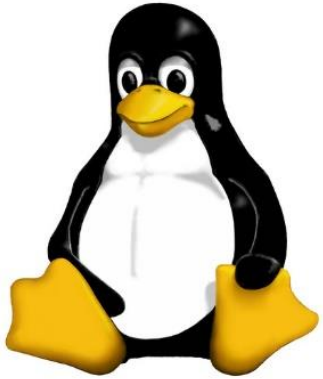


Introduction to Linux

Giorgio Agugiaro

Last update: 22 October 2024

License

Index

Check for the latest version here: https://3d.bk.tudelft.nl/gagugiaro/tutorials.html

Index

Linux in a nutshell

- **Linux** (more precisely: **GNU/Linux**[1]) is a computer operating system that uses the GNU software *and* the Linux kernel
- **GNU** is a collection of free and open software packages which can be *either* used as a stand-alone operating system *or* can be used in parts in other operating systems – as in the case of GNU/Linux. Richard Stallman started the development of GNU in 1984
  - Examples: GCC (GNU C Compiler), GNU Bash shell, etc.

- The **Linux kernel** was originally developed by Linus Torvalds and first released in 1991. A kernel is a computer program, always loaded in memory, and is the core of a computer's operating system. It controls all resources and applications, and it facilitates the interaction between hardware and software.

[1] If you are curious, you can read here more on the long-standing naming controversy: https://en.wikipedia.org/wiki/GNU/Linux_naming_controversy

Linux in a nutshell

Linux is used in personal computers, but not only!

You'll find it, among the rest, in:

- all Android devices
- routers, NAS, TVs, eReaders, smart watches, cars...
- the majority of servers, world-wide (96%)
- high-performance computers, world-wide (nearly 100%)
- ...the International Space Station!! 😀

Image source: https://www.asi.it/wp-content/uploads/2019/03/iss.jpg

Linux in a nutshell

- Linux comes packaged in a so-called **Linux distribution** (short: "distro") that assembles the kernel, the GNU tools and a selection of several other software packages tailored for specific needs
- Some common distributions are Debian, Fedora, openSUSE, ArchLinux, Gentoo. Other distros "derive" from the previous ones: Ubuntu, Red Hat Enterprise Linux, SUSE Linux Enterprise, Manjaro Linux, etc.

- Some distros are tailored to servers, other to personal computers. The latter ones generally come with a Desktop Environment (DE), consisting of several programs with a Graphical User Interface (GUI)
- There are different **Desktop Environments**. Among the most common ones are KDE Plasma, GNOME, MATE, LXDE, etc.

Image adapted from: https://nixwindows.wordpress.com/wp-content/uploads/2015/02/linux-distro-stickers.png

Users

Just like in other operating systems, there are **two types of users**. Each type of users is granted different privileges. They are:

- **root**: is the "name" of the main *superuser* in Linux. A superuser is a special account user used for system administration. The "root" user has all rights and permissions to all files and programs in all modes (single- or multi-user). The "root" user should never ever be used to perform operations on the computer other than system administration!
- **"Normal" users**: are all other users that are not "root". They have limited privileges on what files/directories they can read and write, and what programs they can run. Normal users are associated to a **user name**, such as "giorgio", "luke", "ashoka", etc, and to a **user group**.
  - In many Linux distributions (like Ubuntu), the **sudo** command enables "normal" users to run programs with the security privileges of another user, by default the superuser "root".
    For example: updating the software requires the following two commands to be run by the "root" user – unless a "normal" user adds the sudo command as follows:

    sudo apt update
    sudo apt upgrade

Working with Linux

- Just like in other operating systems, you can interact with Linux in two ways:

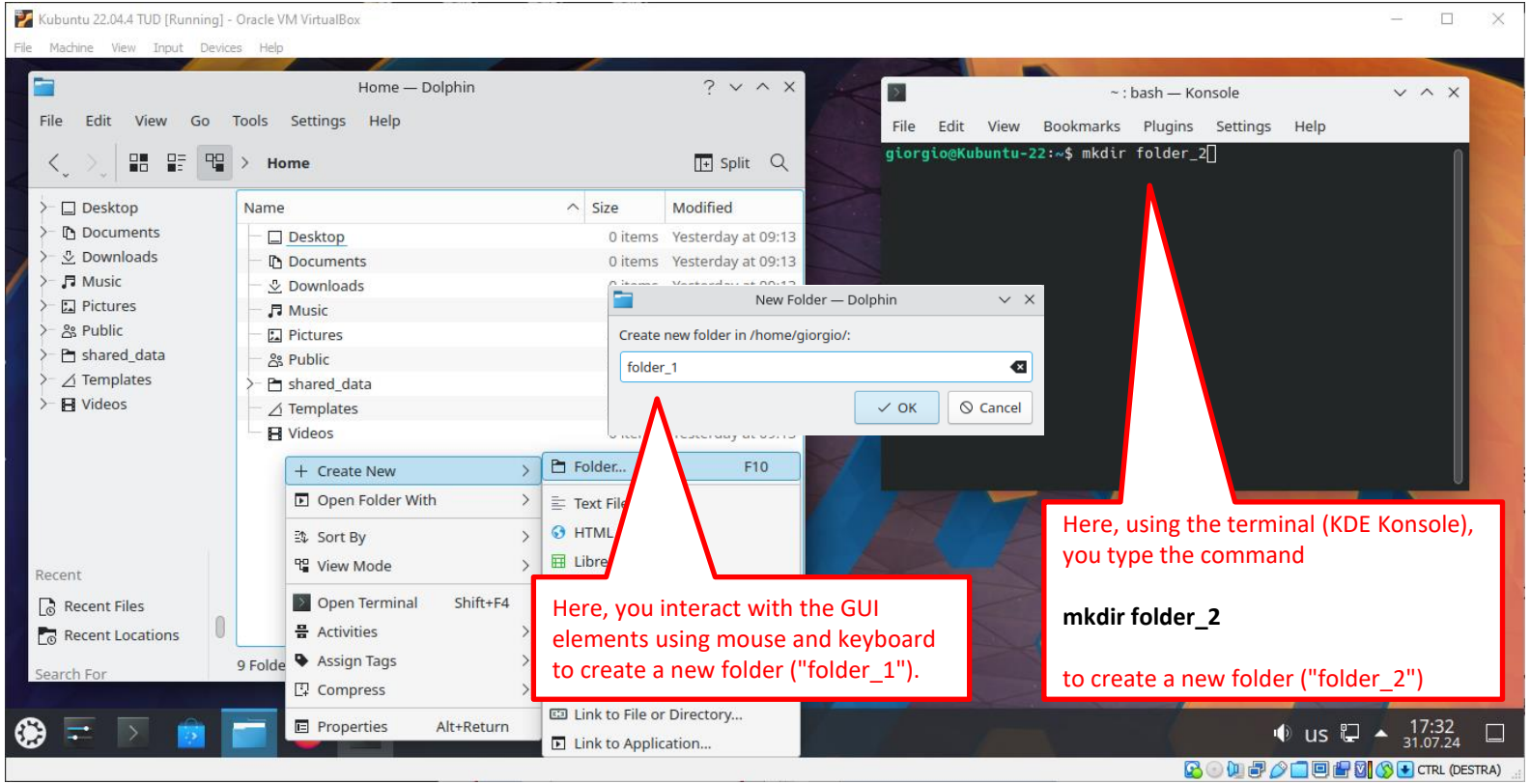
- Using the **terminal** to write commands that the system will carry out
  - Generally faster, but requires to know the command names and their syntax
  - You can cluster and run multiple commands together by writing and running a script

- Using the **GUI elements** (windows, buttons, etc.)
  - More user-friendly, commands can be run clicking buttons or interacting with the GUI
  - It may be more difficult to automatize repetitive operations using the GUI elements (some commands may not have a GUI at all…)
  - GUI elements may not always be available (e.g. working remotely on a server)

- See next slide for an example

Working with Linux

Example: Create a new folder in your home directory



Index

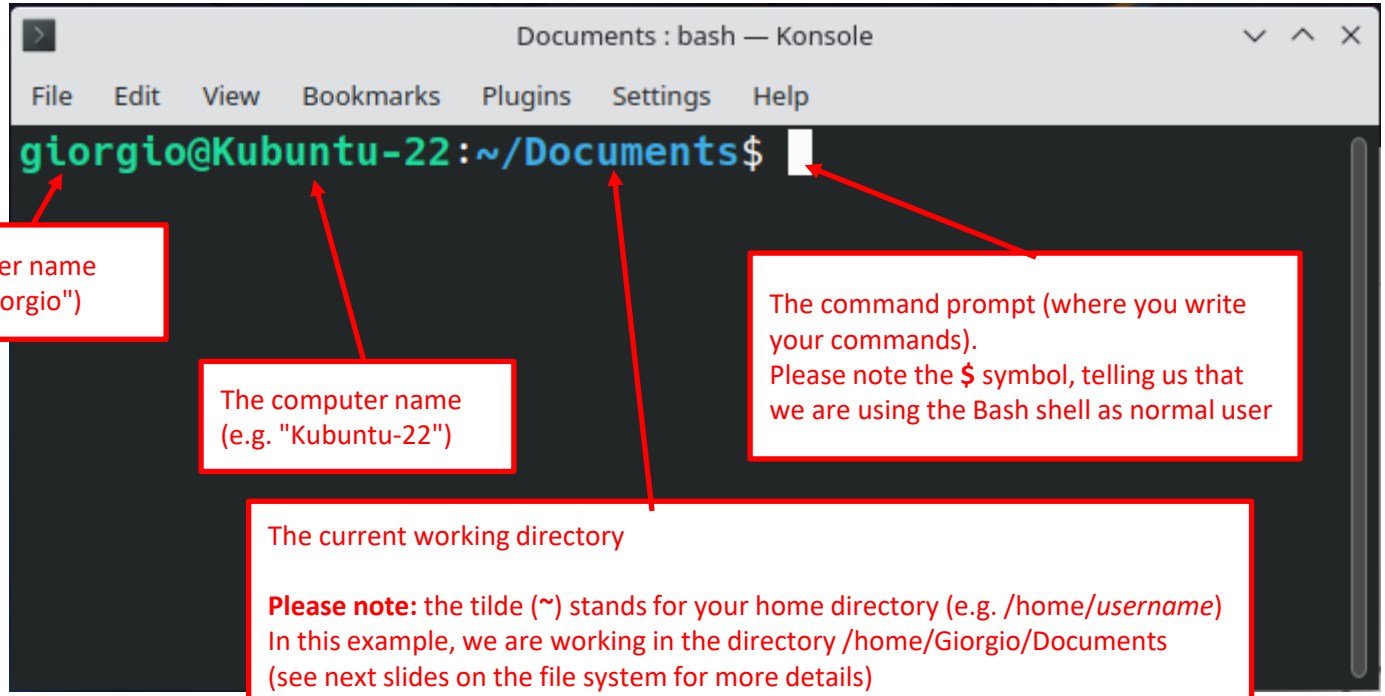
The terminal and the shell(s)

- The **terminal** (also called the **console**, or **command line interface** (CLI)) is a tool to interact with computers by typing textual commands on your keyboard. A terminal is the "window" in which you type commands. It handles user input and output
- A terminal uses a **shell**. A shell is a program that takes the commands you type and translates them into actions that the operating system has to perform. There are several shells, the most common ones are:
  - **bash** (Bourne-Again Shell): the most common one on Linux (and used in this guide)
    - the bash prompt is $ for a regular user and # for root (see next slide)
  - **zsh** (z shell): Extended bash with many improvements
    - the zsh prompt is % for a regular user and # for root
  - **csh** (C-shell): It mimic the C language as the Linux kernel is predominantly written in C
    - the csh prompt is % for a regular user and # for root
  - **ksh** (KornShell): implements and extends features from the C shell and Bourne shell
    - the ksh prompt is $ for a regular user and # for root
  - …more shells can be found here

Please note: while the terms "terminal" and "shell" are often colloquially used interchangeably, they are not the same!

The terminal and the Bash shell

Whenever you open a terminal, you get the following information when using the Bash shell:



Shell commands

Most of the shell commands have a common structure, i.e:

command + options + arguments

Example: The next command copies the source_directory and all its contents to the dest_directory.

```
cp -rf ./source_directory ./dest_directory
```

- **command**: cp (copy)
- **options**: -rf = -r recursively, -f force
- **arguments**: ./source_directory ./dest_directory

If you do not remember the correct syntax? No problem!

