THEORY OF INFORMATION, ARCHITECTURE OF COMPUTERS AND OPERATING SYSTEMS (TIACOS)
Bioinformatics, ESCI
DAC, UPC

**ShellLab** **Spring 2023**

## Objectives

The objectives of this session are practicing with the basic Unix commands to deal with data from a shell. A shell is a software program that interprets and executes command lines. A shell script is a list of such command lines, written down together in a text file. Besides we will understand mechanisms to monitor kernel information associated to the active processes of the system.

## Skills

- Be able to use man pages.
- Be able to use the basic system commands to modify/navigate around the file system: cd, ls, mkdir, cp, rm, rmdir, mv.
- Know the special directories "." and "..".
- Be able to use the basic commands and programs of the system to access files: less, cat, grep, gedit (or another editor).
- Be able to modify the access permissions of a file.
- Be able to consult/modify/define an environment variable.
- Be able to use some special characters of the shell (command-line interpreter):
- `&` to execute a program in the background.
- `>` to store the output of a program (redirecting the output).
- | for inter process communication.
- Be able to program a basic shell script.

## Previous knowledge

This session doesn't require previous knowledge.

## Access to the system

This laboratory requires accessing to a POSIX terminal. Operating systems like GNU-Linux, macOS, GNU-Hurd, FreeBSD or Solaris have one. You can also use a MS Windows 11, if it has installed the Windows Subsystem for Linux. You can emulate a Linux system with a virtual machine, like VirtualBox or Vmware. If you are interested in this option, ask your instructor.

In this wording we suppose you are connected to a Linux system. To start, we will execute a shell or a command-line interpreter. A Shell is a program that the OS offers us to work in an interactive text mode. This environment could seem less intuitive than a graphical environment, but it is really simple and powerful. There are several command-line interpreters in the market, in these labs you will use Bash (GNU-Bourne Again Shell), but in general we will refer to it just as shell. Most of the commands that we will explain in this session can be consulted through the Bash manual (executing the command `man bash`).

To execute a shell, we only need to open a "Terminal" application. This application opens a new window (see figure 1) where a new shell is executed. The text that appears just before the blinking cursor is called prompt. It is used to indicate that the shell is ready to receive new orders or commands. Note: in the laboratory documentation we will use the `$` character to represent the prompt and indicate that what comes next is a command line (therefore, in order to try that command line you must NOT write the `$`, only the command that appears next to it).

