Programming exam in C: Shell

Student:
José Vitor Velozo de Carvalho

September 29, 2022

1 Brief description

This file describes in a very objective manner what was implemented for the shell implementation exercise. Further details regarding the expected behaviour of the Shell can be found in `case_tests.csv`, in which the first column describes the user's input and the second explains how the program should behave. The shell was developed to work accordingly to the description provided in the document at the discipline's Google Classroom.

2 Introduction

The folder contains 3 main files: `func.h`, which stores the function signatures of `func.c`. This file stores a large number of functions utilized by `shell.c`, which is the main program. The folder also contains `redir_file.c`, a program that was developed to test redirections using `"<"`. There are also 3 command files to test batchfile behaviour and lastly, there is `case_tests.csv`.

3 Requirements

3.1 Interactive sequential execution and exit - implemented

The execution was implemented using `execvp()` which receives an `arg_data` struct passed to the function, that is responsible for creating a child process to execute the command. This way, the shell does not terminate after executing the command. Exit is handled by a custom shell handler in the main program. The maximum characters of a line of command is defined by `MAX_LINE`. If a line is longer than `MAX_LINE` - 1, the commands will be executed in the next interaction.

3.2 Batch, exit and interactive sequential execution - implemented

The batch execution executes differently from the interactive, as it stores an array of lines of command, which is limited to 40 lines. When all the lines are finished or the file passes an exit command, the program terminates. Lines of command are separated by line breaks, and the commands are separated by `";"`.

3.3 Batch and interactive execution, exit, and style parallel - implemented

For style parallel, a new process is created for every command. The order in which they finish execution is random, since they are closed after all of them are created, in a different loop. However, generally, faster processes finish earlier.

3.4 Batch and interactive execution, exit, style sequential and PIPE - implemented

Style sequential offers support for pipes, and the data for every pipe is stored in `pipe_arg_data`. The shell offers support for one pipe at a time, which means that chains of pipes are ignored and only the two first commands are executed.

3.5 Batch and interactive execution, exit, style parallel and PIPE - implemented

Just as style sequential, those pipes are stored in `pipe_arg_data`. In this case, the `waitpid()` system call is not called after the pipe's execution in the function.

3.6 Batch and interactive execution, exit, style sequential and redirections - implemented

The output of commands can be written to another file using `">"`. Redirection to commands using `"<"` reutilizes the logic of batchfiles to read the lines of a file line by line, but does not separate by `";"`. Redirections utilizing `">>"` append to a file's content. In order to execute any redirection, an executable must be called using `"./example"`, the redirection and the file's name. If multiple spaces are passed rather than a file's name, an error message is displayed.

3.7 Batch and interactive execution, exit, style parallel and redirections - implemented

Parallel redirection functions also do not have wait system calls in their methods.

3.8 Batch and interactive execution, exit, style sequential or parallel and history - implemented

History stores the last executed command in a runtime variable. Its initial value returns a `"No command"` output, but after execution of a last command different than `"!!"`, the history is updated for the next line of command. It supports up to 40 characters, and values longer than that are ignored.

3.9 Batch and interactive execution, exit, style sequential or parallel and background - not implemented

This functionality is incomplete, and its execution causes the output of the shell to behave badly. The usage of `&` executes a process but does not wait for it to terminate, because of this, new lines of command can be read and executed. However, the shell behaves badly after this. It also does not support the command `fg` nor does it print the specified format established by the document when sent to background. Overall, this functionality is very incomplete and poorly tested.

4 Error handling

Empty lines of command or lines made of only spaces are ignored by the shell. Spaces before and after `";"` are also allowed. `CTRL-D` terminates the shell and even if a batchfile does not have an exit command, the shell exits. If more than one file is passed to the shell executable, it prints an error message and then terminates.

5 General design

Lastly, a `Makefile` is used to compile all the C files in the directory, and their objects and executables should be removed using `make clean`.

The general design of the shell's flow of behaviour is controlled by the main function, while repeatable tasks are included in methods in `func.c`.

If the mode of execution is a batchfile, all of the inputs are stored in an array, line by line. When all of the lines are executed or exit is provided by a command, the program closes (in this case, by the custom commands handler). In interactive mode, those lines are defined by inputs and the shell continues executing until `exit` or `CTRL-D` is provided.

In order to execute a command, the shell also makes some validations. If the command is blank, the shell skips to the next position in the commands array. Else, the command is saved to a local copy of the command, which will be used in the end of the interaction to determine history. If the given command is history, then it substitutes the current string for the value stored in history. The type of the command is also determined. Lastly, there is a custom handler which validates style changes and exit.

After all these steps, depending on the selected style of execution, the commands will be sent to a specific function and be executed. Lastly, there is a loop which closes the possibly created parallel processes, some memory cleaning and freeing, the copy of the local copy of the command to history, and if a batch is being executed, the array position will be incremented.

6 Possible improvements and refactorings

Some of the possible improvements to the shell's functionalities would be to add background processes and change the current usage of parallel processes to threads, which could generate more organized outputs.

There are some blocks of code which could be reduced greatly by creation of more methods. Some of these methods also contain a lot of repetition, which could be refactored.