

Elements of Operating Systems

Chapter 3

In Defence of the Shell

J. Fornés M. Farreras Y. Becerra

Barcelona School of Informatics
Universitat Politècnica de Catalunya

Bachelor's Degree in Informatics Engineering — Operating Systems

Preview for readers · September 28, 2026



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

© Universitat Politècnica de Catalunya

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International licence (CC BY-NC-SA 4.0):

«By Universitat Politècnica de Catalunya — BarcelonaTech (UPC), Any 2026»



<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en>

This text was written with the help of Claude, Anthropic's language model. Any errors that remain are the authors'.

3 In Defence of the Shell

3.1 What a shell is

Every operating system has to provide, by definition, a programming interface and a user interface for interacting with the kernel's services. For programming we have the system calls, which we will study in many of this book's chapters. For the user we have the windowing environment but also, in text mode, we can run commands.

A *shell* is the command interpreter. Put simply: an endless loop that reads lines —strings of characters ending in a newline—, interprets them (splits them into programs and options) and runs the programs with the interpreted options.

Notice that the kernel appears nowhere in this definition. The shell is not part of it: it is a user program like any other, which calls `fork()`, `exec()` and `wait()` just as you would. In Chapter 10 we will write a minimal one, able to run `ls | wc`, and you will see that it fits in thirty lines of C.

Every Unix programmer has been tempted to write one. Some —many— gave in to the temptation, such as Steve Bourne, creator of the *Bourne shell* (`sh`), the origin of the *Bourne Again shell* (`bash`). Linux distributions usually ship `bash` by default, and macOS ships `zsh`; but there are versions of both —and of many others— for both systems. It would not be strange to find others on the lab computers: `csh`, `ksh` or even plain `sh`, the simplest of them all.

You may wonder what the difference is. In the things this book deals with —running programs, redirecting their input and output, connecting them with pipes— almost none: what you learn here works

in all of them. The differences are about convenience and about the scripting language, and they are better learnt by using them than by reading comparisons.

3.2 The manual

Before any other command, the one that explains them all. Every Unix comes with its manual installed, and you read it with **man**:

```
$ man ls
```

The page fills the whole terminal and is read with a pager, usually **less**. A handful of keys is enough:

Key	What it does
space, b	forward or back one screen
/word	search forward for word
n, N	go to the next or the previous match
q	quit
h	the pager's help, with all the others

The manual is divided into numbered **sections**, and in this book you will use four of them:

Section	What is in it	For example
1	commands	ls , chmod , man
2	system calls	write() , fork()
3	C library functions	printf , strlen
7	overviews and conventions	signal , pipe , inode

Sections 1 and 2 are the two interfaces of Chapter 2: the user one and the programming one, each with its own section. That is why in documentation you will find names with a number next to them, **write(2)** or **printf(3)**: it means “in section 2” or “in section 3”, and it is not an argument.

i Careful

The same name may be in more than one section, and if you don't say which, **man** shows you the first one it finds —and it starts with section 1. Try it with the call from Chapter 2:

```
$ man -f write
write (1)          - send a message to another user
write (1p)         - write to another user
write (2)          - write to a file descriptor
write (3p)         - write on a file
```

A plain **man write** explains a command for sending messages to another user, which has nothing to do with the call; the call is **man 2 write**. The sections with a **p** (**1p**, **3p**) are the pages of the POSIX standard: they say what the command or function must do on any Unix, not what Linux in particular does. The same happens with **printf** and with **kill**. When you look up a system call or a function, always give the number.

The **man -f** you have just seen tells you which sections a name you already know is in. When you don't know the name, **man -k** searches for a word in the descriptions of every page:

```
$ man -k 'create pipe'
pipe (2)          - create pipe
pipe2 (2)         - create pipe
```

And if you want to know everything **man** itself does, you know where to look: **man man**.

 Going further

If one day you want more than a call's syntax, section 7 is the one you can read end to end. It describes no command and no function: it explains how one part of the system fits in with the rest, and which calls take part. `man 7 signal` and `man 7 pipe`, for example, are among the best overviews you will find anywhere. Read them when you reach Chapters 7 and 10: they will tell you the same as this book, in the system's own vocabulary.