- Most of the commands have a "--help" option
  - Examples: `cp --help`, `rm --help`, `mkdir --help`
- The `man` ("manual") command loads and shows the extensive manual
  - Example: `man cp` will load the on-line manual of the "cp" command

Some useful shell commands

These are very common Bash commands

- exit: exit and close the terminal window
- echo <text>: displays a line of text
- cat <file> ("concatenate"): print file (or group of files) to screen
- head <file>: print the first lines of a file
- tail <file>: print the last lines of a file
- which <command: locate a command
- whereis <command>: locate the binary, source, and manual page files for a command
- locate <file>: find file(s) by name, quickly, generally using an index created/updated by updatedb
- touch <file>: create a new empty file

With many commands you can use also **wildcards**, such as:

- ?: Matches any single character
- *: Matches any string of characters
- [set]: Matches any character in the set. Example: [adf] will match any occurrence of a, d, f
- [!set]: Matches any character NOT in the set of characters

Examples

- ls *.txt -> list all files having the extension like "txt"

Index

- Linux in a nutshell
- The terminal and the shell(s)
- **The file system**
- File system permissions
- Data streams
- Processes
- Scheduling jobs
- Installation of software applications
- Text editors
- Bash shell scripting
- Further resources

File system

- In Linux (and other Unix-like operating systems) files are written in a **hierarchical file system**
- Such file system is standardised and is called **Filesystem Hierarchy Standard** (FHS)
- In the FHS, all files and directories appear under the **root directory /**, even if they are stored on different physical or virtual devices
- The first-level directories names **(/bin, /usr/, /home, /var**) are consistent over the Unix-like operating systems, and are generally used in the same way

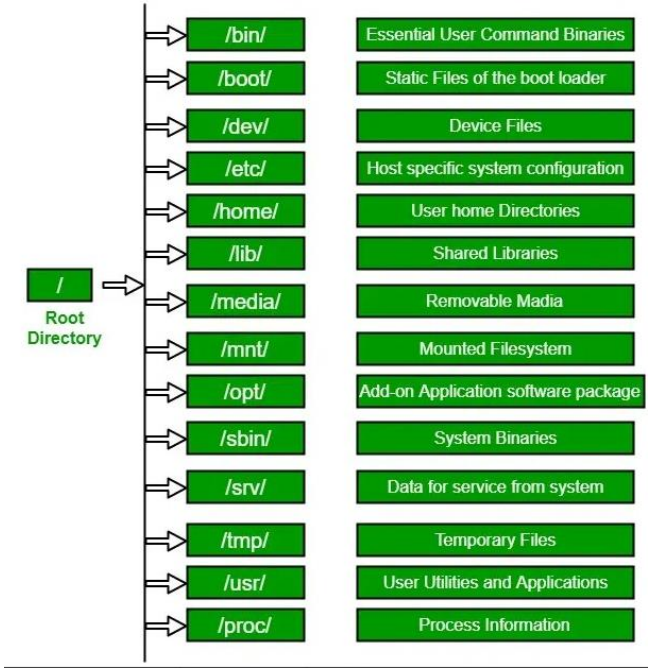


Image source: https://www.geeksforgeeks.org/linux-file-hierarchy-structure

File system

Some of these directories are particularly relevant:

- **/ (aka "root directory")**: Base directory of the entire file system hierarchy
- **/root**: the "root" user's home directory
- **/home**: all users have their own home directory. Normal users can write ONLY inside their home directory (and subdirectories).
  - Example: /home/giorgio, /home/leia, /home/kylo, etc.
- **/sbin** and **/usr/sbin**: contains program executables ("binaries") that are used for system administration and can be run only by a user with superuser privileges
- **/bin** and **/usr/bin**: contains the majority of the binaries that are installed by default and that can be run by "normal" users.
- **/usr** (and subdirectories): contains the majority of the user utilities and applications
- **/lib**: software libraries necessary for the binaries in /bin and /usr/bin
- **/media**: mount points for removable devices (USB sticks, CD-ROMs, etc.)
- **/mnt**: temporarily mounted file systems
- **/dev/null**: is a special device, used to dispose unwanted output streams of a process, or as a convenient empty file for input streams. This is usually done by *redirection* (see later for examples)

File system

Some common Bash commands to work with the file system

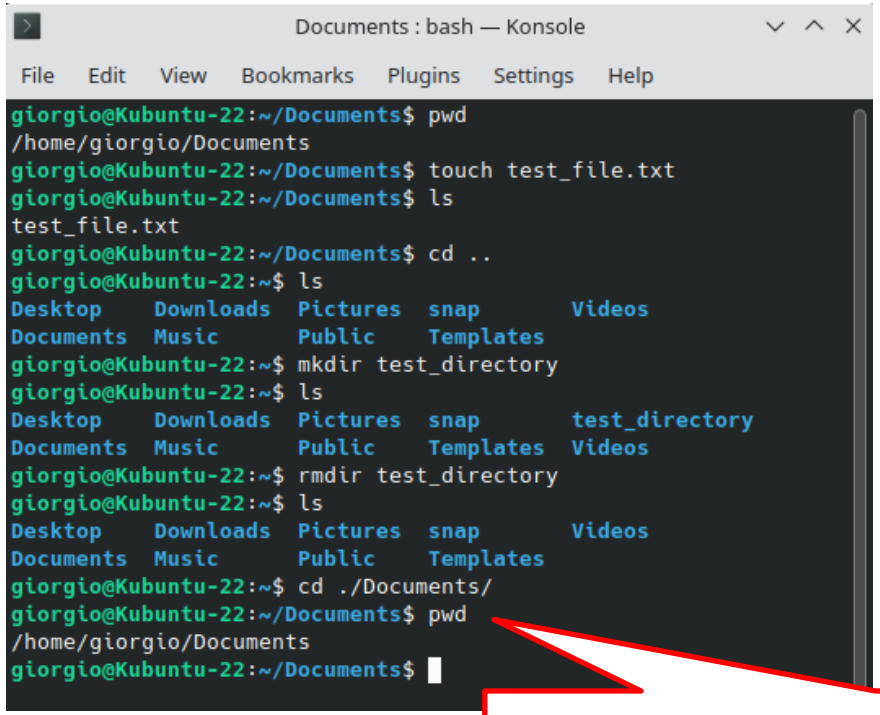
- `pwd` ("print working directory"): tell you in which directory in which you currently are
- `tree`: show all subdirectories from the directory you are in
- `ls` ("list short"): list files and subdirectories in the directory you are in (no details)
- `ll` ("list long", alternative for `ls -l`): list files and subdirectories in the directory you are in (with all details)
- `cd <directory>` ("change directory"): change to another directory
- `mkdir <directory>` ("make directory"): create a new directory
- `rmdir <directory>` ("remove directory"): remove an (empty) directory
- `cp <source-file> <destination-file>` ("copy"): create a copy of a file
- `rm <file> or <directory>` ("remove"): remove a file or a populated directory
- `mv <file> <new-destination>` ("move"): move a file to another directory

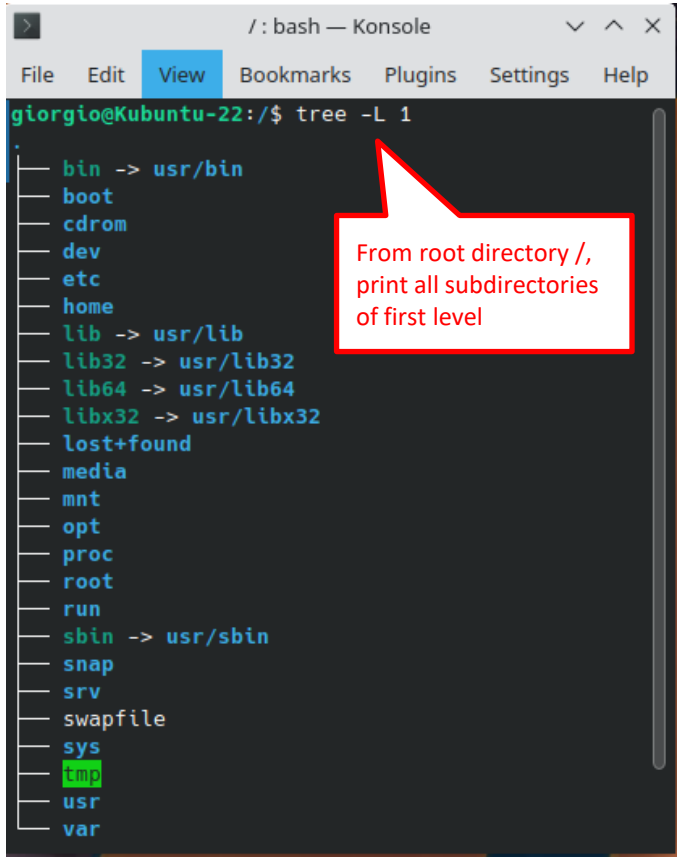
Navigation in the file system

- `.` (dot): is the current directory
- `..` (two dots): is the parent directory
- `~` (tilde): stands for your own home directory

File system

Examples:

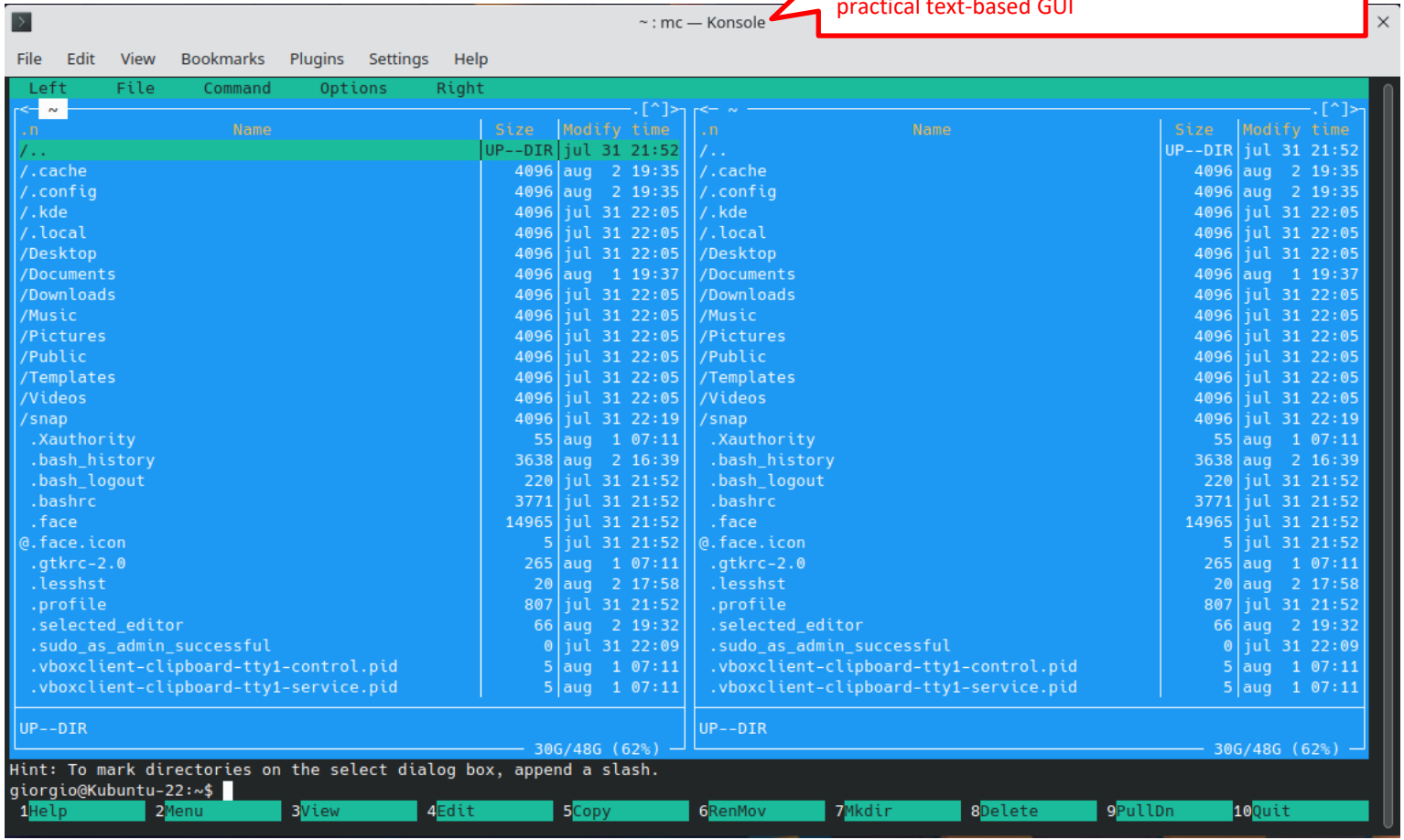




From root directory /, print all subdirectories of first level

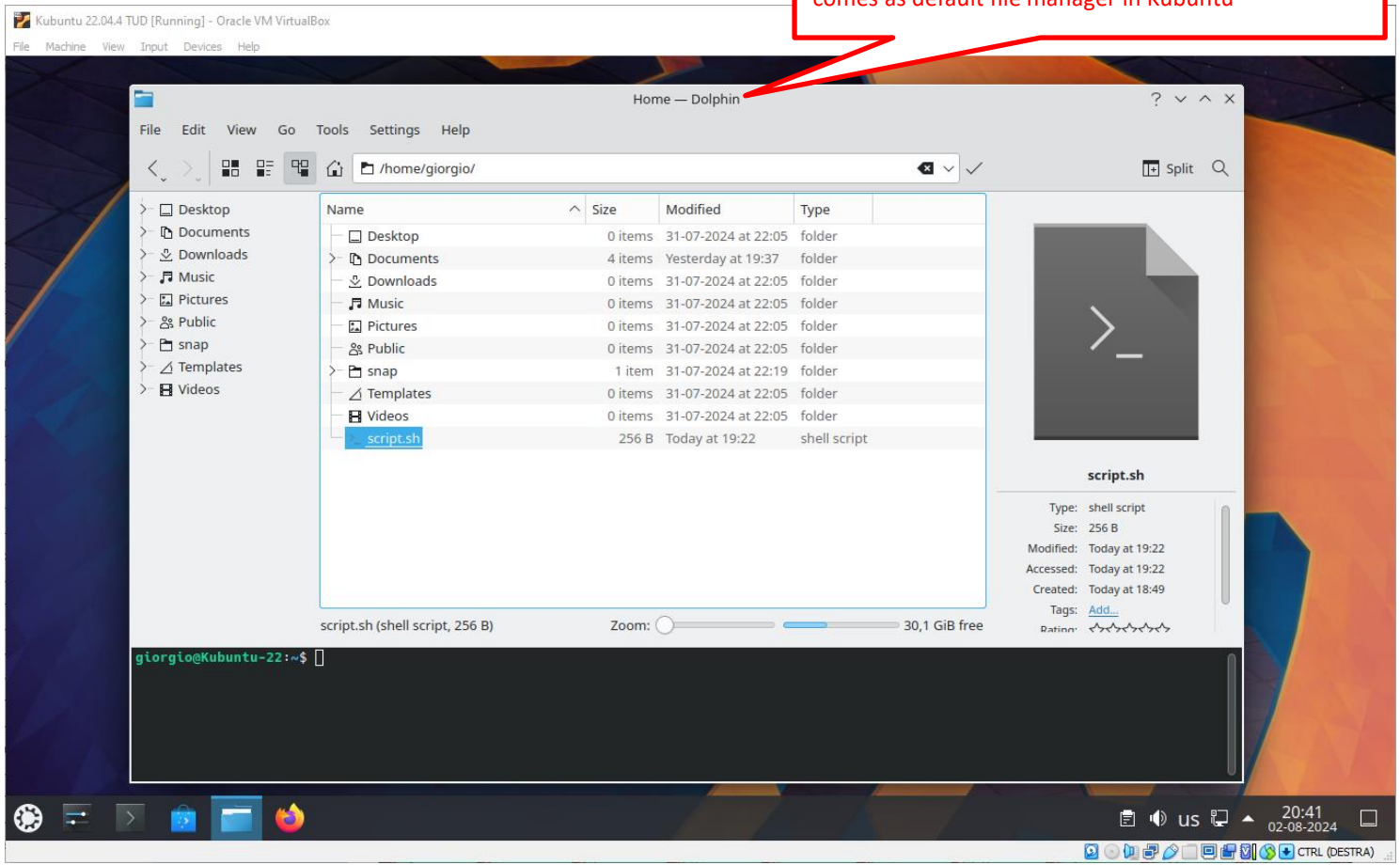
Perform a series of operations using **pwd**, **touch**, **cd**, **mkdir**, **ls**, **rmdir** commands. You can see the output right after the command has been issued

