TP4 Mini-Shell

Ieva Petrulionyte, Nishith Puranik

Implemented shell functionalities

Mandatory

Here we add a table which gives a number of a question from the *tp4.pdf* document, lists the corresponding functionality and its designated test which can be found in the *tests* directory.

Question	Functionality	test
1	interpreting the command line	–
2	quit	1
3	exec	$2, 3$
4	$>, <$	$5, 6, 7$
5	errors	$11, 12, 13$
6	|	$5, 6, 7$
7	multiple |	8
8	&	$9, 10, 14$
9	zombies	4

Pipes and redirection

In our shell, the parent process does any necessary redirection before forking and executing a foreground command. That results in the child process inheriting the redirection. To illustrate this let's consider a pipe between two commands. In this case, the parent will:

1. set the current output to the pipe's input and current input to the pipe's output;
2. make *stdout* point to current output;
3. fork the first child process;
4. the first child will send its output to *stdout*, which points to pipe's input;
5. make *stdin* point to current input;
6. fork the second child process;
7. the second child will take its input from *stdin*, which points to pipe's output.

The main program differentiates between the middle and the last command in the command line. The parent will set the current output to the initial *stdout* or a given file if the command is the last one in the command line. The parent also saves the initial *stdin* and *stdout* in order to restore it at the end of a command line.

Bonus

Question	Functionality	test
1	ctrl-c, ctrl-z	$17, 18$
2	jobs	15
3	fg, stop	$16, 19$

Jobs

Our shell saves each new background job (child process identified by its PID) in the *bg_jobs* structure, continues to execute the next command, and reaps the background job after it terminates. All background jobs are reported on creation by displaying their *id* number and their PID. We changed the *readcmd*() function to detect & characters and identify background commands and created the *bg_jobs* data structure to store and handle background jobs.

Problems

There are two main problems in our shell, the first being some possibly unexpected behavior related to the *pause*() call when waiting for a foreground process to finish. When we only handled foreground jobs our shell would wait for every specific (identified by its PID) child process to finish before calling the next one. However, we had to introduce a $SIGCHLD$ handler for all child processes in order to properly implement background jobs. That meant that if we would wait for the foreground process to finish with a *waitpid*() function in the main program, it would also call the $SIGCHLD$ handler after the process has finished. The problem would arise when would try to get the foreground process PID in the handler. Since the child had already "reported" to the *waitpid*() call in the main function, we would not be able to get its PID in the handler and that would cause errors. That's why we chose to use the *pause*() call, which introduces a possibility of a background process finishing while the shell pauses to wait for the foreground command to finish, which could potentially lead to another foreground process starting while the previous one is still running.

The second problem, or a deviation from the standard shell, is the fact that a job in our shell is considered to be a background process. This is a result of our misunderstanding of the following phrase in the provided project description: "les commandes exécutées en arrière-plan s'appellent des jobs". We implemented the $jobs$, fg and *stop* commands only for background processes, not realizing that in a standard shell a job is both a foreground and a background process. Because of our misunderstanding, the *bg* command doesn't make sense in our shell, since with our definition of a job it would mean "bring a background process to background".

Conclusion

This project has helped us improve our understanding of main shell functionalities such as input and output redirection, pipes, background job control, and exit/stop signals. We painfully discovered that $printf()$ is a $non-async-signal-safe$ function, also learned the difference between $ctrl-c$ and $ctrl-z$ signals.