Training week: System

Master M1 MOSIG

2024

1 About text editors

Many text editors are usually available on a Linux distribution. Some of them are more specifically designed to facilitate code editing (providing functionalities ranging for simple syntax highlighting to advanced debugging features).

Bellow we list a few of them that you can try out.

- Text editors
  - **Gedit**: A simple text editor
  - **Vim**: A rich text editor. Supports many programming languages. Works mostly with keyboard shortcuts.
  - **Emacs**: A competitor of vim
- Integrated development environments
  - **VS Code** (or its `VSCodium` variant)
  - **Code::Blocks**
  - **Geany**
  - **Kate**

2 About man pages

Man pages are the usual way of getting access to documentation in a Linux system. The man pages provide documentation for Commands, System calls and Library functions (in particular for the C standard library). For instance, to access the documentation of command `ls`, simply use in a terminal:

```
man ls
```

Note that a command, a system call and/or a library function may have the same name. To deal with this issue, the man pages are divided into sections. The three main sections are: (1) the commands; (2) the system calls; (3) the library functions. To access the documentation for the system call `open`, use:

```
man 2 open
```

Some websites also host online versions of the man pages. See for example:

- `https://man7.org/linux/man-pages/`
- `https://manpages.ubuntu.com`
- `https://manpages.debian.org`

Note: We strongly advise you to read the english version (which is usually the original version) of the man pages. Indeed, the quality (accuracy and freshness) of the translations in other languages may vary.

3 About version-control systems: Git

A version-control system such as Git offers several features that can be very useful when working on projects and practical sessions. With git, you can:

- Keep track of the modifications in your code. This can help you to undo modifications that break your code and more generally to restore any intermediate versions of your code. This also allows you to easily identify the modifications that have been made between different versions.
- Share your code with others. You can give access to your projects to other persons which can simplify team work.
- Manage multiple versions of the same project through branches (`https://www.atlassian.com/git/tutorials/using-branches`), which can allow multiple persons to work on the same project in parallel without creating inconsistencies.

Last but not least, since Git services are usually hosted on remote servers, pushing your code on a Git server avoids losing your work in case of a crash of your machine and allows you accessing your code from different places easily.

Platforms for hosting projects such as `GitHub` or `GitLab` provide Git-based management of source code as their basic service. Creating private projects (projects that only you and your teammates can access) is supported both in `GitHub` and `GitLab` free offer.

We suggest you to create a project (on the platform of your choice) on which you are going to store the code created during this lab. After creating a project through the web interface, we recall that the main git commands are:

- `git clone` to get a working copy of a repository
- `git add` to select the modifications to be tracked by git
- `git commit` to save the selected modifications in the local copy of the repository
- `git push` to send the modifications present in the local copy of the repository to the remote copy
- `git pull` to bring the modifications made by others to the remote copy of the repository into your local copy.

4 Shell

The goal of this exercise is to practice with the shell. To this end, please write the commands required to run the following operations:

1. In your home directory, create a directory called `training_system`. In this directory, create another directory called `exercise4` and move into this directory.
2. In this directory, create a file `file1.txt` where you write a small text.
3. Count the number of lines in this file and display the result on the screen.
4. List the content of your home directory and store the result in a file `exercise4/file2.txt`
5. Verify experimentally how many of the last lines of a file the command `tail` displays by default. Of course, you should not count manually.
6. Count the number of lines including the word "warning" in the files with a ".log" extension in directory `/var/log`
7. Your previous command might display some error messages in the terminal. How can you get rid of these messages?
8. Still related to question 6 above: how to write the output of your command (the command that lists the lines with the word "warning"), including the error messages, to a file?

5 Fork/Wait

Write a program that executes n processes using `fork()`. Each child process should do a `sleep()` followed by printing its own pid to `stdout`. Then each child exits. What do you have to take care of to ensure that the program is terminated before you can start another program in the same terminal?

6 Fork/Exec

Write a program that creates 2 processes. The second process should be created via `fork()`. After the fork, the child should execute (via `execve` or another primitive of the same family - see `man 2 execve` and `man 3 exec`) the binary `/bin/ls`. The parent process should print a "hello world" message after forking. How many "hello world" messages will be printed?

7 Reading/Writing to a file

Write a C program that performs a copy of `file1.txt` (created previously) into a new file `file_copy.txt`.

To solve this exercise, use system calls: `open()`, `close()`, `read()`, `write()`, etc.

8 Simple Pipe

Write a program that creates two processes. Create a pipe and send the pid of the parent process to the child process. Let the child and the parent print the pid of the parent.

9 Drawing random numbers

(This exercise is a pre-requisite for the next one.)
We would like to write a program executed by n processes: each process picks a random number and prints it at the screen. Here is an example:

```
$ ring1 6
process pid 25387 node 2 val = 1430826605
process pid 25388 node 3 val = 48523501
process pid 25389 node 4 val = 822619539
process pid 25390 node 5 val = 1591287596
process pid 25385 node 0 val = 2047288621
process pid 25386 node 1 val = 1731323093
$
```

This program is composed of 6 processes. The number of processes should be a parameter of the program. The processes are identified by their `pid` (returned by `fork`), a number (the order of creation) and a random number (generated with `rand`). In order to have different sequences of random values in different executions, read carefully the man page of `srand` (`man srand`).

10 Pipes 2 – Determine the winner

The goal is to adapt the previous program in order to decide which process generates the biggest random number. To do this, we will use an election algorithm. We will interconnect the processes

with pipes so as to generate a ring topology. This means that *process 0* will be connected to *process 1* through a pipe, *process 1* will be connected to *process 2* through another pipe, etc. The last process *(n-1)* will be connected to *process 0*.

When the processes and the pipes are created, *process 0* will send its random value to *process 1*. *Process 1* will compare it with its proper random value and will send the bigger one to *process 2*. The other processes will work in the same way. At the end, *process 0* will receive the biggest value, as well as the information concerning the pid and the number of the process that generated it.

```
$ ring2 6
process pid 26603 node 1 val = 982695543
process pid 26604 node 2 val = 679092432
process pid 26605 node 3 val = 1441867048
process pid 26606 node 4 val = 1135529882
process pid 26607 node 5 val = 1906462017
process pid 26602 node 0 val = 208930720
the winner is 1906462017 pid 26607 node 5
$
```