File system



Alternatively, you can use for example **mc** ("**Midnight Commander**"), which is a very powerful file manager with comes with a simple but very practical text-based GUI

File system



Index

- Linux in a nutshell
- The terminal and the shell(s)
- File system
- **File system permissions**
- Data streams
- Processes
- Scheduling jobs
- Installation of software applications
- Text editors
- Bash shell scripting
- Further resources

File system permissions

Every file is described by a 10-character string (called **mode string**) in which:

- Character 1: type of file
- Characters 2-4: privileges of the file owner on that file
- Characters 5-7: privileges of the owner's group on that file
- Characters 8-10: privileges of "everybody else" on that file

Type of file can be:

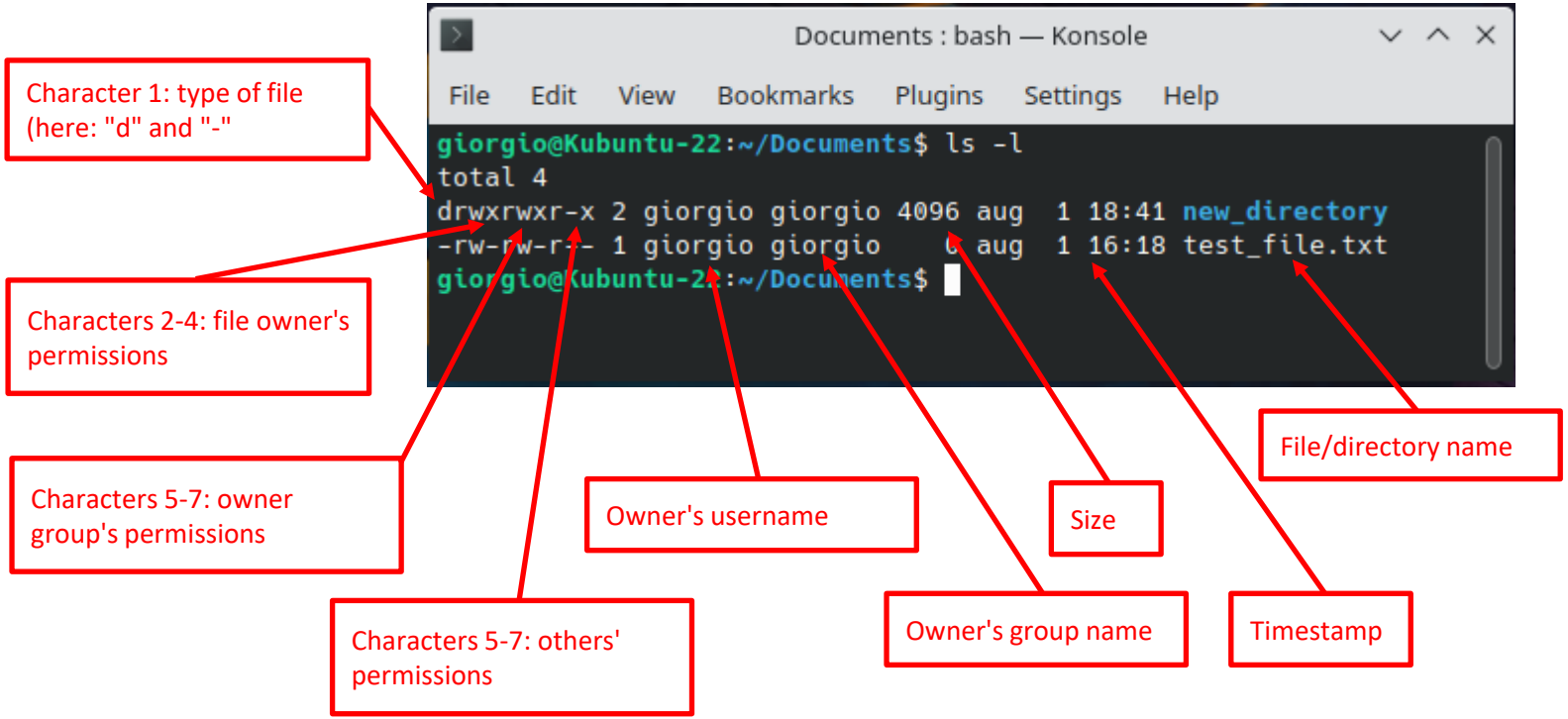
- -: a file
- d: a directory
- l: a symbolic link
- b: a block special file or block device (e.g. /dev/hda, a hard disk)
- c: character special file (e.g. /dev/tty, the terminal of the current process)
- p: a pipe (a temporary file between two linked commands – see later for more examples)
- s: a socket

Type of permission can be:

- r: readable, -: it is not readable
- w: writable, -: it is not writable
- x: executable, or permission to enter a directory, -: it is not executable

File system permissions

Using the command ls -la (or ll -a), you get all details about each file/directory in the current directory (including the hidden files, with the -a parameter).



File system permissions

The types of permissions can be also expressed numerically. In this way, all possible combinations can be expressed with a digit between 0 and 7.

- r: read = 4
- w: write = 2
- x: execute = 1

Both representations, literal and numeric, are commonly used, especially with **chmod** ("change mode"), a command used to change the permissions of a file or a directory.

#	Sum	rwx	Permission
7	4(r) + 2(w) + 1(x)	rwx	read, write and execute
6	4(r) + 2(w)	rw-	read and write
5	4(r) + 1(x)	r-x	read and execute
4	4(r)	r--	read only
3	2(w) + 1(x)	-wx	write and execute
2	2(w)	-w-	write only
1	1(x)	--x	execute only
0	0	---	none

Source: https://en.wikipedia.org/wiki/Chmod

File system permissions

The command **chmod** can be issued in several ways. The permissions to add, remove or change can be expressed using either numerical or symbolic modes.

Here are some examples for using the numerical mode.

- `chmod 664 file_name.txt`: file_name.txt will receive read and write (6) permissions for both the owner and the owner's group, and only read permissions for the "others"
- `chmod 700 file_name.txt`: file_name.txt will receive read, write and execution (7) permissions for the owner and zero permission for the owner's group and the "others"

Please refer to the manual, or **--help** for more details

File system permissions

The command **chmod** can be issued in several ways. The permissions to add, remove or change can be expressed using either numerical or symbolic modes.

The symbolic mode is composed of three components, which are combined to form a single string of text. Specific modes can be modified, leaving the others untouched

`chmod [references][operator][modes] file`

Reference	Class	Description
u	user	file owner
g	group	members of the file's group
o	others	users who are neither the file's owner nor members of the file's group
a	all	all three of the above, same as `ugo`
(empty)	default	same as "all", except that bits in the umask will be unchanged

Operator	Description
+	adds the specified modes to the specified classes
-	removes the specified modes from the specified classes
=	the modes specified are to be made the exact modes for the specified classes

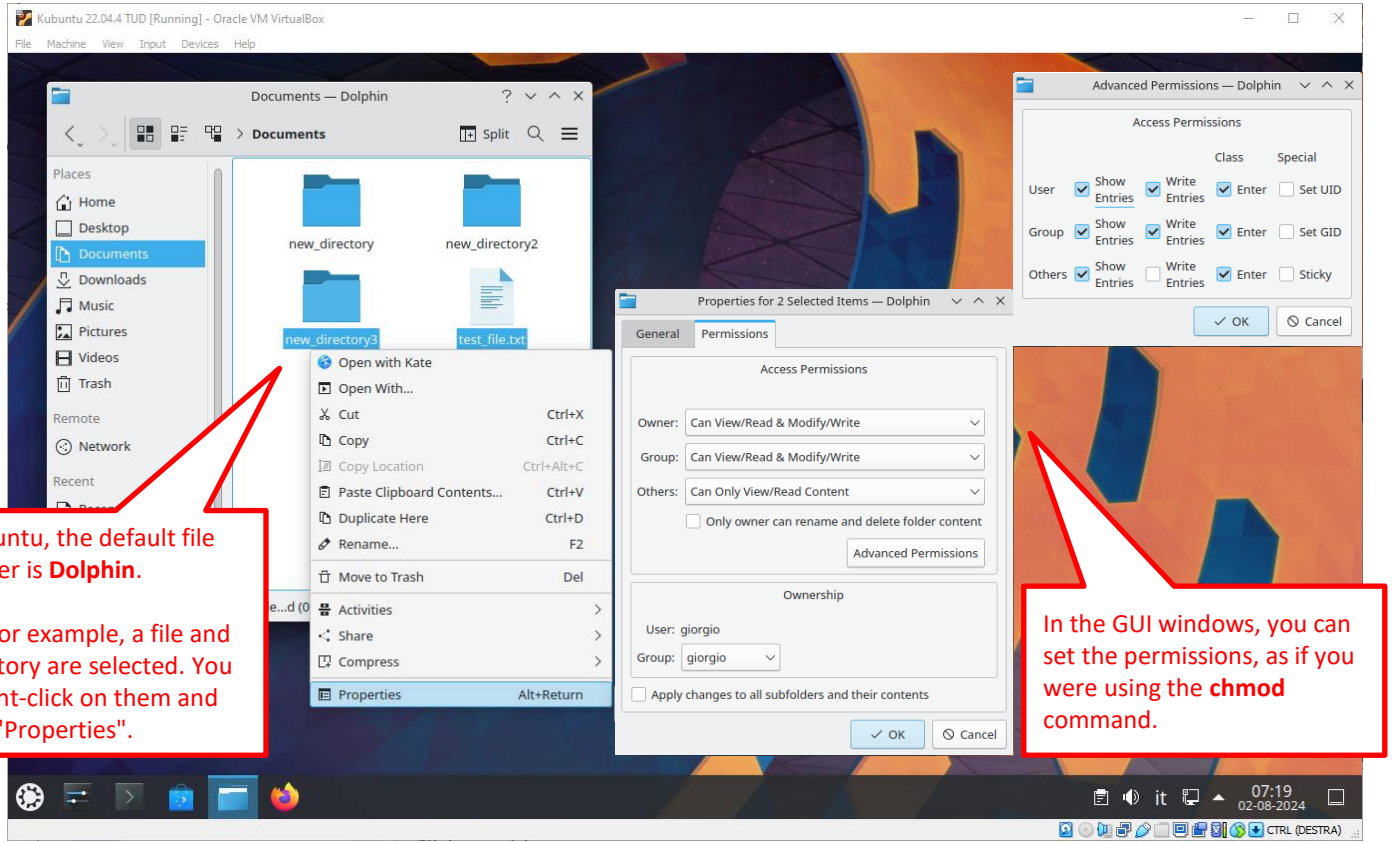
Source: https://en.wikipedia.org/wiki/Chmod

Examples:

- `chmod u+wx filename`: add write and execute privileges to filename for the owner
- `chmod a-w filename`: remove write privigege from filename for everybody
- `chmod ug=rwx filename`: set the privileges of filename to be read, write and execute only for the owner and the owner's group.

File system permissions

Of course, you can also change the permissions of files and directories using the GUI



Index

Data streams

A data stream is, as the name says, a stream of data —especially text data— being passed from one file, device, or program to another.

The GNU Utilities, the Linux core utilities, and many other command-line tools exchange data and perform their work based on data streams.

In Linux and other Unix-like OSes, the use of **Standard Input/Output (STDIO)** is a fundamental way to exchange data between programs: Programs implementing STDIO use standardised file handles for input and output instead of files stored on a disk (or elsewhere).

STDIO is a **buffered data stream**, and its function is to stream data from the output of one program/file/device to the input of another program/file/device.

Data streams

There are **3 STDIO data streams**:

- **STDIN** (File handle 0) is the **standard input** which is usually input from the keyboard. STDIN can be redirected from any file, including device files, instead of the keyboard.
- **STDOUT** (File handle 1) is the **standard output** which sends the data stream to the display by default. It is common to redirect STDOUT to a file or to pipe it to another program for further processing.
- **STDERR** (File handle 2) is the **standard error** data stream, i.e. where the program sends error and diagnostics messages. STDERR is also usually sent to the display. If STDOUT is redirected to a file, STDERR continues to be displayed on the screen. STDERR can also be redirected to the same or passed on to the next transformer program in a pipeline.

