Linux Terminal –
Tutorial for Beginners

Jonas Weßner

5. April 2023

Inhaltsverzeichnis

1 Introduction

The command line (also called terminal, console or shell) is a tool to control a computer merely by typing in text and receiving text as a response. The disadvantage of this is that it is unintuitive for many users when encountering it for the first time. However, on the other side, it also is more powerful than graphical user interfaces and therefore very commonly used in the context of computer science. One of the advantages is that it is available on every computer. On the contrary, a desktop with a graphical user interface (GUI) often is not available on servers or similar machines, which are targeted on high performance. Another reason is, that for programmers it is much easier to develop programs for the command line compared to GUI applications, because of the extra layer of complexity, which comes with the design of a user interface. Therefore, many applications in the context of computer science only have a command line interface (CLI). Another extremely big advantage of the command line is that it can be used to automate tasks by combining multiple commands into a script, whereas with GUIs there is a human required who clicks buttons.

Every operating system has a terminal. However, we will be focusing on the Linux terminal here, which defaults to `bash`. `bash` is the name of the program which interprets the commands you type in. There are also other terminals, but `bash` is by far the most common one.

You may also ask yourself why computer scientists tend to choose Linux over other operation systems. There are many reasons for this, of which some will be presented in the following.

- It is free to use and open source, which makes users independent from companies.
- It is more performant (i.e. faster) than e.g. Windows, which is one of the reasons pretty much all of the servers, computer clusters and supercomputers run Linux.
- Therefore, a lot of software is targeted on Linux as well.
- It tends to be more secure than other operating systems.
- In Linux the user has full control over system updates, whereas with Windows updates might be forced.

2 Navigation

The shell always has a current position, which can be printed using the `pwd` (**p**rint **w**orking **d**irectory) command. When first opening the shell, the position is the home directory of the current user. On a system there can exist multiple users, which are essentially accounts. Each user usually has his own home directory inside of the directory `/home`. Apart from the normal users, there also exists a user called `root`, who is a system user and has privileged access on the system. For security reasons some data on the system can only be modified by the `root` user.

The file system on a linux computer is organized as a tree. The root of the tree is called `/` and is the upper most directory. The `cd <dest>` (**c**hange **d**irectory) command can be used, to navigate through the directory tree. The argument `<dest>` is the destination file path to navigate to. A file path is generally speaking a position in the directory tree pointing to either a directory or a file, which is formed by the names of directories separated by forward slashes. There are two types of file paths: absolute paths and relative paths. Absolute paths start at the root of the directory tree and are therefore independent from the current position of the shell. An example would be `/home/myuser/example.txt`. Relative paths start either with a word, a dot, or a double dot. The dot in file paths is a replacement for the current working directory i.e. the result of the command `pwd`. Therefore the above file path could also be written as `./myuser/example.txt`, if the current working directory is `/home`. Also, generally `bash` assumes a relative path if the first part of a file path is a word. Therefore, the above path could also be simplified to `myuser/example.txt`. Moreover, because the home directory is an often used part of file paths, the character ~ can be used as a replacement for the users home directory. Therefore, if the current user is `myuser`, the file path can be further simplified to `~/example.txt`. A double dot in a file path is used to navigate one directory upwards. For example, if the current position of the shell is `/home`, then the file path `../bin` would point to `/bin`.

All contents of the current directory can be printed using the command `ls` (**l**i**s**t). The contents of a directory can be files and other directories. The behavior of the `ls` command can be modified by additional options: `ls -1` prints all entries on separate lines, `ls -l` (**l**i**s**t **l**ong) prints additional information for each entry. There are many other options for `ls`, of which some will be discovered in later sections. Many programs accept additional modifying parameters just like `ls` does. It is a convention that those options usually start with a hyphen or a double hyphen.

In order to type commands faster, one can use the tab key to complete parts of the command. This is especially helpful when typing long directory names. If there is no unambiguous completion for the command part, one can press the tab key twice to receive a list of all possibilities to complete the command.

Previously used commands can be reused in order to avoid typing the same command over and over again. In order to do that one can use the up and down arrows to navigate through `bash`s command history. Furthermore, the complete history can be printed with the command `history`.

3 Files and Directories

In this section, we take a closer look at files and directories and how to modify them.

