Linux/Bash

Introduction to the Linux command line

Who am I?

Nicolas Mattelaer

- Member of Infogroep
- Second master student (SOFT)
- Linux user since 2017
- Email: nmattela@infogroep.be
- Discord: discord.infogroep.be

Slides: Adapted slides from Bram Vandenbogaerde and Robin Vanderstraeten

Practical

- Slides are available at seminars.infogroep.be
- You can try the commands on your computer while I am explaining them
- Please interrupt when you have a question, don't wait to ask it

Some history

Some history

Linus Torvalds

Creator of Linux, a UNIX-like operating system developed for the GNU project

Some history

Linux turned 30 years!

Some history

Linux turned 30 years!

"I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones."

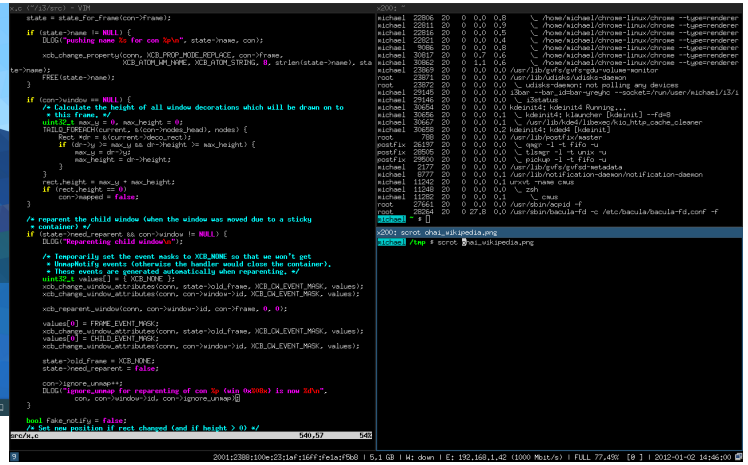
Why Linux?

- Much more **freedom**
- For computer science students
  - lots of tools are made with a Unix mindset
  - for the course "Operating Systems" and "Structure of Computer Programs 2" you will need Linux for certain exercises
  - when developing in C the GNU compiler collection (GCC) works the best on Linux
  - some libraries are only available on Linux

Difference between distributions

Gnome

Kde



i3

Different looks

Difference between distributions

- Different looks
  - Gnome (default in Ubuntu, Fedora, ...)
  - KDE (default in KDE Neon)
  - Xfce (default in Xubuntu)
  - i3
  - ...
- Different ways to install new software
  - Ubuntu: apt-get/snap
  - Arch Linux: pacman
  - Fedora: dnf
- Different default software

Difference between distributions

Linux is very versatile

- You can change how it looks
- You can change how it works
- And it is easier than you may think (if you know the tools)

What people think Linux is:

How Linux actually works:

```
sudo pacman -S firefox
```

Bash

Using the command line

What is Bash?

Definition:

> Bash is a command processor that typically runs in a text window, where the user types commands that cause actions. Bash can also read and execute commands from a file, called a script. (Wikipedia)

Running a Bash shell

Open a "Terminal"

You will be greeted with a **command line prompt**:

nico@tine:~$

- nico: the username of the current user
- tine: the name (hostname) of the computer
- ~: the current working directory

After the `$` sign you can start typing your commands.
We will often omit the username, host and current path in future examples.

Structure of a command

A "command" is just a program or Bash function, that optionally accepts arguments.

Example:

```
$ firefox https://google.com
```

- `firefox` is the program you want to run
- `https://google.com` is the first and only argument of the command

In general:

```
$ program-name argument1 argument2 ...
```

Structure of a command

Commands can also have options. They are passed as arguments but with a `-` sign before them.

Example to download a video from Youtube:

```
$ youtube-dl -o song https://www.youtube.com/watch?v=dQw4w9WgXcQ
```

or

```
$ youtube-dl --output song https://www.youtube.com/watch?v=dQw4w9WgXcQ
www
```

