
|-- pdf_folder
1 directory, 2 files
marcel@dell:~$ mv *.pdf pdf_folder
marcel@dell:~$ tree
.
|-- pdf_folder
    |-- filea.pdf
    |-- fileb.pdf
1 directory, 2 files
```

In this example, two **.pdf** files in the working directory are sent to a folder called **pdf_folder**. Note that the files are not specified by name, but rather by a file globbing expression ***.pdf**. If there were more **.pdf** files in the working directory, they would also be transferred to the **pdf_folder**.

The star wildcard can also be used to specify all files starting with a particular string, or even to specify files that start with a particular string and end with another string. See the examples below.

```
marcel@dell:~$ tree
.
|-- script1.sh
|-- script2.sh
|-- scripts   <-- directory
|-- script_test
|-- test1
|-- test2
1 directory, 5 files
marcel@dell:~$ rm test*
marcel@dell:~$ tree
.
|-- script1.sh
|-- script2.sh
|-- scripts  <-- directory
|-- script_test
1 directory, 3 files
marcel@dell:~$ mv script*.sh scripts
marcel@dell:~$ tree
.
|-- scripts
|   |-- script1.sh
```

```
|   |-- script2.sh
|--  script_test
1 directory, 3 files
```

In this example, all files that start with the string **test** are deleted at once using the command **rm test***. Also, all files that start with the string **script**, and end with **.sh**, such as **script1.sh**, **script2.sh**, etc., are moved to a folder called **scripts**, using a command **mv script*.sh scripts**.

File Globbing: Under the Hood

The file globbing mechanism is quite interesting. For instance, you may have assumed that the commands **rm** and **mv**, when we entered **rm test*** and **mv script*.sh** above, received **test*** and **script*.sh** as arguments, respectively. However, this is incorrect. In fact, it was the **Bash** shell that expanded these file globbing expressions into all files that match them, before calling the **rm** and **mv** commands.
In other words, the shell expanded these expressions so that, from the point of view of the **rm** and **mv** commands, they were called with: **rm test1 test2** and **mv script1.sh script2.sh**. This is clearly a smart system design choice, as it requires only the shell to implement a file globbing algorithm, as opposed to requiring all commands to implement it individually.

11.2 QUESTION MARK - ?

The question mark, as we said before, replaces one character, and one character only. It is frequently used to match sequences of file names that end on numbers, as seen in the following example:

```
marcel@dell:~$ ls
script1  script2  script2a  script_test
marcel@dell:~$ rm script?
marcel@dell:~$ ls
script2a  script_test
```

In this example, the files **script1** and **script2**, were deleted from the current working directory. The files **script2a** and **script_test**, on the other hand were left untouched.

Note that you can use more than one wildcard at once, for example, if we had used **rm script??** in the example above, we would have deleted the file **script2a**, while all other files would be left untouched.

11.3 SQUARE BRACKETS - []

Square brackets can be used to substitute one character, and one character only, by a set of characters or numbers, or even by a range of characters or numbers. Note that they are similar to the question mark wildcard in which they substitute one, and only one character. However, they are more restrictive, as they enforce a list of options for possible replacement. See the following example:

```
marcel@dell:~$ ls
script1  script2  scripta  scriptb
marcel@dell:~$ rm script[1234]
marcel@dell:~$ ls
scripta  scriptb
```

In this example, the files **script1** and **script2**, where deleted from the current working directory, while **scripta** and **scriptb**, were not. Note that if there were files with names **script3** or **script4**, they would also be deleted from the current working directory.

The expression **[1-4]** could have been used in the example above to replace **[1234]**. It means: any number in the range between 1 and 4. We could also have used **[a-z]** or **[A-Z]** to replace any lowercase or uppercase alphabetical character, respectively*. It is possible to select a list of forbiden charactes, as opposed to a list of acceptable characters. To do so, you simply need to start the list with an exclamation mark (**!**). For example, to allow anything but lowercase characters to replace a particular character in a file name, you can use **[!a-z]**.

*In order to replace any alphabetical character, regardless of its case, we could have used the expression **[a-zA-Z]**, or **[A-z]**.

11.4 ESCAPING SPECIAL CHARACTERS

The shell will, by default, assume that the characters *****, **?**, and square brackets **[]** are to be expanded as wildcards. However, there might be scenarios in which your file names have these characters. In order to match these specific characters, as opposed to use them as wildcards, you simply need to escape them with a backslash character *****. See the example below:

*The backslash character also needs to be escaped. For example, to create a file called **script**, we should enter **touch script**.

```
marcel@dell:~$ ls
script*  script1  script2 scripta  scriptb
marcel@dell:~$ rm script\*
marcel@dell:~$ ls
script1  script2 scripta  scriptb
```

In this example, only the file **script*** was deleted with the command **rm script***. Note that if we had used **rm script***, instead, all files would have been deleted from this directory.

EXERCISES

For the exercises below, create an empty directory and run the following command within this directory: **`touch myscript script script.sh script1.sh script2.sh scripta.sh scriptb.sh mytest test test.sh testa.sh testb.sh`**. Also, create subfolders called **`scripts`** and **`tests`** with a **`mkdir scripts tests`** command.

For each question, you need to issue a single command that will perform the required task. The use of wildcards is required. After each question, make sure to restore the directory back to its initial state by running **`rm *`**, and then the **`touch`** and **`mkdir`** commands specified above.

EXERCISES

1. Delete all files containing the string **`script`**.
2. Move all files ending in **`.sh`** to the **`scripts`** folder.
3. Delete the files **`script1.sh`** **`script2.sh`**.
4. Move the files **`scripta.sh`**, **`scriptb.sh`**, and **`script1.sh`** to the **`scripts`** folder.
5. Move all files containing the string **`test`** to the **`tests`** folder.
6. Delete all files containing the string **`test`** in it that end with **`.sh`**.
7. Delete all files from this directory.

12

GREP AND REGULAR EXPRESSIONS

I have, many times in the past, googled excerpts of song lyrics that got stuck in my head, in the hope that I would find the rest of it, or at least the song's title. Similarly, system administrators can find themselves looking for a specific file whose name they cannot recall, but they can remember excerpts of its contents. In other cases, the sysadmin might want to find in which line of a given file (which can be pretty long), a particular string occurs*.

*This is a very common scenario when dealing with configuration files.

To help with the scenarios presented in the previous paragraph, most Linux distributions come by default with a tool called **grep*** to search for specific strings, or string patterns, inside text files.

*Short for global regular expression print.

In wat follows, we will cover some basic usage of the **grep** tool. Following, we will introduce the concept of regular expressions, which are often used while performing grep searches.

12.1 grep

The **grep** command is called with the following syntax:

```
grep "STRING" LIST OF FILENAMES
```

I.e., it takes at least two arguments: first, the string of characters we are looking for*, and then the names of the files in which we are performing this search on. In its output, **grep** prints on the terminal the lines in the specified files in which the string was found. See the example below:

*It is a good practice to always write the string of characters inside double quotation marks.

```
marcel@dell:~$ more poem
Nature's first green is gold,
Her hardest hue to hold.
Her early leafs a flower;
But only so an hour.
Then leaf subsides to leaf.
So Eden sank to grief,
So dawn goes down to day.
Nothing gold can stay.
marcel@dell:~$ grep "gold" poem
Nature's first green is gold,
Nothing gold can stay.
marcel@dell:~$
```

Note that we can use file globbing expressions, as opposed to writing a complete list of files. For example, if we wanted to search for lines

in all files in the current working directory containing the string **gold**, we could have used **grep "gold" ***.

12.1.1 *grep options*

The **grep** command offers to system administrators a plethora of options. For instance, it can print line numbers together with the lines in which the required string was found. As yet another example, it can print just the name of the file in which a particular string was found, which is particularly useful when using file globbing techniques. A small list of **grep** options is listed below in Table 12.1*.

*For a complete list, check **grep**'s manual page.

-i	**grep** is, by default, case sensitive. Using this option makes it become case insensitive.
-w	Searches for isolated words. Without this option, a **grep** search for a string **"cat"** would also return lines with words such as **cat**terpilar, **cat**ch, con**cat**enate, etc. Using this option, **grep** looks for strings that are complete words on their own right.
-v	Reverse grep. Displays only the lines in which there are no matches.
-l	Displays only the name of the files in which at least one match was found.
-n	Shows the line numbers in front of the lines in which matches were found.

Table 12.1: **grep** options.

12.2 REGULAR EXPRESSIONS

Sometimes we want to search for strings that follow a particular pattern, instead of specific strings. For example, postal codes in canada always follow the pattern **LNL NLN**, where each **L** stands for an uppercase alphabetical character, and each **N** stands for a number between 0 and 9. So, **M3J 3M6**, **M5V 1J1**, and **T6G 2R3** are valid postal codes, whereas **ABC 123**, **A12345**, and **456-1234** are not.

Clearly, there are more scenarios in which we might want to search for strings that follow a particular pattern. Among these: phone numbers, addresses, social insurance numbers, emails, are just a handful of examples.

In order to work with patterns, as opposed to specific strings, **grep** makes use of regular expressions, also known as regex. A regex normally contains a number of literal characters, as well as special characters, that are interpreted in order to delineate a desired pattern. In

Table 12.2, we introduce a few regex special characters. Following, we present a few examples.

`\`	If applied to an special character, it indicates that the special characer should be treated as a regular character. *ex: "*" matches the star character.*
`\s`	Matches a space.
`^`	Forces the pattern to match at the beginning of the line. *ex: "^A" provides no matches in a line: "an A".*
`$`	Forces the pattern to match at the end of the line. *ex: "$\." provides no matches in a line: "No periods here"*
`*`	Matches the preceding character 0 or more times. *ex: "bo*" provides matches in lines with "b", "bo", "boo", etc.*
`\+`	Matches the preceding character 1 or more times. *ex: "bo\+" provides matches in lines with "bo", "boo", "booo", etc., but not with "b"s not followed by at least one "o"*
`\?`	Matches the preceding character 0 or 1 time. *ex: "bo\?" provides matches in lines with "b" or "bo". Note that, unless the option* **`-w`** *is used, "boo" would also provide a match, as it contains the string "bo"*
`\{N\}`	Matches the preceding character exactly **`N`** times. *ex: "bo{2}" provides matches in lines with "boo", but not lines with the character "b" by itself or with "booo"*
`\{N,M\}`	Matches the preceding character **`N`** to **`M`** times.
`\{N,\}`	Matches the preceding character **`N`** or more times.
`.`	The period character matches any single character. *ex: "c.t" provides matches in lines with "cat", "cut", etc.*
`[]`	Square brackets matches single characters, as long as they belong to a specified list. *ex: "c[ae]t" provides matches in lines with "cat" and "cet", but not with "cit", "cut", etc.*

Table 12.2: Special characters for regex.

Note that these special characters are often employed together. For example, the regex **`"[0-9]\{5\}"`** matches any number with 5 digits. As another example, the regex **`".*"`** matches everything.

12.2.1 *Examples of* ***`grep`*** *commands using regex*

Email addresses always have the following structure: they start with a few characters, followed by an **`@`**, followed by a few more characters, followed by a period sign, an finished by a few more characters. For example, john@mycommany.com, jane@mycollege.ca. To capture

lines containing emails from a file called **students**, we can use the following command:

```
grep "[a-z]*@[a-z]*\.[a-z]*" students
```

Note that in this example, emails containing numbers, underlines, or period signs before the **@** character would not be detected. In order to detect these emails, we could change our command to to:

```
grep "[a-z0-9\._]*@[a-z0-9\._]*\.[a-z]*" students
```

Postal codes in Canada, as discussed at the beginning of this section, follow the pattern **LNL NLN**. Hence, a simple **grep** command that would capture lines containing postal codes in the **students** file would be:

```
grep "[A-Z][0-9][A-Z] [0-9][A-Z][0-9]" students
```

Regular expressions facts

- In some ways, regular expressions are similar to file globbing expressions. However, they tackle a very different problem. Indeed, while file globbing deals with matching complete file names, regular expressions aim at finding strings within large chunks of text. As a result, they are implemented very differently.
- Regular expressions are used in many areas. In fact, most search and find tools implement them. Also, many programming languages such as Pearl, Python, and JavaScript, among others work with regular expressions.
- Regular expressions can be quite complex. In fact, a number of books have been dedicated solely to them. In our case, we will simple cover the basics.

EXERCISES

For all questions below, you should write **grep** commands to capture all lines containing the required patterns without capturing any lines not containing the required patterns. Use the following **students** text file (copy and paste it into a text file):

```
marcel@dell:~$ more students
John Smith
(416) 123-2345
M4Z 2P3
john.smith@myseneca.ca
K23a!5

Mohammad Ali
(905) 231-3381
N3P 4A1
mali@myseneca.ca
jE_3sa4G

Akira Kurosawa
(905) 872-1221
M4S 1X4
akira@myseneca.ca
2S!aTe6

Priyanka Singh
(416) 431-3231
M3S 5N2
psingh4@myseneca.ca
X32Dsa
```

1. Student names. Doesn't contain special characters, just alphabetical characters and spaces.
2. Telephone numbers. (NNN) NNN-NNNN
3. Postal codes. LNL NLN
4. Student emails. Note that all students should have an email ending with @myseneca.ca
5. Passwords. Only have alphanumerical characters, as well as the special characters _, !, and ?. Should have between 6 to 8 digits.

13

BASIC FILTERS

In this chapter, a number of filters that can be used on text files is presented. These filters can be used to join (concatenate) multiple text files (**cat**), show only the first few lines of a text file (**head**), show only the last lines of a text file (**tail**), sort the contents of a text file based on some criteria pertaining one of its columns (**sort**), remove unwanted columns of a text file (**cut**) or even make small changes to a text file such as substituting one word by another (**sed**).

13.1 cat

The **cat*** command can be used for two distinct goals:

*Short for concatenate.

- To display the contents of a short text file
- To concatenate the contents of multiple text files

For the first goal, **cat** works in a very similar way to **more**. You need only to call it together with the name of the file you wish to display. The contents of the file are then displayed on the terminal and you are granted with a new command prompt. In this regard, the major difference between **cat** and **more** is that, for long text files, **cat** doesn't provide the user with a way to navigate the file page by page. The whole file is displayed at once. See the example below:

```
marcel@dell:~$cat poem
Nature's first green is gold,
Her hardest hue to hold.
Her early leafs a flower;
But only so an hour.
Then leaf subsides to leaf.
So Eden sank to grief,
So dawn goes down to day.
Nothing gold can stay.
marcel@dell:~$
```

For the second goal, you need only to call **cat** together with a list of files you are trying to concatenate. The shell will display the contents of all the required files on your terminal one on top of another, and then providing you with a new command prompt. See the following example, where the following text files were used:

```
Nature's first green is gold,
Her hardest hue to hold.
Her early leafs a flower;
But only so an hour.
Then leaf subsides to leaf.
So Eden sank to grief,
So dawn goes down to day.
Nothing gold can stay.
```

Listing 13.1: Poem1 text file

```
What's in a name? That which we call a rose
By any other name would smell as sweet.
```

Listing 13.2: Poem2 text file

