OS Practice 2:

Implementation of a Simple Command Shell in C

Introduction:

The shell is the primary text interface between the user and the kernel, as it allows users to request these services by launching programs, connecting them together, and controlling their execution. From an operating systems perspective, a shell is an user-level program that relies on system calls to perform its task.

The objective of this laboratory assignment is to design and implement a simplified command shell in the C programming language. Your shell will support the execution of individual commands, simple pipelines, and concurrent background execution. To reduce the complexity associated with parsing the traditional shell syntax, you will use a simplified syntax.

Assignment Description

You are required to implement a command-line shell that reads instructions from standard input and executes them according to a specified execution mode. The shell operates in a loop, repeatedly reading instructions from the standard input, interpreting them, and launching the corresponding processes. Execution continues until an explicit termination instruction is encountered.

Each command is introduced by a keyword that specifies how the command must be executed. This keyword appears on a separate line immediately before the command itself. Instructions are separated by newline characters, and no interactive prompt is required.

The shell must support **SINGLE** (normal) execution of single commands, **PIPED** execution of exactly two commands joined by a pipe, and **CONCURRENT** execution of background commands (only of single commands, not piped). A special keyword **EXIT** must be provided to terminate the shell.

The input syntax will consist of the execution mode (SINGLE | PIPED | CONCURRENT), written in a line, followed by a command in the next line, with possible CLI arguments separated by single spaces. In the case of piped commands, then it will have two lines, each one expressing a command. For this practice, you can assume that there are no extra spaces that you need to remove separating the commands.

Program flow:

The general execution flow of the shell follows a simple pattern, in an infinite loop:

- It will read a line, and determine the execution mode. If it is **EXIT** it will end the process.

- Then it will read the command and arguments in the next line. It will create a new process using `fork()`. The child process then replaces its program image by invoking `execvp()` with the parsed command and its arguments.
    - If it is a **PIPED** execution, it will need to read a second line and create a second process, as well as creating the `pipe` and use `dup2()`to connect both processes before the `execvp`.
- If it is not a CONCURRENT execution, use `waitpid()` to wait for the previous process (in the case of the piped, you will need to wait for both of them). Search in the linux documentation how to use `waitpid`, and how it is slightly different from the `wait` command seen in class.

Hints:

1) You need to use strcmp for comparing strings. Remember that read will also return the '\n'.
2) The command line must be parsed into a NULL-terminated array of strings suitable for use with `execvp`. You can do it yourself, or use the given function `split_command`
3) When reading from the keyboard with a large buffer, each read will return information in a line. However, when the standard input is redirected to a file you will need to use either a circular buffer, as in last practice to correctly split in lines.

Examples:

The following input illustrates the expected format and behavior of the shell:

```
➢ SINGLE
➢ ls -l /home
➢ PIPE
➢ ps aux
➢ grep root
➢ CONCURRENT
➢ sleep 10
➢ EXIT
```

Delivery instructions

Submit through gradescope adding all the integrants of the group. Here you have the instructions.
The submission requires to **exactly** follow the structure below:

- *src* folder, with all the sources. There can only be one C code with a main.
- compile.sh, a bash script which will contain the compile command (and no other commands).
- integrants.txt : the name and NIA of all the group members

Remember to remove all hidden files before submitting.
Non adherence to the submission instructions will result in point penalization.