

Linux terminal

Introdução Engenharia Informática

Mário Antunes

September 22, 2025

Universidade de Aveiro

Welcome to the Command Line!

More Than Just a Black Box

The **Terminal** is your direct, text-based connection to the operating system.

- **Why use it?**

- **Power & Speed:** Execute complex tasks instantly.
- **Automation:** Script repetitive jobs.
- **Efficiency:** Uses minimal system resources.
- **Industry Standard:** Essential for developers and system administrators.

Analogy: A GUI is a restaurant menu. The CLI is speaking directly to the chef.

The Shell & Bash

The **shell** is the program that interprets your commands. The terminal is the window; the shell is the brain inside.

- There are many shells, each with different features:
 - `sh` (Bourne Shell): The original, classic shell.
 - `zsh` (Z Shell): A popular modern shell with extensive customization.
 - `fish` (Friendly Interactive Shell): Focuses on being user-friendly out of the box.
 - **`bash` (Bourne Again SHell)**: The most common shell on Linux. It's the de facto standard we will learn today.

The Linux Filesystem (Part 1: Core Directories)

The filesystem is a tree starting from the **root (/)**.

- `/`: The **root directory**. Everything begins here.
- `/home`: Your personal files are here (e.g., `/home/student`).
- `/bin`: Essential user **binaries** (programs like `ls`).
- `/etc`: System-wide **configuration** files.
- `/var`: **Variable** data, like system logs (`/var/log`).
- `/tmp`: For **temporary** files.

The Linux Filesystem (Part 2: Software & Admin)

More important locations you'll encounter.

- `/opt`: **Optional** software. Used by third-party programs you install manually (e.g., Google Chrome).
- `/usr/local`: A place for software you compile or install for all users that isn't part of the standard OS distribution. You'll often find `/usr/local/bin`.
- `/root`: The home directory for the **superuser** (root user). Do not confuse this with the `/root` directory!

Hidden Files & Directories

In your home directory (~), many configuration files are “hidden” by starting with a dot (.). They control how your programs and shell behave.

- **Examples:**

- `~/.bashrc`: Bash shell configuration script. This is a crucial file.
- `~/.config`: A common directory for application settings.
- `~/.themes` or `~/.local/share/themes`: For desktop themes.
- `~/.gitconfig`: Your Git configuration.

Basic Navigation: pwd and cd

Two fundamental commands for moving around.

- **pwd**: **P**rint **W**orking **D**irectory. Shows your current location.

```
$ pwd  
/home/student
```

- **cd**: **C**hange **D**irectory. Moves you to an absolute or relative path.

```
$ cd /var/log      # Move to an absolute path  
$ cd Documents    # Move to a subdirectory
```

Special Navigation Shortcuts with cd

cd has several useful shortcuts for faster navigation.

- Move up one level:

```
$ cd ..
```

- Go directly to your home directory from anywhere:

```
$ cd ~
```

(Or just cd with no arguments)

- Go back to the last directory you were in:

```
$ cd -
```


Listing Directory Contents: `ls`

The `ls` command **lists** the contents of a directory. It's your eyes in the terminal.

- Use **flags** to change its behavior. The most common is `-l` for a **long** list format.

```
$ ls -l
-rw-r--r-- 1 student student 4096 Sep 19 2025 my_doc.txt
drwxr-xr-x 2 student student 4096 Sep 17 2025 Scripts
```

This shows permissions, owner, size, and modification date.

Seeing Everything with `ls -a`

How do we see those hidden configuration files?

- The `-a` flag tells `ls` to show **a**ll files.

```
$ ls -a
.  ..  .bashrc  .profile  Documents  Downloads
```

- You can combine flags. `ls -la` is a very common command to get a **l**ong list of **a**ll files.

Executing Programs and Editing Files

- **Running a program:** Simply type its name.

```
$ firefox
```

- **Editing a text file:** nano is a simple, beginner-friendly terminal editor.

```
$ nano my_shopping_list.txt
```

(Use `Ctrl+X` to exit, then `Y` to save).

Getting System Information

The terminal is excellent for quickly checking system status.

- `whoami`: Shows your current username.
- `date`: Shows the current date and time.
- `uname -a`: Shows kernel and system info.
- `top`: Shows running processes in real-time (like Task Manager). Press `q` to quit.

Users: Standard vs. Superuser

Linux is a multi-user system.

- **Standard User** (student): Your day-to-day account with limited privileges.
- **Superuser** (root): The administrator. Has complete power over the system.

To run one command with root privileges, use `sudo` (**S**uper**u**ser **d**o).

```
# This needs admin rights, so we use sudo
$ sudo apt update
```

As an administrator, you can manage user accounts from the command line.

- `sudo useradd new_user`: Creates a new user.
- `sudo passwd new_user`: Sets the password for the new user.
- `sudo userdel new_user`: Deletes a user.