```
marcel@dell:~$ cat poem1 poem2
Nature's first green is gold,
Her hardest hue to hold.
Her early leafs a flower;
But only so an hour.
Then leaf subsides to leaf.
So Eden sank to grief,
So dawn goes down to day.
Nothing gold can stay.
What's in a name? That which we call a rose
By any other name would smell as sweet.
marcel@dell:~$
```

It is important to note that the original files are not modified in any way. The shell simply displays the concatenation of their contents on the terminal, without altering them. We will cover methods to create text files based on the results of tools such as **cat** in Chapter XXX.

13.2 **head**

Sometimes, a system adminstrator or developer is not interested in all the contents of a text file, but rather in just a part of it. For example, most professional source codes start with a few lines describing its functionality. For such scenarios, opening the entire file is simply a waste of time.

A tool called **head** is provided for cases in which only the first few lines of a text file is of interest. This tool behaves nearly identically to the first usage of the **cat** described previously. The only difference is that it only displays, by default, the first ten (10) lines of the file.

In case you want to display a number of lines different than ten, this command can be called with the **-n** option, followed by the number of lines you wish to display. See the example below:

```
marcel@dell:~$ head -n 4 poem1
Nature's first green is gold,
Her hardest hue to hold.
Her early leafs a flower;
But only so an hour.
marcel@dell:~$
```

13.3 tail

Quite frequently, when we analize log files, we are most interested into the latest events that have been logged. These events are normally written a the bottom of the log file. For example, in a server, there can be a log file that keeps tracks of all network requests. If a malicious request caused the system to shut down, you can verify which request caused the shut down by looking at the last entry in the log file.

For scenarious such as browsing log files, as discussed above, a tool called **tail** is made available. Its behaviour is nearly identical to that of **head**. The only differece is that, instead of showing the first ten (10) lines of the file, it shows the last ten (10) lines of it. Also, just like with the **head** command, you can specify the number of lines to display using the **-n** option followed by the number of lines you wish to display. See the example below:

```
marcel@dell:~$ tail -n 3 poem1
So Eden sank to grief,
So dawn goes down to day.
Nothing gold can stay.
marcel@dell:~$
```

If you want to look at log files that are currently being updated, you can use **tail** with the **-F** option. For example, **tail -F /var/log/syslog** will display the last ten lines of **syslog***, and update these lines if a new content is added to file. To stop monitoring the file, and get a command prompt back, you need to press **Crt+C**.

**This is a log file that collects messages from various programs and services, including the kernel.*

13.4 tac

In some scenarios, such as when analyzing some log files, you may want to display an entire file with the last lines appearing first. I.e., you may wish to read the a particular file with its lines displayed in reverse order. For these scenarios, the **tac*** is made available. Like **cat**, **tac** can also be used to concatenate files. However, in the result, each individual file will appear with its lines in reverse order. See the example below:

Stands for **cat spellt backwards.*

```
marcel@dell:~$ tac poem2
By any other name would smell as sweet.
What' s in a name? That which we call a rose
marcel@dell:~$
```

13.5 **sort**

Suppose you have data stored in a text file like the one shown below, where each line represents one element, and each column represents one category.

```
USA     321     9,826
China   1,367   9,596
UK      64      243
Brazil  204     8,511
Canada  35      9,984
India   1,251   3,287
Egypt   88      1,001
```

Listing 13.3: Countries file.

In this example, the first column represents the country's name, the second line its population (in millions), and the third line represents its area (in thousands of square kilometers). Each line, or row, represents one country.

It is clear that this file is not sorted in any obvious way. Not alphabetically, not by population, and neither by area. However, there is a tool called **sort** that can be used to sort files like this in any way you choose. The choices of how to sort the file, i.e., by which column, in which order, etc., are made by calling **sort** with the proper set of options. In Table 13.1, we present some of the most common options:

-k N	Sorts the file based on its N*th* column.
-r	Sorts in reverse order, i.e., from Z to A, in case of an alphabetical sorting, or from larger to smaller in case of a numerical sort.
-n	Sorts numerically.
-t	Chooses a column separator other than non-blank to blank. You need to enter the column separator inside double quotation marks

Table 13.1: **sort** options.

In he examples below, we show a number of sorting operations on the Listing 13.3. Note that, just like with **cat**, **sort** does not alter the

files provided as an argument. It simply displays the sorted version of it on your terminal. We will cover a method to save these alterations in Chapter XXX.

```
marcel@dell:~$ sort countries
Brazil    204     8,511
Canada    35      9,984
China     1,367   9,596
Egypt     88      1,001
India     1,251   3,287
UK        64      243
USA       321     9,826
marcel@dell:~$ sort -k2 -n countries
Canada  35  9,984
UK  64  243
Egypt   88  1,001
Brazil  204 8,511
USA 321 9,826
India   1,251 3,287
China 1,367 9,596
marcel@dell:~$ sort -k3 -n -r countries
Canada    35      9,984
USA       321     9,826
China     1,367   9,596
Brazil    204     8,511
India     1,251   3,287
Egypt     88      1,001
UK        64      243
```

In Listing 13.3, different columns (fields) are separated by tabs. By default, **sort** uses blank to non-blank transitions, i.e., transitions from spaces and tabs to visible characters, to mark the beginning of each column. In case we have a different separator*, we need to indicate the separator we are using with the option **-t** followed by the chosen separator written inside double quotes, as seen in the example below:

*Such as in comma-separated values, CSV, files.

```
marcel@dell:~$sort -t"," -k2 -n countries
Canada,35,9984
UK,64,243
Egypt,88,1001
Brazil,204,8511
USA,321,9826
India,1251,3287
China,1367,9596
```

Numerical versus Alphabetical sorting

When we want to sort a number of items alphabetically, we need only to compare the first character of each item. For example, we can use the fact that **Adam** starts with an **A**, and **Bryan** with a **B**, to sort these two items, regardless of the rest of them. In the cases where the first character of two different items is the same, we need to check the second character. For example, we can check that **Adam** comes before **Aidan**, because **d** comes before **i**. The same goes if the first two characters are the same, in which case the third item is compared, and so it goes.
When performing a numerical sort, we always need to look at the whole number. If we only look at the first character, we would have reached the conclusion that the number **9** is greater than **10**, because **9** is greater than **1**. This is clearly not what we want when sorting numbers. Hence, this is why **sort** needs to know if the user is trying to sort things numerically or alphabetically. By default, **sort** assumes that an alphabetical sort is taking place.

13.6 **cut**

When dealing with data stored in text files, we are often only interested in a subset of it. For example, in Listing 13.3, we could be interested only in the countries population, or only in their area, as opposed of both. For scenarios like these, **cut** comes in handy. As the name suggests, it is used to cut, or in other words remove, any contents from a text file that the user doesn't need.

In the example shown below, we first use this filter to remove the information about the countries' area. Following, we use it to remove all information about their population.

```
marcel@dell:~$ cut -f 1,2 countries
USA     321
China   1,367
UK      64
Brazil  204
Canada  35
India   1,251
Egypt   88
marcel@dell:~$ cut -f 1,3 countries
USA     9,826
China   9,596
UK      243
Brazil  8,511
Canada  9,984
```

```
India   3,287
Egypt   1,001
```

Note that, with **cut**, we need to specify the fields, or categories, that we wish to keep, not the ones we want to cut. This is done by using the **-f** option, followed by the column numbers separated by commas. If you are insterested in a range of columns, for example columns one (1) to three (3), you can use a dash, as in **cut -f 1-3 countries**.

Note that, **cut** uses a tab as its default delimiter, not the transition from blank to non-blank characters like **sort**. If multiple tabs, or spaces and tabs occur between two columns, **cut** will probably not behave the way you expect. In case any other separator is used, for example, commas or colons, the user needs to specify the separator using the **-d** option followed by the desired separator, written between single quotes. See the example below for a **cut** command being applied to a CSV version of Listing 13.3.

```
marcel@dell:~$ cut -d',' -f 1,2 countries
USA,321
China,1,367
UK,64
Brazil,204
Canada,35
India,1,251
Egypt,88
```

13.7 sed

sed* can be used to find lines containing a particular word or regex, similarly to **grep**, or to display all lines that do not contain a particular word or regex, which can also be done using **grep**. However, there is one particular task for which **sed** excels, and for which it is arguably the best tool available: replacing particular words or expressions in a text file with other words or expressions*.

*Short for stream editor.

sed works by parsing the provided text file line by line, and then making changes whenever there is a match to the provided word or regex. The syntax to use **sed** to search and replace words or regex inside a text file is the following: **sed "s/original_word/replacement_word/" file_name**. See the example below:

*Like find and replace tools that exist in most text editors.

```
marcel@dell:~$ cat poem
A rose is a rose is a rose
Gertrud Stein
marcel@dell:~$ sed "s/rose/flower/" poem
A flower is a rose is a rose
Gertrud Stein
```

Note that, in the example above, only the first occurrence of the word **rose** was replaced by **flower** on the first line. This happens

because, by default, **sed** skips to the next line after finding a match. In order to replace all words or regex in these scenarios, you need to specify that you want to perform a global search an replace, using the following syntax: **sed "s/original_word/replacement_word/g" file_name**.

```
marcel@dell:~$ cat poem
A rose is a rose is a rose
Gertrud Stein
marcel@dell:~$ sed "s/rose/flower/g" poem
A flower is a flower is a flower
Gertrud Stein
```

Note that, just like all other filters discussed in this chapter, **sed** does not alter the text file provided as an input. It will simply display its result in the terminal. In Chapter XX, we will discuss methods to save the alterations made by **sed**.

EXERCISES

For all questions below, you should use the following text files (copy and paste them into three text files called **poem1**, **poem2**, and **countries** repectively):

```
What's in a name? That which we call a rose
By any other name would smell as sweet.
```

```
A rose is a rose is a rose
```

```
USA     321     9,826
China   1,367   9,596
UK      64      243
Brazil  204     8,511
Canada  35      9,984
India   1,251   3,287
Egypt   88      1,001
```

1. Concatenate both poems, displaying the result on the terminal.
2. Concatenate both poems, but displaying each one of them in reverse order. I.e., each poem should start at its last line, and proceed all the way to its the first.
3. Display only the first three lines of the file **countries**.
4. Display only the last three lines of the file **countries**.
5. Use **tail** with the proper options to keep track of the last five (5) lines of a log file called **sysgen.log** that is currently being updated.

6. Sort the **countries** file based on its population (second column) in ascending order.

7. Sort the **countries** file based on its area (third column) in descending order.

8. Use **sed** to display the contents of **poem2** with all instances of the word **rose** replaced by **flower**.

9. Does the **sed** command changes the file provided as an argument? For example, after running a **sed** command on a file called **poem2**, do the modifications performed by **sed** get stored in memory?

14

AWK, RENAME, AND FIND

In our previous chapter, we have covered a number of simple tools that could be applied to text files, such as **cat**, **sort**, **sed**, etc. In this chapter, we will cover some slightly more complex, but incredible powerful, tools that allow us to filter lines of a text file based on a chosen criteria (**awk**), rename multiple files at once (**rename**), and find specific files according to different search criteria (**find**).

14.1 awk

In previous chapters, we showed how to use **grep** to display lines of text that contained a particular word or regex. However, what if we want to display all lines of a particular text file that satisfy a particular pattern (condition), instead of a regex? For example, see the text file below:

```
toyota corolla 1970 2500
chevy malibu 1999 3000
ford mustang 1965 10000
volvo s80 1998 9850
ford thundbd 2003 10500
chevy malibu 2000 3500
honda civic 1985 450
honda accord 2001 6000
ford taurus 2004 17000
toyota rav4 2002 750
chevy impala 1985 1550
ford explor 2003 9500
```

Listing 14.1: Car dearlership text file.

Listing 14.1 contains a number of columns with information about cars at a car dealership. The first column contains the car model, the second column its manufacturer, the third its year, and the fourth its price.

Clearly, we can use **grep** to display all the lines in which the car manufacturer is **Toyota** or **Ford**. However, we we cannot simply write a regex and use **grep** to display all lines in which the car is from 1999 or newer. Also, we wouldn't be able to use a regex to display all lines in which the cars' prices are less than 15,000 CADs. For these scenarios, another tool, called **awk*** can very useful.

**The name of this command stands for the first letter of the name of its authors.*

awk is a pattern-scanning and processing language designed for Unix, and later rewritten by the GNU project as **gawk***. It searches for patterns and/or conditions in text files and displays the lines in which they were found. For instance, using Listing 14.1, we could have retrieved all lines containing cars made during or after 1999 with the following syntax:

*See the **awk** vs **gawk** text box.

```
marcel@dell:~$ awk '$3 >= 1999' cars
chevy malibu 1999 3000
ford thundbd 2003 10500
chevy malibu 2000 3500
honda accord 2001 6000
ford taurus 2004 17000
toyota rav4 2002 750
ford explor 2003 9500
```

Also, we could have retrieved all lines containing cars with a price tag at or below 9,000 CADs, with the following syntax:

```
marcel@dell:~$ awk '$4 <= 9000' cars
toyota corolla 1970 2500
chevy malibu 1999 3000
chevy malibu 2000 3500
honda civic 1985 450
honda accord 2001 6000
toyota rav4 2002 750
chevy impala 1985 1550
```

Note that we use the notation **$N**, where **N** denotes the number of the column we want to check for an specific pattern. Also, note that we should always put the complete pattern statement within single quotes. Some of the most important patterns that can be used with **awk** are described in Table 14.1.

<	Less than
<=	Less or equal than
==	Equal to
!=	Not equal to
>	Greater than
=>	Greater or equal than

Table 14.1: **awk** patterns.

Note that nothing prevents us from applying multiple patterns at once. For example, if we wanted to display all lines containing cars made during or after 1999 with a price tag at or below 15,000 CADs, we could have used the following syntax:

```
marcel@dell:~$ awk '$4 <= 15000 && $3 >= 1999' cars
```

```
chevy malibu 1999 3000
chevy malibu 2000 3500
honda accord 2001 6000
toyota rav4 2002 750
```

Where the **&&** operator denotes a logical **and**. I.e., it only returns the lines in which both conditions are met. If we wanted to display all lines in which either one condition **or** another condition were met (or both), we could have used the **||** operator.

Before we conclude this section, it is worth to note that **awk** is capable of doing much more than what we have described here. However, most of its other uses can also be carried out by other commands we have previously discussed such as **grep**, **cut**, or by combining commands, as we will discuss in Chapter XXX.

awk vs **gawk**

In this book, we have been using tools such as **ls**, and **cat**, that were rewritten by the GNU Project to replace Unix tools with the same name. For example, when we discussed **cat**, we discussed the GNU Project implementation of the **cat** tool originally written for Unix. However, when the **awk** tool was rewritten, the GNU Project developers added a **g** to its name, and then called it **gawk**.
As a result of this name change, some Linux distributions nowadays require users to write **gawk** to use this tool, whereas other distributions require users to write **awk**. Some Linux distributions even allow users to write **gawk** or **awk** to call the same tool.

14.2 rename

We have seen in Chapter 6 that the we can use **mv** to rename individual files. For example, **mv original_name new_name** changes a file name from **original_name** to **new_name**. However, what if we want to change the name of multiple files at once? For example, what if we want to change the ending of all **.html** files from a website to **.php** so that they could be processed by a server-side script prior to be delivered to the end-user? For such scenarios, **rename** can be a very helpful tool.

rename uses regex to match patterns for filenames, and substitute it by a string literal. It also uses wildcards to indicate in which files the pattern matching should be applied upon. For instance, to change the ending of all **.log** files in a directory to **.bak**, we could have used the following command*:

```
marcel@dell:website$ ls
index.html contact.html about.html
```

*The syntax presented below only works in Debian based systems, such as Ubuntu.

