CS3480 Principles of Secure Application Development

Programming Lab 1

Report

210250D

Jayasooriya D.D.M

Part 1

Introduction

The goal of this task was to create a program that can execute a series of commands separated by + symbols as specified in the command-line arguments. Each command runs in its own process using the `execve` system call.

Approach

- ★ When handling command-line arguments, the program separates commands using the '+' symbol as a delimiter and each command can have up to 8 arguments.
- ★ If a command is empty (two consecutive '+' symbols), the program substitutes it with `/bin/true` to ensure a valid command is executed.
- ★ For executing commands, the program creates a new process for each command and uses `execve` to execute the command in the child process.
- ★ The parent process then waits for all child processes to complete.

Challenges and Resolutions

- ★ Understanding how `fork()` and `execve()` works was important to carry of the task. This was achieved by https://youtu.be/cex9XrZCU14?si=hpdb0oKsc79XHSfB and https://youtu.be/OVFEWSP7n8c?si=OJ67IpCXluTKLPnK
- ★ Handling the separation of commands and their arguments was challenging at first. This was resolved by using `strcmp` and a nested array structure to store commands and their arguments. (https://www.programiz.com/c-programming/library-function/string.h/strcmp)
- ★ Use `wait` to ensure the parent process waits for all child processes to complete. Handle errors with appropriate messages.

Conclusion

The program successfully executes multiple commands in parallel, handling edge cases like empty arguments and incorrect paths. Thorough testing with different command combinations ensured robustness.

Part 2

Introduction

The task involved creating a tool that mimics the behavior of a shell, allowing the execution of repetitive commands with placeholders '%' replaced by subsequent user inputs. The tool supports commands with multiple placeholders and allows for flexibility in the number of inputs.

Approach

- ★ Parse the command line arguments to find placeholders (%) and save the command template.
- ★ Read inputs from the user, replacing each % in the command template.
- ★ For each input, create a separate process to run the command with the replaced placeholders.
- ★ Use `execvp` to carry out the command in each individual process.
- ★ Make sure the parent process waits for all child processes to finish.

Challenges and Resolutions

- ★ Determining how to replace multiple placeholders (%) correctly with user inputs and handling scenarios where inputs are fewer than placeholders. To resolve this, inputs were split into tokens and added to end of the common part of the command iteratively in each process. Ensure dynamic handling of inputs based on the number of % placeholders.
- ★ Reading multiple inputs from the user and correctly associating them with the placeholders in the command template.Use `fgets` to read entire lines, using `strcspn` and `strdup` to strip `\n` and `strtok` to split the line into tokens. This allowed handling commands with multiple placeholders efficiently. (https://www.geeksforgeeks.org/fgets-gets-c-language/ , https://stackoverflow.com/questions/15472299/split-string-into-tokens-and-save-them-in-an-array , https://www.geeksforgeeks.org/removing-trailing-newline-character-from-fgets-input/)
- ★ Managing multiple child processes and ensuring proper synchronization and error handling. Fork a child process for each input and use `wait` to ensure the parent process waits for all child processes to complete. Handle errors with appropriate messages using `fprintf`

Conclusion

The lab1p2 tool acts like a shell for running repeated commands with replaceable placeholders for inputs. It is designed to handle varying numbers of inputs and keep multiple processes in sync. We overcame challenges by carefully parsing, managing dynamic inputs, and handling processes robustly.