Understanding File Permissions

The `ls -l` command shows permissions as a 10-character string like `-rwxr-xr--`.

- **It's read in groups:** Type | Owner | Group | Others
- `r`: Permission to **read** the file.
- `w`: Permission to **write** (modify) the file.
- `x`: Permission to **execute** the file (run as a program).

Managing Permissions with chmod

Use the chmod (**ch**ange **mode**) command to change permissions.

- You can add (+) or remove (-) permissions for the **u**ser, **g**roup, or **o**thers.

Example: Make a script executable for yourself.

```
# Give the user (u) the execute (x) permission  
$ chmod u+x my_script.sh
```


Redirection: Saving Output with >

Don't want to see output on the screen? Save it to a file with >.

Warning: This **overwrites** the file if it already exists.

Example: Save a list of your home directory contents to a file.

```
$ ls -l ~ > my_files.txt
```

Redirection: Appending Output with >>

To **add** output to the end of a file without deleting its contents, use >>.

- This is great for creating log files.

Example: Add a timestamped entry to a log file.

```
$ echo "System rebooted at $(date)" >> system.log
```

The Power of the Pipe |

The **pipe** is one of the most powerful concepts in the shell. It sends the output of one command to be the input of the next.

Think of it as plumbing: Command A -> | -> Command B

Example: Find all `.log` files in a directory.

```
# The output of 'ls' is "piped" to 'grep' to be filtered.  
$ ls /var/log | grep .log
```

Your Environment: Variables

The shell uses variables to store information. By convention, they are in ALL_CAPS.

- \$HOME: Your home directory.
- \$USER: Your username.
- \$PATH: A list of directories where the shell looks for programs.

Example: See the contents of the \$PATH variable.

```
$ echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

Customizing Your Shell: .bashrc

The `~/.bashrc` file is a script that runs every time you open a new terminal. This is the place to personalize your shell.

You can edit it with a text editor:

```
$ nano ~/.bashrc
```

Remember: Changes won't apply until you open a new terminal or run `source ~/.bashrc`.

Customization Example: Aliases

An **alias** is a shortcut or nickname for a longer command. They save you a lot of typing!

- Add this line to your `~/ .bashrc` file:

```
alias ll='ls -a\F'
```

- Now, when you type `ll` in a new terminal, bash will run `ls -a\F` for you.

Introduction to Bash Scripting

A script is simply a text file containing a sequence of commands.

1. The first line **must** be `#!/bin/bash`. This is called a “shebang.”
2. Add your commands.
3. Use `#` for comments to explain your code.
4. Make the file executable with `chmod +x`.

Scripting Example 1: Hello World

This script uses a variable and the echo command. It's the "Hello, World!" of scripting.

File: hello.sh

```
#!/bin/bash
# A simple hello world script

NAME="Student"
echo "Hello, $NAME!"
```

To run it:

```
$chmod +x hello.sh$ ./hello.sh
```


Scripting Example 2: Using if

This script uses an `if` statement to check if a file exists before trying to use it.

File: `check_file.sh`

```
#!/bin/bash
# Checks for the existence of the system log file.

FILENAME="/var/log/syslog"

if [ -f "$FILENAME" ]; then
    echo "$FILENAME exists."
    # We could now do something with the file, e.g.
    # tail -n 5 "$FILENAME"
else
    echo "Warning: $FILENAME not found."
fi
```

Scripting Example 3: Looping Over Files

A for loop lets you perform an action on a list of items, like files.

File: add_prefix.sh

```
#!/bin/bash
# Adds "backup_" prefix to all .txt files.

for file in *.txt
do
    # Check if it's a file before moving it
    if [ -f "$file" ]; then
        mv -- "$file" "backup_$file"
        echo "-> backup_$file"
    fi
done

echo "Batch rename complete."
```

Scripting Example 4: Complex Script

This script combines arguments, if, variables, and a program (tar) to create a useful tool.

File: backup.sh

```
#!/bin/bash
# Backs up specified items into a .tar.gz archive.

# Exit if no arguments are provided.
if [ "$#" -eq 0 ]; then
    echo "Usage: $0 <file1> <dir1> ..."
    exit 1
fi

DEST="$HOME/backups"
TIME=$(date +%Y-%m-%d_%H%M%S)
ARCHIVE="$DEST/$TIME-backup.tar.gz"

mkdir -p "$DEST" # Create backup dir if needed
echo "Creating archive..."

# "$@" holds all command-line arguments.
tar -czf "$ARCHIVE" "$@"

echo "Backup complete: $ARCHIVE"
```

You've now seen the core concepts of the Linux command line:

- **Navigating** the filesystem.
- **Managing** files, permissions, and users.
- **Combining** commands with pipes and redirection.
- **Automating** tasks with shell scripts.

Now, let's apply this knowledge in the practical part of the class.