Miles Davies

@02921243

Introduction:

The provided code is a simple shell implementation in C, capable of executing commands, handling built-in commands, managing environment variables, supporting I/O redirection, and allowing background processes. The code is structured to provide a basic interactive shell experience, akin to a simplified Unix shell.

Design Choices:

Prompt and Current Working Directory:

* The shell prompt is dynamically generated to display the current working directory. This is achieved by using the getcwd function to obtain the current directory and updating the prompt accordingly.
* The update\_prompt function ensures that the prompt is always reflective of the current working directory.

Command Tokenization:

* The code uses a tokenization approach to break down user input into individual command arguments. This is achieved using the strtok function, splitting the input based on whitespace characters.

Built-In Commands:

* Built-in commands such as cd, pwd, echo, exit, env, and setenv are implemented. These commands provide core functionality to navigate the file system, display the current working directory, print messages, exit the shell, display environment variables, and set new environment variables, respectively.

Signal Handling:

* Signal handlers are implemented for SIGINT (Ctrl-C) and SIGALRM (timer). The SIGINT handler ensures that the shell does not exit when Ctrl-C is pressed, but rather returns to the prompt. The SIGALRM handler terminates a foreground process if it exceeds a set time limit.

Foreground and Background Processes:

* The shell supports both foreground and background processes. A background process is denoted by appending "&" at the end of a command. Foreground processes wait for the child process to complete, while background processes return immediately to the prompt.

I/O Redirection and Piping:

* The shell allows for basic I/O redirection using "<" (input redirection) and ">" (output redirection). Additionally, piping is supported using "|". These features enhance the functionality of the shell, enabling users to redirect input/output and chain commands together.

Error Handling:

* The code includes basic error handling for system calls such as fork, execvp, waitpid, open, dup2, and pipe. Error messages are printed to stderr to assist users in identifying issues.

Security Considerations:

* The code does not perform extensive input validation or sanitation, making it susceptible to certain types of attacks. For a production-level shell, additional measures would be necessary to ensure security.

Conclusion:

The implemented shell provides a foundation for a basic command-line interface with essential features. It allows users to interact with the file system, run commands, and leverage I/O redirection and piping. While the code is functional for educational purposes, a production-level shell would require additional features, security measures, and optimizations for performance.