Chapter 3 – Using the Unix command line

Even though a CLI shell may look scary at first, its original purpose really is to serve as a user-friendly everyday environment. Most shells offer a variety of quality-of-life features worth learning. Today we take a quick look at some of them.

1 Entering shell commands in the terminal

1.1 Auto-completion

Exercise 1 Typing is boring (and slow) but the shell comes with various ways of reducing this effort. In Chapter 1, we already used **TAB-completion** to avoid typing **command names** in full. This also works with **command arguments**: type `ls ~/` then press `TAB` instead of `ENTER`. Nothing happens just yet, because there are several possible completions, we have to press `TAB` again to see the full list of matches. Press `shift+D` to type an uppercase D, then press `TAB` once more, and observe that the list of candidate arguments is now reduced. Now type a lowercase `o`, and press `TAB` again. How many more letters do we need before TAB-completion correctly guesses `ls ~/Documents` ?

1.2 In-line editing

Unfortunately, the shell cursor (in the terminal window) and the mouse pointer (in the GUI desktop) are implemented by very different subsystems and they mostly ignore one another. For instance, clicking in the terminal window will typically *not* move the shell cursor.

Exercise 2 Still, keyboard shortcuts are provided to improve comfort when editing: the left/right arrow keys move the cursor in the current line; Ctrl+A and Ctrl+E move to the beginning and end of line, respectively; Ctrl+C discards the current line. Try those !

Exercise 3 **(optional)** If you want to become a programmer, then now is probably a good time to get familiar with the full list of shortcuts: `https://en.wikipedia.org/wiki/GNU_Readline`. Most of these are very useful in practice, especially the internal "clipboard" of the shell, available through Ctrl+K (cut from cursor to end-of-line), Ctrl+Y (paste), and also key sequences like ESC then "d" (cut one word forward), or ESC then BACKSPACE (cut one word backwards). Ask us for help !

1.3 Command history

Exercise 4 The up/down arrows navigate in your **command-line history** so that you can recall (and possibly modify) a previous command without having to type it again. Type `history` to display the full list, and notice the numbers in the left column. You can repeat any previous command using this "history number", for instance typing `!5` will repeat the fifth command in your history.

Exercise 5 You can also **search the history** interactively: press Ctrl+R to enter "reverse-i-search mode" then type a few letters. You can cycle through alternatives by repeatedly pressing Ctrl+R. When you found the right command, press ENTER to run it directly, or left/right arrow to edit it first.

1.4 Wildcards

A command line consists of a command name followed by zero or more arguments. To reduce the effort of entering long lists of arguments, the shell understands **wildcards** i.e. special syntax patterns which it expands to a list of matching filenames before running the actual command:

- an **asterisk "*"** gets replaced by any number of characters (except the slash "/"). For instance, "*cake" can stand for "cheesecake", "carrotcake", just "cake", etc (assuming that files with these names exist). Similarly, "doc/*.txt" will expand to the list of all files in subdirectory "doc" with a name ending in ".txt". You can also write "*/*.txt" to do the same in all subdirectories (each wildcard gets expanded separately)
- a **question mark "?"** matches any single character (again, except the slash "/"). For instance, command `mv ?.txt shorts/` will move all files named with a single character followed by ".txt" from the current directory to directory "shorts", while "??.txt" would match all files whose name consists of 2 characters followed by ".txt".

For more info please read https://en.wikipedia.org/wiki/Glob_(programming)

Exercise 6 Use wildcards to see how many header files (name ending in .h) in /usr/include have a name beginning with "std".

Exercise 7 How many subdirectories of /usr/include contain a file named stdio.h ?

Exercise 8 Wildcard expansion (aka **filename globbing**) is a feature of the shell itself, not of the individual commands. Observe how `ls /tmp/*` yields many error messages (there are "private" folders in /tmp that ls cannot read) but we can say `echo /tmp/*` just fine[1].
By the way, one can prevent wildcard expansion by enclosing the offending pattern in single quotes e.g. `echo '/tmp/*'`, or even just `echo /tmp/'*'`. The shell removes the quotes before running the command.