Manuals

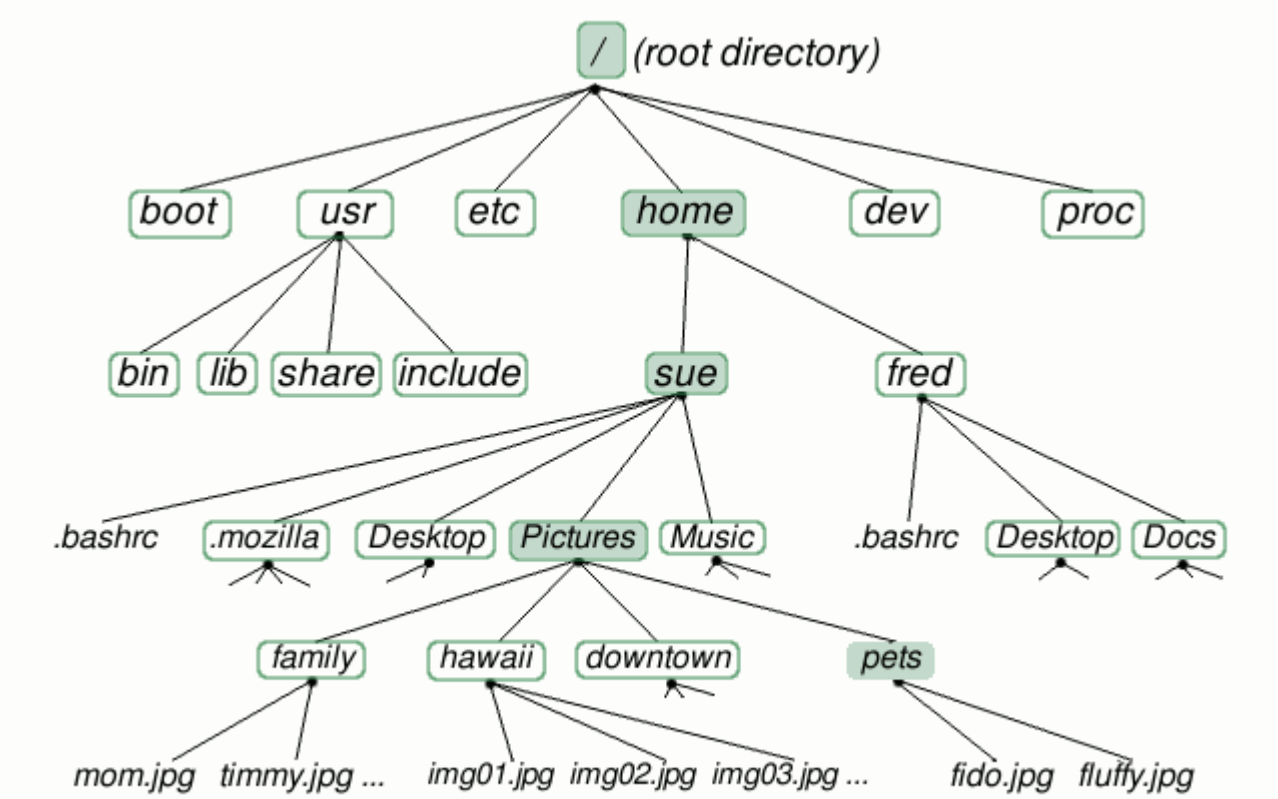
Commands usually have very rich interfaces. It is impossible to memorize every command and its arguments.

Linux has a manual for every command. They can be accessed using the `man` command.

```
$ man man
```

Linux File System

The Linux filesystems can be represented as a tree of directories.



Linux File System

- No drives (no such things as a C:, D:, ... drive)
- Everything lives under the same root
- Other partitions, disks and filesystems can be attached anywhere on the tree
- Some files in the tree can point to other files in the tree (shortcuts or links)

Working directory

- Each program on Linux has a current working directory
- When a terminal is launched its current working directory typically is `~`
- Paths: a path points to an element of the tree
  - absolute paths: the real path from the root of the tree

    ```
    /home/nico/Documents
    ```

  - relative paths: paths relative to the current working directory

    ```
    Documents
    ```

    if the current working directory is `/home/nico` this then points to `/home/nico/Documents`

Working directory

In order to display the current working directory you can use the `pwd` command:

```
nico@tine:/home/nico/Documents/$ pwd [ENTER]
/home/nico/Documents
```

Working directory

The current working directory can be changed using the `cd` command.
`cd` is short for **change directory**.

Example:

```
nico@tine:/home/nico/$ cd Documents [ENTER]
nico@tine:/home/nico/Documents$
```

You could do the same with absolute paths:

```
nico@tine:/home/nico$ cd /home/nico/Documents [ENTER]
nico@tine:/home/nico/Documents$
```

Listing the contents of a directory (1)

You can also show what is inside the directory.

```
$ ls
proposal.tex  infogroep  summaries
```

Default sorting is alphabetically.

