INTRODUCTION TO LINUX

MARCEL JAR

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

CLI	Command Line Interface
GUI	Graphical User Interface
UI	User Interface
OS	Operating System
FHS	File system Hierarchy Standard

Part I

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.

1

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 1.1.

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

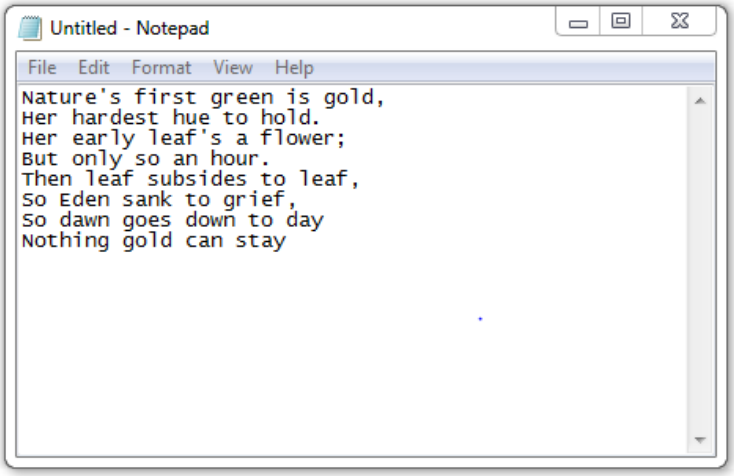


Figure 1.1: Notepad application for Windows 7.

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 1.1 and 1.2.

*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.

1.1 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.

1.2 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 1.1 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 1.1: 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?

2 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.

2.1 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`**.

2.2 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 2.8 , 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***.

2.3 **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.

2.4 **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
```

2.5 `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
```

2.6 `ls`

To obtain a list of all files and folders present in the working directory, as demonstrated in Chapter 1, 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 2.1, 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 2.1: Long list information for the **seneca.pdf** file.

2.7 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.

2.8 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:~$
```

2.9 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 2.1, and you want to switch the current working directory from any folder directly to any other folder.

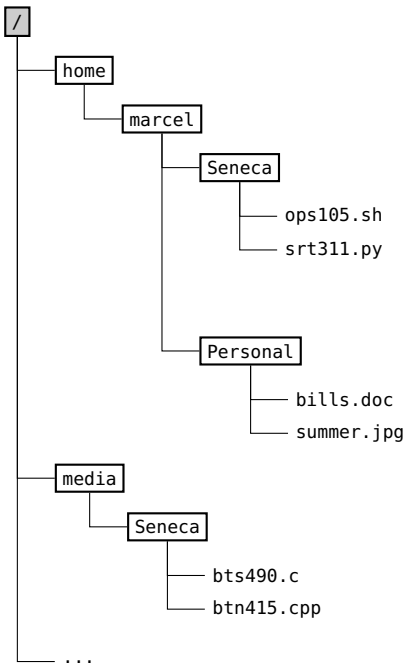


Figure 2.1: Directory tree.

For example, given the directory tree shown in Figure 2.1, 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.

2.9.1 *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
```

2.9.2 *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.

2.10 **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.*

2.11 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 2.2, 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?

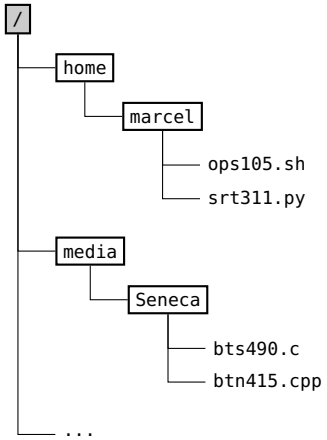


Figure 2.2: Directory tree for questions 9, 10, and 11.

3

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.

3.1 `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**.

3.2 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:~$
```

3.3 touch

As shown in Section 2.6, 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..*

3.4 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.

3.5 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.

3.5.1 *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.

3.5.2 *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.

3.6 **cp**

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

**short for copy.*

3.6.1 *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:~$
```

3.6.2 *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 3.1.
5. Still with regards to the diagram diagram in Figure 3.1, 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?