```
marcel@dell:website$ rename 's/\.html$/\.php/' *
marcel@dell:website$ ls
index.php contact.php about.php
```

Note that **rename** uses a syntax very similar to **sed**, discussed in the previous chapter. I.e., it is called with:

```
rename 's/original\_regex/replacement\_string/' wildcard
```

Also, note that **\.** is used to indicate that we want to use the period as a literal, as opposed to be a regex that can stand for any character.

As another example, we might have wanted to delete the ending of **.log** files, as opposed to switched it to something else. This could have been accomplished with:

```
marcel@dell:log$ ls
netlist.log tcpdump.log read_me
marcel@dell:log$ rename 's/\.log$//' *.log
marcel@dell:log$ ls
netlist tcpdump read_me
```

14.3 **find**

As the name suggests, **find** can be used to find files based on a number of possible search criteria. For example, you can search all files in a given directory whose name matchs a particular wildcard pattern*. As another example, you can use **find** to search for all files in your system that were created after a certain date. You can even use this tool to search for all files that belong to a particular user, or have a particular set of permissions, among other possibilities. The syntax for the **find** is:

*Note that we need to use wildcards, not regex with this tool.

```
find target\_directory criteria\_type criteria\_match
```

In case **criteria_match** uses wildcards, it needs to be placed within single quotes.

On table 14.2, you can find a list of different types of searches that can be done using **find**. Note that this list is not complete. For a complete list, refer to **find**'s man page.

Below we have a few examples of **find** being used in different forms:

```
marcel@dell:~$ ls -l
-rw-rw-r-- 1 marcel marcel    125  Sep 18 10:52 assign.pdf
-rw-rw-r-- 1 john john        3212 Sep 23 21:51 classes.docx
-rw-rw-r-- 1 john john        450  Sep 15 14:22 test1.pdf
-rw-rw-r-- 1 marcel marcel    1232 Jul 19 12:44 honesty.pdf
marcel@dell:~$ find . -name '*.pdf'
.
./assign.pdf
```

```
./test1.pdf
./honesty.pdf
marcel@dell:~$ find . -user john
.
./classes.docx
./test1.pdf
marcel@dell:~$ find . -mtime -10 #today is September 23
.
./assign.pdf
./classes.docx
./test1.pdf
marcel@dell:~$ find . -maxdepth 1 -size +1k
.
./classes.docx
./honesty.pdf
```

By default, **find** searches not only the provided directory, but also all its subfolders. You can change this behaviour by using the **-maxdepth** option with a value of **1**. You can also use it to perform searches on the current working directory and its imediate subfolders, by setting **-maxdepth** to **2**, and so on.

-name	Searches for files that match a provided wildcard.
-size	Searches for files whose size is smaller, equal, or greater than a provided number.
-type	Searches for all files of a particular type. Possible types are: regular files (**-f**), directories (**-d**), symbolic links (**-l**), among others.
-user	Searches for all files that belong to a particular user.
-samefile	Searches for all files with the same inode number as a file provided as an argument. In other words, it finds all hard links of a particular file.

Table 14.2: **find** search criteria types.

EXERCISES

1. Which command can be used to list all cars in Listing 14.1 that were built after 2000?
2. Which command can be used to list all cars in 14.1 that cost less than 15000 CADs?
3. Which command can be used to list all cars in 14.1 that were built after 2000 and cost less than 15000 CADs?
4. Which command can be used to change the ending of all **.jpeg** files in the current working directory to **.jpg**?
5. Which command can be used to change the ending of all **.jpeg** files in the current working directory to **.jpg**?
6. Which command can be used to change all uppercase letters, in all files in the current working directory, to lowercase?
   *hint: check **rename**'s man page.*
7. Write down a command to search all files with more than 20kb inside the current working directory, as well as inside its imediate subfolders (but not on subfolders of subfolders).
8. Write down a command to search all files that have been modified in the past week inside the current working directory, as well as in all its subfolders.
9. Write down a command to search all files that have been accessed after a file named **syslog** has been modified.
10. Write down a command to search for all hard links of a file called **file**, that exists in your current working directory, in your entire file system.

15 REDIRECTION AND PIPING

In Chapters 13 and 14, the results of all examples were displayed on the terminal. I.e, no permanent changes were made to the text files provided as arguments, or any other files in the system. Also, we have worked with each tool by itself. I.e., we did not try to combine tools together, as according to the Unix philosophy.

In this chapter, we will address these two issues. First, we will show how to use redirection to save outputs in text files, as opposed to displaying them in the terminal. Following, we will cover how to combine two or more tools together using the concept of piping.

15.1 REDIRECTION

The concept of redirection is connected to that of data streams, which are the the source of system inputs, as well as the destination of system outputs.

There are three data streams in **bash**, one input, and two outputs, which are listed on Table 15.1.

stdin	The standard input, as the name suggests, refers to the source of the data that is being fed into the shell. It is normally linked to the keyboard.
stdout	The standard output, as the name suggests, refers to where the output of any sucessful commands should go. By default, it is linked to the terminal.
stderror	The standard error stream refers to where the output of unsucessful commands should go. Like **stdout**, it is also linked to the terminal by default.

Table 15.1: Bash data streams.

In Figure 15.1, you can see a depiction of these three data streams, as well as their relationship with the shell.

Even tough both output streams are linked to the terminal display by default, separating them has one great advantage. It allow users to deal with the output of commands in different ways, depending if they were succesful or not. For example, you can choose to save

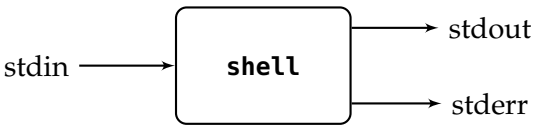


Figure 15.1: Visual representation of data streams.

the output of succesful commands to a particular file, but send error messages to **/dev/null**[*].

[*] *See **/dev/null** box below.*

/dev/null

In Linux systems, **/dev/null** is a simple device implemented in software, which does not correpond to any hardware device. Simply put, it is used to throw information away, or to test reading with empty information. In a nutshell:

- **/dev/null** looks as an empty file when you try to read from it using commands such as **cat** or **less**.
- Redirecting data to **/dev/null** results in that data being thrown away. It simply "disappears" from the system.

15.1.1 *Redirection Syntax*

stdout

To redirect the standard output, **stdout**, to a given file, you need just to add **> file_name**, or **1> file_name**, to the end of the command. In the example below, which uses the file **countries** presented in Listing 13.3, we save the output of a sorting operation into a file called **countries_sorted**.

```
marcel@dell:~$ sort countries > countries\_sorted
marcel@dell:~$ cat countries\_sorted
Brazil    204     8,511
Canada    35      9,984
China     1,367   9,596
Egypt     88      1,001
India     1,251   3,287
UK        64      243
USA       321     9,826
```

It is important to note that because we are redirecting the output of the sorting operation, we don't get any feedback on our terminal. If we want to see the result, we need to use a command to read the **countries_sorted** file. In the example above, we used **cat**. Also, note that if the file **countries_sorted** already exists, it is overwritten. In

case the command results in an error, the file **countries_sorted** will be empty.

In fact, when applying redirection, the first action taken by the shell is to erase any data from the destination file. This can lead, sometimes, to an unexpected behaviour. For example, a command **sort -k2 countries > countries** will result in an empty file. This happens because the file **countries** will be emptied prior to the sorting operation.

stderr

To redirect the standard error, **stdout**, to a given file, you need just to add **2> file_name**, to the end of the command.

You can also redirect both output streams to the same file by adding **&> file_name**, or you can redirect each stream to an specified file using **1> file_name1 2> file_name2** at the end of the command.

stdin

The standard input normally comes from the keyboard, in the form of arguments that we provide. For example, **ls folder_name** uses the **stdin** to provide the argument **folder_name** to the **ls** command.

More often than not, commands that are regularly applied to files accept both the file name, or a redirection from the file. For example, both commands below are equivalent:

```
marcel@dell:~$ head -n3 countries\_sorted
Brazil    204     8,511
Canada    35      9,984
China     1,367   9,596
marcel@dell:~$ head -n3 < countries\_sorted
Brazil    204     8,511
Canada    35      9,984
China     1,367   9,596
```

Because of equivalences such as the one showed in the example above, scenarios in which you need to redirect the **stdin** are not as common as scenarios in which **stderr** and **stdout** are redirected.

15.2 PIPING

The concept of piping commands together is quite simple. It can be summarized as: *"using the output of one command as an input to another command"*. In fact, the name piping derives from an analogy with a real world pipeline, where the output of each pipe becomes the input of another pipe further down the pipeline.

Regardless of its simplicity, piping is a very powerful concept. To better understand its implications, we will discuss in what follows a scenario in which piping commands together solves a particular problem.

15.2.1 *Piping Application Scenario*

A college keeps the records of all their students in separate files, with each file corresponding to the programs they are enrolled such as the ones shown in Listings 15.1 and 15.2*. Now, suppose that you are requested to come up with a list of the top three students in the entire college, based on their GPAs.

*In a real world scenario, these files would be much bigger.

```
John Smith john.smith@mycollege.ca 2.9
Jane Doe jane.doe@mycollege.ca 3.9
Mohammad Ali mohammad.ali@mycollege.ca 3.4
Chun Li chun.li@mycollege.ca 3.1
```

Listing 15.1: Computing Science students information (cs.info).

```
Aditya Kapoor aditya.kapoor@mycollege.ca 3.9
Juan Sanchez juan.sanchez@mycollege.ca 2.8
Klaus Klein klaus.klein@mycollege.ca 3.6
Marc Belleville marc.beleville@mycollege.ca 2.6
```

Listing 15.2: Fine Arts students information (fa.info).

Clearly, this scenario could use a tool that combines the power of **`cat`** to concatenate text files, **`cut`** to get rid of unnecessary columns, and **`sort`** to sort the students according to their GPAs (as well as **`head`** or **`tail`**, to limit the output to only three students).

With piping, the solution to this problem is quite simple. We first use **`cat`** to join all text files with students information. However, instead of displaying the results on the terminal, or redirect it to a file, we pipe it to **`cut`** which uses a space as its delimiter and keeps only the first, second, and fourth fields. Following, we send the output of **`cut`** to **`sort`**, which should sort students using its third column,* numerically, and in descending order. Finally, we pipe the output of **`sort`** to **`head`**, which is used to display only the first three lines of the result, as seen below:

*Note that the output of the **`cut`**, which is the input of **`sort`**, has only three columns.

```
marcel@dell:~$ cat cs.info fa.info | cut -d" " -f 1,2,4 |
    sort -r -n -k3 | head -n3
Jane Doe 3.9
Aditya Kapoor 3.9
```

```
Klaus Klein 3.6
marcel@dell:~$
```

15.2.2 *Piping Syntax*

To pipe the output of one command as an input to another command, as shown in the example above, we use the piping operator **|**.

```
marcel@dell:~$ command1 options input_arguments | command2
    options | command3 options | ...
```

Note that there is no limit on the number of pipes that can be used. Also, note that we exclude the input arguments* for all pipelined commands besides the first. This happens because, for these commands, the input should come from the pipe, not from another file. In other words, the input coming from the pipe replaces the input arguments we normally use.

*Which are normally file names.

15.2.3 *Piping examples*

The application scenario we presented had a quite complex pipeline of commands. To get a better grasp of the concept of piping, see the examples below, were simpler pipelined commands are used:

```
marcel@dell:ls | less
```

In this example, we simply redirect the list of files and folders in the current directory to be displayed in **less**, as opposed to be printed in the terminal. This is a very common use of pipes, as it allows users to check the contents of large directories that would not fit the terminal screen.

```
marcel@dell:cat *.log | grep "error" | less
```

In this example, we first concatenate all files with a **.log** ending, and then we use **grep** to find the lines in which the string **error** can be found. Finally, the result is displayed in **less**.

```
marcel@dell:find /  -user john | xargs grep "password"
```

In this example, we first use **find** to get the name of all files that belong to a user called **john**. Then, we use **grep** to search for the string **password** inside all of these files. Note that we used a command called **xargs*** between the pipe and **grep**. This was done because we wanted **grep** to search inside the files whose names and paths were passed by the pipe. Without **xargs**, **grep** would perform its search on the list of filenames, as opposed to performing the search inside all files presented by this list.

*The **xargs** command breaks the list of arguments passed by the pipe into its individual elements.

15.2.4 *Piping and Redirection*

Just like we did for regular commands, you can redirect the final result of a pipeline into a text file using the **>** operator.

In all examples from the previous section, we could have written the results into a text file called **file**, as opposed to displaying it in **less** or printing it in the terminal, by using:

```
marcel@dell:ls > file
marcel@dell:cat *.log | grep "error" > file
marcel@dell:find / -user john | xargs grep "password" > file
```

15.2.5 ***tee***

In our previous section, we showed how to redirect the final result of a pipeline. However, we cannot simply redirect the output of a pipeline midway. For example, the command below will result in an error.

```
marcel@dell:cat *.log > my\_file | grep "error" | less
```

This should come to no surprise, if we think about our pipeline real world analogy. If we dump the contents of a pipe, we cannot expect it to proceed further down the pipeline.

In order to be able to save intermediate results, we need to use **tee***. This tool allow us to copy the output of a pipe into a file, but also send it further down the pipeline. For example, we can fix the previous example by using **tee** to send the output of **cat** to a file called **my_file**, but also send it further down the pipeline, using the following syntax:

*The name of this tool derives from the format of the letter **t**.

