By Bartolomeu Rodrigues
https://www.linkedin.com/in/bartmr/

Introduction to the Operative System

The operative system is the interface between the user and their apps, and the hardware of their computer.

There are two main operative system architectures: Unix and Windows NT kernel.

Unix is the architecture used in Mac OS and Linux. Windows NT kernel is used in Windows. Most software development and data science tooling is designed for Unix-based operative systems, so that is what we will cover.

You can simulate a Unix-based environment on Windows by installing Windows Subsystem for Linux (WSL) or Git Bash.

Users and groups

Long before everyone had a computer just for them, computers used to be shared across many people. A single computer would be used in an entire company or university.

Very soon, users started to understand that they needed their own private spaces in a machine that was shared by everyone, much like a company has a cubicle for every employee, where their personal belongings are safely stored, sometimes behind a locked cabinet.

Each operative system **user** has their own files and can run their own software with their own settings, as long as their software does not affect the other users. A user can give granular permissions to its files, and allow another users to read or edit their files, much like you do with Google Drive or Microsoft Office, where you can allow other people to read, edit or review your own files.

Sometimes a company had many users, and it was cumbersome to list all people every time we needed to change permissions. Imagine if you had written some employee reports in your huge and widely-shared computer. You now need to give permissions to the Human Resources team to allow them to read your reports. It would be a pain to change the file's permissions for every member of the Human Resources team. So, operative systems allow you to group many users into a **group**, in this case, you would change the permissions once in your file to allow the whole Human Resources group to access your file.

Today, most people have their own personal computer to themselves, but the model lives on, because it is still useful to isolate parts of your computer into individual users and because the users and groups model is so ingrained in our operative systems, it would just be a lot of trouble to roll it back.

Today, companies use the operative systems users and groups model to limit what their employees can do in their computers (like stopping them from installing potentially malicious software that might compromise the employee and their colleagues). Some special software that simulates a computer inside a computer uses the users and groups model to avoid simulated software from "spilling out" of the simulation black box. The Chrome browser, for example, uses disposable users to stop web pages from directly snooping around your computer.

Files

Everything that is stored in your computer is a file.

Files are grouped by and stored in directories. These directories can contain files or more directories under them. You can picture this structure as roots of a tree. It's no coincidence that we call the **root directory** to the place in your disk that stores the first directories and files.

Paths

Files have a place to be stored. They are organized. To point to a file, we use **paths**.

Paths are the coordinates that point to the place where the file is.

Each directory is delimited with `/`, and it goes in a downward direction, the same way coordinates are delimited by `,`, from the biggest structure to the smallest structure.

Absolute paths

Paths can start from the root directory of your computer (called **absolute paths**)

Here is how the computer will interpret the following absolute path:
`/home/myuser/Documents/2024/presentation.pptx`

1. from the root of the computer
2. Go to `home`
3. Go to `myuser`
4. Go to `Documents`
5. Go to `2024`
6. Here is the file `presentation.pptx`

Relative paths

You can also specify paths that start from the directory we are currently in (called the **current working directory**). Paths that start from the current working directory are called **relative paths** because they are relative to some place

Here is how the computer will interpret the following relative path
`Documents/2024/presentation.pptx` :

1. from the directory we are currently in (called the **current working directory**)
2. Go to `Documents`
3. Go to `2024`
4. Here is the file `presentation.pptx`

Go up with `..`

If you want to reference a directory one level up from where you are currently in, you can use `..` as a directory name. Each `..` signals the computer to go to the upper directory.

Here is how the computer would interpret the relative path `../../Documents/2024/presentation.pptx`

5. from the directory we are currently in (called the **current working directory)
6. Go to the upper level folder
7. Go one level up again
8. Start going downwards from here
9. Go to `Documents`
10. Go to `2024`
11. Here is the file `presentation.pptx`

Stay with `.`

You can also point to a file starting from the directory you are in by using a single `.`.