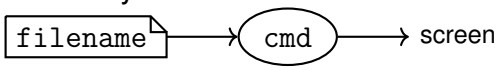
2 Paths and files

One of the genius ideas of Unix is that **"everything is a file"**[2]. More precisely, a lot of system features offer an interface that can be treated like a file, i.e. a stream of bytes associated with a path in the filesystem. This is true of reading and writing ordinary files on disk, of course, but also of network communications (aka sockets), inter-process communication channels, hardware peripherals (e.g. keyboard, screen) and many more.
Every process starts executing with three of these communication channels known as the three **standard streams**[3], namely **standard input (stdin), standard output (stdout)** and **standard error (stderr)**. By default all these streams are connected to the terminal: stdin reads text from the keyboard (this is how we type commands in the shell), and stdout/stderr both print text on the screen.

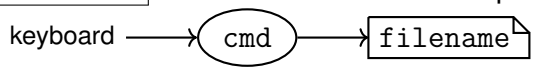
2.1 File redirection

Command-line shells fully embrace this "everything is a file" philosophy and provide many features to help us work with files. One of the most useful is **stream redirection** operators, which let us choose where standard streams are connected for a particular command:

- command `cmd < filename` will run cmd with no arguments but with its standard input reading from filename instead of the keyboard:



- similarly, command `cmd > filename` redirects the standard output of the process into some file:



[1] The echo simply prints its command-line arguments
[2] https://en.wikipedia.org/wiki/Everything_is_a_file
[3] https://en.wikipedia.org/wiki/Standard_streams

Less frequent, but still sometimes useful are the `>>` and `2>` operators :

- the `>>` operator is similar to `>` but it preserves the previous contents of the file and just appends new lines at the end, for example `ls >> files.txt`
- you can also redirect the standard error stream of any command with `2> filename`, for example `cmd argA argB > stdout.txt 2> stderr.txt`.

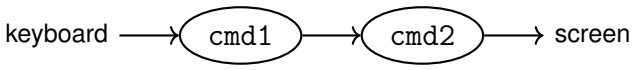
As illustrated by the last example, multiple redirection operators can be combined together, and everything is fully compatible with ordinary command-line arguments. For more info about file redirection syntax, see `https://www.gnu.org/software/bash/manual/html_node/Redirections.html`

Exercise 9 Play with these operators in simple commands and ask us questions until you feel comfortable with them. Remember: we've used file redirection in Chapter 2 already ! What was the exact command we used ?

Exercise 10 Invoke an ordinary command (e.g. `ls`) in such a way that it produces output both on stdout and stderr. Now run it again, but redirect both streams into distinct files. Check that the contents of the files, together, is the same as to what you see on screen with no redirection.

2.2 Pipes

Even more powerful, Unix shells allow for writing complex commands involving several programs cooperating together in a so-called **command pipeline**[4]. Writing `cmd1 | cmd2` will run both programs (in two distinct processes) with command1's stdout "connected" to command2's stdin:



This is useful because many Unix programs are designed to operate on lines of text within such a pipeline. Here are a few examples:

- `grep STRING` acts as a filter: it reads lines from stdin and prints (on stdout) all lines containing the given string.
- `grep -v STRING` does the reverse: it only prints lines which do not contain the string.
- `find PATH` searches all files and subdirectories recursively under a given path and prints their paths.
- `find PATH -name PATTERN` does the same but only prints filenames matching the given pattern. Warning: the pattern syntax is almost the same as shell wildcards, so you should use **single quotes** to tell the shell that it should not interpret the pattern, for instance `find /usr/include/ -name 'std*.h'` (try it !)
- `sort` reads all lines from stdin then prints them out in alphabetical order.
- `head -n 42` prints only the first 42 lines of input. Similarly, `tail -n 15` only prints the *last* 15 lines of stdin.
- `wc -l` reads all lines from stdin then prints a line count.

Exercise 11 Try each of the commands above once, and open their man page. Ask us for help if you're struggling.

Exercise 12 How many header files are there in total under `/usr/include` ?

Exercise 13 Still starting from there, how many files in total have the name `stdio.h` ?