Data streams: Redirection

Linux includes redirection commands for each stream. These can be used to write STDIN, STDOUT and STDERR to a file. If you write to a file that does not exist, a new file with that name will be created prior to writing.

Commands with a **single bracket overwrite** the destination's existing contents.

> : send to standard output

< : read from standard input

2> : send to standard error

Commands with a **double bracket append** (do not overwrite) the destination's existing contents:

>> append to standard output

<< read from standard input, line by line

2>> append to standard error

Data streams: Redirection

Examples:

`ls /home/giorgio/Documents > giorgio_documents.txt` : List all files/directories in the given directory and write the results to a file

`mkdir '' 2> error.txt` : creating a directory with an empty name is not permitted. The error message will be written to file error.txt

`echo Write this message to a new file > file.txt`

`echo Append this line to an existing file >> file.txt` : Example to highlight the difference between overwriting and appending when using > and >> redirectors

Assuming we have a script called "script.sh":

`script.sh < input_file` : run the script and read from input_file

`script.sh > output_file` : run the script and write to output_file

`script.sh < input_file > output_file` : run the script, read from input_file and write to output_file

`script.sh 2> error_file` : run the script and write the STDERR to error_file

`script.sh > all_output_file 2>&1` : run the script and write both STDOUT and STDERR to all_output_file. **Please note:** In Bash, the last command can be also written as `script.sh >& all_output_file` or `script.sh &> all_output_file`

`script.sh > /dev/null` : Discard the STDOUT, i.e. redirect it to the special device /dev/null

`script.sh 2> /dev/null` : Discard the STDERR, i.e. redirect it to the special device /dev/null

Data streams: Pipes

Pipes are a particular type of stream redirection. They are used to redirect a stream from one program directly to another. When a program's STIOUT is sent to another through a pipe, the first program's output will be used as STIN to the second, rather than being printed to the terminal. Only the data returned by the second program will be displayed.

In Linux, and other Unix-like OSes, the pipe is represented by a **vertical bar "|"**

The general syntax is:

```
command_1 | command_2 | command_3 | .... | command_N
```

Pipes are unidirectional i.e., **data flows from left to right through the pipeline**.

More details: https://en.wikipedia.org/wiki/Pipeline_(Unix)

Data streams: Filters

Filters are a class of programs that take plain text as standard input, transforms it into a meaningful format, and then returns it as standard output. They are commonly used with output piped from another program.

cut: extract sections from each line of input, usually from a file
find: returns files with filenames that match the argument passed to find
grep: returns text that matches the string pattern passed to grep
sort: prints the lines of its input in sorted order
tee: redirects standard input to both standard output and one or more files
tr: finds-and-replaces one string with another
uniq: outputs the text with adjacent identical lines collapsed to one, unique line of text
wc ("word count"): counts characters, lines, and words

More details: https://en.wikipedia.org/wiki/Filter_(software)

Data streams: Filters

Examples:

`ls | grep image1.jpg` : List all files and directories, pass them to grep to search for image1.jpg

`ls -l sort` : List all files and directories and return them sorted

`cat long_text.txt | head -15 | tail -5` : selects the first 15 lines, from which the last 5 lines will be eventually displayed

`cat energy_result.csv | grep "PV production" | tee ned.txt` : reads the content of the energy_result file, send it to grep to search for the "PV production" values and finally, with tee, display them in the terminal AND save them to file ned.txt

Chaining operators

The data stream operators are part of a larger family: the **chaining operators**. They are used chain multiple commands together. They are:

| (pipe) : The output of the first command acts as input to the second command (see previous slides)

>,>>, < (redirection) : Redirects the output of a command (see previous slides)

**** (concatenation) : Allows to concatenate long commands spanning over several lines in the shell

() (precedence) : Allows to define the precedence order to execute commands

& (ampersand) : Run a process/script/command in the background

&& (logical "and") : The command following && is only executed if the command preceding && has been successfully executed

|| (logical "or") : The command succeeding || is only executed if the command preceding || has failed.

&&-|| (and-or) : Combination of the && and || operators, similar to the if-then-else statement.

! (not) : Negates an expression within a command

; (semi-colon) : The command following ; is executed even if the command preceding ; has failed

{} (combination) : The execution of the command list inside **{}** depends on the execution of the first command in the list

Chaining operators

Examples:

`ping -c20 3d.bk.tudelft.nl &` : Ping the webpage for 20 times, run the process in the background (test it also without **&** to see the difference: can you interact with the terminal?)

`who ; pwd ; ls` : Simply run the 3 commands one after the other, no matter if one fails or succeeds

`echo "Print this!" && echo -e "\nThe first echo command succeeded"` : The second echo is run only if the first one succeeds

`mkdir "" || echo -e "\nThe first command failed"` : the echo is run only because the first command fails (you cannot create a nameless directory)

`ping -c1 3d.bk.tudelft.nl && echo “Successful ping” || echo “Failed ping"` : ping the URL, if there is an answer then execute the first echo, else execute the second echo

`rm !(*.jpg)` : remove (delete) all files that do not have a .jpg extension

`ping -c1 www.brickset.com && { echo -e "\n\n*** The webpage exists!" ; firefox www.brickset.com & }` **:** Ping the webpage. Then consider the list of commands inside **{}**. Run the second in the list only if the first (echo) succeeds. Beware the syntax: please note the empty spaces after the **{** and before the **}** parentheses

Index

Processes

A running instance of a program is called a **process**

- **Example:** If you have opened two terminal windows, most likely you have two processes of the same program (e.g. "Konsole"). Each terminal loads a shell program (e.g. the Bash shell): each running shell is itself another process. Whenever you issue a command from the shell (e.g. "cp"), the corresponding program is executed in a new process, too.

Processes in Linux are organized as a tree.

- The *init* process is the root process
- Each process as its own ID (PID, process ID)
- Each process has the ID of the parent process (PPID, parent process ID)

Multiple processes, running in parallel or in series, can be grouped in jobs. A **job** is a scheduled process or set of processes.

Processes

In Linux, there are **5 types** of processes.

- **Parent process:** The process created by the user on the terminal. All processes have a parent process. If it was created directly by the user, then the parent process will be the kernel process
- **Child process:** A child process is a process that is created by another process known as the parent process
- **Orphan process**: A child process becomes an orphan process when the parent gets executed before its own child process. In such a case, the orphan process has a “Init” process (PID 0) as its PPID
- **Zombie process**: A process that is already dead but shows up in process status. Zombie processes have zero CPU consumption
- **Daemon process**: A system-related background processes. These processes often run the permissions of root and service requests from other processes. A Daemon process often runs in the background. A Daemon process can be recognized if it has “?” in its TTY field (see later)

Processes

Linux processes can be run in foreground or in background

- **Foreground processes** are started by the user and are the default. They accept command-line input and output it to the computer screen. A running foreground process prevents the start or execution of other, following processes because the command prompt will not be available until the currently running program completes its processing and comes out.
- **Background processes** run, as the name says, in the background. They are non-interactive and do not need keyboard input. While one process is running in the background, it is possible to start another process from the terminal.
  By adding a **single ampersand ("&")** at the end of a command, the command can be executed as a background process

  Example:
  - `sleep 5 && echo "Ciao!"`: Force the terminal to wait for 5 seconds, then print the message to the screen. Nothing can be done in-between in the terminal.
  - `sleep 10 && echo "Ciao!" &`: Note the last, single **&**! After issuing the command, you can still interact with the terminal. After 5 seconds, the message will be printed to the screen.

Processes

Useful BASH commands for processes

`ps` ("process status"): list processes, with different levels of information, e.g. `ps -fu`

`ps <pid>`: check status of a single process, identified by its PID

`pstree`: print the tree of the processes

`kill <pid>`: end/terminate a process, identified by its PID

`top`: display Linux processes

`htop`: an interactive system control, process viewer, and process manager

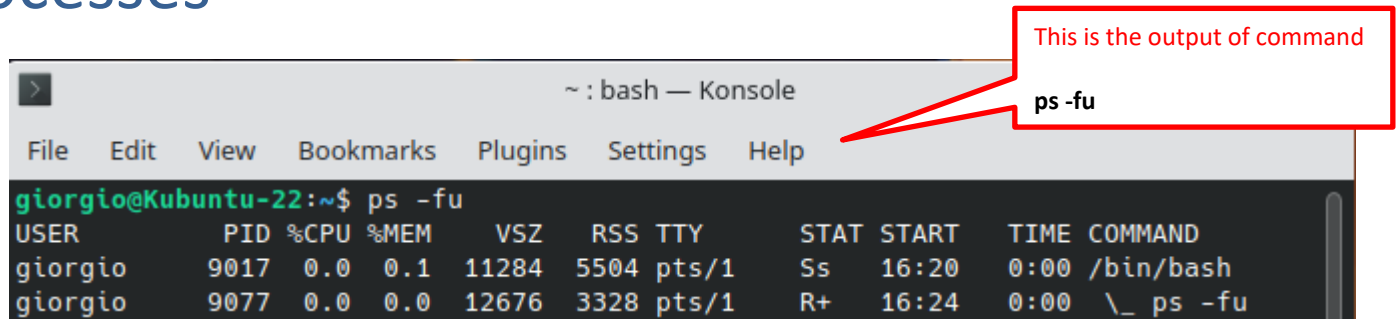
`jobs`: display status of jobs

Miscellaneous commands:

`free`: display the total amount of free and used memory (RAM) on the Linux system

`df`: display the free disk space(Hard Disk) on all the file systems

Processes



USER: Process owner

PID: The process ID

%CPU, %MEM: Percentage of CPU/memory used

VZS: Virtual memory size, virtual memory used by the process (in kB)

RSS: Resident set size, physical memory used by the process (in kB)

TTY: The terminal associated with the process

STAT: The state code of the process; there are many values, but the common ones are S (sleeping) and R (running)

START: The time when the process started

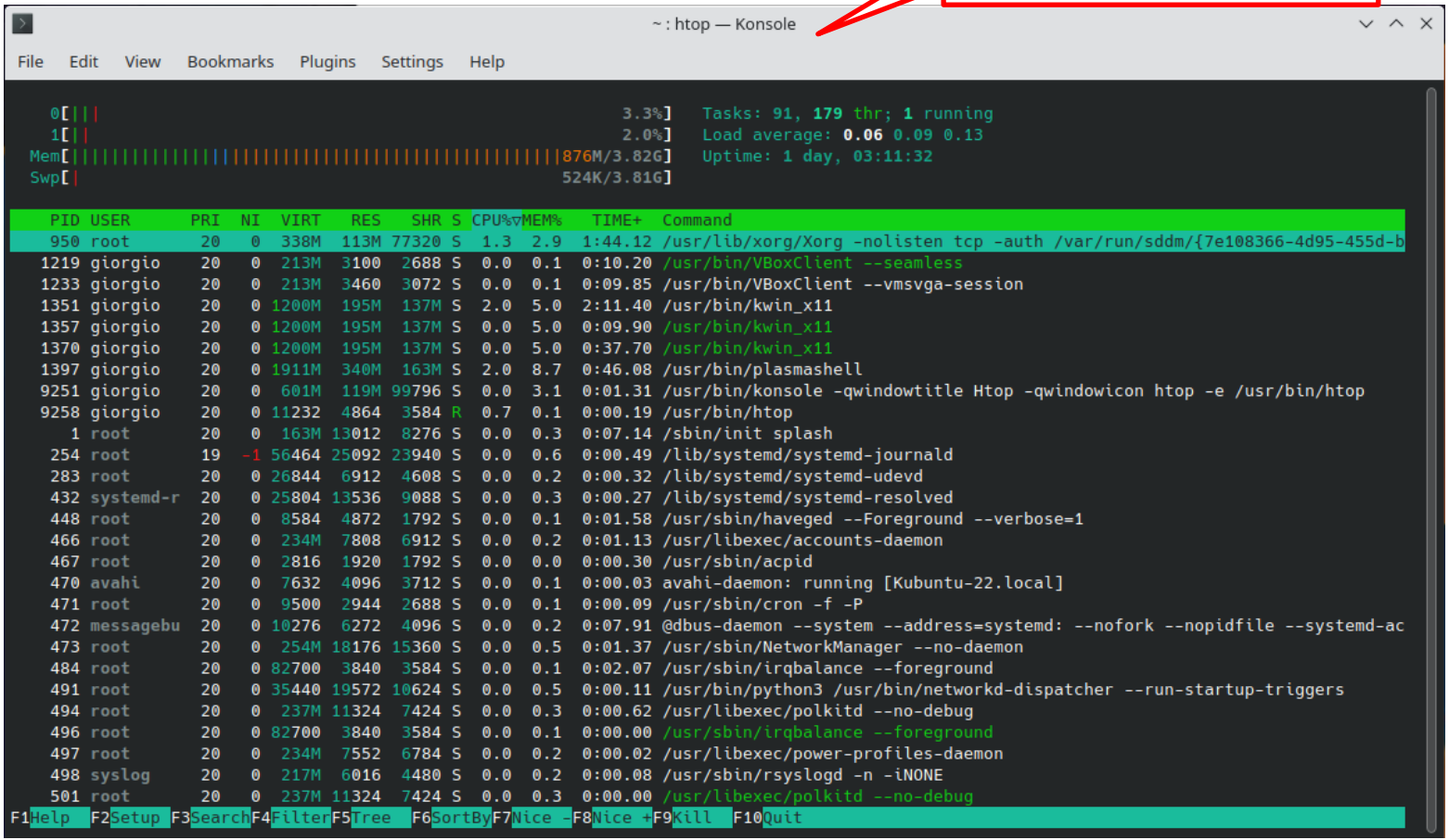
TIME: The CPU time consumed by the process