3.3 A few commands to whet your appetite

A glance at the `/bin` directory is enough to see how many commands you have at your disposal. You can call all of them from a shell, and you will find them, with more or less the same arguments, on every Unix system. Here we only give you a small taste. We hope it gets you hooked on the keyboard.

The `ls` command lists the contents of a directory. Those contents—the data block of a directory—consist of *[inode, name]* pairs. But those names do not identify the files: you can have the same name in different directories and have them belong to completely different files. What really identifies a file is its index node, or *inode*. Inside that data structure the filesystem keeps the file's management information—owner, permissions, access times—and also which data blocks have been allocated to it.

With no arguments, `ls` shows only the names, that is, one element of each pair. If you add the `-l` option (for *long*), to show more information about each file, it will have to read not only the directory's data block but also the contents of the inodes.

Let's go through an example of all the information this option shows. We have added the options `-i` (to show the inode number) and `-a` (to show all the files, even `.` and `..`) so that we can explain it better. The listing comes from a different machine, with the system set to Catalan—hence `set` for September—; the lab's `/dev` is not the same, but it reads the same way:

```
alumne@labS0:/dev$ ls -lia
total 4
  2 drwxr-xr-x 17 root root      3840 set 25 15:11 .
  2 drwxr-xr-x 20 root root     4096 set 15  2021 ..
342 crw-r--r--  1 root root    10, 235 set 25 15:06 autofs
215 drwxr-xr-x  2 root root      340 set 25 15:11 block
153 drwxr-xr-x  2 root root     3100 set 25 15:06 char
 15 crw--w----  1 root tty       5,   1 set 25 15:06 console
148 lrwxrwxrwx  1 root root       11 set 25 15:06 core ->
    /proc/kcore
328 brw-rw----  1 root disk   253,   0 set 25 15:06 dm-0
312 drwxr-xr-x  3 root root      100 set 25 15:06 dri
```

One step at a time.

Don't worry about the first line (`total 4`) for now. It shows how much room these files really take up on the disk, but we will get to that in Chapter 11.

All the other lines follow the same pattern. Let's take any one of them to fix ideas:

```
15 crw--w----  1 root tty  5,  1 set 25 15:06 console
```

3.3.1 The inode

```
15 crw--w----  1 root tty  5,  1 set 25 15:06 console
```

15 is the file's inode. This number is unique across the whole filesystem, and it is what really identifies the file. We can have many different names for the same file* and many different files with the same name, though not in the same directory. This number only appears with the `-li` option.

3.3.2 The type and the permissions

```
15 crw--w----  1 root tty  5,  1 set 25 15:06 console
```

*What we call *links*. Look at the manual page for `ln`.

First comes the letter **c**, which means that this file is a character device: the smallest unit of access is the character*.

Then come nine characters, **rw--w----**, which you must group in threes for *read*, *write* and *execute*. They are the file's permissions for the owner, for the group of users it belongs to, and for everybody else. So the owner—the user **root**—has read and write permission (**rw-** = 110) but not execute. The members of the **tty** group have write permission, but neither read nor execute (**-w-** = 010). And the rest of the world has none (**---** = 000).

i Careful

On Unix a file can only be executed if it has execute permission, whatever its name. Windows' **name.extension** business means nothing here.

To change permissions you have the **chmod** command (*change mode*). If you think of the permission bits in octal, you already have the arguments. For example, if you want to give read permission to the members of the **tty** group and leave everything else as it is, you would want **rw-**, that is, $110_2 = 6_8$. For the owner, **rw-** too, $110_2 = 6_8$, and for the rest of the world **---**, $000_2 = 0_8$. In short:

```
$ chmod 660 console
$ ls -li console
15 crw-rw---- 1 root tty 5, 1 set 25 15:06 console
```

3.3.3 The number of links

```
15 crw-rw---- 1 root tty 5, 1 set 25 15:06 console
```

This 1 is the file's number of *links*: how many different names this inode has across the whole filesystem or, put another way, how many different pathnames lead to the same file. In this case, just one.

