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 chap. 1, we already used **TAB-completion** to avoid typing **command names** in full. This also works with **command arguments**: type `ls /home/gs` 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 list of matches. Press "a" then TAB once more, and observe that the list of candidate arguments is now reduced. How many more letters do we need before TAB-completion correctly guesses `ls /home/gsalagnac` ?

1.2 In-line editing

Exercise 2 Unfortunately, the shell cursor (in the terminal window) has no interaction with the mouse pointer (in the GUI desktop). Still, keyboard shortcuts are provided to improve our 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 so-called **wildcards**, 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. 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 /home/gs*` yields many error messages (because homedirs are private by default, so `ls` cannot read their contents) but we can say `echo /home/gs*` just fine. Note that **the `echo` command** simply prints its command-line arguments one by one.
By the way, one can prevent wildcard expansion by enclosing the offending pattern in single quotes e.g. `echo '/home/gs*'`, or even just `echo /home/gs'*'`. The shell removes the quotes before running the command. Try it !

2 Paths and files

One of the genius ideas of Unix is that **"everything is a file"**[1]. 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 communications, hardware peripherals (e.g. keyboard, screen) and many more.
Every process starts executing with three of these communication channels known as the **standard streams**[2]: **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 obvious is **stream redirection** operators, which let us control where standard streams are connected for a particular command:

- writing `command < filename` will run `command` with no arguments but with its standard input reading from `filename` instead of the keyboard.
- similarly, `command > filename` redirects the standard output of the process into some file.
- finally, the ">>" operator is similar to ">" but it preserves the previous contents of the file and just appends new lines at the end.

Exercise 9 Play with these operators in simple commands and ask us questions until you feel comfortable with them. Remember: we used that in chapter 2 already !

2.2 Pipes

Even more powerful, Unix shells allow for writing complex commands involving several programs cooperating together in a so-called **command pipeline**.[3] Writing `command1 | command2` will run both programs 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.

[1] `https://en.wikipedia.org/wiki/Everything_is_a_file`
[2] `https://en.wikipedia.org/wiki/Standard_streams`
[3] `https://en.wikipedia.org/wiki/Pipeline_(Unix)#Pipelines_in_command_line_interfaces`

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

Exercise 11 How many header files are there in total under `/usr/include` ?

Exercise 12 Still starting from there, how many files in total have the name `stdio.h` ?

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[4].

Exercise 13 In the man page of `ls`, find which option causes the listing to show everyting, even the dotfiles. Try it on your homedir.

Exercise 14 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 15 While doing the previous exercises, you may have noticed two hidden directories with names "`.`" (pronounced "**dot**") and "`..`" (pronounced "**dot dot**"). These names exist in every single directory: "`.`" refers to the **directory itself**, and "`..`" refers to its **parent directory**. This is why we can type `cd ..` (cf chap 1) to move up the tree, and why we can type `./myprog` (cf chap 2) to run an executable from the current directory.
Guess what is the effect of command `cd /../../../../` then run it to verify your hypothesis.

2.4 Miscellaneous

Exercise 16 Every process has its own **current working directory** aka CWD, which can be modified through the appropriate system call (this is what the shell does when we use the `cd` command). 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 explicit argument. Try command `pwd` to print your current working directory, then read its manual page.

Exercise 17 Type `ls /dev` and observe that the kernel exposes many **devices** as file-like objects. You can ignore most of those for everyday use. Most useful, however is the **`/dev/null` special file**[5], which acts as a data sink: all writes to `/dev/null` are simply discarded. Thus, we can redirect the standard output of any command to hide it from the terminal.

[4] https://en.wikipedia.org/wiki/dotfile
[5] https://en.wikipedia.org/wiki/Null_device

3 Process control

3.1 Basic process manipulation

Commands we have seen so far generally perform a simple task and terminate quickly. But commands can be used to start any type of program in the system, including long-term ones (e.g. a web brower, a text editor, etc.).
When we say that commands typed in the shell actually start programs, the exact term is *process*. A **process** is *an instance of a program that is currently being executed by system*. It generally includes, the program binary code and a region of the memory for the running program to store its data. Chapter 6 will further delve into the topic of processes. For now, suffice it to say that when typing the name of a program we want to execute, you actually ask the system to create a process for this program and execute that process.
One first observation when asking to run a program from the shell, is that, if this program does not terminate right away, you "loose control" over the shell.

Exercise 18 Type `xeyes`. This lauches a useless, yet funny, graphical program. In that state, we say that `xeyes` is running **in the foreground**, preventing any interaction with the shell.
You can regain control over the shell, by pressing `Ctrl-C` in the terminal window, which will **kill** the foreground process.