COMMAND: Command issued that created the process

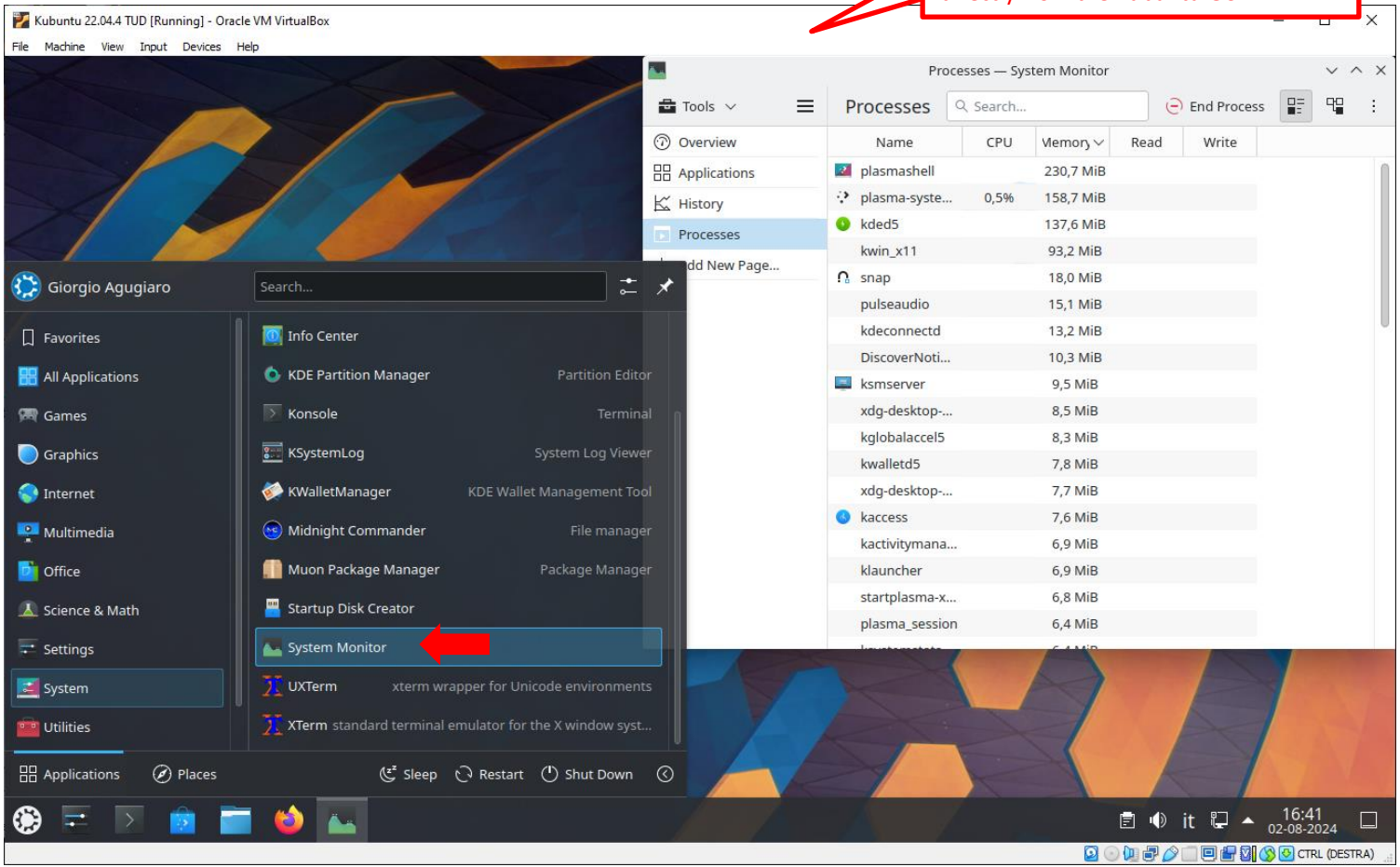
More details: https://it.wikipedia.org/wiki/Ps_(Unix)

Processes

htop is interactive system control, process viewer, and process manager



Processes



Index

Scheduling jobs

A process or a group of processes (a job) can be schedule to be run at a certain point in time, or with a certain schedule. Relevant commands are:

`at`: execute commands for one time at a specified time (in future)

`atq` (at queue): print the list of user's pending jobs

`cron`: a Daemon process. It reads every minute the crontab table and executes the scheduled jobs

`crontab`: show and manage the table containing the list of scheduled jobs. In particular

- `crontab -l`: list all jobs in the crontab table
- `crontab -e`: open the editor to add, remove, change the scheduled jobs in the crontab table
- `crontab -r`: remove the complete list of scheduled jobs

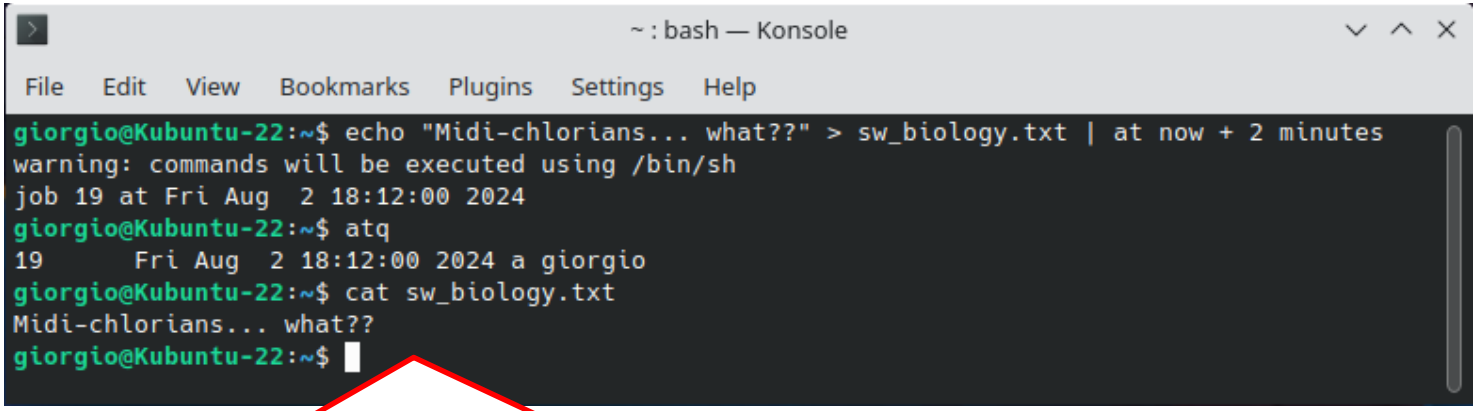
Please note: For `at` and `cron` there are several command options and some ancillary commands. Please refer to the documentation for further details. The next slides will provide only some simple examples

at: https://en.wikipedia.org/wiki/At_(Unix)

cron: https://en.wikipedia.org/wiki/Cron

Scheduling jobs: at

The command **at** can be run in different ways: **Example 1**



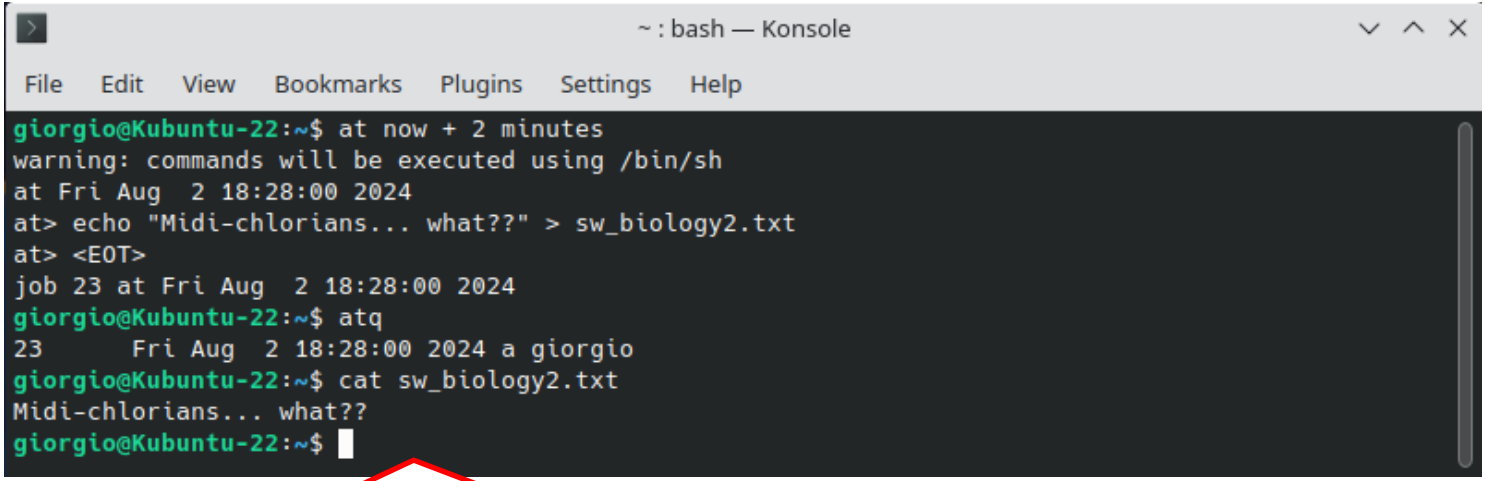
Using piping, you echo a message that will be written to a text file. This operation is to be carried out in 2 minutes from now. A job is therefore created.

You can check the pending job with **atq**.

After 2 minutes, you can check that the file has been written and contains the original message.

Scheduling jobs: at

The command **at** can be run in different ways: **Example 2**



Here you perform the same operation as before, however using at's **interactive prompt** that allows you to enter which commands to run at the specified time. A warning stating which shell the command will use is also printed.

You can exit the interactive prompt and save the scheduled job by pressing **Ctrl + D**. You can cancel the job with **Ctrl + C**.

Scheduling jobs: cron

Scheduling jobs for **cron** is consists in adding lines to the crontan table. Assume that you have a shell script named "script.sh" that you want to run at regular intervals. The generic syntax to add a job entry to crontab is

```
A B C D E COMMAND
```

with:

A: Minutes range from 0 to 59; default is * (i.e. all values)

B: Hours range from 0 to 23; default is * (i.e. all values)

C: Days range from 0 to 31; default is * (i.e. all values)

D: Months range from 0 to 12; default is * (i.e. all values)

E: Days of the week range from 0 to 7 (Sunday is 0 or 7); default is * (i.e. all values)

COMMAND: command to be executed.

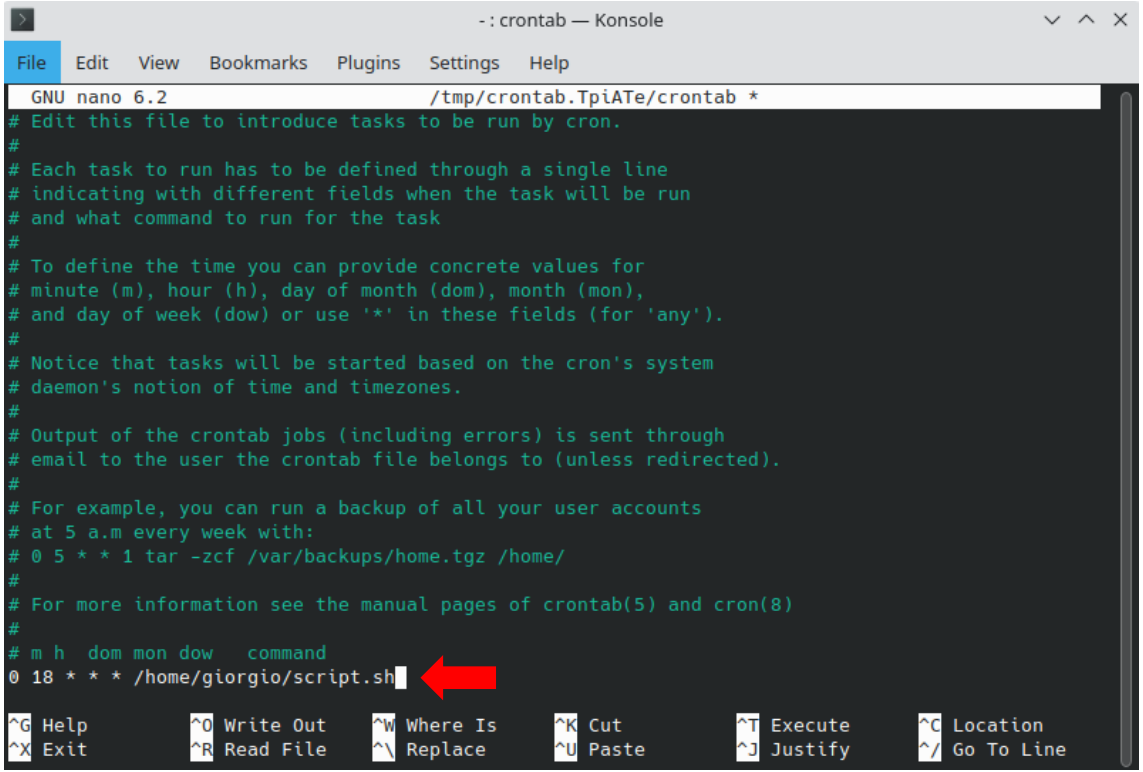
Example:

Line to add to run the script every day of the year exactly at 18:00 (6:00 PM)

```
0 18 * * * /home/giorgio/script.sh
```