```
marcel@dell:cat *.log | tee my\_file | grep "error" | less
```

EXERCISES

1. Write down a command to concatenate the contents of three files, called **`syslog`**, **`syslog1`**, **`syslog2`**, and display it on the terminal

2. Write down a command to concatenate the contents of three files, called **`syslog`**, **`syslog1`**, **`syslog2`**, and display it on **`less`**.

3. Write down a command to concatenate the contents of three files, called **`syslog`**, **`syslog1`**, **`syslog2`**, save it in a file called **`syslogs`**, and also display it on **`less`**.

   For the questions below, assume that you have two files: **`cs.info`** and **`fa.info`** in your current working directory. The contents of these files are displayed in Listings 15.1 and 15.2, respectively, on page 100.

4. Write down a command to display in the terminal a list of all students emails, without displaying the information in other columns.

5. Write down a command to display only the students first and last names in the terminal in alphabetical order according to their last names.

6. Write down a command to get the students first and last names, as well as their GPAs, and save them in descending order based on their GPAs in a file called **`all_students.info`**.

Part IV

SCRIPTING

In the following chapters, we introduce the concept of scripting, i.e., creating files containing a sequence of instructions that perform a specific task. Scripts can be used for automating repetitive tasks. They are widely used by system administrators to take care of tasks such as creating hundreds of new users, or configuring multiple machines. With scirpts, such tasks cab be done in a matter of minutes, instead of taking hours of days of repetitive work.

16 INTRODUCTION TO SCRIPTING

In movies and plays, a script is a document containing the lines that the actors and actresses should say, as well as the actions they should perform, i.e., running, jumping, fighting, etc. Similarly, in computing, a script is a text file containing a sequence of lines of code that we issue to a system in order for it to print outputs and execute particular actions.

In this chapter, we will start by introducing a very basic script. Then, we explain how to run scripts using the terminal. Following, we discuss the concepts of variables and shell environments. Finally, we end this chapter by showing how a script can work with inputs provided by the user in real-time.

16.1 HELLO WORLD SCRIPT

Most programming languages books start by introducing a source code to print the sentence "*Hello World*" on the terminal. This book is no exception, in Listing 16.1 we show a very basic **bash** script* that, once executed, prints this sentence.

*There is a discussion about the meaning of the words ***script*** and ***source code*** in the Compile languages vs Interpreted languages box on page 106.*

```
1 #!/bin/bash
2 #Hello World example
3
4 echo "Hello World!"
```

Listing 16.1: Script for a Hello World example.

Even though this is a very simple script, it already contains a number of basic features that will be present in almost all **bash** scripts we are going to be writing in the course of this book. These features are discussed in what follows.

16.1.1 ***shebang*** *line*

The first line of any terminal script, no matter which language they were written with, must tell the shell which interpreter to use in order to read the script. This line is usually called a **shebang** line.

A **shebang** line starts with a pound sign (hashtag), **#**, followed by an exclamation sign, **!**, followed by the location of the tool used to interpret the script. Because we are writing **bash** scripts in this book, we need to provide the location of **bash**. In most distributions, **bash**

is at the **/bin** folder*. Hence, in the first line of our script we have a shebang line: **#!/bin/bash**.

*You can confirm **bash**'s location in different distributions by issuing a **which bash** command.

16.1.2 *Comments*

Many times we want to write some information into a script that we don't want the interpreter to process. For example, you may want to write the name of the author of the script, or provide a brief description of what it does. In both cases, the information you are providing is just for yourself, or other users, to read. The interpreter should simply ignore these lines.

For scenarios such as the ones described above, we can make use of comments. A comment is a block of text that follows a pound sign (hashtag) **#**. With the sole exception of the **shebang** line we discussed in the previous section, the interpreter ignores everything written between a **#** sign and the end of the line. In Listing 16.1, we have a comment on the second line. Note that in our example, we wrote a comment on a blank line. However, nothing prevents us from adding a comment to the right of a non-empty line.

In large scripts, comments are often used to describe specific parts of it. They can be very helpful for others, and even your future self, to understand the script. It is a good practice, in such cases, to explain what is it that the script does, not how it does it.

16.1.3 ***echo***

echo prints on the terminal the arguments passed to it. In Listing 16.1, we use it in line **4** to print the string "Hello World!".

Note that whenever we want to print a string, i.e., a sequence of words, we need to write it inside quotes. There are two types of quotes that can be used:

SINGLE QUOTES Also know as hard quotes, allows **echo** to print what is inside the quotes in verbatim. I.e., exactly as it is written.

DOUBLE QUOTES Allow **echo** to replace the name of variables preceded by a dollar sign **$** with the value stored in it. We will cover variables later in this chapter.

Compiled languages vs Interpreted languages

There is a plethora of computer languages available today, each one catering for a different kind of public, or different kind of application scenarios. There are multiple ways to categorize these languages. However, one of the most fundamental divides between computer languages is that some are compiled languages, and others are interpreted languages.

A compiled languaged requires a tool called compiler to translate a **source code**, which is written by a software developer, into an executable file. Then, this executable file can be easilly called by the user to perform whatever tasks it was designed to do. If the developer needs to change the application, he or she will need to alter the original source code and recompile it. Note that, once the executable file has been created, the system does not need the source code anymore. **`C`**, **`C++`**, and **`Java`**, are some examples of compiled languages.
An interpreted language, on the other hand, requires a tool called interpreter. This tool reads the script line by line, processing all commands it finds as it goes through the script. **`Bash`**, **`Python`**, **`PHP`**, and **`JavaScript`** are some examples of interpreted languages.
Note that in the paragraphs above, I am calling the text files that serve as input to compilers **source codes**, and files that serve as input to interpreters **scripts**. This distiction is applied by this book's author, but is not universal. Some developers might talk about **source codes** for Python applications. That said, you will rarely see the expression **bash source code** being used instead of **`bash script`**.

16.2 USING THE TERMINAL TO RUN SCRIPTS

To run a **script**, you simple have to call it by its name using the terminal so that the shell can execute it. In fact, most of the tools this book has covered so far, such as **`ls`**, **`cat`**, **`awk`**, etc., are nothing but binary scripts that are located at the **`/bin`** directory.

Clearly, in order to execute scripts, the shell needs first to know where to find them. That is where a variable called **`PATH`** plays an important role, as we discuss in what follows:

16.2.1 ***`PATH`*** *variable*

A variable called **`PATH`** is used to store the paths of all directories that **`bash`** searches while it looks for scripts to execute. By default, in an Ubuntu distribution, it contains:

```
marcel@dell:~$ echo $PATH
/home/marcel/bin:/home/marcel/.local/bin:/usr/local/sbin:/
   usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/
   usr/local/games:/snap/bin
```

Note that in your own system, your username should repalce mine (**`marcel`**) in the PATH fodlers that are under the user's home folder.

When a user enters any command, the shell searches through all these directories, one by one, while trying to find a script which name matches the provided command. Once a match is found, the search is interrupted and the script is executed*.

*This means that if two scripts with the same name exist in different directories in the **PATH**, the one that comes first in the **PATH** is the one that gets executed.

It is important to note that the current folder is not usually in the path. However, we can always execute a script by providing the shell with its path. This means that, in case we want the shell to execute a script with name **script_name.sh** within the current folder*, we use the following syntax:

*Which is always indicated by a period sign (.).

```
marcel@dell:~$./script_name.sh
```

Note that we could have used a similar notation to run scripts in any other directories by providing the shell with its relative or absolute paths.

16.2.2 *Granting permission to execute a Script*

Suppose that you save the script provided in Listing 16.1 in a text file called **script.sh*** in your working directory. You should expect to be able to run it by simply writing down the command: **./script.sh**. However, when you do so, you get the error message shown below:

*Ending **bash** script names with **.sh** is not necessary, but it is a good convention to follow.

```
marcel@dell:~$./script_name.sh
bash: ./script.sh: Permission denied
```

What happens is that, by default, files created in most linux systems are created with only reading and writing permissions, not with executing permissions*. To change this, you need to grant yourself permission to run your script. This can be accomplished by issuing the command: **chmod +x script_name**, as shown below:

*A comprehensive discussion on file permissions can be seen in Chapter XXX.

```
marcel@dell:~$./script_name.sh
bash: ./script.sh: Permission denied
marcel@dell:~$chmod +x script.sh
marcel@dell:~$./script.sh
hello world
```

Note that, once we granted the script permission to be executed, we were able to call it without any problems. As a result, it printed **Hello World** on the terminal as expected, and exited.

It is important to note that, apart from the fact that it requires executing permission, a script is no different than any other text file. In fact, just like any other text fiIe, it can be opened for reading using **more**, **less**, **cat**, or opened for editing in **nano**, **vim** or even with GUI tools such as **Gedit**.

16.3 VARIABLES

A variable is a name we assign to a place in memory that we can use to store a particular value. For example, if we enter **var=6**, in our terminal, the shell saves the value **6** in memory on a variable called **var**. To retrieve the contents of a variable, we need only to call its name with a preceding dollar sign, **$**. See the example below:

```
marcel@dell:~$var=6
marcel@dell:~$echo "The value stored in var is $var"
The value stored in var is 6
marcel@dell:~$var=8
marcel@dell:~$echo "The value stored in var is $var"
The value stored in var is 8
```

As shown above, a variable can have its value changed after it was first defined. Hence the name variable. Note that we used double quotes in the example above, instead of single quotes. This was done to ensure that the shell would translate **$var** into the value stored in **var**, before printing the message. In case we had used single quotes, the result would have been quite different, as shown below:

```
marcel@dell:~$var=6
marcel@dell:~$echo 'The value stored in var is $var'
The value stored in var is $var
```

Variables are extensively used when working with scripts. They can be used to store numbers, words or sentences*, and even filenames.

*Which in computing jargon are called strings.

16.3.1 *Mathematical Operations*

To perform mathematical operations with variables, one needs to use the syntax shown below:

```
$((OPERATION))
```

Where valid operations are addition (+), subtraction (-), multiplication (*), and division (/). Note that, for division, the result will only have the integer part. See the examples below:

```
marcel@dell:~$var1=6
marcel@dell:~$var1=4
marcel@dell:~$echo "$var1 + $var2 = $(($var1+$var2))"
6 + 4 = 10
marcel@dell:~$echo "$var1 - $var2 = $(($var1-$var2))"
6 - 4 = 2
marcel@dell:~$echo "$var1 * $var2 = $(($var1*$var2))"
6 * 4 = 24
marcel@dell:~$echo "$var1 / $var2 = $(($var1/$var2))"
6 / 4 = 1
```

16.4 ENVIRONMENTS

When a user logs into the system, the shell normally sets up an environment for this specific user with number of variables that control the behaviour of the terminal. The **PATH** variable we discussed before is an example of a variable that is normally set with default values. Other examples are **LOGUSER**, and **HOSTNAME**, which store the user name of the user that is currently logged, and the host name, respectively. You can see all default variables by issuing a **env** command.

It is important to note that each user that logs into the system is assigned his or her own exclusive environment. Also, it is important to know that a variable is only defined within its own environment. This guarantees that multiple users can access the system at the same time without the risk of one user altering another user's variables.

16.4.1 *source*

When a user calls a script, the shell creates a separate environment to store all variables created by the script. I.e., the shell executes the script within its own environment*. Executing scripts in their own environments results in two important features:

*You can think about environments as sandboxes.

- A script does not have access to any variables set prior to the start of the script by the user, or by the system.
- Any variables set by the script will cease to exist as soon as the script finishes execution.

By creating an exclusive evironment for each script that is executed, the shell guarantees that scripts will not accidently overwrite any variables that exist outside them. For example, if a script defines a variable called **PATH**, this variable will have no relationship to the default **PATH** variable defined by the shell. Furthermore, once the script has finished execution, the value stored in the **PATH** will be the same value it had before the script was executed. The example below demonstrates this behaviour for the **USER** variable. It calls a script called **change_user.sh**, shown in Listing 21.5.

```
1 #!/bin/bash
2 #script that changes the LOGNAME variable
3 LOGNAME=Jar
4 echo "The current username is: $LOGNAME"
```

Listing 16.2: Script for changing the **USER** variable.

```
marcel@dell:~$echo "The current username is: $LOGNAME"
```

```
The current username is: marcel
marcel@dell:~$./script.sh
The current username is: Jar
marcel@dell:~$echo "The current username is: $LOGNAME"
The current username is: marcel
```

It is possible to run scripts without creating an exclusive environment for them. I.e., running scripts in the same environment as the shell itself. To do so, you simply need to use the **source** command in front of the script call, using the syntax: **source ./script_name.sh***. This is called sourcing the script. An obvious scenario for which sourcing a script is useful is the one in which a user wants to alter or use shell variables from within the script. See the example below:

*Assuming the script is in your current working directory.

```
marcel@dell:~$echo "The current username is: $LOGNAME"
The current username is: marcel
marcel@dell:~$source ./script.sh
The current username is: Jar
marcel@dell:~$echo "The current username is: $LOGNAME"
The current username is: Jar
```

16.5 GATHERING USER'S INPUT

There are two ways in which we can gather inputs from users that are calling a script. Using the **read** keyword, or passing user input as arguments. Both methods are discussed in what follows.

16.5.1 *read*

The **read** keyword, as the name suggests, reads an input from the user and saves it into a variable. Its syntax is: **read variable_name**. Note that the user can enter numbers, words, or even full sentences. See the example below:

```
marcel@dell:~$./read_input.sh
Enter your name:
Marcel
Hello Marcel
marcel@dell:~$./read_input.sh
Enter your name:
Marcel Jar
Hello Marcel Jar
```

This example uses the script **read_input.sh** in Listing 16.3, shown below:

```
#!/bin/bash
echo "Enter your name:"
read name
echo "Hello $name"
```

Listing 16.3: Script that uses the **read** keyword.

16.5.2 *Passing arguments to a script*

You can also gather inputs from users by having them passing arguments to the script during its call. This is done by writing all arguments, separated by spaces, when the script is called. The value of each argument can be retrieved from inside the script using the syntax **$N**, where **N** stands for the argument number. I.e., if it is the first argument, the second, etc. You can also get the total number of arguments passed to the script using the syntax **$#**. In the example below, we pass three arguments to a script call:

```
marcel@dell:~$./read_arguments.sh Marcel Jar 25
Hello Marcel Jar you are 25 years old
I was given 3 arguments.
```

This example uses the script **read_arguments.sh** in Listing 16.4.

```
#!/bin/bash
echo "Hello $1 $2 you are $3 years old"
echo "I was given $# arguments."
```

Listing 16.4: Script that takes arguments from the user's script call.

EXERCISES

1. Explain, with your own words, what is the purpose of a **shebang** line. Also, write the proper shebang lines for **Bash**, **Python** and **Pearl** scripts. *hint: use the **which** tool to find the location of the proper script interpreter.*
2. Write a **bash** script that simply prints on your screen a message: **This is my first script!**. Make sure to include a **shebang** line, and one comment line containing your name as the script's author. Also, remember that you need to grant the script permission to be executed with: **chmod +x script_name**.
3. Using your own words, explain what is a variable?
4. How can you create a variable called **name** and assign to it your own name using **bash**.

5. How can you create a variable called **`age`** and assign to it your own age?

6. Explain, with your own words, the advantage of having the shell creating a different environment for each user that logs into the system. Also, explain the advantage of creating, by default, separate environments for scripts, when the user calls them.

7. Explain, with your own words, what is the difference between calling a script with or without the **`source`** command.

8. Write a script that asks the user to enter its name, its job, and its age. Following, your script should print: **`Hi USERNAME, you are a USERJOB, and you are USERAGE years old.`**, where the values of **`USERNAME`**, **`USERJOB`**, and **`USERAGE`** are taken from the user's inputs. *Hint: Use the* ***`read`*** *keyword*

9. Rewrite the script from the previous question. In this new version, instead of using **`read`** to gather the user's input, the script takes three arguments during its call: **`USERNAME`**, **`USERJOB`**, and **`USERAGE`**, respectively.

17

LOGICAL EXPRESSIONS

In real life, we often need to make decisions based on what is happening around us. For example, during a weekend, you might decide to go to the beach **if** it is sunny and warm, or **else** in case it is cold and rainy, you might go to the movies. In computing, these **if/else** blocks of sentences, which allow us to make decision based on **logical expressions**, arise quite frequently. In this chapter, we will study how to make decisions in **bash** scripts based on results of previous commands, based on information stored in variables, or even based on the input from users.

17.1 BASIC `if/else` SYNTAX

The script presented in Listing 17.1 demonstrates the proper syntax for working with a simple **if/else** block.

