

# Lab 2.3. Processes

## Objectives

In this lab, we will review the basic system functions for process management: scheduling policies, process creation, process groups, sessions, process resources and signals.

## Contents


- Environment Preparation
- Scheduling Policies
- Process Groups and Sessions. Process Resources
- Program Execution
- Signals


## Environment Preparation

Some of the exercises require superuser privileges to set some attributes of a process, like real time scheduling policies. Therefore, this lab should be done in a **virtual machine**.

## Scheduling Policies

In this section, we will study parameters of the Linux scheduler that allow getting and setting the priority of a process. We will see both the system interface and some important commands.

**Exercise 1.** The scheduling policy and priority of a process can be consulted with the `chrt` command. Additionally, the `nice` and `renice` commands allow adjusting the *nice* value. Consult the man page of these commands and check their behaviour by changing the *nice* value of the shell process to -10 and then changing its scheduling policy to `SCHED_FIFO` with priority 12. 

**Exercise 2.** Write a program that shows the scheduling policy and priority of the current process. Also, show the maximum and minimum priority values for the scheduling policy. 

**Exercise 3.** Execute the previous program in a shell with priority 12 and scheduling policy `SCHED_FIFO` as in exercise 1. What is the priority of the program? Are the scheduling attributes inherited?

## Process Groups and Sessions. Process Resources

Process groups and sessions simplify the management done by the shell, since it allows sending signals to a group of process (to suspend, resume, finalize...). In this section, we will see this relationship and will study the system interface to control it.

**Exercise 1.** The `ps` command is specially important to see the processes in the system and their status. Consult the man page and:

- Show all processes of the current user in extended format.
- Show all system processes, including the PID, PGID, SID and command line.
- Observe the PID, PGID and SID of processes. Which identifiers are shared between the shell and the programs executed by it? What is the identifier of a PGID when a new process is created?

**Exercise 2.** Write a program that shows the identifiers of the process: PID, PGID and SID. Also, show the maximum number of files that can be opened by the process and the current working directory.

**Exercise 3.** Usually, a daemon has its own session and process group. To guarantee that it is possible to create a new session and group, the process calls `fork()` to execute the daemon logic and create the new session. Write a template for a daemon (new process and session creation) that only shows its process attributes (as in the previous exercise). Also, a daemon defines a working directory, so set it to `/tmp`.

What happens if the parent process exits before the child gets its information (observe the PPID of the child process)? What if the child exits before (observe the status of the child process with `ps`)?

**Note:** Use `sleep()` or `pause()` to force the desired termination order.

## Program Execution

**Exercise 1.** Main functions for program executions are `system()` and the `exec` family. Write two programs, one with `system` and the other with the appropriate `exec` call, that executes a program passed as an argument. In each case, after execution, show the string “The command finalized execution”.

**Note:** Consider how the arguments must be passed in each case to ease the implementation. For example, what is the difference between `./execute ps -e1` and `./execute “ps -e1”`?

**Exercise 2.** Using the `exec` version of the previous exercise, and the daemon template of the previous section, write a program that executes any program as if it was a daemon. Also, redirect the standard streams associated to the terminal using `dup2`, so that:

- Standard output to `/tmp/daemon.out`.
- Standard error output to `/tmp/daemon.err`.
- Standard input to `/dev/null`.

Check that the process keeps running after exiting the shell.

## Signals

**Exercise 1.** The `kill` command can send signals to a process or process group by their identifier (`pkill` can do it by process name). Consult the man page of the command and the signals that can be sent.

**Exercise 2.** In a terminal, start a long duration process (e.g. `sleep 600`). In other terminal, send different signals to the process, checking its behaviour. Observe the exit code of the process. What is the relation to the signal sent?

**Exercise 3.** Write a program that blocks signals `SIGINT` and `SIGTSTP`. After blocking them, the program will suspend its execution with `sleep()` for the number of seconds specified in the environment variable `SLEEP_SECS`. After waking up from the `sleep` call, the process will show if it received the signals `SIGINT` or `SIGTSTP`. In the last case, it will unblock it, so the process will stop and will have to be resumed from the shell. Write a string before finalizing the program to check this behaviour.

**Exercise 4.** Write a program that sets a simple handler for signals `SIGINT` and `SIGTSTP`. The handler

will count the number of times the signal has been received. The main program will be in a loop that stops when the process receives 10 signals. The number of signals of each type will be shown before the program exits.

**Exercise 5.** Write a program that performs the scheduled removal of its own executable file. The program has one argument with the number of seconds to wait before removing the file. It will be possible to stop file removal if the signal SIGUSR1 is received.

**Note:** The main program can not be suspended with `sleep()`. Use the system functions to remove the file.