Building a Simple Shell

Operating Systems, COSC 4302

Instructor: Dr. Bo Sun

Project Members:

Andrew Ingram

Caleb Joiner

Andrew Kalathra

Introduction

The Operating System (OS) is sometimes referred to as the “system call interface”. As the OS gives services for function calls, its utility lies in its interactivity with the user. User level processes cannot be put on the system level to run, but the OS should not need to make an interface just for this purpose. Instead, there are “one or more command line interpreter programs that use the conventional system call interface to invoke OS services” (*Operating Systems*, 76-77).

The term “shell” is used to describe this command line interpreter. A simple shell, such as the one implemented in this project, should be able to take in simple commands and execute what is valid. The typical application of this is through the use of either threads or forks for execution of any new command/process. As described in *Operating Systems*, this is useful in that it allows the extra process to carry the risk of computation, in case the command caused any fatal errors (*Operating Systems*, 78).

Code Components

As this project was coded in C++, there will be a more focused discussion of components of the program in reference to the C++ language. Topics that easily defined in C will be discussed in C.

In part 1, we were tasked with using fork to create a child process to run the command. In part 2, we were tasked with using thread to create a parallel process to run the command. The main structure, however, is maintained between the two.

- A cpp file, shared.cpp, defines instructions and data shared between part 1 and 2.
- In part_1.cpp, we have a variety of functions to execute this shell.
  - The external files necessary are all listed at the top portion of the code. It should be noted that some of the libraries can only be found/run on unix-based systems.
  - The executeCommand function in both part 1 and 2 are prototyped before the main function so the main function will be seen first.

- part_1.cpp and part_2.cpp consist of the main function to each of the programs with executeCommand functions that execute the given command in another process/thread.
- A while loop is used to keep taking in user commands. This program can only be terminated by exiting out of while loop, which can only be done by entering the exit command of “quit”.
- The user is prompted to enter their command, which is then retrieved by the program.
  - A command can be a full pathname or just a relative pathname.
  - Full pathnames, or absolute pathnames, begin with ‘/’, and can be used to directly launch the execution (*Operating Systems*, pg 80).

- The command is then parsed into a component of a struct detailed in the shared.cpp file.
- Preceding this is where the command is checked. Special commands are checked here, and others move on.
    - If the command was “quit”, the program will terminate.
    - If the command was “cd”, the user is able to traverse directories
- Depending on which project was run, the following portion may be viewed differently. This, however, maintains the same algorithm.
    - If project 1, the child process is created here. The child process must then execute the command given. The parent process must wait till the child process finishes execution before continuing.
    - If project 2, the working thread is created here. The working thread is then tasked with executing the command given.
- If the user quits the program, an exit message is printed, and the terminal is then given control (the original shell).

This is the general overview of the program. However, some functions that could use a brief explanation are given below.

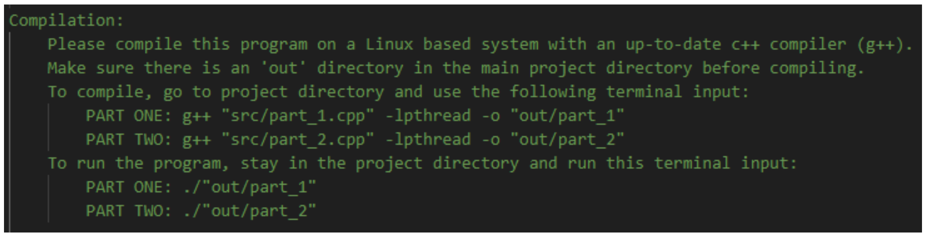
- parsePath() - this function gets and parses the path string by using ‘:’ as a delemiter/splitter. This function returns a vector string of all directories in the path string. This method is referenced in executeCommand(). This function references getEnv() (to get the PATH environment variable).
- lookupPath() - this function checks if the input is an absolute path, in which the input is returned. If not, this method either returns the command directory of the command or an empty string (signifying an error). This function is referenced in executeCommand(). This function does not make any major references.
- parseCommand() - this function parses the command input and pushes the commands individually into a vector string defined in the a struct which was defined in the shared.cpp header file. This function is referenced in the main function and references arguments,
- executeCommand() - this function executes the command. This function returns a success integer. This function is referenced in the main function (in either the child process or working thread). This function references parsePath() and lookupPath(). Special notice should be given as it uses pipe (which could create errors of its own).
- getWorkingDirectory() - this function gets the current working directory. This function returns a string of the working directory. This function is referenced in the main function (to change directories) and in printPrompt(). This function does not make any major references.

Compilation and Execution

This project can be found on our github:

https://github.com/InsomniaWins/operating-systems-basic-shell-project

Instructions to compile are listed in both part_1.cpp and part_2.cpp:



```
Compilation:
    Please compile this program on a Linux based system with an up-to-date c++ compiler (g++).
    Make sure there is an 'out' directory in the main project directory before compiling.
    To compile, go to project directory and use the following terminal input:
        PART ONE: g++ "src/part_1.cpp" -lpthread -o "out/part_1"
        PART TWO: g++ "src/part_2.cpp" -lpthread -o "out/part_2"
    To run the program, stay in the project directory and run this terminal input:
        PART ONE: ./"out/part_1"
        PART TWO: ./"out/part_2"
```

It is recommended to run the program on an updated Linux systems. The current versions (as of December 28, 2023) may not support this as certain required C++ libraries are absent (such as some GLIBCXX versions).

Conclusion

Shell's are very useful, not just in development of operating systems, but in customization of systems. This brief overview outlines the bare necessities of a shell program, in that input functionality is necessary, alongside an execution functionality. To protect the core operating system, the execution of a command cannot proceed unhindered, so additional checks must be done by utilizing other system files, such as environment variables. There are a variety of ways to accomplish this, however, so strictly adhering to the method shown in this overview is not necessary. The reader of this report should research other ways, perhaps even more effective ways, to make a shell than what was shown.