Operating Systems
Course Project

Scope and goals

The goal of this programming project is to implement an **alternative Linux shell in C**. As seen in class, the shell is the part of the operating system that interacts with the user either via the command line or a graphical UI. Our shell will be called **imcsh** (short for *IMC shell*) and will accept commands from the user using the standard input (`stdin`). The shell will execute the provided commands from the user and print the result to the standard output (`stdout`).

Important: It is **not allowed** to call the standard Linux shell to implement the commands! (i.e. using the `system` function is not permitted).

IMCSH description

After executing the shell, a command prompt of the form `user@host>` will be shown on a new line in the terminal. The shell will accept the following syntax:

- `exec <program_to_execute>`: accepts a program to execute with its parameters and executes it by creating a child process. The program to execute will be a string (eventually with blank spaces in between if command-line parameters are needed). After termination of the child process, the shell will output the PID of the finished process in a new line.
  - Example: `exec ps -uax` will execute the command `ps -uax`.
  - **Hint:** use the `strtok` function to tokenize (parse) the input string.
- Modifier `&`: if this modifier is used at the end of an exec command, `imcsh` will execute the command in the background and continue immediately (without waiting for the command to finish). As soon as the process ends, it will output the PID of the finished process in a new line.
- `globalusage`: this internal command will show a string with details on the version of `imcsh` being executed. Example output: `IMCSH Version 1.1 created by <author(s)>`.
- Modifier `>`: if this modifier is used at the end of a command, `imcsh` will redirect the output as text to the file given after the symbol. If the file does not exist, the shell will automatically create it. If the file exists, the shell will try to append the new data to the end of the file. In this case, the only visible output in the console will be the PID of the process after termination.
  - Example 1: `exec ls -l > directory_output.txt` → Executes the command `ls -l` and writes the output to the file `directory_output.txt`.
  - Example 2: `globalusage > usage.txt` → Executes the command `globalusage` and writes the output to the file `usage.txt`.
  - **Hint:** Use the `dup2` system call (`unistd.h`) to redirect standard output (`stdout`) to a file descriptor.
- `quit`: quits the shell. If there are any running processes, the following question will be prompted to the user: `The following processes are running, are you sure you want to quit? [Y/n]`. A list of all currently running processes will follow. If the user enters Y, the shell will quit and all running processes will be terminated.

Submission and assessment

Deadline: Feb 2, 2025 23:59

Submission contents:

- Implementation in C including `makefile` (running the `make` command in the directory will create the executable).
- Technical documentation (in Word, Markdown or PDF format).

This assignment is worth 50 points, which will be assessed as follows:

- Working implementation of `exec`: 25 points.
- Working implementation of `&`: 10 points.
- Working implementation of `globalusage`: 2 points.
- Working implementation of `>`: 5 points.
- Working implementation of `quit`: 3 points.
- Clean code and documentation: 5 points.

All files will be enclosed in a single zip file.