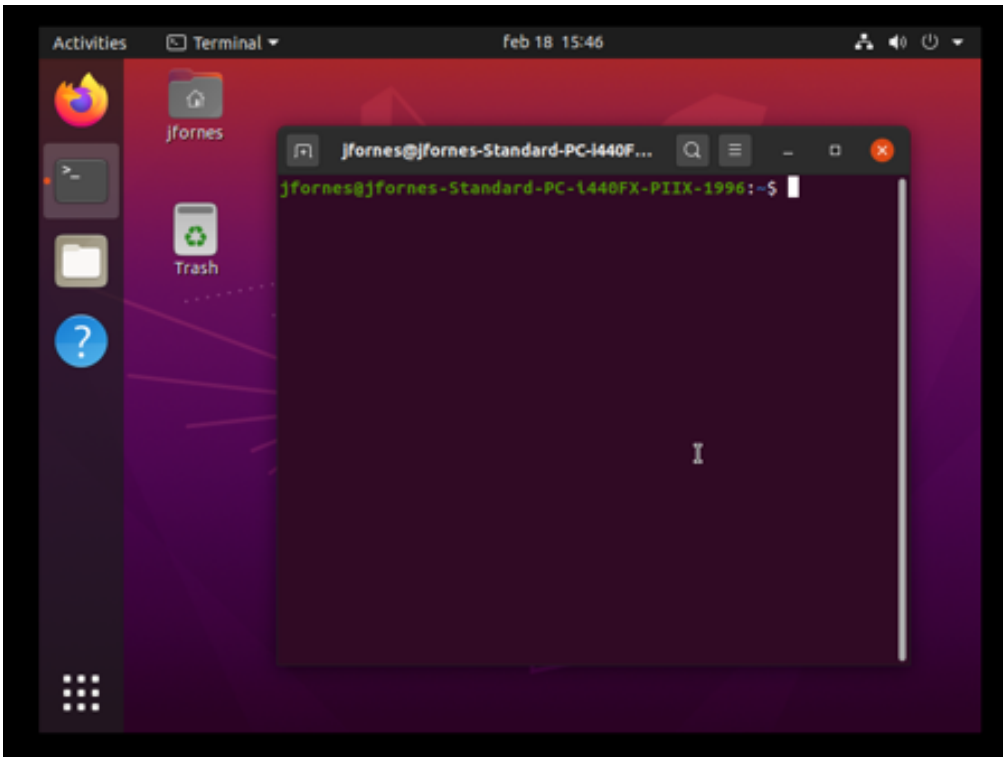


Figure 1: A terminal.

There are two types of commands: internal commands and external commands. The external commands are any program installed in the machine and the internal commands are functions implemented by the command-line interpreter (each interpreter implements its own, some of them

common and some of them particular to the interpreter).

## Commands to obtain help

In Linux, there are two commands that we can execute locally in the machine to obtain interactive help: the man command, which offers help about external commands (as part of the installation, the manual pages that we can consult using the man command are also installed), and the help command, which offers help about internal commands. Read the guide about "How to use the Linux man" that you have at the end of this section. Afterwards, query the man (`$ man` *command*) of the following commands (see 1). Specifically, for each command you have to read and understand perfectly: the SYNOPSIS, the DESCRIPTION and the options that appear under the "Options" column of the table.

| Command | Basic description | Options |
|---|---|---|
| man | Format and display the on-line manual pages | |
| bash | GNU Bourne-Again SHell | |
| ls | List directory contents | -l, -a |
| alias | Defines or displays an alternative name for a command | |
| mkdir | Make directories | |
| rmdir | Removes an empty directory | |
| mv | Move (rename) files | -i |
| cp | Copies files and directories | -i |
| rm | Deletes files or directories | -i |
| echo | Visualize a text (can be an environment variable) | |
| less | Shows files in a format suitable for the terminal | |
| cat | Concatenates files and shows them through the standard output | |
| grep | Searches text (or patterns of text) in files | -l |
| gedit | Text editor for GNOME | |
| vim | vim - Vi IMproved, a programmers text editor | |
| chmod | Modifies the access permissions of a file | |
| which | Finds a command | |
| tr | translate characters | -d |
| awk | pattern-directed scanning and processing language | -F |
| head | display first lines of a file | -n |
| xargs | construct argument list(s) and execute utility | -I |
| sort | sort or merge records (lines) of text and binary files | -u |
| wc | word, line, character, and byte count | |
| wget | The non-interactive network downloader | |
| curl | transfer a URL | |

Table 1: GNU Linux Commands

## Redirection

A Unix process reads from standard input (file descriptor 0) and writes to the standard output (file descriptor 1). Errors, if any, will be written to the standard error (file descriptor 2). By default, standard input points to the keyboard (terminal) and both standard output and standard error point to the screen (terminal). Before a command is executed, its input and output may be redirected using a special notation (redirection operator) interpreted by the shell. Redirection may also be used to open and close files for the current shell execution environment. If the redirection operator is `>` the redirection refers to the standard output. If the redirection operator is `<` the redirection refers to the standard input.

For instance:

`$ ls > dirlist`

This command line redirects the output of `ls` to a new created file named `dirlist`. If it existed, it will be truncated.

`$  ls >> dirlist`

The same, but now `dirlist` will not be truncated, but appended with the output of `ls`.

`$ grep root < /etc/passwd`

Searchs the string 'root' in the file '/etc/passwd'.

## Pipes

Brian Kernighan writes in his memoirs [**?**] that the idea of UNIX pipes should be pointed out to Doug McIlroy who has been pissing off Ken Thompson since 1964 asking him that UNIX programs should be able to attach "like garden hoses". Ken, who at first saw no way out of connecting programs, continued thinking of a solution that would silence Doug. In the end, well into 1972, he came up with the idea. Ken was able to add the system call in an hour, as the input/output redirection mechanisms were already in place. With the *syscall* implemented, Thompson added the mechanism to the shell and was amazed at the result.