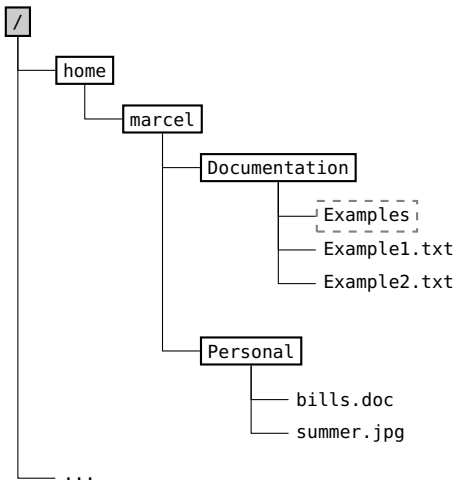


Figure 3.1: Directory tree for questions 4 and 5.

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.

4

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 2 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 2.1, 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.

4.1 `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.

4.2 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 4.1 (page 28). 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.

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
```

4.2.1 *Sections*

Note that the **mkdir** command appears with a **(1)** next to it at the top of Listing 4.1 (see page 28). 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 4.1.

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 4.1: 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
```

4.3 `whatis`

As shown in Listing 4.1, 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
```

4.4 `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*:

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

```
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
```

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.

4.5 `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 4.2.

**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 4.2: 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.*

```
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 4.1: Manual page for the **mkdir** command.

5

READING AND EDITING TEXT FILES FROM THE SHELL

In Chapter 2, 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**).

5.1 READING TEXT FILES

5.1.1 *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.

5.1.2 *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 2 and 3. 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 5.1, 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 5.1*.

*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

```
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 5.1: 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 5.1: Less navigation keys.

in this chapter, **nano** and **vim**, as well as the **top** command covered in Chapter XXX.
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**).

5.2 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.

5.2.1 *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:

```
1  GNU nano 2.2.6         File: poem
2
3  Nature's first green is gold,
4  Her hardest hue to hold.
5  Her early leafs a flower;
6  But only so an hour.
7  Then leaf subsides to leaf.
8  So Eden sank to grief,
9  So dawn goes down to day.
10 Nothing gold can stay.
11
12 ^G Get He^O WriteOut^R Read Fi^Y Prev Pag^K Cut Te^C Cur Pos
13 ^X Exit  ^J Justify ^W WhereIs^V Next Pag^U UnCut ^T To Spel
```

Listing 5.2: Nano's user interface.

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.

5.2.2 *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** creates a new file.

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

Not that in Listing 5.3, 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 5.2 presents a list of all information that is displayed, as well as the corresponding values displayed in Listing 5.3.

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.