Exercise 19 Restart `xeyes`, and in the shell window, now press `Ctrl-Z` to **suspend** the `xeyes` program. The shell then says something like `‘‘[3]  + 286397 suspended  xeyes’’`. By doing so, you get control over the shell back, but the `xeyes` is not responsive anymore. You can resume it (and loose again control over the shell) by typing `fg`. Alternatively, you can force it to execute **in the background** by typing `bg`. the shell now says `‘‘[3]  - 286397 continued  xeyes’’`. You get your cake and eat it too: the `xeyes` now runs normally and you can still use your shell.

Exercise 20 We could have directly launched the `xeyes` program in the background by appending an ampersand symbol & to its name. Try typing `xeyes &` in the shell, and observe that this both launches the `xeyes` program *and* directly gives you control over the shell back.

3.2 Process identification

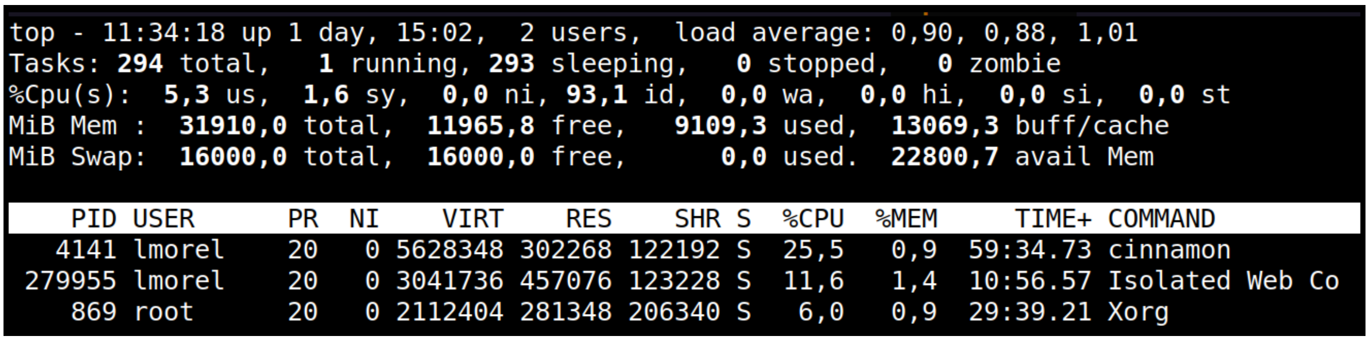
You may wonder what is the large number (in our case `286397`) that appears alongside the xeyes name. It actually is the **Process Identifier**, or **PID**. It's a unique number that identifies every running process in the system.

Exercise 21 Type command `ps` (*process snapshot*) to get a list of processes currently running on your system. This list is incomplete, but you can see your `xeyes` program there, associated with its `PID`.

Exercise 22 Another view of running processes is available through the `top` command. It displays a real-time of all processes in the system, ordered by CPU usage (cf figure 1 on the facing page) For each process, top gives its `PID`, `USER`, then a few numbers we won't get into now, then the fraction of CPU time (`%CPU`) and memory space (`%MEM`) consumed by the process. The name of the command that started the process is given at the far right of the line.

As mentionned above, **processes can be killed** on demand. Alongside Ctrl-C that can only be used to kill a process running in the foreground, you can use the `kill` command, which takes a `PID` as argument.

Exercise 23 Try it on the `xeyes` program.



```
top - 11:34:18 up 1 day, 15:02,  2 users,  load average: 0,90, 0,88, 1,01
Tasks: 294 total,   1 running, 293 sleeping,   0 stopped,   0 zombie
%Cpu(s):  5,3 us,  1,6 sy,  0,0 ni, 93,1 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
MiB Mem :  31910,0 total,  11965,8 free,   9109,3 used,  13069,3 buff/cache
MiB Swap:  16000,0 total,  16000,0 free,      0,0 used.  22800,7 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   4141 lmorel    20   0 5628348 302268 122192 S  25,5   0,9  59:34.73 cinnamon
 279955 lmorel    20   0 3041736 457076 123228 S  11,6   1,4  10:56.57 Isolated Web Co
    869 root      20   0 2112404 281348 206340 S   6,0   0,9  29:39.21 Xorg
```

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

4 Optional: shell scripting

Learn about shell scripting at `https://www.shellscript.sh/`

5 Optional: build automation with `make`

Learn about the `make` tool at `https://makefiletutorial.com/`