The notation was simple and elegant. Ken Thompson and Denis Ritchie had to rewrite the code for all 1972 UNIX commands (which they did in one night) and along the way they invented the standard error output.

All in all it wasn't a very cumbersome task, but instead the result was one of UNIX's most amazing contributions. Now it was not necessary to create a new program for a new task, but to combine existing programs. For example, can you answer how many sessions you have open on jfornes? Nothing simpler:

```
$ who | grep jfornes | wc
```

The pipes have been on UNIX since then, despite some attempts to subsume them. A few years later, with the introduction of networks (and multiprocessors), other IPC mechanisms (*Interprocess Communications)* also appeared. 4.2BSD UNIX (August 1983) added `sockets`. Also, like pipes, sockets used the file input/output mechanism to communicate processes, using file descriptors. The big difference with pipes was (and still is) that sockets extended the IPC to process communication on different machines . But that is another story and will have to be told at another time.

Returning to our example (`who | grep jfornes | wc`). To understand the purpose of this *pipeline*, you need to go back to when the machines had a lot of users, connected from terminals (physical: green phosphor screen, keyboard and serial cable). The first program, `who`, displays the users connected to the machine. By default, they appear in a column and if, as is often the case on large machines, there are many open sessions, it is difficult to find what you want:

```
jfornes@sert-entry-2:~$ who
xavim    pts/0        2020-10-30 15:35 (gw-4.ac.upc.es)
marc     pts/1        2020-11-02 16:57 (pcbona-i-amargos.ac.upc.es)
alopez   pts/4        2020-10-09 10:48 (tmux(25651).%4)
xavim    pts/5        2020-11-02 17:38 (gw-4.ac.upc.es)
gvavouli pts/7        2020-10-23 09:26 (gw-5.ac.upc.es)
victor   pts/12       2020-11-03 06:47 (gw-5-vpn-i.ac.upc.edu)
alopez   pts/13       2020-10-09 10:19 (tmux(25651).%0)
alopez   pts/14       2020-10-09 10:21 (tmux(25651).%1)
alopez   pts/15       2020-10-09 10:22 (tmux(25651).%2)
antoniof pts/20       2020-11-02 10:54 (tmux(27966).%5)
victor   pts/22       2020-11-03 07:08 (gw-5-vpn-i.ac.upc.es)
isaac    pts/23       2020-11-03 08:32 (gw-4.ac.upc.es)
jfornes  pts/25       2020-11-03 09:17 (gw-5.ac.upc.edu)
antoniof pts/26       2020-10-29 12:03 (tmux(27966).%1)
antoniof pts/27       2020-10-29 12:03 (tmux(27966).%2)
jfornes  pts/28       2020-11-03 09:18 (sert-entry-1.ac.upc.es)
jfornes  pts/29       2020-11-03 09:19 (sert-entry-1.ac.upc.es)
antoniof pts/30       2020-10-29 15:06 (tmux(27966).%4)
```

The next program in the *pipeline*, `grep`, is used to search for regular expressions. The acronym for `grep` means **g**eneral sea**r**ch r**e**gular **e**xpression and **p**rint. Before pipes, we could have saved the output of `who` to a temporary file and then searched for the occurrences of the expression `jfornes` in the file:

```
$ who > fitxer_who.tmp
$ grep jfornes fitxer.tmp
jfornes  pts/25       2020-11-03 09:17 (gw-5.ac.upc.edu)
jfornes  pts/28       2020-11-03 09:18 (sert-entry-1.ac.upc.es)
jfornes  pts/29       2020-11-03 09:19 (sert-entry-1.ac.upc.es)
```

This would not solve the problem at all. As you can see, users usually open more than one terminal (they are no longer physical and each `pts` is a window with a separate shell). To know how many sessions a user has open we can use the command `wc` (**w**ord **c**ount) which shows how many lines, words and letters the entry has passem. However, the output of the previous `grep` should be saved:

```
$ who > fitxer_who.tmp
$ grep jfornes fitxer_who.tmp > fitxer_grep.tmp
$ wc fitxer_grep.tmp
 3      15     185
$ rm  fitxer_who.tmp fitxer_grep.tmp
```