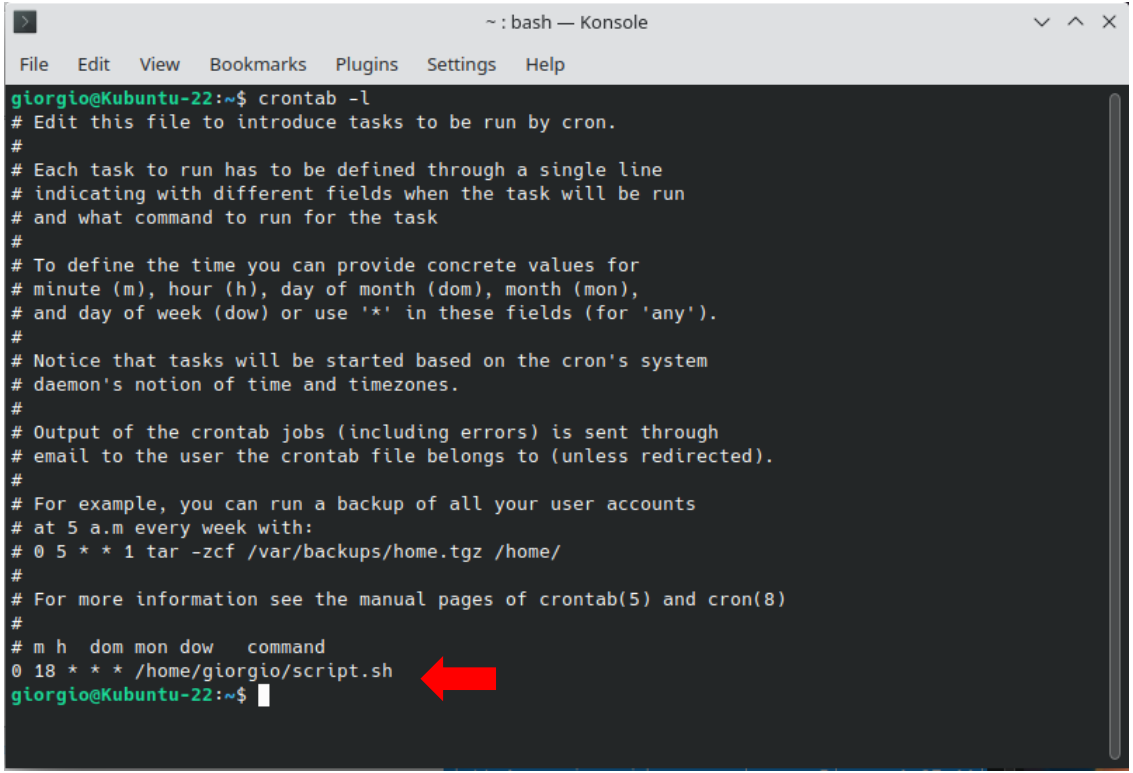
Scheduling jobs: cron

Run `crontab -e` to edit the crontab table and add a line for each job at the end of the file. Save and exit. Suggestion: always use ABSOLUTE PATHS to your script.



Scheduling jobs: cron

Run `crontab -l` to print to screen the crontab table. Here you can see that the job has been added and will be carried out at the set time.



Index

- Linux in a nutshell
- The terminal and the shell(s)
- File system
- File system permissions
- Data streams
- Processes
- Scheduling jobs
- **Installation of software applications**
- Text editors
- Bash shell scripting
- Further resources

Installation of software applications

- In Linux there are several ways to install a software application
- The easiest, and by far the most common way, is to install software applications from a **repository**. A repository is a public server that hosts software packages
- A **software package** (or just: package) is a collection of files and metadata that contains a specific software application. The purpose is to simplify the process of distributing, installing, and managing software on a computer system
- There are different types of software **package formats**, depending on the distribution. The two most common ones are **rpm** and **deb**. The *rpm* format is tailored to Red Hat Linux and its derivatives, while the *deb* format is for Debian-based distributions, such as (K)Ubuntu
- A Linux distribution provides a command, and usually a GUI-based program, that retrieves the software from a repository and installs it onto your computer
  - It is conceptually the same as searching for and installing an app on your smartphone from the Android or Apple stores!

Installation from a repository

- The **installation from a repository** can be carried out, as usual, in two ways:
  a) Using the terminal
  b) Using a GUI based software installation program
- When using the terminal, you only need to know the exact name of the package containing the program you want to install
- Before installing the software itself, it is a good habit to refresh the database(s) on your machine that contains the list of applications available in the repositories
- All these operations require superuser privileges, but can be carried out also by a normal user thanks to **sudo**
  - In (K)Ubuntu, you use the **apt** command to perform software installation and other related operations
- Alternatively, you can use the GUI-based package manager to search for the package name, and to install or remove it without using the terminal at all
  - In Kubuntu, the GUI-based package manager is called **Discover**
  - In Ubuntu, the GUI-based package manager is called **Ubuntu Software Center**

Installation from a repository

Example 1:

Let's assume we want to install the file manager **Midnight Commander**. Its package name is **mc**. Let's open a terminal and type (you will be asked your password):

```
sudo apt update
sudo apt install mc
```

That's all! 😀 The first line updates the database(s) containing information on the software in the repositories. The second line downloads and installs Midnight Commander. At the end, you can type mc, and the program will be launched. Using command chaining, the above commands can be also written as

```
sudo apt update && sudo apt install mc
```

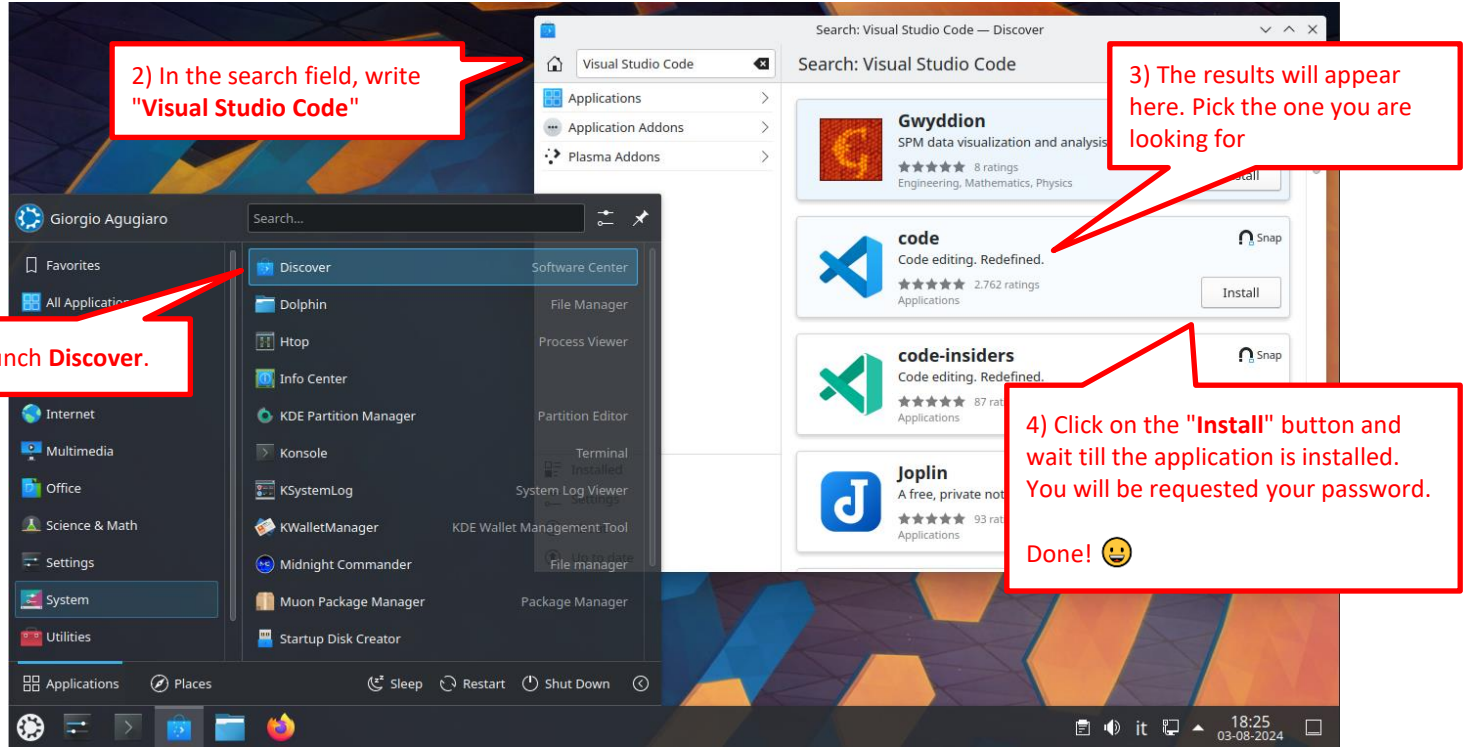
Finally, if you want to remove/uninstall a software package (e.g. **mc**), you simply type:

```
sudo apt remove mc
```

Installation from a repository

Example 2:

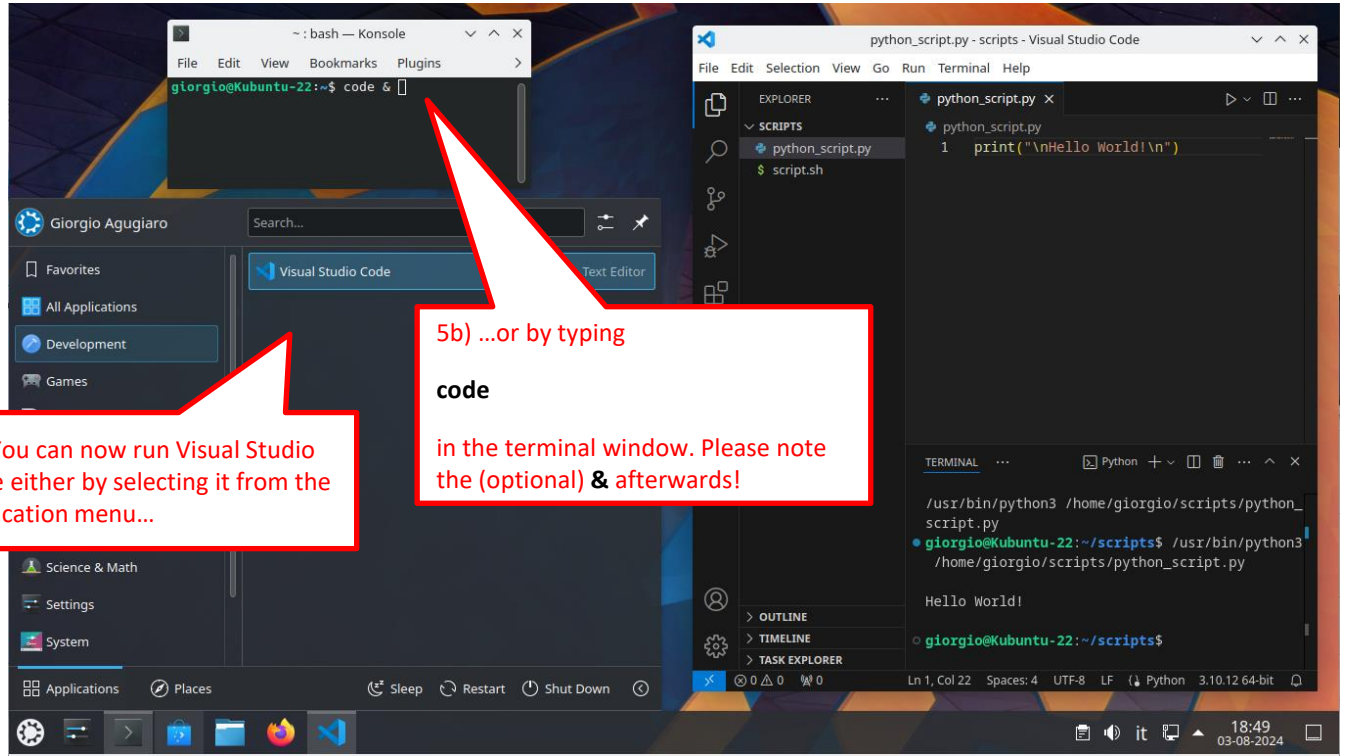
Let's assume we want to install **Visual Studio Code**. But you do not know the exact package name. Let's use **Discover** in Kubuntu.