Exercise 14 Run `seq 10` to print numbers from 1 to 10. Then, use pipes to filter the output of `seq 100` so as to keep only lines 40 to 50. Hint: you will need multiple pipes in a single command line.

[4] `https://en.wikipedia.org/wiki/Pipeline_(Unix)#Pipelines_in_command_line_interfaces`

2.3 Special files

You're already familiar with the `ls` command which displays the contents of a directory, i.e. its files and subdirectories. But by default, `ls` omits entries whose name starts with a dot, like `.bashrc` in your homedir. By convention, those are called **hidden files** aka dotfiles[5]

Exercise 15 In the man page of `ls`, find which option causes the listing to show everyting, even the dotfiles. Try it on your homedir.

Exercise 16 By default, the "*" wildcard omits hidden files too: type `ls ~/*` and observe that file `.bashrc` is not listed. Compare with `ls ~/.*` which shows only dotfiles. What `ls` command line should we type to list everything in a directory without using any option ?

Exercise 17 While doing the previous exercises, you may have noticed two hidden subdirectories with names "`.`" (pronounced "**dot**") and "`..`" (pronounced "**dot dot**"). These names exist for technical reasons, and they are here in every directory. The name "`.`" refers to the **directory itself**, and "`..`" refers to its **parent directory**. This is why we can type `cd ..` (cf Chapter 1) to move up the tree, and why we can type `./myprog` (cf Chapter 2) to run an executable from the current directory.
Guess what is the effect of command `cd ./../../../../` then run it to verify your hypothesis.

Exercise 18 Type `ls /dev` and observe that the kernel exposes many hardware peripherals (aka **devices**) as file-like objects. You can ignore most of those for everyday use. Most useful, however is the **`/dev/null` special file**[6], which acts as a data sink: all writes to `/dev/null` are simply discarded.
Redo exercise 10 but this time redirect `stdout` to be ignored.

2.4 Paths, `$PATH` and builtin commands

One of the key reasons for Unix's success lies in its modular architecture, which offers great extensibility. The system comes with many commands available by default, but adding new commands is as simple as writing new programs and then use them from the command line !

Exercise 19 The most obvious way to invoke a program from the shell is to type its path, spelled either as an **absolute path** starting from the root directory "/" (pronounced "slash") or as a **relative path** starting from the current directory (denoted as "`.`" (pronounced "dot").
Read our Chapter 1 again to remember where the `ls` executable is located, then invoke it using its absolute path, then using a relative path.

Exercise 20 (optional) Until now, we have been running commands like `ls` with no path information at all, how is that possible ? When no path is given, the shell uses a configurable list of directories[7] where to search for an executable with the correct filename. Type `echo $PATH` to see that list.
If you want to learn more about these **environment variables**, go read wikipedia[8] and/or ask us questions. Otherwise you can safely forget about it.

Exercise 21 Every process has its own **current working directory** aka CWD. When a new process is created, it inherits the CWD from its **parent process**. This is how `ls` knows what to show when given no argument. Try command `pwd` to print your current working directory.

[5] `https://en.wikipedia.org/wiki/dotfile`
[6] `https://en.wikipedia.org/wiki/Null_device`
[7] `https://en.wikipedia.org/wiki/PATH_(variable)`
[8] `https://en.wikipedia.org/wiki/Environment_variable`

Exercise 22 A process can freely change its own CWD, but there is no way to affect the CWD of other processes. For this reason, some shell commands (not a lot, but a few) can not be implemented as external programs nor executed in a distinct processes. This is the case for the `cd` command, for instance. To know how `cmdname` will be interpreted by the shell, you can use `type cmdname`. Do this with all commands we've learned about (`ls, date, cd, mkdir...`), and try to predict the answers.

3 Process control

Note: If you are using macOS or WSL, instructions involving graphical programs won't apply. You might want to use the INSA VDI "Bureau Virtuel" instead, or the Virtual Machine provided in Chapter 1, for the exercises of this section.