```
1  #!/bin/bash
2  #Simple if/else statement
3
4  echo "How is the weather today?"
5  read weather
6
7  if [ $weather == "sunny" ] ; then
8      echo "I am going to the beach"
9  else
10     echo "I am going to the movies"
11 fi
```

Listing 17.1: Script containing a simple **if/else** block.

In this script, we start by asking the user about the weather. Then, we save the user's input into a variable called **weather**. Following, we write an **if/else** block to display different sentences depending on the value stored in the variable **weather**. This is done using the following syntax:

```
if [ LOGICAL EXPRESSION ] ; then
    #Lines executed if the LOGICAL EXPRESSION is true
else
    #Lines executed if the LOGICAL EXPRESSION is false
fi
```

A logical expression is always a statement that evaluates to either **true** or **false**. In our example, we check if the string stored in the **weather** variable is identical to the string **sunny** using the **==** operator*. This will result in the script executing line 8 of Listing 17.1, and skipping line 10 of the same script. Otherwise*, the script will skip line 8 and execute line 10.

*This will be true if the user answers **sunny**.

*I.e., in case the user answers anything but **sunny**.

Note that in **bash** scripting, proper syntax needs to be followed very closely. Even omiting spaces can result in errors. For example, imagine that we had written "**if [$weather == "sunny"] ; then**" on line 7 of Listing 17.1. I.e., we omitted the spaces between the brackets and the logical expression. In this case, running the script would have resulted in an error. This is due to the fact that there must be spaces between the opening and closing brackets of the **if** statement and the logical expression. We would also have an error if we omitted the **then** keyword, or the **fi** keyword that ends the **if/else** block.

17.2 LOGICAL EXPRESSIONS

There are three basic types of logical expressions in **bash**. There are **string expressions,** where we can compare two strings to see if they are the same, different, or to see if one comes after the other*. There are also **integer expressions**, where we can do the same with two numbers. Finally, there are **file expressions**, where we can check if a file exists, if it is a directory or a regular file, or if a file is older than another, among other expressions.

*In alphabetical order.

17.2.1 *String Expressions*

The most common string expressions available in **bash** are listed in Table 17.1. You can see an example of a script that makes use of these expressions in Listing 17.2 .

EXPRESSION	RETURNS **true** IF:
string1 \== string2	The two strings are identical.
string1 \!= string2	The two strings are not identical.
string1 \< string2	**string1** comes before **string2**.
string1 \> string2	**string1** comes after **string2**.
-n string	The length of **string** is greater than zero.
-z string	The length of **string** is equal to zero.
string	The **string** exists.

Table 17.1: String Expressions.

```
#!/bin/bash
#String expression example

echo "What is your name?"
read name

if [ $name \< "Thomas" ] ; then
    echo "You come before Thomas in the attendance list."
fi
if [ $name \< "James" ] ; then
    echo "You come before James in the attendance list, or
    your name is also James."
fi
```

Listing 17.2: Script containing two string expressions.

In this example, we ask the user for his name, and then we store it in a variable called **name**. Then, we compare this name with **Thomas**, to see who would come first in the attendance list. Following, we do the same process and compare the user's name to **James**.

This example demonstrates a few interesting aspects of **if/else** blocks, as well as string expressions. First, it shows that we can ommit the **else** part of an **if/else** statement. Second, it shows that we can have multiple **if/else** blocks in the same script. Finally, it is important to notice that string comparisons are made based on the integer values of each character in the ASCII* table (http://www.asciitable.com/).

*which shows, for example, that lowercase letters are considered greater in value than uppercase letters.

17.2.2 *Integer Expressions*

The most important integer expressions available in **bash** are listed in Table 17.2. You can see an example of a script that makes use of some of these expressions in Listing 17.3 .

EXPRESSION	RETURNS **true** IF:
number1 -lt number2	**number1** is lower than **number2**.
number1 -le number2	**number1** is lower or equal to **number2**.
number1 -eq number2	**number1** is equal to **number2**.
number1 -ne number2	**number1** is not equal to **number2**
number1 -ge number2	**number1** is greater or equal to **number2** .
number1 -gt number2	**number1** is greater than **number2**

Table 17.2: Integer Expressions.

```
#!/bin/bash
#Integer expression example

echo "Enter a number?"
read number

if [ $number -le 10 ] ; then
    echo "Your number is lower or equal to ten."
else
    echo "Your number is greater then ten."
fi
```

Listing 17.3: Script containing an integer expression.

In this example, we start by asking for a user input and we store it in the variable **number**. Then, we check if this number is lower or equal to 10 or not, and we display a different message for each scenario.

17.2.3 *File Expressions*

The most important file expressions available in **bash** are listed in Table 17.3. You can see an example of a script that makes use of some of these expressions in Listing 17.4 .

EXPRESSION	RETURNS **true** IF:
-e file1	**file** exists.
-s file1	**file** exists and has a size greater than 0.
-f file1	**file** exists and it is a regular file.
-d file1	**file** exists and it is a directory.
-w file1	**file** exists and the current user has permission to write (edit).
-x file1	**file** exists and the current user has permission to execute it as a script.
file1 -nt file2	**file1** is newer than **file2**.
file1 -ot file2	**file1** is older than **file2**.

Table 17.3: File Expressions.

In this example, we start by asking the user for an input containing the name of a file in the current working directory. Following, we store this input in the variable **file_name**. Finally, we check if the file exists and if it is non-empty.

```
#!/bin/bash
#Integer expression example

echo "Enter a file or directory name?"
read file_name

if [ -e $file_name ] ; then
    echo "The file $file_name exists"
fi
if [ -s $file_name ] ; then
    echo "The file $file_name exists and it is not empty"
fi
```

Listing 17.4: Script containing two file expressions.

17.3 COMBINING EXPRESSIONS USING && AND | |

There are situations in which we might want to take one course of action in case more than one condition is met. For example, you might decide to go to the beach if, and only if it is a weekend and it is sunny. In this scenario, you would not go to the beach if it is a weekday, and you would also not go in case the weather isn't nice. For such scenarios, **bash** makes use of the **&&** operator*. An example of a script making use of a **&&** operator is shown in Listing 17.5 .

*which corresponds to a logical **and**.

```
#!/bin/bash
#and operator script example

echo "Is today a weekend day?"
read weekend
echo "Is the weather sunny?"
read weather

if [ $weekend == "yes" ] && [ $weather == "yes" ] ; then
    echo "I am going to the beach"
else
    echo "I am going work or I will just stay home."
fi
```

Listing 17.5: Script using a **&&** operator.

Note that in the example in Listing 17.5, the script will print "*I am going to the beach*" if and only if the user has answered **yes** to both *Is today a weekend day?* **and** *Is the weather sunny?* If the user replies **no** to

any one of these questions, or if he replies **no** to both questions, the script will print "*I am going work or I will just stay home.*".

In other scenarios, you might want to follow one particular course of action if at least one out of a number conditions is met. For example, you can become an American citizen if one of your parents is American, or if you were born in American soil. I.e., as long as at least one of these two conditions is met, you are eligible to apply to an American citizenship. For such scenarios, **bash** makes use of the `||` operator*. An example of a script making use of a `||` operator is shown in Listing 17.6 .

*which corresponds to a logical **or**.

```
#!/bin/bash
#or operator script example

echo "Is one of your parents American?"
read parents
echo "Where you born in American soil?"
read soil

if [ $parents == "yes" ] || [ $soil == "yes" ] ; then
    echo "You can apply for an American citizenship."
else
    echo "You can apply for an American citizenship. "
fi
```

Listing 17.6: Script using a `||` operator.

Note that in the example in Listing 17.6, the script will print "*You are eligible to apply for an American citizenship.*" if the user has answered **yes** to *Is one of your parents American?* or if the user has answered **yes** to *Where you born in American soil?*, or if he has answered **yes** to both questions. The only case in which the script will print "*You are not eligible to apply for an American citizenship.*" is if the user has answered something different than **yes** for both questions.

17.4 BASIC **if/elif/else** SYNTAX

So far, we have only dealt with situations in which we would could only follow two courses of action. One if the logic expression being evaluated was **true**, or else we would follow a different course of action. However, there are situations in which we might have multiple courses of action. For example, in North America, we have the following age restricitons for movies:

- People over or at the age of 17 can watch any movies.

- People with ages between 13 and 17 can watch **G**, **PG**, and **PG-13** movies, but they need a guardian to watch **R** rated movies, and they cannot watch **NC-17** rated movies.
- Finally, anyone under the age of 13 can only watch **G** and **PG** movies by themselves, but need a guardian to watch **PG-13** movies or **R** rated movies.

For scenarios such as these, **bash** provides **if/elif/else*** blocks. In an **if/elif/else** block, the shell works as follows:

**elif is a contraction of else if.*

1. First, the shell checks the logical expression in the line with the **if** keyword. In case this expression is true, the shell executes the lines associated with this expression being true, and skips all the rest of the **if/elif/else** block.
2. If the logical expression associated with the **if** keyword is false, the shell will move to check the logical expression in the line of the first **elif** keyword. In case this expression evaluates to true, the shell executes the lines associated with it and skips the rest of the **if/elif/else** block. Note that multiple **elif** keywords can be used, each one associated with a different logical expression.
3. In case all logical expressions associated with the **if** keyword, as well as all **elif** keywords are false, the shell executes the lines of code associated with the **else** keyword.

In the script presented in Listing 17.7, we provide the user with information about what types of movies he or she can watch, based on the age they provide as an input*.

**Note that, just like the line with the if keyword, the lines with elif keywords require a then keyword.*

```
1  #!/bin/bash
2  #if/elif/else script example
3
4  echo "How old are you"
5  read age
6
7  if [ $age -ge 17 ] ; then
8      echo "You can watch any movie you want"
9  elif [ $ age -ge 13 ] ; then
10     echo "You need a guardian for R-rated movies."
11 else
12     echo "You need a guardian for PG-13 and R-rated movies."
13 fi
```

Listing 17.7: Script using an **if/elif/else** syntax.

Note that, for the script in Listing 17.7, we did not have to check if the user was below the age of **17** in the **elif** expression on line 9. We simply checked if he or she was over the age of **13**. As it was explained before, in case the logical expression associated with the **if** keyword*, the lines of code associated with this expression would be executed and the script would skip the rest of the **if/elif/else** block. In other words, this script only checks if the user is above **13** if it already knows that the user is below **17**.

*I.e., if the user were **17** years old or older.

17.5 NESTED LOGICAL EXPRESSIONS

In some scenarios, making one decision affects other decisions you can possibly make in the future. For example, choosing to eat an entire chocolat cake in the morning will prevent you from running a marathon or an "iron man" in the afternoon. As another example, choosing to learn another language will allow you to apply for a number of positions in which knowledge of this language is mandatory.

In the script presented in Listing 17.8, a user is asked a series of questions about possible positions to apply to at a hotel in Montreal. This script starts by asking the user if he or she is fluent in French because some positions, such as manager and receptionist, are only available for fluent French speakers. Then, depending on the answer for this question, the script asks more questions offering the user different kinds of jobs.

Note that, on Listing 17.8, all the script lines from 8 to 27 will only be executed if the user answers **yes** to the first question. Then, depending on the user's position choice, the script might execute line 12, lines 14 and 15, lines 17 to 24, or line 26 (in case he or she has entered an invalid option). If the user enters anything but **yes** for the first question, the line 29 will be executed.

It is important to notice that because some choices depend on previous choices, we had to write **if/elif/else** blocks inside parts of another **if/elif/else** blocks. For example, lines 11 to 27 correspond to an **if/elif/else** block that is only executed if the user has entered **yes** for the first question. Moreover, lines 19 to 24 represent an **if/else** block that is only executed if the user has entered **yes** to the first question and **electrician** for the second question.

```
#!/bin/bash
#nested if/else block

echo "Are you fluent in French?"
read language_question

if [ $language_question == "yes" ] ; then
    echo "Which job do you want to apply to?"
    echo "options: receptionist, manager, electrician"
    read job_question
    if [ $job_question == "receptionist" ] ; then
        echo "We have no openings for at the moment."
    elif [ $job_question == "manager" ] ; then
        echo "We have 3 openings for at the moment."
        echo "Please submit your resume"
    elif [ $job_question == "electrician" ] ; then
        echo "Do you have an electrician diploma or degree?"
        read electrician_question
        if [ $electrician_question == "yes" ] ; then
            echo "We have 2 openings at the moment."
            echo "Please submit your resume"
        else
            echo "Sorry, certification is required."
        fi
    else
        echo "Please, provide a valid answer"
    fi
else
    echo "Sorry, fluency in French is required"
fi
```

Listing 17.8: Script using nested **`if/elif/else`** blocks.

The importance of indentation

The **bash** interpreter ignores whitespace. I.e., you can add as many spaces or even tabs to the beginning or end of any line of code without affecting how that line will be interpreted. This fact is often used to help us structure our scripts in a way that makes them easier for us, humans, to read.
As an example, take a close look at the script in Listing 17.8. Note that we added four spaces at the beginning of lines 8 to 27. This was done to make it easier for the reader to see that all these lines would only be executed in case the logical expression associated with the **if** keyword in line 7 was true. The same way, we added four spaces at the beginning of line 29 to make it clear that this line is associated to the **else** expression. Finally, note that whenever we have nested **if/elif/else** blocks, we just add additional whitespace characters to make it visually clear that those lines are only called when two or more decisions were already made. For example, we have 8 spaces at the beginning of lines 12, 14 to 15, and 17 to 24, to indicate the fact that they belong to a nested **if/elif/else** block.

EXERCISES

1. Create a script that asks what day of the week it is. Then, in case the user enters any day from Monday to Friday, the script outputs: *"Today is a weekday. Go to work"*. In case the user enters Saturday or Sunday, the script should output *"Today is a weeked day. Time to chill"*.

2. Create a script that asks the user for a filename. Then, in case there exists a file with the provided filename in the current working directory, the script should delete the file, and warn the user that the file has been deleted. Otherwise, your script should warn the user that no file has neen found.

3. Create a script that asks the user what is his favorite OS. If the user answers **Linux**, the script should output: *"That's awesome"*, if the user answers **OSX**, the script should output: *"Not bad, but you can do better"*, if the user answers **Windows**, the script should output *"Some people are just beyond help"*, and finally if the user enters anything else, the script should output: *"Are you sure this is an OS?"*.

4. Create a script for a robot waiter. It should start by asking if the costumer wants some eggs. In case the answer is **yes**, then your script should then ask how many eggs, how the costumer wants his eggs, and save these answers in variables called **qtd** and

egg. Then, your script should output the costumer order. For example, if the user entered **yes**, **2**, and **scrambled**, the script should output: *"I will bring 2 scrambled eggs soon"*. In case the costumer says **no** to the eggs question, the script should then ask if he wants some pancakes instead. In case the costumer says **yes** to this question, your script should then ask how many pancakes and store this value in a **pck** variable. Then, your script should display: *"I will bring 3 pancakes soon"*. Finally, if the user also says no to the pancakes question, the script should say: *"I have no other food for you. Go away!"*

18

LOOPS

One of the areas in which computers really excel is in their ability to perform repetitive tasks in a fast and precise manner. Users can request a particular command to be executed hundreds or thousands of times, and the system will hapilly oblige, executing the command exactly the same way each time. In this chapter, we will learn how to execute a set of instructions multiple times using **`for`**, **`while`**, and **`until`** loops.

18.1 `for` LOOPS

A **`for`** loop defines a variable with the first value in a given list. Then, it runs all lines of code between a **`do`** and **`done`** keywords. Following, the loop will redefine the variable with the next value in the given list, and execute all lines of code between **`do`** and **`done`** again. This procedure continues until the variable has assumed all possible values in the provided list. Listing 18.1 provides an example of a **`for`** loop.