Installation from a repository

Example 2:

Launch Visual Studio Code



5a) You can now run Visual Studio Code either by selecting it from the application menu…

5b) …or by typing

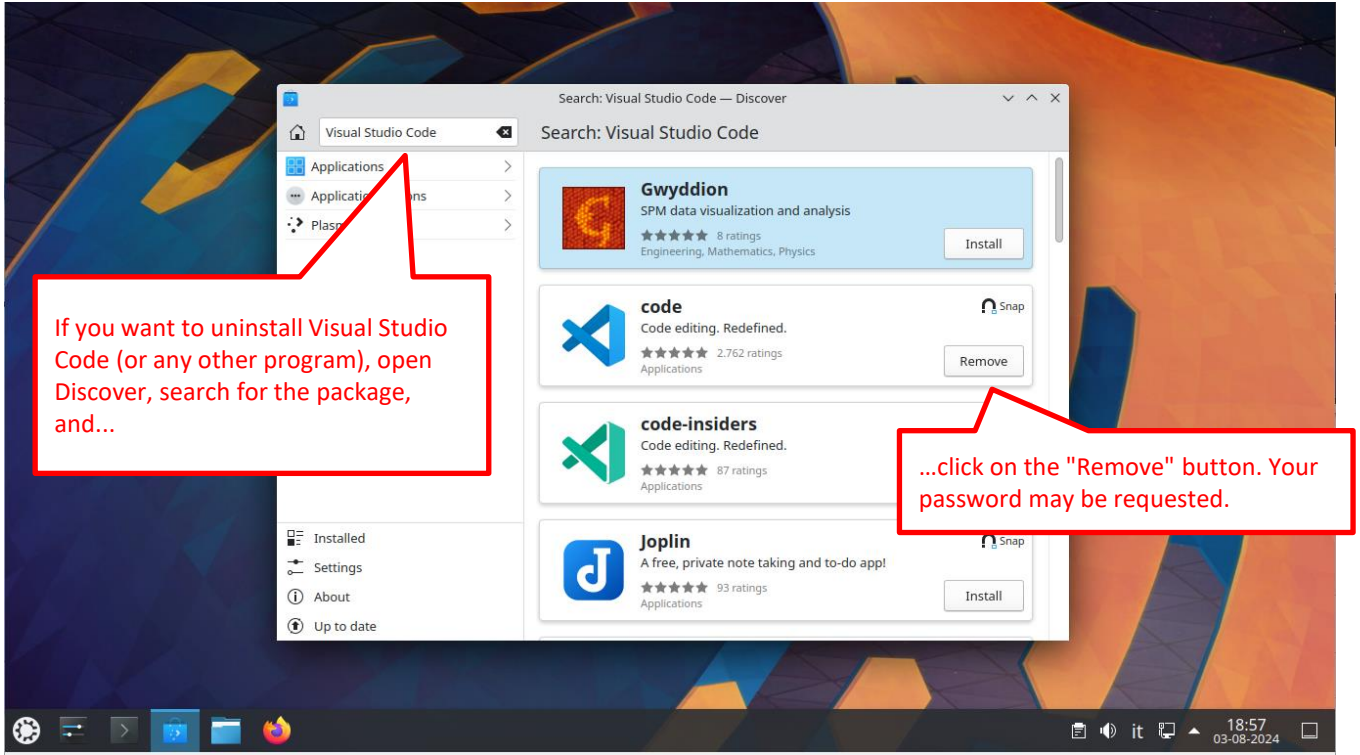
code

in the terminal window. Please note the (optional) **&** afterwards!

Installation from a repository

Example 2:

Uninstall Visual Studio Code



Index

Text editors

- A fundamental application that can't miss in a Linux machine is a **text editor**. As a matter of fact, there are several text editors for Linux. Some are based on the command-line, others exploit the GUI possibilities of the Desktop Environment they are part of
- In terms of command-line text editors, the two most well-known are **vim** and **nano**

- **Kate** is the text editor shipped with KDE (e.g. in Kubuntu), while **gedit** is the "equivalent" that comes with GNOME (e.g. in Ubuntu).

- But there are many more, even several extensions for **Visual Studio Code** that provide support for the Bash shell!

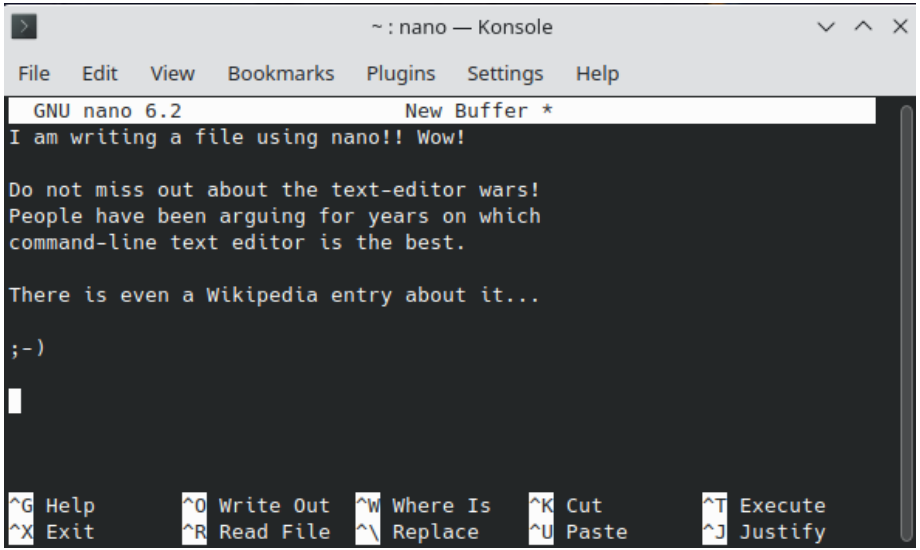
Further details: https://en.wikipedia.org/wiki/Category:Unix_text_editors

Text editors: Nano

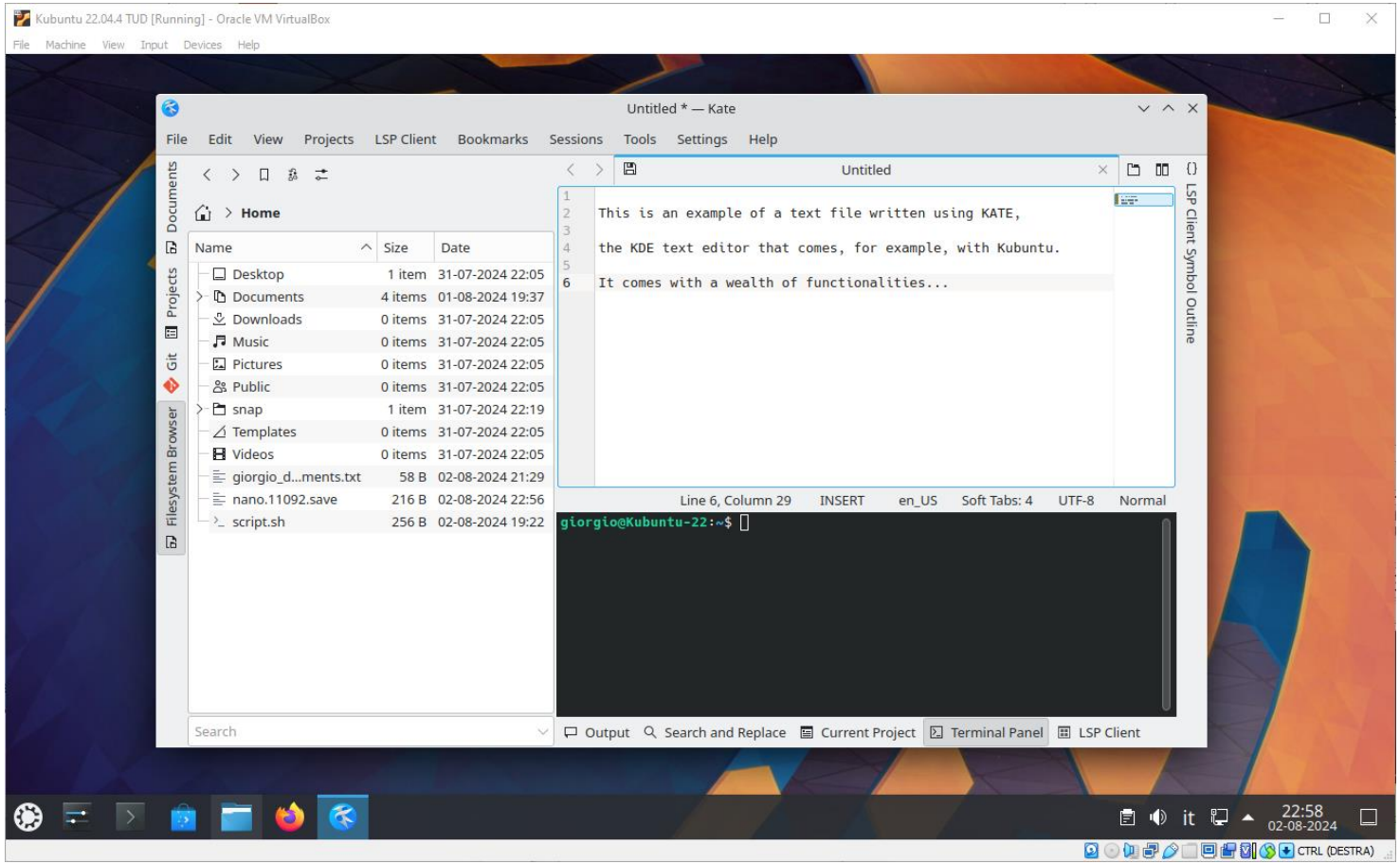
In the realm of command-line text editors, **nano** (more precisely: GNU nano) is less powerful than **vim**, but much easier to learn and use. It can be started simply typing `nano` or `nano <file_name>`. It provides a two-line shortcut bar at the bottom of the screen which lists the available commands.

Some of these commands are:

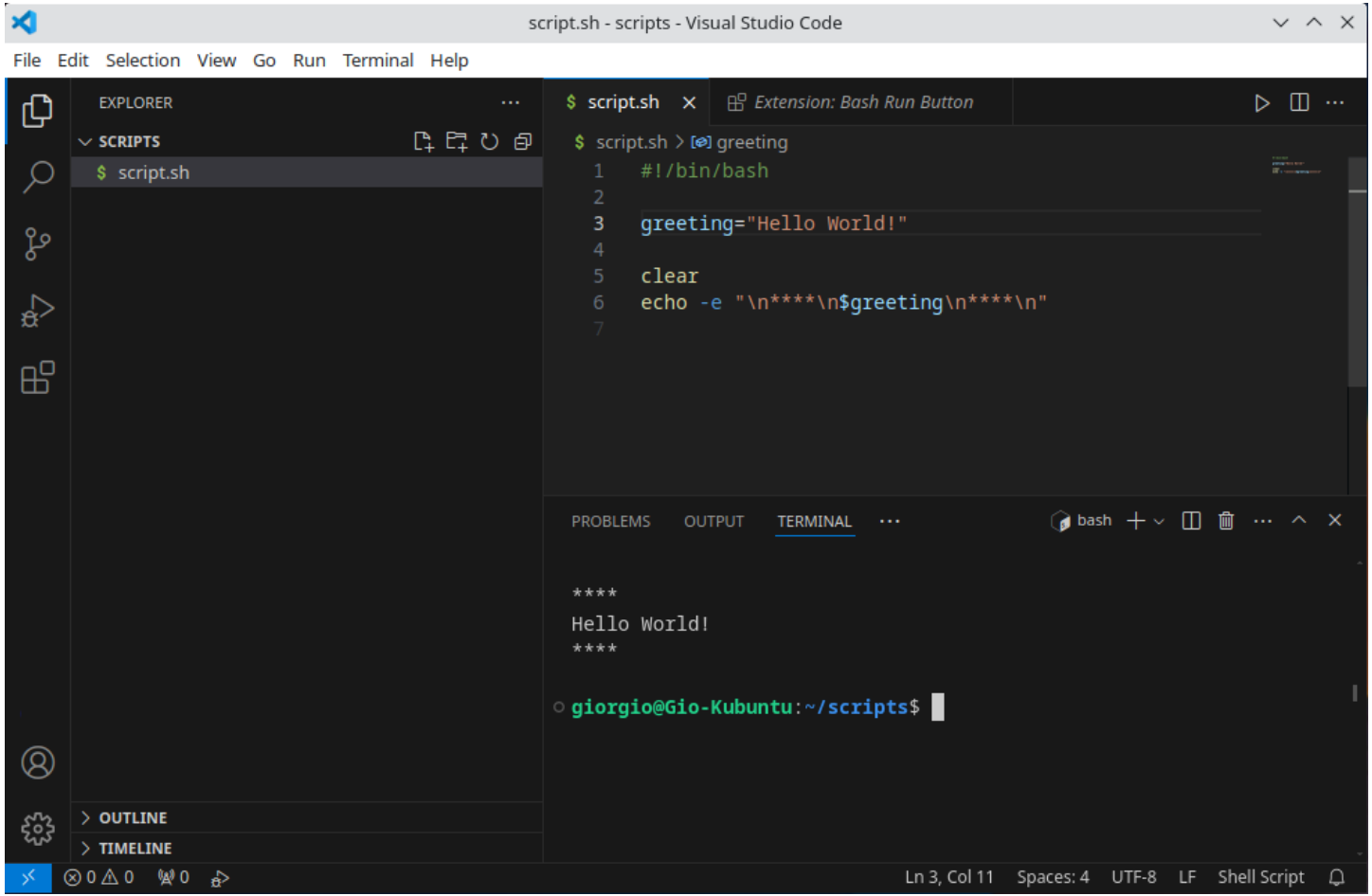
CTRL-R: Instert contents from another file to the current buffer
CTRL-G: Display the help screen
CTRL-O: Write to a file
CTRL-X: Exit a file
CTRL-C: Show cursor position



Text editors: Kate



Text editors: Visual Studio Code



Index

Shell scripting

- **Shell scripting** consists in turning a series of commands into a script that can be run as many times as needed Shell scripting is primarily meant to automate repetitive tasks, test solutions, and increase efficiency
- Examples of tasks that benefit from shell scripting can be performing backups of files, monitoring system resources, and managing user accounts
- A shell script is in essence a text file with a list of commands that instruct an operating system to perform certain operations
- A shell script must be read and interpreted by a shell program
- A shell script file has generally a .sh extension
- A shell script can be run in two ways:
  - As argument of the shell binary/executable file (e.g. /bin/bash)
    - Example: `/bin/bash ./my_script.sh`
  - As an executable file, which however must be made executable with `chmod u+x my_script.sh`
    - Example: `./my_script.sh`

Bash shell scripting

- Shell script files must begin with the so-called **shebang**. The first line of the script contains the absolute path to the shell interpreter. This is relevant especially when there exist different shells in the same machine
- The shebang is written as `#!/bin/bash` (for the Bash shell)
- The name comes from the combination of the terms sharp (#) and bang (!). Besides shebang, it is also known as sha-bang, hashbang, etc.

```
1  #!/bin/bash
2
3  echo "May the Bash be with you!"
```

Example of a simple shell script the just prints a message to the screen. The first line is the shebang

More details: https://en.wikipedia.org/wiki/Shebang_(Unix)

Bash shell scripting

Please note: Providing a full guide to shell scripting is beyond the purpose of this introdutory guide. Only the basics will be mentioned here. A good starting point is the open-source book "**Introduction to Bash scripting**" by Bobby Iliev, available also on GitHub at https://github.com/bobbyiliev/introduction-to-bash-scripting.

