Project #1 Report
CSCI 401 - Operating Systems
Khloe Wright & Maize Booker
10/25/2023

**Design Choices in Developing a Shell Program**

The development of a simple shell program is a multi-faceted project involving various design considerations to ensure functionality, user-friendliness, and robustness. This paper discusses the design choices made during the implementation of the shell program following the instructions provided in the assignment, which includes Task 1 through Task 5.

**Task 1 - Printing the Shell Prompt and Adding Built-in Commands**

**Displaying the Shell Prompt**

The design choice for this task was to create a shell prompt that provides users with information about their current working directory using getcwd(). The `getcwd()` function is imported from the ‘unistd.h’ file, which provides access to the our operating system’s API allowing us to “get” the current filepath. The function returns a string so the current working directory was easily printed along with the rest of the prompt using “%s%s” as the format specifier and without the need to store it in an extra variable.

**Implementation of Built-in Commands**

The design of built-in commands was a critical aspect of Task 1. We enabled the shell to interpret and execute a set of built-in commands, including cd, pwd, echo, exit, env, and setenv. These built-in commands provide essential functionalities for directory navigation, environment variable management, and interaction with the shell.

- $var Variables: The shell was designed to support variables in the form of $var. To implement this, the tokenization function was modified to detect strings that start with a $. When a string begins with $, the getenv function is employed to retrieve the variable value, which is then placed in the argument array. This design choice allows variables to be used with various commands, ensuring their universality and convenience.
- echo Command: The echo command was implemented with a design choice to run a loop over the command arguments. This loop is responsible for printing the string values, making it flexible and accommodating for user-defined messages and environment variable values.
- env Command: To implement the env command, access to the global environment variables array was required. The design choice was to include the declaration extern char **environ in the code, allowing the shell to access and iterate through environment variables efficiently.
- setenv Command: The setenv command was designed to set environment variables. To achieve this, the code uses the setenv function with overwrite=1, assuming that the assigned value is a single string token. This design simplifies the process of setting and managing environment variables within the shell.

**Task 2 - Adding Processes**

The core design choice in Task 2 was to extend the shell program with process forking, enabling it to execute external commands from the command line.

Process Forking: The primary design choice here was to implement process forking for external commands. When a user enters a command, the shell checks whether it is a built-in command. If it's not a built-in command, the shell forks a child process to execute the external command. This design ensures that both built-in and external commands can be executed within the shell environment.

**Task 3 - Adding Background Processes**

Task 3 introduced support for running processes in the background. The primary design choice was to enable background execution by checking for the `&` symbol at the end of a command line.

Background Processes: Users can initiate background processes by appending a `&` at the end of a command line. This design choice allows the shell to return to the prompt immediately for the next command while the background process runs independently. It adds significant flexibility to the shell, allowing users to execute parallel tasks efficiently.

**Task 4 - Signal Handling**

Task 4 allowed us to handle the attempt of termination of a foreground process using ‘CTRL-C’. We chose to implement a signal handler to catch SIGINT (‘CTRL-C’) and prevent the shell from quitting.

Signal Handling: A signal handler was introduced to catch SIGINT. When a user depresses `CTRL-C` to terminate a foreground process, the shell no longer quits. Instead, the signal handler allows the shell to return to the prompt, ensuring that the user's interaction with the shell remains uninterrupted. This design choice significantly enhances the robustness of the shell.

**Task 5 - Killing off Long Running Processes**

In Task 5, the design choice was to implement a timer that monitors foreground processes and terminates them if they exceed a specified time limit. The key design considerations were as follows:

Timer for Long-Running Processes: The timer was designed to terminate foreground processes if they run for more than 10 seconds and have not completed. To achieve this, the `kill` function with SIGTERM is used. If the process completes before the timer expires, the shell immediately returns with the prompt. This design choice prevents long-running processes from causing delays in the shell environment, ensuring that the shell remains responsive and efficient.

In conclusion, the design choices made during the development of the shell program have focused on creating a user-friendly and versatile environment. The shell can execute a variety of commands, manage environment variables, run processes in the background, handle signals, and ensure that long-running processes do not disrupt the user's experience. These design choices collectively contribute to a functional and efficient shell program.