To get a deeper understanding of files and directories in Linux, one can use the `ls -l` command (**l**ist **l**ong), which prints all contents of the current directory including additional information. An example of the output of this command is shown below:

```
-rw-rw-r-- 1 myuser myuser 13980 Sep 19 18:18 big_data.zip
drwxr-xr-x 2 myuser myuser  4096 Sep 28 10:34 docs
-rw-r--r-- 1 myuser myuser     0 Sep 28 10:34 notes.txt
-rwxrwxr-x 1 myuser myuser   853 Sep 19 19:00 program.sh
```

Each entry begins with information about the permissions on the left side. For each entry there are 10 characters, which all have a special meaning. For directory entries the first character is a `d`, which tells us that this entry is a directory, for file entries this is a hyphen. If all permissions are given for a file or directory, the following 9 characters will be three blocks of `rwx`, which results in `rwxrwxrwx`. These have the following meaning:

- `r` is the permission to read the file, or for directories to look at its contents.
- `w` is the permission to write, which means for files to modify them and for directories to create, delete or rename entries inside of the directory.
- `x` is the permission to execute. For files this means to run them as a program. For directories this means to navigate into the directory i.e. to open it.

If any permission is not set, instead of the respective character, a hyphen (-) will appear. The permissions exist three times, because they are specific to the role the user is assigned to. Each file or directory belongs to a user and a group. All files shown in the example belong to the user called `myuser` and a group with the same name. Initially a file or directory belongs to the user who created it and to the group associated with the user. The first bundle of `rwx` then represents the permissions for the owning user of the entry, the second one represents the permissions for owning group and the third one the permissions for everyone else. For example, the file `big_data.zip` can be modified and read by the user `myuser` or anyone in the group called `myuser`. All other users can only read the file. Nobody can execute the file as a program. After the owning users and groups, the next information emitted by `ls -l` is the size of the file or directory in bytes. On the right side of that one can see the last modification date of the entry.

File and directory names in Linux and any other operation system are a way of uniquely identifying files and directories inside of their parent directory. Therefore, two files or directories with the same name in the same directory are not permitted. An interesting fact is, that the file extension has no effect on the contents of the file at all. It merely is a conventional hint on the format which the file is written in. E.g. plain text files often have the extension `.txt` and pdf files the extension `.pdf`. If the file extension does not match the actual format of the file, programs working with the file might handle the files the wrong way. E.g. a PDF-viewer might not want to open a file with the extension `.zip` even if the actual content of the file is in pdf format. This allows programs to make an educated guess about the format of the file without having to read and analyze its contents.

Files whose names start with a dot are hidden files, which are not shown to the user by default. To show hidden files with the `ls` command, the option `-A` (**a**ll) can be used.

The content of files can be looked at using the `cat` command, which reads the file and prints everything read to the console.

To create a new empty file, the command `touch <filename>` can be used. To create a new directory, the command `mkdir <dirname>` (**m**a**k**e**dir**ectory) can be used.

To remove a file, the command `rm <filename>` (**rm**ove) can be used. In order to remove a directory the option `-r` (**r**ecursive) must be given like so: `rm -r <dirname>`.

To copy a file from one location to another, the command `cp <source> <dest>` (**c**o**p**y) can be used, where `<source>` and `<dest>` are both file paths. If the destination file path is an existing directory,

the file will be copied into that directory and maintain its name. If the destination file path ends with a non-existing file name, the file is copied and the copy will receive this new name. To copy whole directories, once again the option `-r` must be specified: `cp -r <dir> <dest>`.

In order to move a file from one location to another, the command `mv <source> <dest>` (**m**o**v**e) can be used. Similar to the `cp` command, if the destination is an existing directory, the source will be moved into that directory and maintain its name. If the destination ends with a non-existing name, the source will be moved to that location and receive the given name. Therefore, the move command can also be used for renaming files like so: `mv <old_name> <new_name>`.

In order to address multiple files or directories whose names share a common pattern, wildcards can be helpful. The wildcard symbol is a star (*) and matches any combination of characters. E.g. `rm -r *` would remove all files and directories in the current directory and `rm -r useless_*` would remove all files and directories with the prefix `useless_`. It is important to mention that wildcards, for security reasons, do not match hidden files.

In order to change the earlier mentioned permissions of files, the command `chmod <opt> <filename>` can be used. It expects an option `<opt>` which specifies how to change the file permissions. The option `+x` adds execute permissions for everyone and the option `-x` removes them. The option `u+x` adds execute permissions only for the owning user. Other combinations `<role>+/-<permission>` work in the same manner.

4 Editing Files Using Vim

Vim is a command line text editor installed on most versions of Linux.