`./Documents/2024/presentation.pptx` is the same as `Documents/2024/presentation.pptx`

Hidden files

Files starting with `.` in their name, like `.env` are considered hidden, meaning they won't show in most apps unless you explicitly tell the app to do so.

One example is when calling `ls` in the terminal to see the files in the current working directory. Unless you explicitly call `ls -a`, the hidden files will not be shown.

Permissions

Files contain some small extra fields that determine which users and groups in your computer can access them.

Usually, files created inside the home directory of your user belong to you and you have full-access to do whatever you want with them.

Files created by other users will be out of your reach, unless the original user changes the files permissions with the `chmod` command, which is used to change the file permissions,

much like when you share a file on Google Drive or Microsoft Office, and decide who gets to read it or change it.

Files created by the root user are generally available for you to read, since the root user creates system files, and these system files are read and run by all users.

Processes

Processes are everything that is currently running in your computer. If a file were a cog, a process would be the cog spinning.

A process is the running result of an executable file being loaded and its instructions being run.

Open your computer's Task Manager (the famous Ctrl + Alt + Delete) and you will see all processes currently running in your computer: the ones that you see on the screen like your browser or messaging app, and the ones that are hidden, doing things like controlling the CPU, watching your mouse for movements, updating your software, scanning for viruses, and even drawing the current Task Manager window that you see.

That's right: what you see on your screen right now is drawn by many processes that are hidden to you. We will talk more about that in the **shell** chapter.

When you turn on your computer, a whole chain of running processes start. Thousands of files are read and loaded into the computer's RAM as processes.

Processes can open more processes under them. We call them **child processes**. These **child processes** can open more child processes of their own and so on. You can see this with the Google Chrome browser. Every time you open a new tab, a new Chrome process for tab is created. As we saw with files, processes themselves are organized as a tree too.

Your whole computer being turned on right now can be seen as a tree of processes and child processes, just like the way files are organized.

The life cycle of a process is this:

- a process is started by another process
- the process runs
- a process exits:
    - after it finishes running all instructions
    - it encounters an unexpected scenario so fatal that it can't continue
    - receives a signal from the shell to stop and exit
- a numeric **exit code** is issued by the process. A zero exit code means the process finished successfully, a number other than 0 means the process failed

While a process runs, it sends and receives feedback. A process might want to show the user important events, or return some data. A process might also await input from the user or

another process. It's a two way street.

Processes send output messages and data to the `stdout` stream. They can also send warnings and error messages to the `stderr` stream. If you are running commands in the terminal, these streams will be shown on your screen.

Processes also listen for input from the keyboard, like a password, or other commands, by listening to the `stdin` stream.

Shell

The shell is the interface that allows you and other processes to interact with your computer and what is in it. It's the waiter at the restaurant that gets your order, and the orders of other processes.

When you turn on your computer, your operative starts a long chain of processes from the shell, and those processes will do everything from showing the graphical interface on the screen, from managing your audio, to keeping your hardware in check, so that you can use the computer.

The app that allows you to interact with the shell is commonly known as the **terminal** or the **command-line**.

Before computers had graphical interfaces with windows like today, the only way to use a computer was to interact with its shell through the terminal.

As technology evolved, and chips got faster, so did new ways to make computers more accessible to every-day people. One day, computers were powerful enough to draw windows and buttons on a screen, instead of forcing users to type every instruction.

Today, graphical interfaces like Microsoft Office or Chrome or the windows in our screen exist for the common end-users. But building graphical interfaces is still very expensive time-wise for developers. It is also less flexible if you want to connect and chain instructions and data from different computer programs. It is easier to build software that just accepts instructions as simple text commands. You will discover that a lot of the software available out there is used in the terminal, and does not have a graphical interface.