- -t: sort by modification time
- -l: create a detailed list
- -r: reverse the sorting
- -u: do not sort
- -a: show hidden files

Listing the contents of a directory (2)

```
$ ls -lt
-rw-r--r-- 1 nico nico    0 okt  6 12:25 info.docx
drwxr-xr-x 2 nico nico 4096 okt  6 12:25 infogroep
drwxr-xr-x 2 nico nico 4096 okt  6 12:25 samenvattingen
```

How to read:

- The first column lists the permissions of the files and directories
- The second column specifies the number of elements in a directory or links to a file
- The third and fourth column specify the user and group that owns the element
- The fourth field specifies the size of the element
- The fifth field specifies the last modification date
- The last file specifies the name of the element

Creating directories (1)

You can create a directory using the `mkdir` command.

Example:

```
$ mkdir new_directory_name
```

Useful options:

- -p: create the full path if elements on the path do not exists yet

Creating directories (2)

Full example:

```
$ ls -rtl [ENTER]
-rw-r--r-- 1 nico nico    0 okt  6 12:25 info.docx
drwxr-xr-x 2 nico nico 4096 okt  6 12:25 infogroep
drwxr-xr-x 2 nico nico 4096 okt  6 12:25 samenvattingen
$ pwd [ENTER]
/home/nico/Documents/
$ mkdir infogroep/workshop-linux [ENTER]
$ cd infogroep [ENTER]
$ ls -rtl [ENTER]
drwxr-xr-x 2 nico nico 4096 okt  6 12:25 samenvattingen
drwxr-xr-x 2 nico nico 4096 okt  6 12:25 infogroep
-rw-r--r-- 1 nico nico    0 okt  6 12:25 proposal.tex
drwxr-xr-x 2 nico nico 4096 okt  6 13:09 workshop-linux
```

Removing files and directories (1)

You can remove files with `rm` command:

```
$ rm filename
```

Options:

- -r: remove recursively, this can be used to remove entire directories. But be careful with this one!
- -f: forcefully remove and ignore any warnings
- -d: remove empty directories

Removing files and directories (2)

Full example:

```
$ ls -rtl [ENTER]
drwxr-xr-x 2 nico nico 4096 okt  6 12:25 samenvattingen
drwxr-xr-x 2 nico nico 4096 okt  6 12:25 infogroep
-rw-r--r-- 1 nico nico    0 okt  6 12:25 proposal.tex
drwxr-xr-x 2 nico nico 4096 okt  6 13:09 workshop-linux
$ rm proposal.tex
$ ls -rtl
drwxr-xr-x 2 nico nico 4096 okt  6 12:25 samenvattingen
drwxr-xr-x 2 nico nico 4096 okt  6 12:25 infogroep
drwxr-xr-x 2 nico nico 4096 okt  6 13:09 workshop-linux
```

Creating empty files

You can create empty files using the `touch` command.

Example:

```
$ touch example.txt
```

Will create an empty file named `example.txt`.

Permissions of files and directories (1)

- Files and directories are owned by a specific user and group
- A group consists of one or more users
- The permissions of those owners and others can be changed on a directory or file basis
- The permissions are grouped in the following groups:
  - user permissions (u): one user who owns the file or directory
  - group permissions (g): all users that are within that group
  - other users permissions (o): everyone else that isn't an owner

Permissions of files and directories (2)

You can give or revoke the following permissions to those groups

- read (r): the ability to read a file
- write (w): the ability to change the contents of a file
- execute (x) or list (x): the ability to execute a file (as code) or if a directory to list its contents

Example:

```
$ ls -rtl
-rw------- 1 nico nico 0 okt  6 13:29 example.txt
```

File can only be read or written by the user that owns it (nico).

Changing the owner

You can change the owner of a file or directory with the `chown` command.

Example

```
$ chown joske:infogroep example.txt
```

- `joske` is the user that will own the file
- `infogroep` is the group that will own the file
- `example.txt` is of which the ownership will be changed

If you want to change ownership of a directory and its contents use `chown -R`.
Use `chown -R` with caution since it broke entire filesystems in the past!

Changing the permissions

You can change the permissions of a file or directory with the `chmod` command.

Revoke the owners permission to read the file:

```
$ chmod u-r example.txt
```

Use the `-` sign to revoke permissions and the `+` sign to grant permissions.