But with the use of pipes, we can do all three command lines in one and save our temporary files.

```
$ who | grep jfornes | wc
 3      15     185
```

**The pipe is a one-way communication channel between processes**. It's a FIFO structure, everything that comes in comes out in the order of entry or stays in the pipe until some process drains the contents (and then disappears from the pipe). We don't have to worry (for now) about the storage capacity of the pipe [1]. All the characters that `who` writes to the pipe stay there until `grep` reads them. When you read them they disappear.

It should be noted that there are four processes involved in this command line; `who`, `grep` and `wc` are three sibling processes. His parent is the shell, `$`, who created both the processes and the pipes.

One more note. The result of the command line is not entirely appropriate. There are three numbers, when what we wanted to know was one, the first, which answers the initial question: how many sessions are open by jfornes? Would you know how to make the answer only the first number?

Of couse! Adding another pipe and a command. The `awk` command is very powerful, it contains a whole language. We will put here only the minimum necessary to solve the exercise:

```
$ who | grep jfornes | wc |  awk '{ print $1 }'
3
```

As you may have guessed, `awk` writes the first column that reaches you through the pipe. By default, the column separator is blank. If you want to change it, use the `-F` option.

One last example. This UNIX script prints the 10 most common words in the entry [2].

```
$ cat $* | tr -sc A-Za-z '\012' | sort | uniq -c | sort -n | tail  | pr -t -5
```

You can find out, with the help of the `man`, what each command does, but the part we are interested in right now is to understand how the pipes connect the task done by `cat` with the input of `tr` and so on until `pr` writes by standard output what it has read from the pipe, formatted in 5 columns.

## Bash scripting

A shell script is a list of command lines, wrote down together in a text file. A shell is, in fact, a complete interpreted language, with data types, flow control and functions. Let's begin with the "Hello world" program. Edit a text file[3], named "hello_world.sh" with this content:

```
echo "Hello world"
```

Save it and change its permission so that it can be executed (see `man chmod` for more info):

```
$ chmod +x hello_world.sh
```

And now, execute it:

```
$ ./hello_world.sh
```

[1] At all events, you can find the answer here: `man 7 pipe`

[2] I updated an example of Kernighan that already appeared in his book [Kernighan (1984)].

[3] You can edit with your favourite editor, but I prefer `vim`. Vim is a very powerful editor that has many commands. And it does not need to open a new window to run. See the manual, also you can run the `vimtutor` program.

So, you already have your first bash script. Now, you can write down in a file any of the command lines that you tested before. The extension ".sh" is just a convention[4] but it is not mandatory.

Your shell scripts can have arguments, they are saved, in order, in the variables `$1`, `$2`, ...

For instance, this shellscript (named "say.sh") adds "What !" to the first argument:

```
# say.sh
echo "What "$1"!"
```

A word beginning with # causes that word and all remaining characters on that line to be ignored. They are comments. So, here some examples of execution of "say.sh":

```
$ ./say hello
What hello!
$ ./say bye
What bye!
```

You can read the Bash manual to know more about variables, control flow, expressions and operators, and so on. However, you won't need much more to make this lab.

### Shell variables

There are several intrinsic bash variables that affect the bash script behavior. Here we list just a few:

| | |
|---|---|
| $# | Expands to the positional number of arguments in decimal. |
| $* | Expands to the positional arguments, starting from one. |
| $$ | Expands to the process id of the shell. |
| $HOME | Expands to the home directory of the current user. |
| $PATH | Expands to the search path for commands. |

### Control flow

You already know how to express a sequence of instructions; one commmand line after another.