```
1 #!/bin/bash
2 #Simple for loop
3
4 for colour in "white" "blue" "green" ; do
5     echo "my variable has colour: $colour"
6 done
7 echo "Your loop has ended"
```

Listing 18.1: Script containing a simple **`for`** loop.

In the example provided in Listing 18.1, our loop defines a variable called **`colour`** and starts by storing a string **`white`** in it. Then, it prints: "*my variable has colour: white*" on the terminal. Following, the loop replaces the value stored in the variable **`colour`**, from **`white`** to **`blue`**, and prints "*my variable has colour: blue*" on the terminal. Finally, the loop replaces the value śtored in the variable **`colour`**, from **`blue`** to **`green`**, and prints "*my variable has colour: green*" on the terminal. Because **`green`** is the last value in the provided list, the loop finishes execution, and the script will proceed with the first line after the **`done`** keyword. In this particualr example, it will simply print "*Your loop has ended*".

18.1.1 *Using wildcards*

In our previous example, we specified a complete list for our **for** loop to go through. However, it is important to keep in mind that **bash** scripts can use many of the tecniques we have used in the past while working with shell commands. One such technique is making use of wildcards. In Listing 18.2, we show a script that loops through all the **.pdf** files in the current working directory and asks the user if he or she wants to delete it.

```
#!/bin/bash
#For loop with wildcard
for file in *.pdf ; do
   echo "Do you want to delete file: $file"
   read answer
   if [ $answer == "yes" ] ; then
       echo "deleting file: $file"
       rm $file
   fi
done
echo "End of script"
```

Listing 18.2: Script using a **for** loop with wildcards.

Note that in this example the loop will go through all files that end on **.pdf**. In case there are no files with this end, the script will simply assign a value ***.pdf** to the variable **file**.

18.1.2 *Using brace expansion*

Brace expansion is a method in **bash** to create a sequence of numbers or characters. For example, **touch file{1..3}** creates three files: **file1**, **file2**, and **file3***.

Brace expansion is often used with **for** loops. For example, Listing 18.3 can be used to print the same message, "*I will not play games in class*" a hundred times.

It can also be used with letters, for example* **touch file{a..z} *creates 26 files:* **filea**, **fileb**, *etc.*

```
#!/bin/bash
#For loop with brace expansion
for number in {1..100} ; do
   echo "$number: I will not play games in class"
done
echo "End of script"
```

Listing 18.3: Script using a **for** loop with brace expansion.

It is important to note that brace expansion does not expand **bash** variables. This occurs because the brace expansion is the first step of the shell expansion. Hence, a script like the one shown in Listing 18.4 does not work properly.

```
#!/bin/bash
#For loop with wrong brace expansion

var=100
for number in {1..$var} ; do
   echo "$number: I will not play games in class"
done
echo "End of script"
```

Listing 18.4: Script using a **for** loop with **wrong** brace expansion.

The problem with the script in Listing 18.4 is that the shell will first try to apply the expansion: **1** to the string **$var**, before replacing the string **$var** with the value contained in the variable **var**, which is **100**. As a result, the variable number will assume a value: **{1..$var}**, as opposed to a sequence of values from **1** to **100**.

18.1.3 *Using c-style **for** loops*

Given that brace expansion cannot work with variables, it cannot be easily used to create loops with user-define ranges. For example, the script presented in Listing 18.5 would not work properly because brace expansion does not work with variables.

```
#!/bin/bash
#For loop with usr input wrong brace expansion

echo "Write down a starting point"
read start
echo "Write down an end point"
read end

for number in {$start..$end} ; do
   echo "$number"
done
echo "End of script"
```

Listing 18.5: Script using a **for** loop with user input and **wrong** brace expansion.

In order to get a **for** loop to work properly with variables, we need to use a differeent type of syntax for **for**, called **c-style***. A c-style for loop uses the following syntax:

*Due to the fact that they are similar to the syntax used in the **C** language.

```
for ((ASSIGNMENT ; LOGICAL EXPRESSION ; INCREMENT)) ; do
    #lines to run for each iteration
done
```

Where each element of the c-style syntax is ecplained in what follows.

ASSIGNMENT Refers to the initial value assigned to the variable that will be used in our iterations. For example, we can have **var=1** or **var=$start***.

*Where **start** is a previously defined variable.

LOGICAL EXPRESSION Refers to a logical expression that will interrupt the loop once met. For example, we can have **var==10** or **var<=$end***.

*Where **end** is a previously defined variable.

INCREMENT Refers to how the variable changes with each iteration. For example, you can have **var=var+1***, **var=var-1***, or **var=var+3**, which would increment the value stored in **var** by **3** at each iteration.

*Which can be shortened to **var++**.

*Which can be shortened to **var-**.

As, an example, on Listing 18.6, we fix the script with errors introduced in Listing18.5 by replacing brace expansion with a **c-style for** loop.

```
1  #!/bin/bash
2  #For loop with usr input wrong brace expansion
3
4  echo "Write down a starting point"
5  read start
6  echo "Write down an end point"
7  read end
8
9  for ((number=$start; number<=end; number++)) ; do
10    echo "$number"
11 done
12 echo "End of script"
```

Listing 18.6: Script using a **c-style for** loop with user input.

18.2 **while** LOOPS

Another way of creating a loop in **bash** is to run iterations while a logical expression is **true**, using a **while** loop. The syntax for a **while** loop is shown in what follows:

```
while [ LOGICAL EXPRESSION ]; do
    #lines to run while LOGICAL EXPRESSION is true
done
```

Where the **LOGICAL EXPRESSION** could be any of the types of logical expressions covered in the previous chapter.

For example, on Listing 18.7, we present a loop that keeps asking for a magic word until the user enters the word **abracadabra**.

```
1 #!/bin/bash
2 #Simple while loop
3
4 word=""
5 while [ "$word" \!= "abracadabra" ] ; do
6    echo "Enter the magic word"
7    read word
8 done
9 echo "You have entered the magic word!"
```

Listing 18.7: Script using a **while** loop.

Note that in Listing 18.7, we started by defining the variable **word** as an empty string before starting to compare it to tge magic word in the loop. Also, note that, just like with **if/else** blocks, there must be spaces between the logical expression and the square brackets.

18.2.1 *Reading files line by line*

While loops can also be used to read files, line by line, as shown in Listing 18.8.

```
1 #!/bin/bash
2 #reading a file, line by line, using a while loop
3
4 number=1
5 while read line ; do
6    echo "The contents of line $number is: $line"
7    number=$(($number+1))
8 done < file_name
```

Listing 18.8: Reading file contents using a **while** loop.

Note that, on Listing 18.8, we redirect the **stdin** to send the contents of a file called **file_name** to the **while** loop. The loop will then read the first line of this file, save it in a variable called **line**, and then run its first iteration. Following, the loop will save the contents of the

second line of **file_name** into **line** and run its second iteration. This process continues until the loop reaches the final line of the file. At each iteration, we add one to the value stored in the variable **number**.

18.2.2 ***until*** *Loops*

An **until** loop is similar to a **while** loop except for one major difference. It run iterations while a logical expression is **false**, as opposed to **true**. The syntax for an **until** loop is shown in what follows:

```
until [ LOGICAL EXPRESSION ]; do
    #lines to run while LOGICAL EXPRESSION is false
done
```

Where the **LOGICAL EXPRESSION** could be any of the types of logical expressions covered in the previous chapter. On Listing 18.9, we present a loop that keeps echoing the user's input until the user enters the string **-q**.

```
1 #!/bin/bash
2 #Simple until loop
3
4 word=""
5 until [ $word == "-q" ] ; do
6    echo "Enter any word to have it echoed, or -q to exit"
7    read word
8    echo "You entered: $word"
9 done
```

Listing 18.9: Script using an **until** loop.

18.3 **break**

There might be situations in which we want to get out of a **for** loop even though there are still more items to run through. In the same manner, there might be situations in which we want to exit a **while** loop even though its logical expression is true*

*Or to exit an **until** loop, even though its logical expression is still false..

For these scenarios, the keyword **break** can be used. Whenever the shell interpreter finds a **break**, it immediatly finishes the loop, no matter if it is a **for**, a **while**, r an **until** loop.

One example of a scenario in which using a **break** statement could be used is in a refactoring of Listing 18.9. In its present format, this script would print "*You entered: -q*," before quitting. This script could be refactored*, using a **break**, so that it exits immediatly after the user enters **-q** without printing any messages. This is exactly what was done in Listing 18.10.

*Which is the technical jargon for improved.

```
#!/bin/bash
#Loop with a break

while [ true ] ; do
   echo "Enter any word to have it echoed, or -q to exit"
   read word
   if [ $word == "-q" ] ; then
       break
   fi
   echo "You entered: $word"
done
```

Listing 18.10: Script using a **break**.

Note that in Listing 18.10, the logical expression we used was simply a keyword **true***. This means that, no matter what happens inside each iteration, this expression will be true and the loop will proceed with the next iteration. The only way to exit a loop such as this is by reaching a **break**. Also, note that once the logical expression for the **if** block is true, the interpreter will break out of the loop and, therefore, it will not read the code in line 10.

We could also have used an **until loop with a **false** keyword.*

18.4 **continue**

A **continue** keyword is similar to the **break** keyword in which they both interrupt the normal execution of a loop. However, whereas a **break** completely exits the loop, a **continue** just exits the **current iteration** and moves ahead to the next one.

On Listing 18.11, we present a simple refactoring of the script presented in Listing 18.1. In this new version, the script will skip intepreting line 8, and move to the next loop iteration, in case the value stored in **colour** is blue.

```
#!/bin/bash
#For loop with a continue
for colour in "white" "blue" "green" ; do
    if [ $colour == "blue" ] ; then
        continue
    fi
    echo "my variable has colour: $colour"
done
echo "Your loop has ended"
```

Listing 18.11: Script using a **continue**.

18.5 NESTED LOOPS

Just like with **if/else** blocks, nothing prevents a loop to be placed inside another loop. In such scenarios, the inner loop will run all of it iterations for each iteration of the outer loop.

In Listing 18.12, we present a scrpt that displays the first 5 columns of the multiplicative table.

```
#!/bin/bash
#multiplicative table
for i in {1..10} ; do
   for j in {1..5} ; do
      echo -n -e "$j x $i = $(($j*$i)) \t"
   done
   echo ""
done
```

Listing 18.12: Script using nested **for** loops.

Note that, in Listing 18.12, we used the option **-n** and **-e** for the **echo** command in order to avoid the default newline at the end of each link, as well as to ensure that echpo will interpret the special character **\t** as a tab.

EXERCISES

1. Write a script that will display: "*Bash scripting is easy*" 50 times on the terminal.
2. Write a script that displays the number of files in the current folder that ends on **.pdf**.
3. Write a script that will display the name of all **.txt** files in the current working directory that are not empty.
4. Write a script that will read all numbers contained in the file displayed below, and them display their sum on the terminal. Make sure that your file will not have empty lines at the end.

```
3
4
2
7
```

Listing 18.13: File to be used with question 4.

5. Write a script that starts by creating a variable called **password** containing the string *qwerty*. Following the script asks for the user's password, until the user enters a valid password, in which case your script should display a "*Welcome*" message, or until the user enters three invalid passwords, in which case your script should enter a "*Could not log in*" message.

6. Explain, with your own words,t he difference between the **break** and **continue** keywords.

7. Write a script that uses an **until** loop to count down from 10 to 1.

8. Rewrite the nested loop in Listing 18.12 using **while** loops instead of **for** loops.

19 FUNCTIONS

The concept of creating and calling functions is a very powerful one. In fact, it is a concept present in nearly all programming languages. The main idea behind this concept is to group lines of code that perform a particular task together, and allow them to be executed multiple times with a simple function call. By properly using functions, users can write scripts that are much easier to understand, as well as to avoid having to write the same lines of code in multiple places inside your script.

To create a function in **bash**, the following syntax can be used:

```
function FUNCTION_NAME {
    #line or lines of code that are executed every time the
    function is called
}
```

Where **FUNCTION_NAME** stands for the name the function was given. It is also possible to create a function using a different syntax as shown below:

```
FUNCTION_NAME (){
    #line or lines of code that are executed every time the
    function is called
}
```

Both syntaxes are interpreted the same way by the shell, so the choice of usign one or another is a matter of personnal preference*. Note that a function can be called at any point in your script, after the lines in which it has been defined, by simply writing its name.

*This book's author, for instance, prefers the first one.

To exemplify how powerful functions are, take a look at Listing 19.1. In it we have created a little calculator. It keeps asking the user for which operation to use, as well as for two numbers, before outputing the desired result. Note that we repeat the lines in which we ask for the numbers at four different points in the script.

We can avoid having to repeat the same lines of code multiple times by creating a function called **get_inputs**, and calling it from multiple points in our script. See Listing 19.2.

By calling the **get_inputs** function at line 17 of Listing 19.2, we are in fact executing the code in lines 4 to 7, which asks for two inputs and save these inputs on variables **int1** and **int2**. The same process happens on lines 20, 23, and 26. Note that, before the function **get_inputs** is called for the first time, the code in lines 4 to 7 has not yet been executed. This means that the variables **int1** and **int2** are only going to be defined after the first function call. As a result, trying

```
#!/bin/bash
while(true)
do
    clear
    echo "Choose from the following operations:"
    echo "Addition [+], Subtraction [-], Multiplication [/]
    Division"
    read -p "Your choice: " choice
    if [ $choice == "a" ] ; then
        echo "Enter first integer: "
        read int1
        echo "Enter second integer: "
        read int2
        res=$((int1+int2))
     elif [ $choice == "-" ] ; then
        echo "Enter first integer: "
        read int1
        echo "Enter second integer: "
        read int2
        res=$((int1-int2))
     elif [ $choice == "*" ] ; then
        echo "Enter first integer: "
        read int1
        echo "Enter second integer: "
        read int2
        res=$((int1*int2))
    elif [ $choice == "/" ] ; then
        echo "Enter first integer: "
        read int1
        echo "Enter second integer: "
        read int2
        res=$((int1/int2))
    else
        res=0
        echo "wrong choice!"
    fi

    echo "The result is:  $res"
    echo "Do you wish to continue? [y]es or [n]o: "
    echo  ans
    if [ $ans == 'n' ]
        then
         echo "Exiting the script. Have a nice day!"
        break
    fi
done
```

Listing 19.1: Script for a calculator program.

