Operating Systems 2. Processes

In this practical activity, the student will become familiar with the routines of the UNIX Operating System concerning the creation and management of processes, and also discover the information provided by the Operating System about running processes. After its completion, you should be able to understand the basics of the Operating System in relation to the concept of process and its surroundings.

Suggested Shell commands: `man, ps (-l, aux...), top, kill`…
Suggested C functions: `fork, execvp, wait, exit, getpid, getppid, getcwd, chdir...`
Include (header files): `<unistd.h>, <sys/wait.h>, <sys/types.h>`

*** Note: Find out how these commands and C functions will help you in the development of this practical activity.**

1. Processes in the system and Shell handling.

Remember that a process is a program in execution, and it is identified on the system by an integer (process identifier - **PID**)).

(1) Making use of the appropriate Shell commands, determine the process ID of the Shell you are working on (*bash*), and which process the Shell depends on (the parent process of the Shell - **PPID**).

Execute a command in background that requires sufficient time.

For example: `grep - rIi file / > output.txt & > error.txt &`

Check the **pid** of this new process and who its parent process is (**ppid**).

(2) It is possible to find information about the processes running on the system within the Linux directory **`/proc`**. Check the contents of the directory for the bash Shell **`/proc/<PID of the shell>`**. Check the contents of the file "`status`". How many context switches did the Shell make?

(3) In the "**`fd`**" directory (inside **/proc/<pid>/**), there are descriptors of the files opened by the process. What descriptors are associated with the Shell? Find out what those descriptors mean. From another terminal, try to redirect the output of a command to the file "**1**" in that directory (for example: "`echo $HOME > 1`" within the directory **fd** of the Shell)

(4) Explore the contents of the following files inside **/proc/<pid>/**, what information do they contain?

- **`IO`**

- **`Sched`**

(5) A way of communicating with the processes running on the system is sending signals (**`kill`** command). Explore the possible signals that can be sent to a process. Execute a long-running process and send it a signal of completion. There are two signals that can end a process, which are the differences between them?

2 Process management using the C API

In this practical activity, we will make use of the processes API seen in theory. In order to use **fork, exec** and **wait** we need to include the library `unistd.h`, and for the **exit** macros: `sys/wait.h` and `sys/types.h`. We provide a short description of every function we will use:

- `pid_t fork(void)` – Create a new process. Returns 0 to the child process and the pid of the child to the parent. Returns - 1 if there are errors (managed with errno).

- `int execvp(const char *file, char *const argv[])` – Replaces the calling process with a new process. The first argument "`file`" points to the filename of the program binary to be executed. "`argv`" is an array of pointers to null-terminated "strings", being the first one the name of the executable and the rest are the argument list of the new program (input parameters). The last element of the array "`argv`" must be a pointer to `NULL`.

- `pid_t wait(int *status)` – Called from the parent to wait for any child to finish. The function returns the pid of the child that finishes, while the variable `status` stores the return value of the child's exit.

- `pid_t waitpid(pid t pid, int *status, int opciones)` – Same as above, but to wait for a particular process (pid = - 1 → behaves just like wait).

- `exit(int status)` – Called from the child, allows it to exit and return status to the father (if waited). The father can use the macros `WEXITSTATUS` and `WIFEXITED` to evaluate the return value (there are more macros, search in: `man wait (2)).`

- `pid_t getpid()` – To find out the pid of the process that calls it.

- `pid_t getppid()` – To find out the pid of the parent process of the process that calls it.

In this practical activity, we will design and develop a mini-shell that will allow us to launch other commands from their own interpreter.

1) Develop a command interpreter in C that is able to receive commands from the standard input, i.e. the keyboard.

The interpreter will show a prompt with the name of our shell and the current working directory. Example:

```
MISHELL: / home/user >
```

(2) The program must translate the contents of the entered command and verify if it is within the list of commands allowed.

The initial list will be the following (**`ls, cat, ps, help, pwd, cd, history`**)

At this point, the program will simply show if the command is on the list or if it is an incorrect command.

(3) Once checked if the command is accepted, the interpreter needs to run it in the system guaranteeing the passage of arguments.

The mini-shell must wait for the completion of the command and, once finished, display on screen the result of the execution, as well as any error message if needed. In this first attempt, implement the first 5 commands (`ls, cat, ps, help, pwd`).

[Try with the following executions: **`ls - la, ps - aux, cat - nA fichero.c`**]

(4) Add the command **cd** to the mini-shell to be able to move through the directories. You must accept absolute and relative paths.

[Check it using the mini-shell prompt and through the command **pwd**]

(5) Add the **history** functionality to your mini-shell at a basic level. Each command executed in the minishell should be stored in a text file within the user's HOME [/ home/usuario/mishell-historial.txt]

(6) Add the command **history** to the mini-shell. You should make a program that, given a history file, shows on screen each of the previously executed commands and assigns them a number in execution order. The history command should execute that program.

```
 MISHELL: / home/user > history

[1] ls - l

[2] cat fichero.c

[3] ps - edf

[4] cd...
```

(7) Add the option [-d] as an argument to the program **history**. This option should delete all the history.

(8) Add the command **repeats** to the mini-shell, which, passing the history line number as an argument, executes the corresponding command again.

```
 MISHELL: / home/user > repeats 2

cat fichero.c
```

3. Creating a Makefile

Develop a basic Makefile that helps you with the task of compilation. The Makefile should facilitate changing the compilation options, and should delete all the objects and compiled files through the clean command.