In order to open a file using Vim, the command `vim <filename>` can be used. After opening, Vim is in the command mode, which means, it accepts commands to change to a more specific mode. If one has accidentally entered some other mode, one can press the ESC key, to come back to the command mode.

The second most important mode is the insert mode. To change into the insert mode, one can press the `i` key. In the insert mode one can modify the file just like in any other text editor. To leave the insert mode and come back to the command mode, one must press the ESC key.

In order to leave Vim, one can type `:q` (**q**uit). If there are unsaved changes made to the file, one can either use `:q!` to exit the editor and discard the changes or `:wq` (**w**rite and **q**uit) to save the changes and then leave the program.

If one would like to delete a whole line, he can press the `d` key twice while being in the command mode.

Also, many other features are available in Vim. In fact, it can also be used as an integrated development environment (IDE) to write code if set up properly. However, in this document we leave it for just the basics shown above.

5 Streams and Redirection

Every program has 3 streams of information: the standard input stream `stdin`, the standard output stream `stdout` and the standard error stream `stderr`. Streams are a common way of a program to communicate with other programs or interact with the user. If required, programs read from `stdin` to receive input from the user. If a program wants to emit some output, it writes that to `stdout`. Error messages are written to `stderr`.

The streams can be connected to other programs. For example, `stdout` and `stderr` of each program running in the terminal are by default connected to the shell, which then displays their content to the user. E.g. `ls` writes its output to its standard output stream, which the terminal then prints to the computers display. Also `stdin` of any program running in the terminal is by default connected to the computers keyboard. Therefore, when a program expects input, it can read whatever the user types in.

Sometimes it can be useful to redirect the stream of a program to somewhere else than the terminal. Using the greater-than-symbol (`>`), the standard output stream of a command can be written to a file. For instance, to save the names of the current directories contents to a file, one could use the command `ls > result.txt`. In order to not overwrite the contents of a file, but instead append to the file, two greater-than-symbols (`>>`) must be used. Similarly, the less-than-symbol (`<`) can be used to connect the content of a file to the standard input stream of a program. As an example, The `egrep` program reads from its standard input stream and searches for lines containing a given text. The command `egrep "abc" < file.txt` would connect the contents of the file to the input stream of the `egrep` program and therefore result in searching for the text ”abc” in the file called `file.txt`. Furthermore, it is possible to connect the standard output stream of a program to the standard input stream of another program. This is called piping and requires the use of the symbol `|`. For instance, to search for zip-files in the current directory, one could use the command `ls | grep ".zip"`. This command pipes the standard output of the `ls` command to the input stream of the `grep` command, which then searches for the given pattern in its input stream.

6 Subshells

The term subshell describes the execution of a command in another shell inside of the current shell. This can be useful, if the result of one command is required as the argument to another command. A subshell is created by a dollar sign followed by a pair of brackets encapsulating the command to be executed in the subshell like so: `$(subcommand)`. The shells are interpreted from inside out starting from the most inner subshell. After each execution of a subshell, its code is replaced by the content of the standard output stream of that subcommand and is then subject to execution of the next higher order shell. As a very simple example, one could use the command `echo "Today is $(date)"` to print a text which says what todays date is. First, the subshell will be executed, which results in the current date and time. Then the outer command will be executed, which takes the text as an argument, which contains the current date. Another example would be the command `mkdir $(pwd)/new_dir`, which creates a directory called `new_dir` inside of the home directory of the user who executes the command. Of course, this could also be achieved by using the tilde symbol (~) as a replacement of the users home directory.

Advanced users can use subshells to create powerful functionalities, especially as a part of scripting. These are not shown in this document, but are based on the same principles shown above.

7 Process Management

A process is an instance of an executing program. For example `firefox` or `ls` are programs. Although `firefox` is only installed once on a computer, one can open multiple windows of `firefox`, which are independent running instances of the program – independent processes.

The terminal allows the user to execute programs as foreground or as background processes. All commands shown so far have been executed as foreground processes. Foreground processes block the shell they are running in as long as the foreground process runs. For the commands shown so far that is not an issue, as they complete in a couple of milliseconds. However, there are tasks e.g. installing programs, which may take a lot longer. In case a user wants to install five programs, of which each takes three minutes to install, he can start all installations as background processes, which means they all run in parallel and do not block the current shell. That way he can continue using the current shell for other things in the meantime.