```
#!/bin/bash

function get_inputs{
    echo "Enter first integer: "
    read int1
    echo "Enter second integer: "
    read int2
}

while(true)
do
    clear
    echo "Choose from the following operations:"
    echo "Addition [+], Subtraction [-], Multiplication [/]
    Division: "
    read -p "Your choice: " choice
    if [ $choice == "a" ] ; then
        get_inputs
        res=$((int1+int2))
     elif [ $choice == "-" ] ; then
        get_inputs
        res=$((int1-int2))
     elif [ $choice == "*" ] ; then
        get_inputs
        res=$((int1*int2))
    elif [ $choice == "/" ] ; then
        get_inputs
        res=$((int1/int2))
    else
        res=0
        echo "wrong choice!"
    fi

    echo "The result is:  $res"
    echo "Do you wish to continue? [y]es or [n]o: "
    echo  ans
    if [ $ans == 'n' ]
        then
         echo "Exiting the script. Have a nice day!"
        break
    fi
done
```

Listing 19.2: Script for a calculator program using a function.

to access the contents of these variables before the first function call will result in empty values.

19.1 ESCOPE OF VARIABLES

By default, all variables defined inside a function are available anywhere inside our script. I.e., we can use these variables in the main body of our script, or even inside other functions defined in the same script*. In technical terms, these variables are **global** variables.

There are scenarios in which you may not want to have a variable created inside a function to have global escope. For example, you might want to prevent this variable from overwritting the value of other variables with identical names defined elsewhere. For these scenarios, you can set a variable to have a local escope* by defining them using the **`local`** keyword before assigning them a value. See the example in Listing 19.3.

*As long as these functions are only called after the function that creates them.

*I.e., it is only defined inside the function.

```
1  #!/bin/bash
2  function get_name {
3      echo "Enter your name (function): "
4      local name
5      read name
6      echo "Hello $name" #displays name defined inside the
       function
7  }
8  echo "Enter your name (main):
9  read name
10 get_name
11 echo "Hello $name" #displays name defined in main
```

Listing 19.3: Script containing a local variable.

Note that in Listing 19.3, the **`echo`** statement in line 14 will print the value for the variable **`name`** as defined in the main body of the script (on line 12), not the value of the variable **`name`** defined inside the function (on line 7). As a matter of fact, the local variable **`name`** defined inside the function ceases to exist as soon as we exit the function.

19.2 PASSING ARGUMENTS TO FUNCTIONS

Functions can take arguments in the same manner as a script takes arguments. I.e., we can call a function together with one or more arguments, separated by spaces, as we discussed in Section 16.5.2. These arguments are accessed inside the function according to their position. I.e., the first argument can be accessed via **`$1`**, the second argument via **`$2`**, and so it goes. See the example in Listing 19.4.

```
#!/bin/bash
function print_message {
    echo "Hello $1 $2,  today is : " ‘date‘
}
echo "Enter your first name:
read first_name
echo "Enter your last name: "
read last_name
print_message $first_name $last_name
```

Listing 19.4: Script containing a function that takes arguments.

Note that in Listing 19.4, the values stored in variables **first_name** and **last_name** are accessed from inside the function using **$1** and **$2**, respectively. Also, note that the backticks (‘ ‘) that surround the **date** command inside the function are necessary. They guarantee that the script will display the output of the **date** command, as opposed to the string "*date*".

It is important to mention that functions do not have direct access to arguments passed to the script during the script call. For example, as shown in Section 16.5.2, when calling a script with: **./myscript $arg1 $arg2**, you can access the values stored in **arg1** and **arg2** from inside the main body of the script using **$1**, and **$2**. However, you cannot use **$1** and **$2** to directly access these values from within a function defined inside this script. In order to access these values from within a function, your script must redirect them as arguments during the function call. See the example in Listing 19.5.

```
#!/bin/bash
function print_message {
    echo "Hello $1 $2,  today is : " ‘date‘
}
print_message $1 $2
```

Listing 19.5: Script function that takes redirected arguments.

19.3 RETURNING AN EXIT STATUS FROM A FUNCTION

Functions in **bash** can return an exit status*. This can be used in logical expressions, for example, where an exit status of **0** is considered a logical **true**, and any other exit status is considered a logical **false**. See the example in Listing 19.6.

*Only integers between 0 and 255 can be used.

Note that, in Listing 19.6 we do not put the **right_credential** function call within square brackets, as we normally do with other logical

```
#!/bin/bash
function right_credential {
    echo "enter your username"
    read user
    if [ $user == "marcel" ] || [ $user == "john" ] ; then
        return 0
    else
        return 1
    fi
}
if right_credential ; then
    echo "You entered a valid username"
else
    echo "You did not enter a valid username"
fi
```

Listing 19.6: Script containing a function that returns an exit status.

expressions. This is due to the fact that the return value of the function is already either **true** or **false**. In our previous examples, we had expressions that needed to be evaluated as **true** or **false** such as **[$var -leq 10]**, or the logical expressions in 19.2.

It is important to note that the **return** keyword in **bash** is only used to return exit codes. This is quite different than what most other computer languages do. For example, in most computer languages you can directly assign the return value of a function to a variable with an statement such as: **my_var=my_function**. However, in **bash** to pass a value from a function directly to a variable, we need to echo it and evaluate the output of the echo. See Listing 19.7.

```
#!/bin/bash

function my_funct1 {
    return 3
}
function my_funct2 {
    echo 3
}
my_var1=$my_funct1 #wrong
echo "The value in my_var1 is: $my_var1"
my_var2=$(my_funct2) #correct
echo "The value in my_var2 is: $my_var2"
```

Listing 19.7: Script showing how to save values echoed from functions into variables.

EXERCISES

For the exercises 1 to 4, you will need to refer to the script shown below:

```
#!/bin/bash
function my_func1{
   local var1
   fvar1=10
   fvar2=20
}
function my_func2{
   local var3
   fvar3=30
   fvar4=15
}
mvar1=10
```

1. Can you access the contents of **fvar1** and **fvar2** from within the function **my_func2**?
2. Can you access the contents of **fvar1** and **fvar3** from within the main body of the script?
3. Can you access the contents of **fvar2** and **fvar4** from within the main body of the script?
4. Can you access the contents of **mvar1** within the functions **my_func1** or **my_func2**?
5. Create a function called **is_weekend** that asks what day of the week it is, saves the answer in a variable, and then returns a value of **true** if the answer was **Saturday** or **Sunday**. Otherwise, it should return a value of false. Your function should be able to work properly with the script shown below:

```
#!/bin/bash
function is_weekend{
    #write your function code here
}
if is_weekend ; then
   echo "Today is a weekend day"
else
   echo "Today is a work day"
fi
```

6. The following script uses a lot of repeated lines of code. Rewrite this script using a function called **greeting** in order to avoid repetition. *Hint: This function might take an argument*.

```
#!/bin/bash

echo "Provide your username: "
read username

if [ $username == "john" ] ; then
    echo "Hi john, welcome back to our system "
    echo "Today is" ‘date‘
    echo "Remember to save all your information before
    logging off"
elif [ $username == "paul" ] ; then
    echo "Hi paul, welcome back to our system "
    echo "Today is" ‘date‘
    echo "Remember to save all your information before
    logging off"
elif [ $username == "george" ] ; then
    echo "Hi george, welcome back to our system "
    echo "Today is" ‘date‘
    echo "Remember to save all your information before
    logging off"
elif [ $username == "ringo" ] ; then
    echo "Hi ringo, welcome back to our system "
    echo "Today is" ‘date‘
    echo "Remember to save all your information before
    logging off"
else
    echo "You have entered a wrong username"
fi
```

20 ARRAYS

So far in this book, each variable we have defined in our previous scripts holds a single value. For example, a variable **my_var** created with the command **my_var=7** holds a value of **7**. However, there are situations in which it makes sense to group a number of values under the same variable. To achieve this, **bash** makes use of arrays. Arrays are, simply put, variables that can hold multiple values. In what follows, we will discuss hot to create arrays, how to access their contents, how to edit arrays, and finally, how to delete arrays.

20.1 CREATING ARRAYS

To create an array with **N** elements, you can use the following syntax:

```
my_array=(element0 element1 element2 ... elementN-1)
```

Note that, in the syntax above, we have started at element **0** instead of **1**. This is due to the fact that, as we will see later, **bash** indexes array elements starting with a **0**. As a result, the first element of an array has index **0**, the second element has index **1**, the third has index **2**, and the N-*th* element has index **N-1***.

*Note that each element in an array can be an integer, a string, or even a filename.

An array can also be created by assigning to each one of its indexes a proper value. For example, we can create an array with three elements using the following syntax:

```
my_array[0]=element0
my_array[1]=element1
my_array[2]=element2
```

The choice of which syntax to use depends on the scenario. For example, if you want to create an array with all the days of the week, using the first syntax makes perfect sense. However, if you want to create an array with all numbers from **1** to **100**, using the second syntax inside a loop can save a lot of typing, as shown in Listing 20.1.

20.2 ACCESSING ELEMENTS OF AN ARRAY

To access the value contained in an individual element of an array, we need to use the syntax: **${my_array[index]}**, where **my_array** denotes the name of the array, and **index** is an integer*. For example, **echo ${my_var[2]}** prints the contents of the second element of an array **my_var**, and **today=${week[2]}** saves the value stored in the third element of an array called **week** into a variable called **today**.

*Note that the curly braces are mandatory.

```
#!/bin/bash

index=0
for number in {1..100} ; do
    my_array[index]=number
    index=$(($index+1))
done
```

Listing 20.1: Script creating an array with numebrs from 1 to 100.

It is also possible to expand an array to use it with a **for** loop. This can be accomplished using the syntax **for var in "${my_array[@]}"**, where **var** denotes the variable that will, at each iteration, receive one value stored in the array. In Listing 20.2, we use this syntax to print the names of all days of the week:

```
#!/bin/bash
week=(Monday Tuesday Wednesday Thursday Friday Saturday
    Sunday)
for day in "${week[@]}"; do
    echo "$day"
done
```

Listing 20.2: Script that displays all elements from an array.

Finally, it is also possible to get the number of elements of an array*, using the syntax: **${#my_array[@]}**.

Which is normally, but not always equal to its last index plus one. See the Sparse Arrays box.

Sparse Arrays

All arrays we have created in our examples so far are full arrays. I.e., all indexes from **0** until its last occupied index have values stored in them. However, this does not need to be the case. For example, if we run the commands **new_array[3]=34**, and **new_array[10]=24**. We are in fact creating an array called **new_array** with only two elements. However, the occupied indexes are **3** and **10**, as opposed to **0**, and **1**.
Arrays that only have a handful of indexes occupied, i.e., an array in which there are gaps between indexes, are useful in a few scenarios. They are normally called sparse arrays.
Attempting to access the values stored in non-occupied indexes of an array in **bash** will not result in an error, like it would be the case in many other computer languages. It will simply result in an empty answer.

20.3 EDITING THE CONTENTS OF AN ARRAY

Arrays, just like any other type of variable, can have the values that are stored in them changed at any time during the script's execution. In Listing 20.3, we create an array with usernames on line 2. Following, we ask the user to provide and index and a new username, and we use this information to change the value stored in the corresponding element. Finally, the script prints the new contents of the array.

```
1 #!/bin/bash
2 usernames=(john paul george ringo)
3 echo "Provide the index of the element you want to change"
4 read index
5 echo "Provide the new username"
6 read username
7
8 usernames[$index]=username
9 for user in "${usernames[@]}"; do
10     echo $user
11 done
```

Listing 20.3: Script that allows users to change elements from an array.

It is possible to append items to the end of an array using the **+=** operator, as shown in the example below:

```
marcel@dell:~$my_array=(bananas apples grapes)
marcel@dell:~$echo "{my_array[@]}"
bananas apples grapes
marcel@dell:~$my_array+=(strawberries pears)
marcel@dell:~$echo "{my_array[@]}"
bananas apples grapes strawberries pears
```

This keyword can also be used to avoid the need of creating a variable to keep track of the index, as we did in Listing 20.1. In fact, we can simplify that script significantly, as shown in Listing 20.4.

```
1 #!/bin/bash
2 for number in {1..100} ; do
3    my_array+=($number)
4 done
5 for index in {0..99} ; do
6    echo ${my_array[$index]}
7 done
```

Listing 20.4: Simplified script to create an array with numbers from 1 to 100.

Note that, the round brackets shown in Listing 20.4 are necessary. Without them, the new contents will be appended to the last element of the list, as opposed to in a new element.

20.4 DELETING AN ARRAY

Arrays that are created programatically using loops can end up taking a lot of memory*. Hence, it is important to delete these arrays, once they are no longer needed, in order to free memory for your script to perform other tasks.

*Listing 20.1 could be easily changed to create an array of a million numbers.

To delete an entire array, we simply need to use the **unset** keyword followed by the array name, as shown below:

```
unset my_array
```

Note that you can also apply the **unset** keyword to a particular element of an array, for example using **unset my_array[1]** to delete its second element. In fact, you can use this keyword to delete any variable previously created.

EXERCISES

1. What is the difference between a regular variable and an array?
2. How can you create an array containing the names of all months of the year?
3. Write down a script that will create an array with all even numbers between **0** and **100**. *Hint: Use a loop.*
4. How can you print the number of elements inside an array?
5. How can you delete an array called **my_array**?

21 PERMISSIONS, USERS, AND GROUPS

As mentioned in Chapter XX, Unix was created in an era in which large central computers were shared among multiple users. Hence, one of its main design goals was to support multiple users. As an Operating System (OS) inspired by Unix, Linux has inherited its multiuser support. As a result, nowadays, Linux systems are often used in academic, as well as in industrial systems, in which multiple users need to share the same resources. Examples of such systems are supercomputers, servers, as well as cloud services. In a multiuser system, the following requirements are of fundamental importance:

- Each user should be able to control who have access to his/her files and folders.
- Regular users should not be allowed to interfere with processes initiated by other users. However, sysadmins should be able to take action in case some processes are not behaving properly.
- Sysadmins should be able to create new users, edit the configurations of existing users, and delete users from the system.

In this chapter, we start by discussing files and folders permissions, which allow for a fine control of which users can access which files and folders. Following, we introduce the concepts of **`sudo`** access and the **`root`** user, which allow system admins to perform a large number of administrative tasks, such as process control, as well as software and user management. Finally, we end by explaining how to create, edit, and delete users and groups.

21.1 FILE AND FOLDER PERMISSIONS

When asking for a list of all files and folders in the current working directory with **`ls -l`**, you should get an output similar to the one provided in Listing 21.1:

This output has seven fields. In Table 5.1, we discussed what each field represents, using the file **`seneca.pdf`** as an example. In this section, we are focusing on the first field: files and folders permission sets. Given the fact that files and folders behave differently*, their permissions settings are slightly different. In what follows, we cover permissions for files first. Following, then we cover permissions for folders.

*For example, you can write or execute files, but not folders.