Comments:

Comments must be preceded by the **#** symbol. Example:

```
# This is a comments
```

Variables:

Variables are generally declared using the = symbol and no spaces before and after it. Variables are accessed using the **$** symbol, or (better) using also curly brackets **{}**

Example: **set_a_var.sh**

```
#!/bin/bash
my_var="Ciao!"
my_list="Luke Leia Anakin Obi-Wan"
echo "The value of my_var is: ${my_var}"
echo "The value of my_list is: ${my_list}"
```

```
giorgio@Kubuntu-22:~/scripts$ ./set_vars.sh
The value of my_var is: Ciao!
The value of my_list is: Luke Leia Anakin Obi-Wan
giorgio@Kubuntu-22:~/scripts$
```

Bash shell scripting

Variables: "Arithmetic expansion"

Bash allows for some maths on integers. For floating-point numbers, you can pipe the expression to **bc** (basic calculator).

`$(( expression ))` : This operator called "arithmetic expansion". It is used to perform some (integer-based) maths in Bash. **(())** evaluates the expression, **$** stores the result

Example:

```
a=2
echo "$((${a}**2))"  # prints a*a = 4

b=2.5
echo "${b}^2" | bc -l "  # prints b*b = 6.25
echo "$((${b}**2))"  # will cause an error: no integer!

i=10
echo "$((++i))" # will return 11
```

Syntax	Description
++x, x++	Pre and post-increment
--x, x--	Pre and post-decrement
+, -, *, /	Addition, subtraction, multiplication, division
%, ** or ^	Modulo (remainder) and exponentiation
&&, ||, !	Logical AND, OR, and negation
&, |, ^, ~	Bitwise AND, OR, XOR, and negation
<=, <, >, =>	Less than or equal to, less than, greater than, and greater than or equal to comparison operators
==, !=	Equality and inequality comparison operators

Bash shell scripting

User input: User input (e.g. from the keyboard) can be assigned to a variable using **read**.

Example: **origin.sh**

```
#!/bin/bash
echo "From which country do you come from?"
read country
echo "You come from ${country}"
```

```
giorgio@Kubuntu-22:~/scripts$ ./origin.sh
From which country do you come from?
SOKOVIA
You come from SOKOVIA
giorgio@Kubuntu-22:~/scripts$
```

Bash arguments: You can pass arguments to your Bash scripts. They can be accessed from the script using **$1,$2,$3, ..., $n**, with n the order they are passed. **$@** is a reference to ALL passed arguments.

Example: **fruit_salad.sh**

```
#!/bin/bash
echo "First fruit is $1"
echo "Second fruit is $2"
echo "Third fruit is $3"
echo "All fruits are $@"
```

```
giorgio@Kubuntu-22:~/scripts$ ./fruit_salad.sh apples kiwis bananas
First fruit is apples
Second fruit is kiwis
Third fruit is bananas
All fruits are apples kiwis bananas
giorgio@Kubuntu-22:~/scripts$
```

Bash shell scripting

Arrays: An array is initialised by assigning values separated by spaces and enclosed in round parentheses **()**. You can access array values in different ways.

Example: **arrays.sh**

```
my_array=("X-Wing" "A-Wing" "B-Wing" "Y-Wing")
echo "All spaceships: ${my_array[@]}"
echo "Indices of items are: ${!my_array[@]}"
echo "Number of items in the array is: ${#my_array[@]}"
echo "First spaceship is: ${my_array[0]}" # 0-index based!
echo "Second spaceship is: ${my_array[1]}"
echo "Last spaceship is: ${my_array[-1]}"
echo "First two spaceships are: ${my_array[@]:0:2}" # 2 is excluded
```

```
giorgio@Kubuntu-22:~/scripts$ ./arrays.sh
All spaceships: X-Wing A-Wing B-Wing Y-Wing
Indices of items are: 0 1 2 3
Number of items in the array is: 4
First spaceship is: X-Wing
Second spaceship is: A-Wing
Last spaceship is: Y-Wing
First two spaceships are: X-Wing A-Wing
giorgio@Kubuntu-22:~/scripts$
```

Bash shell scripting

Conditional expressions: the [[compound command and the [built-in command are used to test file attributes and perform string and arithmetic comparisons.

(Some) examples of file expressions:

`[[ -e ${file} ]]` : returns true if file exists

`[[ -d ${file} ]]` : returns true if file exists and is a directory

`[[ -x ${file} ]]` : returns true if file is executable

(Some) examples of string expressions:

`[[ ${string1} == ${string2} ]]` : returns true if the strings are equal

`[[ ${string1} != ${string2} ]]` : returns true if the strings are different

Bash shell scripting

Conditional expressions (ctd)

(Some) examples of arithmetic operators:

`[[ ${arg1} -eq ${arg2} ]]` : returns true if the 2 numbers are equal

`[[ ${arg1} -ne ${arg2} ]]` : returns true if the 2 numbers are different

`[[ ${arg1} -gt ${arg2} ]]` : returns true if arg1 is greater than arg2

`[[ ${arg1} -le ${arg2} ]]` : returns true if arg1 is less or equal than arg2

`[[ test_case_1 ]] && [[ test_case_2 ]]` : returns true if both cases are true (AND)

`[[ test_case_1 ]] || [[ test_case_2 ]]` : returns true if at least one of the cases is true (OR)

Examples of Exit status operators:

`[[ $? -eq 0 ]]` : returns true if the command was successful without any errors

`[[ $? -gt 0 ]]` : returns true if the command was not successful or had errors

Bash shell scripting

Conditional statements: The conditional expressions seen in the previous slides can be used to build conditional statements such as *if-then*, *if-then-else*, etc.

Example: **if_then.sh**

```
#!/bin/bash
correct_answer="Vienna"
echo "What is the capital of Austria?"
read answer
if [ ${answer} == ${correct_answer} ]
then
    echo "Correct!"
fi # please note the "fi" to close the statement
```

```
giorgio@Kubuntu-22:~/scripts$ ./if_then.sh
What is the capital of Austria?
Vienna
Correct!
giorgio@Kubuntu-22:~/scripts$
```

Bash shell scripting

Conditional statement (ctd)

Example: **if_then_else.sh**

```
#!/bin/bash
correct_answer="Glera"
echo "Prosecco wine is made with grapes named...?"
read answer
if [ ${answer} == ${correct_answer} ]
then
   echo "Correct! You deserve a glass of Prosecco! :-)"
else
   echo "Oh no, you seem to lack some basic knowledge :-("
fi
```

```
giorgio@Kubuntu-22:~/scripts$ ./if_then_else.sh
Prosecco wine is made with grapes named...?
Glera
Correct! You deserve a glass of Prosecco! :-)
giorgio@Kubuntu-22:~/scripts$ ./if_then_else.sh
Prosecco wine is made with grapes named...?
Sauvignon
Oh no, you seem to lack some basic knowledge :-(
```

Bash shell scripting

Conditional statement (ctd)

Example: **case.sh**

```
#!/bin/bash
echo "Enter the name of a Pixar movie:"
read -r  pixar_movie # -r allows to have spaces in input
case ${pixar_movie} in
 "Toy Story 5")
   echo "${pixar_movie} will be released in 2026"
   ;;
 "Toy Story" | "Toy Story 2" | "Toy Story 3" | "Toy Story 4")
   echo "${pixar_movie} tells the adventures of Woody, Buzz etc."
   ;;
 "Monsters Inc.")
   echo "With ${pixar_movie} you'll fall in love with Boo!"
   ;;
 *)
   echo "At the moment, I can't tell you anything about ${pixar_movie}..."
   ;;
esac
```

```
giorgio@Kubuntu-22:~/scripts$ ./case.sh
Enter the name of a Pixar movie:
Toy Story 5
Toy Story 5 will be released in 2026
giorgio@Kubuntu-22:~/scripts$ ./case.sh
Enter the name of a Pixar movie:
Toy Story 3
Toy Story 3 tells the adventures of Woody, Buzz etc.
giorgio@Kubuntu-22:~/scripts$ ./case.sh
Enter the name of a Pixar movie:
Monsters Inc.
With Monsters Inc. you'll fall in love with Boo!
giorgio@Kubuntu-22:~/scripts$ ./case.sh
Enter the name of a Pixar movie:
Wall-E
At the moment, I can't tell you anything about Wall-E...
giorgio@Kubuntu-22:~/scripts$
```

Bash shell scripting

Loop statements: In Bash there are loops, while-loops, and until-loops. Beware: you loop over lists! Arrays, if applicable, must be cast to lists.

Example: **loop.sh**

```
#!/bin/bash
fruits_list="apples kiwis bananas strawberries"
fruits_array=("mangos" "peaches" "apricots" "pears")
echo "*** Iterating over a list"
for fruit in ${fruits_list}
do
    echo "Printing: ${fruit}"
done
echo -e "\n"
echo "*** Iterating over an array"
for fruit in ${fruits_array[@]} # Array cast to list!
do
    echo "Printing: ${fruit}"
done
echo -e "\n"
```

```
giorgio@Kubuntu-22:~/scripts$ ./loop.sh
*** Iterating over a list
Printing: apples
Printing: kiwis
Printing: bananas
Printing: strawberries


*** Iterating over an array
Printing: mangos
Printing: peaches
Printing: apricots
Printing: pears


giorgio@Kubuntu-22:~/scripts$
```

Bash shell scripting

Example: **while_until_loop.sh**

```
#!/bin/bash
echo "*** Example of a while-loop"
counter=1
while [[ $counter -le 5 ]]
do
   echo "While-loop counter is: ${counter}"
   ((counter++))
done

echo -e  "\n*** Example of an until-loop"
counter2=1
until [[ $counter2 -gt 5 ]]
do
   echo "Until-loop counter is: ${counter2}"
   ((counter2++))
done
```

```
giorgio@Kubuntu-22:~/scripts$ ./while_until.sh
*** Example of a while-loop
While-loop counter is: 1
While-loop counter is: 2
While-loop counter is: 3
While-loop counter is: 4
While-loop counter is: 5

*** Example of an until-loop
Until-loop counter is: 1
Until-loop counter is: 2
Until-loop counter is: 3
Until-loop counter is: 4
Until-loop counter is: 5
giorgio@Kubuntu-22:~/scripts$
```

Bash shell scripting

Loop statements (ctd):

Inside a loop statement, you can use commands continue and break. With **continue** you can stop the current iteration of the loop and start with the next one.

Example: **continue.sh**

```
#!/bin/bash
for n in 1 2 3 4 5
do
  if [ ${n} == 3 ]
  then
    echo "Skipping value ${n}"
    continue
  else
    n_squared=$(($n*$n))  # $(( )) is called "arithmetic expansion"
    echo "Current value is: ${n}, its square is: ${n_squared}"
  fi
done
```

```
giorgio@Kubuntu-22:~/scripts$ ./break_continue.sh
Current value is: 1, its square is: 1
Current value is: 2, its square is: 4
Skipping value 3
Current value is: 4, its square is: 16
Current value is: 5, its square is: 25
giorgio@Kubuntu-22:~/scripts$
```

Bash shell scripting

Loop statements (ctd):

With **break** you can exit a loop if a certain condition is met.

Example: **break.sh**

```
#!/bin/bash
for n in 1 2 3 4 5
do
  if [ ${n} == 3 ]
  then
    echo "Skipping value ${n}"
    echo -e "\nExiting loop!"
    break
  else
    n_squared=$(($n*$n))  # $(( )) is called "arithmetic expansion"
    echo "Current value is: ${n}, its square is: ${n_squared}"
  fi
done
```

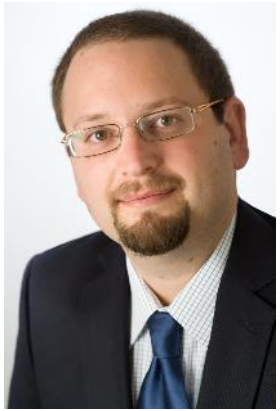
```
giorgio@Kubuntu-22:~/scripts$ ./break.sh
Current value is: 1, its square is: 1
Current value is: 2, its square is: 4
Skipping value 3

Exiting loop!
giorgio@Kubuntu-22:~/scripts$
```

Index

Further resources

- This Introduction is part of **TU Delft's GeoGeeks**
  - https://tudelft3d.github.io/geogeeks/
- "**Introduction to Linux**", by the Linux Foundation
  - https://training.linuxfoundation.org/training/introduction-to-linux/
- "**Linux Tutorial**", by GeeksforGeeks
  - https://www.geeksforgeeks.org/linux-tutorial/
- "**Introduction to Linux**", by CodeAcademy
  - https://www.codecademy.com/learn/introduction-to-linux



Dr. Giorgio Agugiaro
g.agugiaro@tudelft.nl
3D Geoinformation Group
TU Delft
The Netherlands
https://3d.bk.tudelft.nl/gagugiaro

Acknowledgements
Some parts of this document take inspiration and expand previous material by Clara García-Sánchez and Akshay Patil (TU Delft), originally developed for GeoGeeks.