Running commands as the superuser (administrator)

On Linux you can run commands as the superuser (if you have sufficient permissions) using the `sudo` command.

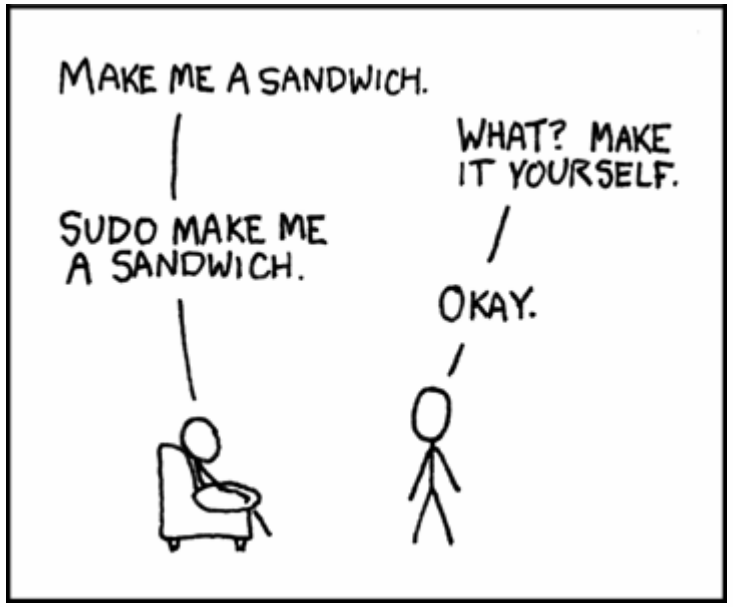
This will execute that command that follows is at the `root` user. The `root` user has all permissions on the system.

Example:

```
$ sudo touch /creating_files_on_the_root_is_normally_restricted.txt
```

Running commands as the superuser

Use with caution, with great power comes great responsibility!



Source: https://xkcd.com/149/

Execute permission (1)

The execute permission grants the user the ability to execute an ordinary file.

So any file can be used as a "program". This is not the case in windows where all executables must have the `.exe` extension.

For example when we execute:

```
$ google-chrome
```

A file named `google-chrome` will be looked up and will be executed.

Execute permission (2)

A text file can also be used as an executable program.

```
$ ./my-racket-file.rkt
```

Will execute a Racket file in the current directory.

This only works if we tell Linux which program to use in order to execute the Racket file.

You can achieve this by adding the following line at the top of the Racket file:

```
#!/usr/bin/env racket
```

and give the file the execute permission using `chmod +x`

Execute permission (3)

Full example:

```
#!/usr/bin/env racket
#lang r5rs

(define (fac n)
  (if (= n 0)
      1
      (* n (fac (- n 1)))))

(display (fac 5))
(newline)
```

Environment variables

Each process (running program) has a list of environment variables. These variables tell the process things about the environment in which it is running.

Examples of important environment variables:

- PATH: where to look for executables (see later)
- LD_LIBRARY_PATH: where to look for shared libraries
- USER: the user who is executing the program
- PWD: the current working directory
- UID: the unique identifier for the user
- see `env` for environment vairables that are set in Bash

Setting environment variables

You can change or add environment variables using the `export` command.

Example:

```
export DATABASE_PASSWORD=123
```

In general:

```
export VARIABLE_NAME=variable_value
```

Retrieving environment variables

With `env` to display all environment variables.

Or with `$VARIABLE_NAME`.

Example:

```
$ echo $DATABASE_PASSWORD [ENTER]
123
```

Where to look for executables (1)

Bash will always look at the directories defined in the `$PATH` environment variables for executables.

For example firefox is installed at: `/usr/bin/firefox`.

You can find out where an executable was found using the `which` command.

Example:

```
$ which firefox [ENTER]
/usr/bin/firefox
```

Where to look for executables (2)

You can add a directory for lookup by adding it the list in the `$PATH` environment variable:

```
$ echo $PATH [ENTER]
/home/nico/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:
/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
$ export PATH="/home/nico/infogroep-bin:$PATH"
```

Each item in the list is seperated by a `:`.

Redirection (1)

We have two outputs and one input in each Linux process:

- `STDIN (0)`: the default input for each proces
- `STDOUT (1)`: the default output for each process
- `STDERR (2)`: output used for printing errors

We can redirect those outputs files using the redirect `>` operator.