3.1 Basic process manipulation

Most shell commands we have used so far perform a simple task and terminate quickly. But the shell can be used to run any program in the system, even long-running applications like a web browser, text editor, etc.
We often say that typing a command will "start a program", but technically the correct term is "**create a process**". A process is *a program that is currently being executed*, including both its actual machine code but also its current *execution state*: all values of variables stored in memory, and the contents of the CPU (in particular the PC register, indicating "where we are" in the code). One of the main roles of the operating system kernel is to run many such processes at the same time. We will learn more about processes in Chapter 6.

Exercise 23 One first observation when running programs from the shell is that, if the program does not terminate right away, you "loose control" over the shell: try it with command `xclock -update 1`. By default, every command is executed **in the foreground** of the terminal: keyboard input is sent only to that process' stdin, preventing any further interaction with the shell itself. Regain control over the shell by pressing `Ctrl+C` in the terminal to **kill** the foreground process.

Exercise 24 Start `xclock` again, and in the shell window, press `Ctrl+Z` to **suspend** the `xclock` process. The shell then says something like ``[3] + 286397 suspended xclock -update 1''. You can now interact with the shell again, but the `xclock` process is paused. You can resume it (and loose control over the shell again) by typing `fg`.

Exercise 25 Alternatively, you can type `bg` to tell the shell to unpause the child process **in the background**. The shell now says something like ``[3] - 286397 continued xclock -update 1''. You can have your cake and eat it too: both the `xclock` process and the shell process run simultaneously side by side. Their standard output streams are both connected to the same terminal.

Exercise 26 We could have directly launched the `xclock` program in the background by appending an ampersand symbol & at the end of the command line. Try typing `xclock -update 1 &` in the shell, and observe that this both launches `xclock` *and* immediately shows the command prompt again.

3.2 Process identification

You may wonder what is the large number (in our case `286397`) which appeared in the messages above. This is the **Process Identifier**, or **PID**, a unique number that identifies every process in the system.

Exercise 27 Type `ps -aux` to get a **process snapshot** i.e. a list of all current processes. Pipe the output of `ps` into `grep` to find the PID of your `xclock` process.

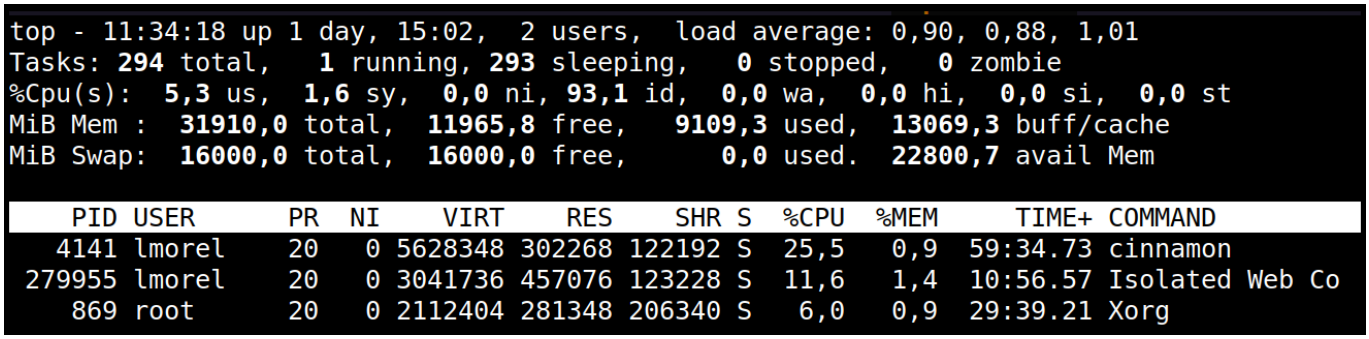


Figure 1: `top`: displaying the real-time view of a running system.

Exercise 28 Another very useful tool is the `top` command. It displays a real-time list of all processes, ordered by CPU usage (cf Figure 1 below). For each process, `top` shows (among other info) its process number (`PID`), the fraction of processor time (`%CPU`) and memory space (`%MEM`) being consumed by the process, and finally the command itself.

Exercise 29 As mentionned above, **processes can be killed** on demand. You already know about Ctrl+C, but that only works for the foreground process of the terminal. To have the kernel kill any process, you can use the `kill` command: find the PID of your `xclock` process then terminate it using `kill THEPID`.