In order to execute a command as a background process, an ampersand has to be appended to the command. For example the command `sleep 60 &` executes the `sleep 60` command, which lets the current shell do nothing for 60 seconds, in the background. After a background process has been started, its status can be checked using the `jobs` command, which shows all running or stopped background processes. In order to pull a process from the background back into the foreground, the command `fg <id>` (**f**ore**g**round) can be used. Running foreground processes can be stopped (not canceled) by pressing the Ctrl+Z. The `jobs` command then shows them as stopped processes. In order to resume a process in the foreground, the `fg <id>` command can be used as well. In order to resume it in the background, the `bg <id>` (**b**ack**g**round) command can be used. To cancel a foreground process completely, one can press Ctrl+C.

8 Installing New Programs With apt

Many programs are preinstalled on Linux. Examples are `ls`, `pwd`, `egrep`, `date`, `find` and many more. The exact set of preinstalled programs varies depending on the Linux distribution. However, it is for sure that at some point the user wants to install new programs. In Linux this is often done by using package managers. The default package managers for Ubuntu (the most common Linux distribution) are `apt` and `apt get`, which only have minor differences.

The `apt` package manager maintains a list of current available packages (i.e. programs) and a list of currently installed packages.

- The command `apt update` updates this list of available packages to the latest versions of the packages found online.
- `apt upgrade` updates all already installed packages to their newest available version.
- `apt install <package>` searches for a package with a given name in the list of available packages and tries to install it.
- `apt remove <package>` uninstalls a package with a given name.

It is always recommended to run `apt update` before running `apt upgrade` or `apt install` in order to upgrade or install the newest version of a package available instead of a deprecated one.

Installing, removing and updating programs is a task that requires administrator privileges. Therefore, all apt commands must be executed as a super user e.g. `sudo apt update`.

A big advantage of package managers over installation wizards is the possibility of automation. Users might want to install many programs in a very specific way e.g. for their work. Instead of the company requiring each employee to follow long and tedious installation guides and clicking through the wizards, they can provide a script, which installs all programs automatically.

Not all programs are always available through package managers. Less popular software must often be downloaded from a website and then installed by running a script.

9 Automation Using Scripts

One of the major advantages of using the terminal is the power of scripting. A script is basically a computer program written in many commands, which are executed just in time. In the easiest case, a script is just a few commands written to a file and executed one after another. However, `bash` actually provides a whole programming language enabling users to create scripts that can achieve pretty much everything. Scripts are a nice way of automating tedious and repetitive tasks.

The following shows a script, which installs two programs and afterwards emits a message saying it is finished:

```
#!/bin/bash

sudo apt update
sudo apt upgrade
sudo apt install -y unzip
sudo apt install -y firefox
echo "Script finished"
```

Each `bash`-script should start with the line `#!/bin/bash`, which is also called shebang. This line tells the computer which program shall be used to interpret the scripts contents. In the case of `bash`-scripts that is the program `bash`, which is located in the directory `/bin` of the system.

In order to execute a script, it must be executable by the current user (see section 3). Then the user can start a script by entering the path to the script in the terminal. Note that if the script is located in the current directory, its path must be specified with an explicit dot at the beginning like `./<script>`, because otherwise the shell only searches for programs installed in certain system directories.

When starting a script, the calling shell will then look at the shebang and start a new instance of the given interpreter (in this case `bash`) and execute the contents of the script in it.

Advanced users can use `bash`-scripts to write programs like in any other programming language. For this use case `bash` also supports if-statements, loops and other flow control statements, which one might already be familiar with.

10 Exercises

There is a docker image with interesting exercises for the sections 2 to 9. Docker is essentially a way to run lightweight virtual machines. This makes it possible to practice `bash` and Linux independent from your computers host operating system.

First you need to install Docker from their official website https://docs.docker.com/engine/install/

Afterwards you can start right away by entering the following command in your operating systems terminal:

```
docker run --name ubuntu-playground-c -it jonaswessner/ubuntu-
    playground:latest
```

This will first pull the prepared image from the internet and then start a shell session inside of a container running the image. If you execute `ls` you will find one directory for each exercise. You should also check out the `README` file located there.

To exit the shell, you can execute the command `exit`. If you want to resume the container afterwards and continue working on the exercises, you can use the following command:

```
docker start -i ubuntu-playground-c
```

If you want to completely reset the container, you can use the following command. However in the home directory of the image, there is also a script, which can reset single directories for you, in case you just want to reset one exercise.

```
docker container rm ubuntu-playground-c
```