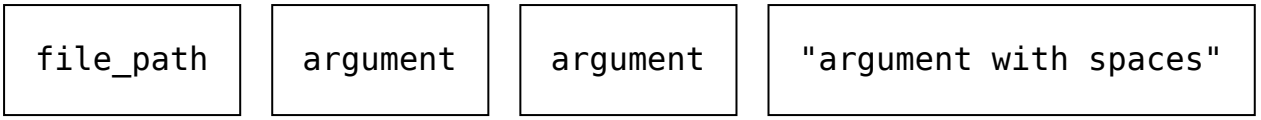
The shell accepts commands, composed of **arguments**, runs them as processes, and these processes send their feedback to the `stdout` and `stderr` streams, which are shown on your screen.

Everything running in the computer is a process, including the running shells themselves. A shell can load a file or another shell as a child process.

In this walk-through, we will be using the Bash shell. Bash is the most widespread shell you will find in many operative systems. Windows comes with a different shell (called the **cmd**, or more recently, the **PowerShell**), but you can use the bash shell too by installing **Git Bash** or **Windows Subsystem for Linux**.

Writing commands

- everything is an **argument**. arguments are the most basic blocks of interacting with the shell
- arguments are delimited by white space. If you need an argument with white spaces, you group it with quotes



a basic command consists of calling an executable file (by writing it's file path) and passing it some arguments specifying how we want to use it.

Running a command

To run something in the shell, you need to point the shell to the file you want to run it, plus any argument that the file is expecting to receive. This will load the file in a new process.

Most operative systems come with built-in files that do many things.

We will start by using the `echo` built-in file, which prints the arguments we give to it to the `stdout`, where we will see them be printed.

We write down its file path, and then it's arguments.

```
/bin/echo "Hello World!"
```

Notice how we use quotes to define an argument with white spaces.

When you run a command in the shell, it will monitor the `stdout` and `stderr` process streams and print them on the terminal you are using to interact with the shell.

Environment Variables

We can store text in an environment variable. An environment variable is like a box where you can put text. Environment variables are made available to the commands you run in the shell, and all child processes that are created from that command, and so on.

Environment variables are used to pass configuration values to child processes when they are started.

Environment variables. can also impact the behavior of the shell you are running, as we will see later.

Environment variables disappear as soon as the shell they were created on is closed.

To create or change an environment variable, just run:

```
export OUR_TEXT="Hello World!"
```

The `OUR_TEXT` environment variable will now be accessible to any command you run from now on, and its child processes.

You can also use it in arguments, like the following example:

```
/bin/echo "$OUR_TEXT 1"
/bin/echo "$OUR_TEXT 2"
```

`PATH` environment variable

As you probably noticed, a lot of the commands that we run in the shell don't need to have their full file path written in order to be executed.

You can simply call `echo` and the shell knows where it is placed.

This is because the shell reads from an environment variable called `PATH` that tells the shell all the directories it should look for the file you called, before trying to find the file in the directory you specified. This is an example of how an environment variable can affect the behavior of a process, and the shell itself is a process.

Let's see what directories the shell looks up for when trying to find the files you try to run:

```
echo "$PATH"
```

Why does `PATH` exists? To make it easier for software to call common commands in many different operative systems with their own configuration. It hides the complexity (the technical word is **abstracts**) and differences of where files are placed from the software that needs to use them.

Example: When you install Python, the Python executable file can be placed in many different locations depending on your operative system and your choices. Operative systems usually have their install locations in the `PATH` so that other software in your computer, like VSCode, can just call `python` and quickly start using it, without needing to know the exact directory where it is placed. This is how different software interacts with each other magically.

Current working directory

The shell is always pointing to a directory, called the current working directory or `cwd`. This is the directory that your commands will start from and point to.

You can see the current working directory of your shell by either:

- looking at the path that precedes your commands

- by running the `pwd` command

By running

```
ls
```

you can see the files placed in the current working directory.

You can change the current working directory by running

```
cd path-of-directory-i-want-to-be-in
```

Creating our first computer program

A computer program is an executable file that will be called in the shell, loaded as a process and run instructions. Programs can be written and packaged in many programming languages and forms, but today we will write a program in the **bash shell scripting language**.

