Rachel Ibihwiori
10/25/2023
Operating Systems
Project 1

Code Report

Introduction

The code provided is a basic shell program implemented in C. It allows the user to interact with the program as if they were typing in terminal because multiple commands have been programmed into C functions. The shell accepts user commands, executes built-in commands, and handles external command execution in child processes. This report will discuss the design choices and provide code documentation as requested.

Design Choices

1. Tokenization of User Input

The code utilizes a tokenize function to break down user input into individual tokens. It employs the strtok function to split the command line input into arguments, using an array containing various delimiters including whitespace as delimiters. When the function encounters the ‘$’ character, it separates the character from the word and replaces the current argument with it. Using array pointers allows you to change values as the program runs, which is really important in a program that is acting like terminal.

2. Implementation of Commands

The code includes built-in commands to provide functionality similar to shell. These commands are as follows:

Rachel Ibihwiori
10/25/2023
Operating Systems
Project 1

- excd = cd: Change directory. If no directory is specified, it changes to the user's home directory, if this can't be done an error is printed. If the directory is specified, then it takes that argument and runs it through chdir, which will return a null value and print an error if this is not possible/valid.
- expwd = pwd: Print the current working directory. It gets the current working directory using the argument in the getcwd function, stores it in a character array, and then prints the directory path.
- execho = echo: If given an environment variable this function prints its contents, otherwise it prints back the argument. This function is made possible by the function tokenize as it recognizes the '$' symbol and saves the argument as the result of running getenv on the input. If this weren't the case, it wouldn't be able to print environment variables properly.
- exexit = exit: Terminates the shell. Rather straightforward, but it's better to have a function for this so it is known to run this function when it receives the word exit. This was purely for formatting purposes when all of the if else if statements determine what the argument is asking for.
- exenv = env: Display environment variables. It first checks if 'env' is followed by anything in the argument, if the value at arguments[1] is null then it recognizes it as being a single env which just prints the environment. If there is a value following 'env' then a while loop starts until the word is null, where it saves the content of running getenv on the word and resets the argument to be this value. To achieve this the echo function was taken into consideration, as well as how tokenize works.
- exsetenv = setenv: Takes in pointer to arguments array. The argument is then sorted into two variables "variable" and "value" to respectively split the string by delimiter into the lefthand side of the argument and the righthand side of the argument that are separated

by an equal sign(=) with no spaces. This was important to do because the program doesn't normally sort items by delimiters such as '='. After this it either sets or overwrites the environment variable with the value received only if arguments at position 1 is not NULL. If arguments were to be null at position 1, then it would print and error message.

3. Child and Parent Processes

The code implements a forking mechanism to create child processes for executing external commands. This is done using a Parent Process function where the parent process waits for the child process to complete its execution. The function accepts two values: a pointer to an array and an integer value called is_background which acts like a boolean to show whether or not the program is a background process depending on the argument received in the command line contains an & symbol at the end. An integer was used instead of a boolean because in the standard C library boolean is not included, which I had discovered because I intended to use booleans first. If a command is intended to run in the background, the code returns the prompt while the program runs in the background. On the other hand, If a command is running in the foreground, it sets up an alarm signal handler for SIGALRM and uses an alarm to limit the execution time to 10 seconds. Lastly it checks if the program has exited and what it's exit status is, if it returns a 1 an error message is printed.

4. Signal Handling

The code includes signal handling for SIGINT (Ctrl+C) and SIGALRM to provide a clean termination of the current process and return to the shell prompt. For SIGINT the function created was signalhandler which prints the prompt when it receives Ctrl+C so that the entire shell doesn't quit but the program does. The signalhandler function is called before the first while loop in main, and this placement is imperative so that at any point during the program if

CTRL+C is used, the signalhandler will react as necessary. SIGALRM was implemented using the function alarmhandler that kills the process once it receives SIGALRM meaning that the alarm has gone off. This function was used in the ParentProcess function before the alarm line of code because the handler must be employed before it's usage, which I discovered after some trial and error on this component.

Turbulence

It should be noted that C acts differently than many other languages. For example, when I needed to use a boolean I learned that booleans don't exist in C on its own, for this you would have to import a boolean header file "#include <stdbool.h>". C does not allow for function overloading, so when creating the function exenv I had to use a creative approach. Function overloading is known from C++ as allowing a function to have multiple declarations with different inputs. I needed this for exenv specifically because when env is run by itself it has a different output than when env is followed by an argument. So instead of function overloading, I accepted one parameter and just checked for null values. In the future if I were needing to do this, I would accept multiple parameters and check for each one.

Conclusion

The provided C program serves as a basic shell with clear code documentation and built-in commands for common shell operations. The code adheres to best practices for readability and error handling. Although it provides a functional shell, there is no user interface, as specified. Further enhancements could be considered, such as adding support for advanced features like piping and redirection, as well as improving user feedback and error messages.