Example:

```
$ ls > listing.txt
```

Redirection (2)

Now the file `listing.txt` will contain the listing of the current directory.

`STDERR` can be redirected as follows:

```
$ rm nonexistingfile 2> err.txt
```

Note that nothing is printed in the terminal.

Redirection (3)

We can also redirect a file to `STDIN`.

```
$ echo < err.txt [ENTER]
rm: cannot remove `nonextingsfile.txt`: No such file or directory
```

Note that for simply reading and printing a file to the terminal the `cat` command can be used.

Redirection is a powerful tool. Often used for logging purposes.

Piping

When the output of one command is redirected to another command we speak of **piping**.

Piping is achieved using the `|` operator.

Example:

```
$ ls -rtl | grep "example.txt"
```

grep would search for the string `example.txt` in the output of `ls -rtl` and would print the line with the match to the screen.

Other useful tools and commands

Text editing

Terminal based text editors:

- vim (hard to learn)
- emacs (also hard to learn)
- nano (quite easy to use)

Graphical text editors:

- gedit (very easy to use)

Other useful tools and commands

Running things in the background

You can use the `&` symbol to run a command in the background so that you can execute other commands while the program is running.

Example:

```
$ firefox &
```

Don't confuse `&` with `&&`, `&&` is used for running one command after another.

Example:

```
$ mkdir test && touch test/test.txt
```

Other useful tools and commands

Secure Shell (ssh)

ssh can be used to connect to a remote server.

Example:

```
$ ssh nmattela@infogroep.be
```

Other useful tools and commands

Text search

Search for text in a file:

```
$ grep "SomeText" someFile
```

grep can use regular expressions:

```
$ grep "^!" someFile
$ grep "!$" someFile
```

- -R: search recursively through directories
- -l: only return filenames
- -A, -B, -C: show context around matching lines
- -i: case insensitive search

Other useful tools and commands

Handling files

Copy a file:

```
$ cp from to
```

Add `-r` to copy directories.

Remove a file:

```
$ rm file
```

Add `-r` to remove directories.

Warning: Linux is unforgiving, recovering files can be quite difficult.

Other useful tools and commands

Handling files

Move a file:

```
$ mv from to
```

Link a file:

```
$ ln -s file link
```

Print the contents of a file:

```
$ cat file
```

Other useful tools and commands

Wildcards

The symbol `*` means: any string of unspecified length (including the empty string).

For example:

```
$ mv *.jpg ~/images/
```

Moves all files ending in .jpg to the `images` directory in the home folder.

```
$ cat *
```

Concatenates the content of all files in the current directory.

Other useful tools and commands

More wildcards

`?` matches a single character.

```
$ rm *.?
```

Removes all files that have a dot in the name and only one character after the dot.

```
$ mv *.[jpg][jpeg][png] ~/images/
```

Moves all files ending in .jpg, .jpeg and .png to the `images` directory.

Advanced tools

Text processing with AWK and sed

AWK is a domain specific language for text processing.

Example:

```
$ awk 'length($0) > 80' file
```

sed (stream editor) can be used to manipulate text from the command line.

Example:

```
$ sed 's/bliep/blaap/g' someFile > result
```

Bash scripts

Bash scripts

General

- Bash is a programming language with support for control structures, variables, ...
- Useful for automating repetitive tasks
- Very strict syntax
- I recommend using a different language for complex scripts

Bash scripts

Variables

Creating and accessing variables in Bash:

```
myvar="Hello world"

echo $myvar
```

Command line arguments are accessible as variables:

```
echo $0
echo $1
echo $#
...
```

Bash scripts

Datatypes

Bash has 3 datatypes:

- Strings
- Arrays
- Associative arrays

Bash scripts

If tests

```
#!/bin/bash

if [ $# -eq 1 ]
    then
        cat $1
    else
        cat /dev/stdin
fi
```

Bash scripts

While loops

```
#!/bin/bash

while true
do
    echo "Hello"
    echo "World"
done
```

Bash scripts

For loops

```
#!/bin/bash

for i in $(ls)
do
    echo $i
done
```

Real example

Summary

Summary

- Linux is the core (kernel) for operating systems. Multiple **distributions** exist that are built on top of Linux
- Linux is very versatile and customizable
- Linux has a very powerful command line (i.e. Bash)
  - creating directories
  - creating files
  - managing permissions
  - searching for text (using grep and cat)
  - ...
- Bash can be used as a scripting language