Expressing alternative actions has this general form:

```
if command
then
       commands if the condition is true
else
       commands if the condition is false
fi
```

An example. This program writes the two arguments it receives:

```
# 2args.sh
if [[ $#<2 ]]
then
```

[4] It is customary but not required that makes easy to recognize a shell script.

```
   echo "Usage:␣"$0 "argument1␣argument2"
else
   echo "first␣argument:␣" $1
   echo "second␣argument:␣" $2
```

Finally, to express iteration, you can use the `while` loop:

```
i=0
while [[ $i<10 ]]
do
        # commands here
        ((i++))
done
```

Or the `for` loop:

```
for (( i=0 ; i<10 ;i++ ))
do
        # commands here
done
```

## Putting it together

You are going to program a shell script to retrive information of a chemical compound. In order to do this, follow the questions, for each one you have to program a short shell script. Eventually, all the shell scripts should be packed and delivered by Moodle. Each question begins with the name of the shell script you have to program.

1. `getProtByChebiId.sh`. Download the data file (using `curl`). By now, let's fixed the chemical compound to deal with. Download protein information about caffeine (CHEBI:27732)[5] from the EBI (European Bioinfomatics Institute). We want data about Caffeine in CSV format (**Co**mma **S**eparated **V**alues)[6] that has this url (just one line):

   ```
   https://www.ebi.ac.uk/chebi/viewDbAutoXrefs.do?
   d-1169080-e=1&6578706f7274=1&chebiId=27732&dbName=UniProt
   ```

   Analize the construction of the url. Choose one of these commands: `wget` or `curl` to download the .csv file. Program a shell script named "getProtByChebiId.sh" that accepts one argument (the chebiId). The shell script has to download the .csv file of the chebiId and write its contents to the standard output. Check that it works for differents chebiIds, such as 27732, 16040, 27958 or 27518.

   Example of execution:

   ```
   $ ./getProtByChebiId.sh 27732
   ```

   The downloaded file must have 77 lines, similar to this one:

[5] CHEBI means **C**hemical **E**ntities of **B**iological **I**nterest and 27732 is the identifier of the entry in CHEBI describing caffeine.

[6] CSV is a common data exchange format that is widely supported by scientific applications, see wikipedia.

```
Q9UTN3,Poly(A) RNA polymerase cid14,DE
```

2. `getProtByChebiId-d.sh`. Your shellscript will download the corresponding file and after that will write to the standard output only the lines that have one of these patterns in the third column: MISCELLANEOUS or DISRUPTION PHENOTYPE. You must use the command `grep` and a pipe.

3. `getProtByChebiId-dc.sh`. Now you need to select just the first column, the one that contains the protein identifiers. Add a pipe to the `grep` output from the previous script. Select just the first column and write the result to the standard output. You must use the command `awk`. Your shellscript should display a column with the protein identifiers. Something like that:

```
$ ./getProtByChebiId-dc.sh 27732
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  4678  100  4678    0     0   5725      0 --:--:-- --:--:-- --:--:--  5718
A2AGL3
B0LPN4
E9PZQ0
E9Q401
F1LMY4
...
```

4. `getProtByChebiId-dcl.sh`. Now you are going to download information about all of these proteins from Uniprot. In order to do that you must construct the url of the protein. The Url starts by `https://www.uniprot.org/uniprot/` followed by the name of the protein, and the data format extension. For instance, to download info about A2AGL3 protein the url will be https://www.uniprot.org/uniprot/A2AGL3.txt. Add a pipe to the output of the command `awk` of the previous script. You must use the command `xargs` to execute a `curl` or `wget` for each protein name.

5. `getProtByChebiId-dclp.sh`. This script will show for standard output the identifiers of the scientific papers published about the proteins. In this final bash script you must use again `grep` to search for the string "DOI" and `awk` to show only the column that has the DOI (Digital Object Identifier) of the paper. Add a pipe to the output of the previous script.

## Deliverable

Create a tar compressed file with all the scripts you have programmed. Add a file named "readme.txt" with the names of the group members.

```
$ tar -cvzf shellLab.tar.gz readme.txt getProtByChebiId*.sh
```

## References

[Kernighan (1984)] Kernighan, B. W., & Pike, R. *The UNIX Programming Environment*. Prentice-Hall, Inc., Englewood Cliffs, NY.

[Cooper (2014)] Mendel Cooper. *Advanced Bash-Scripting Guide*
https://tldp.org/LDP/abs/html/index.html