The shell scripting language is just a programming language that allows you to run the commands you usually type with the keyboard, but in an automated, top to bottom order. Like a series of steps for a recipe.

Let's start by creating a simple text file by calling

```
touch my-program.sh
```

let's open our file in a text editor (like VSCode) and add:

```
echo "My first program"
```

Executable files

One special permission is used to determine if your file contains data or it's a file with software ready to run.

Files containing this permission are **executable files**, meaning you can load their contents into a **process** and run them.

To classify our new file as an **executable** that is ready to run, run the following command in your terminal:

```
chmod +x my-program.sh
```

You can now call this file in the terminal and its contents will run in your computer.

```
./my-program.sh
```

Understanding errors

Software can fail. As we talked about earlier, when processes end, they signal to the shell if they ended successfully, or with an error, by returning a numerical code:

- 0 means success
- anything else means an error happened.
    - Processes might also print a message about the error in `stderr`, but that is a choice from the software developer.

It's important for processes to signal if they ended successfully or not, so that our software knows what to do when the unexpected happens, and change its course of action or stop when an error happens.

Bash stores the last command's exit code in a variable called `?`

Try to run:

```
cd this-directory-does-not-exist
```

and then print `?` by running:

```
echo "$?"
```

You can confirm that `cd` failed to change to a directory that does not exist by returning an error code that is not 0.

We will see how we can use exit codes up next.

Conditions

Conditions are a common concept in programming. It's basically *if this happens, do this; else, do that*.

To write a condition, we need **if** statements and a command, that depending on its exit code, will signal what to do next.

The **if** statement will look at the command that defines the condition, and if it returns a 0 exit code, it will run its code.

A popular command to define comparisons as conditions is the `test` command. `test` is also a built-in command that is registered in the `PATH` environment variable, so there is no need to type its full file path

We call the `test` command and we pass the comparison we want to do as **arguments**. `test` will then return 0 if the condition is true...

```
export BASKET="oranges"

if test "$BASKET" = "oranges"
then
    echo "basket is oranges"
fi
```

... and other exit code if its false, and the code inside `if` will not run.

```
export BASKET="apples"

if test "$BASKET" = "oranges"
then
    echo "basket is oranges"
fi
```

You can also decide to run the code inside `if` if you expect the command to fail and return a non-zero exit code. You do this by **inverting** the condition with a `!` sign, meaning that you want the `if` to run if the condition fails:

```
export BASKET="apples"

if ! test "$BASKET" = "oranges"
then
    echo "basket IS NOT oranges"
fi
```

There might be cases where you want to run code the command returns a zero exit code, or also when the command fails and returns a non-zero exit code. You can do that by using **else**:

```
export BASKET="apples"

if test "$BASKET" = "oranges"
then
    echo "basket is oranges"
else
    echo "basket IS NOT oranges"
fi
```

Using environment variables

Let's try an example using environment variables.

Let's change our script to the following

```
if test "$MY_NUMBER" > 3
then
    echo "my number is bigger than 3"
else
    echo "my number is smaller than 3"
fi
```

and let's run it in the terminal like this:

- define the `MY_NUMBER` environment variable by running `export MY_NUMBER=4`
- run your script with `bash my-program.sh`

Short-hand conditions

More often that not, the bash scripts you see in the real world are actually short. In order to keep them short, bash allows you to write shorter versions of conditions, to chain commands one after the other. Here is are some examples:

`&&`

`&&` means *and*. The following expression means: if `command_1` exits successfully (exit code is 0), run `command_2`

```
command_1 && command_2
```

our basket example would be

```
export BASKET="oranges"

test "$BASKET" = "oranges" && echo "basket is oranges"
```

`||`

`||` means *or*. The following expression means: if `command_1` fails (exit code is not 0), run `command_2`

```
command_1 || command_2
```

our basket example would be

```
export BASKET="apples"
```

```
test "$BASKET" = "oranges" || echo "basket IS NOT oranges"
```