*There are quite a few different file types. How many? Run **man 7 inode** to see them all, and while you are at it, the rest of the information kept in the inode.

But look at the first line of the listing, the one for the directory `.`: inode 2 has **seventeen** links. How can that be? In fact that is few; that filesystem is small, and you will find installations where inode 2 has many more. The reason is that inode 2 is the root directory*.

Question 1

We know that inode 2 is the root because it is the only directory where the files `.` and `..` are the same inode. Why? And why does every new directory you create add one to its parent's link count?

Answer

Because `..` is a name of the parent directory, and so one more link to its inode. The root has no parent, and that is why its `..` points to itself. The answer to the second part follows from this: any directory has at least two links—its name inside the parent and its own `..`, and it gains one for every subdirectory hanging from it.

We will leave it here. How the filesystem stores all this, why the inode does not contain the file's name and what exactly happens when you delete one are covered in Chapter 11.

3.4 Redirections

So far, commands write to the screen and read from the keyboard. That is a half-truth, and the missing half is the important one.

Every Unix process starts life with three open file descriptors (`fd`, for short): 0, standard input; 1, standard output, and 2, standard

*Most Unix file systems reserve inodes 0 and 1 for bookkeeping. In Ext4, 0 means error—empty, null, does not exist—and 1 holds all the bad blocks the system has come across; assigning them to inode 1 avoids trying them again. See [Ext4 — Special inodes](#).

error*. Normally all three end up at the terminal, but the program neither knows nor cares: it writes to fd 1 and that's that.

And since it doesn't know, the shell can change them for it before running it. That is a redirection:

```
$ ls > listing.txt      # fd 1 to the file, not the screen
$ ls >> listing.txt     # the same, but appending at the end
$ wc -l < listing.txt   # fd 0 from the file, not the keyboard
$ gcc prog.c 2> errors  # fd 2 to the file; 1 stays on screen
```

Note that `>` creates the file if it doesn't exist and empties it if it does. `>>` doesn't empty it. It is a distinction you pay dearly for, once.

An fd can be redirected to another fd instead of to a file, with the notation `n>&m`, which means “let fd *n* go where *m* goes” —the `bash` manual calls it *duplicating* a file descriptor:

```
$ gcc prog.c > out 2>&1 # first 1 to the file, then 2 to where 1
    goes
$ gcc prog.c 2>&1 > out  # NOT the same: look at the order
```

Question 2

The two lines of the previous example differ only in the order of the redirections. Where do the error messages end up in each one?

Answer

In the first, fd 1 goes to the file `out`, and right after that 2 is sent “to where 1 goes”, which is already the file. The errors end up in the file.

*This third fd was not there at the beginning. Thompson and Ritchie invented it the same night they rewrote all of the system's commands to fit pipes in; see Section 1.7.1.

In the second, 2 is sent to where 1 goes *while that is still* the terminal, and only afterwards is 1 switched to the file. The errors stay on the screen. The shell processes redirections from left to right, and each one makes its copy at the moment it is carried out.

Notice what we have just done: we have changed where a program reads from and writes to **without touching it**. `ls` doesn't know it is writing to a file, and `wc` doesn't know it is reading from one.

This idea —that programs talk through numbered file descriptors and that someone else decides where they end up— is what makes the whole of Chapter 10 possible. When McIlroy asked for programs to be coupled together “like garden hoses”, that is exactly what he was asking for: that the output of one could be plugged into the input of another. With redirections already in place, Thompson was able to write the system call in an hour.

3.5 Exercises

1. Run `ls /bin | wc -l` and see how many commands you have at hand. Pick three you don't know and read their manual pages.
2. In the `/dev` listing there is a line beginning with `l` and another beginning with `b`. What are they? (Hint: `man 7 inode`.)
3. Create an empty directory and look at its number of links with `ls -ldi`. Create a subdirectory inside it and look again. Does it match the answer to Question 1?
4. What does `ls > listing.txt` do if `listing.txt` appears in the listing? Try it and explain why the file's size is what it is.
5. Redirect a command's standard error to `/dev/null` and explain, looking at this chapter's `/dev` listing, what exactly that file is.