```
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" 9L, 210C                          1,1           All
```

Listing 5.3: **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 5.2: Information displayed in **vim**.

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 5.3. To switch back from **insert mode** to **command mode**, you need to press **Esc**

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 5.1, 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.

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 5.3: Keys to switch to insert mode.

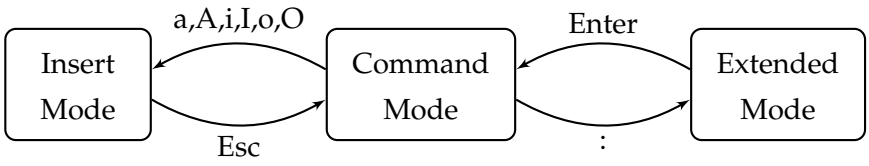


Figure 5.1: Vi operational modes and the keys required to change modes.

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 5.4 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..

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 5.5, 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.

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 5.4: List of some **command mode** important commands.

: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 5.5: 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 II

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.

6 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.

6.1 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.

6.1.1 *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.

6.1.2 *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 6.1.4.

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.

6.1.3 ***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, modify, and change. 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..

6.1.4 *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 6.1 below depicts the contents of the directory file for **/home/marcel**.

*As we explain in Section 6.2, everything in Linux is a file.

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

Table 6.1: Contents of the **/home/marcel** directory file.

6.1.5 *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 6.1.

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

Even though the sequence of steps presented in Figure 6.1 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.*

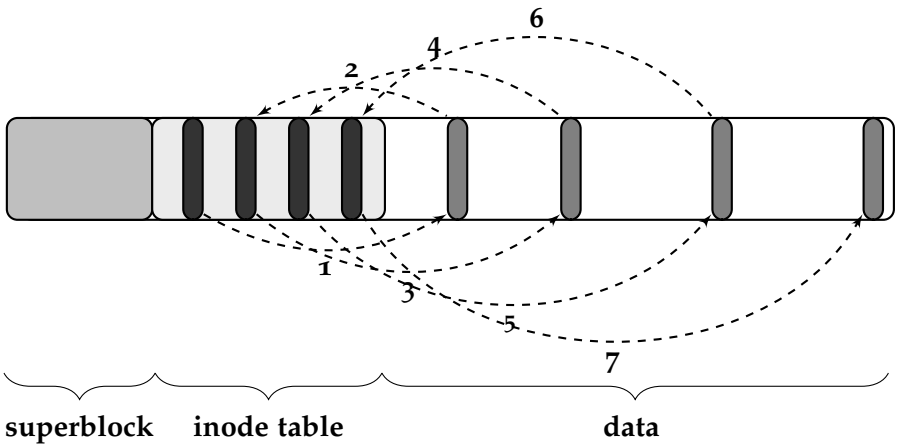


Figure 6.1: Sequence of steps to access a text file.

- 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.

6.1.7 *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.

6.2 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 6.2, 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 contained data that are 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 6.2: 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?