Subshells

Wrapping your commands with `()` will run them in a new shell under a child process, also called a **subshell**

subshells are useful when you want to quickly run a command in a different working directory without affecting the rest of the script. Example:

```
# The HOME environment variable contains your user directory path
cd "$HOME"

# Everything inside () will run in the Documents directory
(
    cd Documents
    && pwd
)

# Outside the (), the shell is still running in the initial working
dictory
pwd
```

Setting environment variables with subshells

You can quickly set environment variables with subshells. When subshells are prefixed with `$`, the shell will grab and store the processes outputs coming from its `stdout` stream

```
export THIS_DIRECTORY_PATH=$(pwd)
```

Using subshells to store the process output is very useful to grab a process's data, and inject it into another command. A common use case is to get sensible passwords and keys, and inject them into a command that needs them. Here is a loose example:

```
export MY_SCRIPT_SECRET_KEY=$(my_password_manager get "secret-key")

# this command will have MY_SCRIPT_SECRET_KEY available in it
my-script
```

Other operators

There are many more operators in the bash shell that allow you to do useful things, like writing a command's output directly to a file, or setting how bash runs your scripts (a common example is `set -e`, which will make bash stop if any command in your script exits

with a non-zero code). You will them in examples online and you will learn as you use them for the first time.

Other common operators are:

- `for` which allows you to iterate a list of arguments and run instructions for each one of them
- `>` which allows you to take the `stdout` output of a command and write it into a file instead of showing it in the terminal
- `|` (called the pipe) which allows you to take the output of a command and pipe it into the next command's `stdin` input, like it was just typed by you with your keyboard
    - the pipe is commonly used to install software. The first command pulls the install commands from a website and pipes it to a second command that usually runs those downloaded install commands.

There are also operators that make it shorter to write commands. In software development, we call those operators **syntactic sugar**, because they allow for the same logic to have a smaller syntax.

An example of syntactic sugar happens with the `test` command. You can write a `test` condition as `if test "$BASKET" = "oranges"`, but also as `if [[ "$BASKET" = "oranges" ]]`, as you will see in many examples online.

Finally, let's not forget that computers can be expanded with more software, software that also runs in the terminal. There is software to browse the web and pipe data into other commands, read and convert excel spreadsheets, and even control the lights in your house, using the shell. There is an endless world of software designed to run in the terminal that I recommend exploring. They can be chained and piped together like Legos, allowing you to automate your life.

A lot of the software that runs in the terminal is installed using a **package manager**. In software development, a package is a piece of software.

The most popular package manager for Mac OS is **Homebrew**

Linux-based operative systems (including Windows Subsystem for Linux) usually come with the `apt` package manager pre-installed. Here is an example of how we can install `curl`, a tool to download files, using apt

```
sudo apt install curl
```

`sudo`

You may have noticed that the `apt` command is prefixed with `sudo`. `sudo` is short for *super user do*.

Operative systems have users, and these users are isolated from each other for security reasons. Decades before the invention of the personal computer, a single computer was not-personal, and was used by many people. This required the creation of boundaries for each user.

When you install software or do system changes, those changes affect all users. This means that only a **super user** with superior privileges, usually the `root` user that comes with the operative system, can make changes that affect all users in the computer.

If the computer you are using is personal and only yours, you can run commands with superior privileges by just invoking `sudo` and writing your password. This will take the command and run it as the `root` **super user**.

If your computer is not totally yours and is shared or connected to your organization's network, you will probably not be able to use `sudo`, as the IT help desk in your organization worries that you might accidentally give up control of the computer to viruses by running commands as the **super user**.

You should always be careful about the commands you install and run, **specially if they require super user privileges**. **super users** can do anything to your computer, including breaking it, change dangerous configurations like CPU cooling, or logging what you write on your keyboard and even recording your webcam and your screen!

With great power comes great responsibility.

History

- AT&T Archives: The UNIX Operating System
  - https://www.youtube.com/watch?v=tc4ROCJYbm0