```
marcel@dell:~$ ls  -l
drwxrwxr-x  2 marcel marcel  4096 Jun 21 23:06 Music
drwxrwxr-x  2 marcel marcel  4096 Jun 21 23:06 Video
-rw-rw-r--  1 marcel marcel 12238 Jun 29 22:54 read_me
-rwxrwxr-x  1 marcel marcel   126 Jun 28 20:52 script.sh
-rw-r-----  1 marcel admin   2238 Jun 12 21:24 guide.pdf
```

Listing 21.1: ls -l.

21.1.1 *File permissions*

There are three different types of file permissions. They are shown on Table 21.1 together with their acronyms and descriptions.

TYPE	AC.	DESCRIPTION
read	**r**	Grants permission to access the file's contents, but not to edit it or execute it
write	**w**	Grants permission to edit the file's contents. Given the fact that a user needs to access the contents of a file in order to edit it, this permission is only effective if granted together with a **read** permission.
execute	**x**	Grants permission to execute the file's contents as a script. Given the fact that the shell needs to access the contents of a file in order to execute it, this permission is only effective if granted together with a **read** permission.

Table 21.1: Types of file permissions.

A full file permission set is comprised of ten characters that are divided in four fields: one to represent the type of file, one to define the permissions set for the user who owns the file, one to define the permissions set for the members of the group owner of the file, and finally one to define the permissions for everyone else in the system. We present a detailed description of each field in what follows.

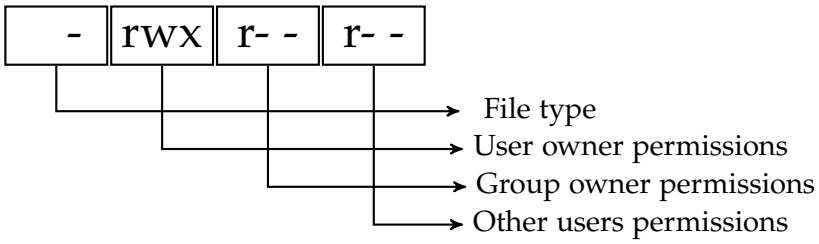


Figure 21.1: File Permissions.

FILE TYPE The first character of a file's permission set contains an acronym that determines what kind of file it is*. A list containing the most important types of files, as well as their acronyms is provided in Table 21.2.

*Remember that everything in Linux is considered a file.

AC.	DESCRIPTION
-	A dash represents a regular file, such as a text file, a script, an mp3 fle, etc.
d	A **d** stands for a directory. Permission sets for directories are covered in the next section.
l	An **l** stands for a symbolic link. Symbolic links were covered in Section XXX.
p	A **p** stands for a named pipe. Pipilines were overed in Section XXX.
s	An **s** stands for a socket, which are used for data communications between different processes.
b	A **b** stands for a block device.
c	An **c** stands for a character device.

Table 21.2: File types.

USER OWNER PERMISSION SET Characters two to four define the read, write, and execute permissions for the user who owns the file.

GROUP OWNER PERMISSION SET Characters five to seven define the read, write, and execute permissions for all users in the group that owns the file*.

*Groups are discussed in Section XXX.

OTHERS PERMISSION SET Characters eight to ten define the read, write, and execute permissions for all other users. I.e., users that are neither the user owner, nor members of the group owner of the file.

In Listing 21.1 on page 147, we showed the output of a **ls -l** command. In it, you can see that there are two directories and three files in the current working directory, each one with different sets of permissions. In what follows, we explain what the permission sets of each one of the three files represent.

read_me This file has a permission set **-rw-rw-r-**, which is the default set when a file is first created. It allows both the user owner (**marcel**), as well as users that are members of the group owner (also called **marcel***), to read and edit (write) this file. Other users can only read this file. Note that no user can execute this file.

*We will explain why the group owner has often the same name as the user owner in Section XXX..

script.sh This file has a permission set **-rwxrwxr-x**. It allows both the user owner (**marcel**), as well as users that are members of the group owner (also called **marcel**), to read, edit (write), and execute this file. Other users can also read and execute this file, but cannot edit (write) it.

config.pdf This file has a permission set **-rw-r---**. It allows the user owner (**marcel**) to access and edit (write) this file. Users that are members of the group owner (called **admin**) can access the contents of this file, but cannot edit (write) it. Other users cannot access or edit this file. Note that no user can execute this file.

21.1.2 *Directory Permissions*

Like regular files, directories also have three different types of file permissions. Also, like files, these permissions have acronyms **r**, **w**, and **x**. However, the meaning of these permissions are slightly different than the ones shown on Table 21.1. On Table **??**, we present a list of directory permission acronyms together with a description adn examples of usage.

AC.	DESCRIPTION
r	Grants permission to read the contents of the directory. For example, it allows users to use the get a list of all fiels and folders using the **ls** command.
w	Grants permission to create files inside the directory. For example, it allow users to create new text files inside the directory using **vim** or **nano**.
x	Grants permission to set the directory as the current working directory. For example, it allow users to use the **cd** command to set it as its current working directory.

Table 21.3: Types of dirctory permissions.

The topic of file permissions is closely related to that of users and groups. In our next sections, we cover these two topics.

21.2 USERS

Linux stores information about users with access to the system in a file called **\etc\passwd**, as shown in Listing 21.2, where its last 5 lines are displayed using a **tail -5 \etc\passwd**:

In this file, each line corresponds to one user, and each column (field) correspond to one property*. The first lines of this file are normally filled with users that are actually not real users, but act as users on behalf of the operating system or different applications.

**Fields are separated by colons(:).*

```
tail -5 /etc/passwd
inge:x:518:524:art  dealer:/home/inge:/bin/ksh
marcel:x:100:100:instructor:/home/marcel:/bin/bash
john:x:101:101:student:/home/john:/bin/sh
george:x:102:102:musician:/home/george:/bin/sh
linus:x:103:103:linux enthusiast:/home/linus:/bin/ksh
```

Listing 21.2: Last five lines of a **passwd** file.

These users, often called daemons, are frequenty used to start processes or create log files. It is easy to identify which users are real users. Real users need a password to access the system, hence they should have an **x** appearing in its second field.

As you can see, each line in this file contains seven fields. In Table 21.4, we provide a description of each one of these fields, using the line that set the properties of the user **marcel** as an example.

21.2.1 *Adding new users*

To add a new user into the system, the **useradd** command can be used. This command has the following syntax**We cover sudo in Section XXX.**:

```
sudo useradd [options] USER_NAME
```

useradd is most often used with the **-m** and **-s** options, in order to grant to this new user a home folder and provide him with a description. In Listing 21.3 we use the **useradd** command to add a user with username **john** to our system. This user is granted a home folder, **/home/john** as well as a description, **newbie**.

```
sudo useradd -m -c "newbie" john
tail -5 /etc/passwd
marcel:x:100:100:instructor:/home/marcel:/bin/bash
john:x:101:101:student:/home/john:/bin/sh
george:x:102:102:musician:/home/george:/bin/sh
linus:x:103:103:linux enthusiast:/home/linus:/bin/ksh
john:x:104:104:newbie:/home/john:/bin/sh
```

Listing 21.3: Adding a user to a Linux system.

Note that, after adding a user to the system, a new line was added at the bottom of the **/etc/passwd** file. Note that this user has, **sh**, as opposed to **bash** as its default shell. This is due to the fact that **useradd** uses a file **/et/default/useradd** to retrieve its default set-

FIELD	EXAMPLE	DESCRIPTION
username	**marcel**	Name that the user enters in order to be granted access to the system.
password	**x**	This field is kept here mainly for historical reasons. An **x** indicates that a password, if it exists, is stored in the **/etc/shadow** file, as we will cover in Section XXX.
user id	**100**	Linux assigns a unique **user id** for each one of its users. This is similar to governamental agencies assigning numbers such as driver license numbers, or social insurance numbers, to its citizens.
primary group id	**100**	Linux assigns a unique **group id** to the primary group of each user. We cover groups on Section XXX.
description	**instructor**	Linux allows a small description to be assigned to each user. In our example, these descriptions are used to display the job of each user.
home directory	**/home/marcel**	This field specifies which directory works as the user's home directory. Every time the user logs into the system, this is the folder used as the first working directory.
login shell	**/bin/bash**	This field specifies which shell should be used when the user logs into the system.

Table 21.4: Description of the fields in the **passwd** file.

tings. And this file, by default, sets **sh** as its default shell. This behaviour can be changed by altering the **/etc/default/useradd** file.

The new user, **john**, still cannot get access to the system as it was not yet granted a password. To grant a password to a new user, we need to use the **passwd** command, as shown below:

```
sudo tail -1 /etc/shadow
ADD THE LINE HERE XXXX
sudo passwd john
Enter new UNIX password:
Retype new UNIX password:
passwd: Password updated succesfully
sudo tail -1 /etc/shadow
ADD THE LINE HERE XXXX
```

Listing 21.4: Providing or changing a password for a given user.

In this example, you can see that the password field for the user john in the **/etc/shadow** had only an exclamation mark (!). After granting this user a password, this field contains a hashed version of the provided password. A hashing algorithm takes a sequence of characters, and outputs another sequence of characters. It is a one-way algorithm. I.e., given a password it is easy to obtain its hash. However, given a hashed version of password, it is nearly impossible to retrieve the original password. Storing a password in a hashed format, as opposed to storing it in plain-text, greatly enhances the security of the system.

21.2.2 *Changing users*

To switch from one user to another, the switch user command, **su**, is used. Its syntax is a shown below:

```
su [options] USER_NAME
```

For example, to switch from user **marcel** to user **john**, the following command can be used: Note that the **su** command, by default, does

```
su john
Password:
pwd
```

Listing 21.5: Switching users.

not change the current working directory. However, it is possible to switch users and, at the same time, switch to the new user's home

folder. This cna be done by entering a dash before the name of the new user, as in **su - john**.

The **su** can also be used without providing a username as an argument. In this case, it switch to the **root** user. In most Linux systems, a **root** user is created during installation. Ubuntu is an exception. In order to use the **root** user in Ubuntu, you need to first povide it a password using **sudo passwd root**.

21.2.3 *Editing existing users*

It is possible to change the settings of particular users by using the **usermod** command together with the proper options. Its syntax is:

```
sudo usermod [options] USER_NAME
```

Among the possible changes that **usermod** can perform are: locking and unlocking a user, changing the user's default shell, changing the user's default home folder, changing the user's password expire date, change the user's primary group, or even add the user as a member of existing groups. For a comprehensive list of all its options, access its manual with **man usermod**. In Listing 21.6, we provide a few examples of usege of the **usermod** tool.

```
usermod -c "new description john
usermod -L john #lock the user john
usermod -U unlock the user john
uermod -s sh john # change the shell for user john
usermod -e 2016-11-28 # set the user's john password to
    expire at the provided date
usermod -aG linux john # add john to the group linux
```

Listing 21.6: Switching users.

21.2.4 *Removing existng users from the system*

To remove a user from the system, the **userdel** command can be used. This command has the following syntax:

```
sudo userdel [options] USER_NAME
```

By default, this command does not delete the user's home folder, which can lead to wasted space in the system' s memory. In order to do remove the user's home folder, at the same time the user is being removed, the option **-r** must be used.

21.3 GROUPS

Groups are used when a number of files and folders need to be shared with multiple users. They allow systems administrators to provide a set of permissions for these files and folders for some users, while preventing other users from gettign the same set of permissions. For example, in the output of the ls -l command shown in Listing 21.1, both the user owner **marcel**, as well as all users that are members of the group **admin** can read and edit the file XXX. All other users are allowed to read, but are prevented from editing this file.

In Linux, the file **/etc/group** file, as shown in Listing 21.7tores information about the current groups in he system, as well as its members.

```
tail -5 /etc/group
john:x:101:
john:x:101:
john:x:101:
john:x:101:
```

Listing 21.7: Contents of the **/etc/group** file.

In this file, each row corresponds to a particular group. There are four fields in each row. In Table 21.5 we explain what each field represent, using the last row in Listing 21.7 as an example.

FIELD	EXAMPLE	DESCRIPTION
group name	**marcel**	Name of the group.
password	**!**	The group's password. It is normally empty, which is represented by an exclamation mark **!**. We cover groups passwords in Section XXX.
group ID	**1201**	ID number associated with the group.
list of users	**john**	All users within a group are listed in this field. In case of multiple users, they are separated by commas.

Table 21.5: Description of the fields in the **group** file.

21.3.1 *Primary Groups*

By inspecting the **/etc/passwd** and **/etc/group** files, you can see that each username in **/etc/passwd** has a matching group in **/etc/etc**. This is not a coincidence. When a user is created in a Linux system, a

group with the same name as the username is created and assigned as its primary group. A primairy group of a given user is the group that is assigned by default as the group owner of all files created by the user. For example, when the user **marcel** creates a new file, this file will have **marcel** as its user owner, and the primary group of **marcel***, as its group owner.

*Also named **marcel**.

21.3.2 *Creating, configuring, and deleting groups*

To add a new group to a Linux system, the **groupadd** command can be used. It has the following syntax:

```
sudo groupadd [options] GROUP_NAME
```

In Listing 21.8, two groups called **students** and **faculty** are added to the system. Note that, after adding groups to the system, new line

```
sudo groupadd students
sudo groupadd faculty
tail -5 /etc/group
XXX
XXX
XXX
```

Listing 21.8: Adding a group to a Linux system.

were added at the bottom of the **/etc/group** file.

After being created, a group can have some of its properties changed with the **groupmod** command. This command has the following syntax:

```
sudo groupmod [options] GROUP_NAME
```

In Listing XXX, we use the **groupmod** command to alter a few properties of some groups such as: the group's name, the XXX ofthe group, etc.

```
sudo groupmod -n new\_name old\_name}
XXX
```

Listing 21.9: Changing group properties with **groupmod**.

To delete a group from the system, the **groupdel** command is used. It has the following syntax:

```
sudo groupdel GROUP_NAME
```

21.3.3 *Adding and removing users from groups*

You can add a user to a group with usermod -aG group_name username without the -a, all groups are repalced by a new one.

Or change the user's primary group with usermod -G GROUP_NAME USERNAME

21.3.4 *Granting group ownership to regular users*

By default, group operations can only be performed using **sudo** access. However, it is possible to assign ownership of a group to a regular user so that this user can add other users to the group. This is accomplished with the **gpasswd** command using the syntax below:

```
sudo gpasswd -A GROUP_NAME USERNAME
```

After being granted administration rights over a group, regular users can modify the group using the **gpasswd** command together with a proper choice of options. In Listing 21.10, we provide examples of a few changes that a group administrator regular can do to a group.

```
gpasswd -a group\_name username
XXXX
XXXX
XXXX
```

Listing 21.10: Making changes to groups using **gpasswd**.

info about group passwd in /etc/gpasswd and etc/gshadow
tools to identify users: id.

EXERCISES

1.