7 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.

7.1 HARD LINKS

Following our discussion in Section 6.1, 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.

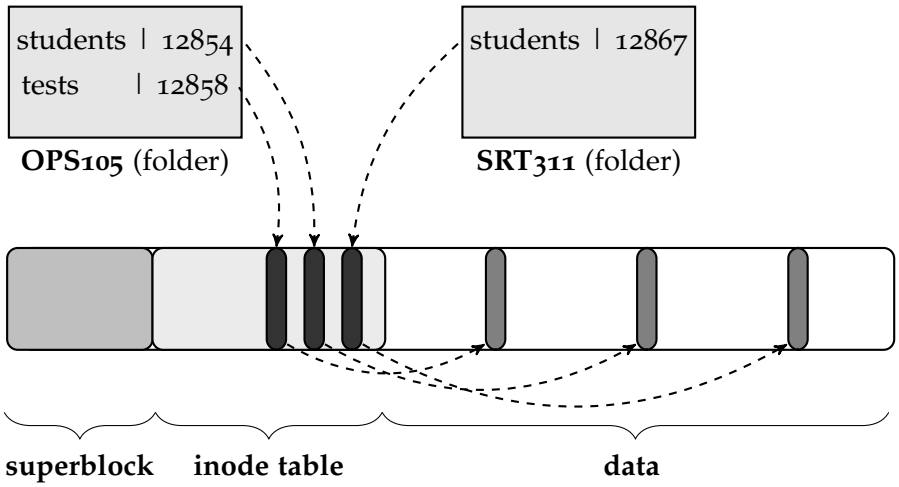
When a hard link is created, the OS simply adds a new entry to the chosen directory file with the same inode number as the original file. Also, this procedure will automatically increment the hard link count associated with the file by one.

In Figure 7.1, 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 7.1a. Then, a hard link to this file, called **`tests_ln`**, is created inside a folder **`/SRT311`**. You can see in 7.1b how both the original file **`tests`** and the hard link **`tests_ln`** point towards the same inode entry.

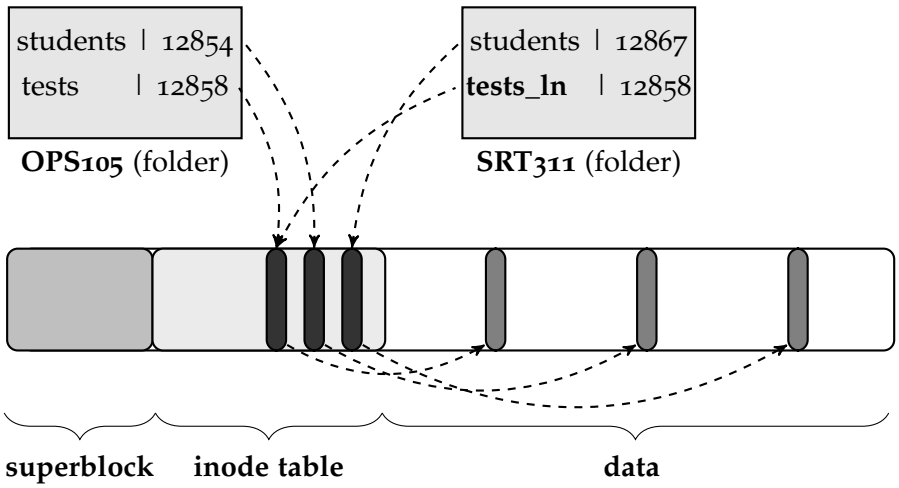
7.1.1 *`ln`*

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 51.

*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 7.1: 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).

7.1.2 *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 7.1b. 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.

7.1.3 *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.

7.1.4 *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.

7.2 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*.

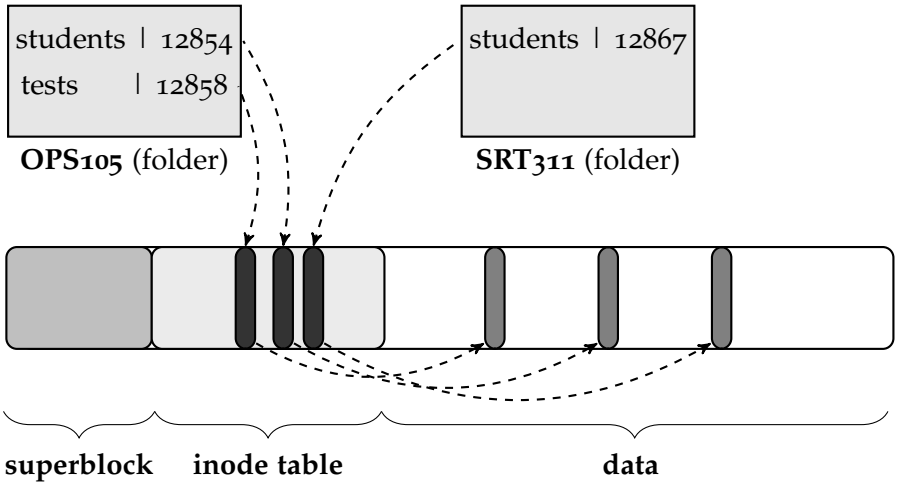
*They predate Windows shortcuts by many years, though.

In Figure 7.2, 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 7.2a. Then, a soft link to this file, called **tests_ln**, is created inside a folder **/SRT311**. You can see in 7.2b 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**.

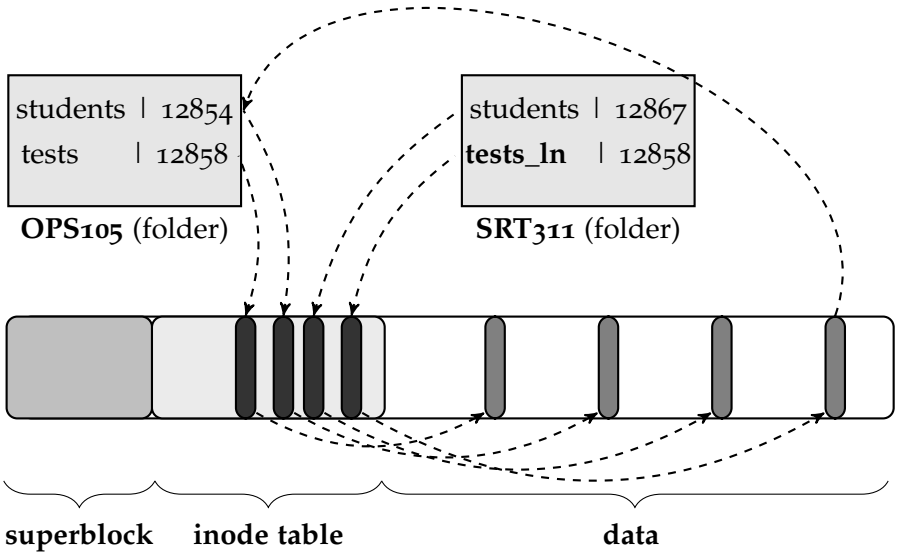
7.2.1 *ln -s*

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 55):

```
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 7.2: 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.

7.2.2 *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 7.3, answer the following questions:

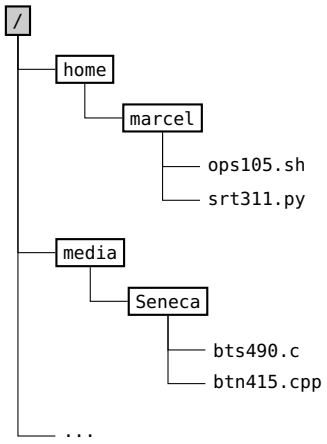


Figure 7.3: Directory tree for questions 2 and 3.

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?

8

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.

8.1 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.

8.2 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**, where 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.

8.3 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]**.

8.4 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.

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.