Lab Assignment 1: Shell and System Calls

Notes

You need to work on this project individually.
This project must be implemented in C.

Objectives

- To get familiar with Linux and its programming environment.
- To understand the relationship between OS command interpreters (shells), system calls, and the kernel.
- To learn how processes are handled (i.e., starting and waiting for their termination).

Overview

You are required to implement a command line interpreter or shell. The shell should display a user prompt, for example: `Shell>`, through which a user can enter for example, `ls l` command, as follows: `Shell> ls l`. Next, your shell creates a child process to execute this command. Finally, when its execution is finished, it prompts for the next command from the user.

A Unix shell is a command-line interpreter or shell that provides a traditional user interface for the Unix operating system and for Unix-like systems. The shell can run in two modes: **interactive** and **batch**. In interactive mode, you will display a prompt (e.g. `Shell>`) and the user of the shell will type in a command at the prompt. Your shell terminates when the user enters the `exit` command at the prompt. In batch mode, your shell is started by specifying a batch file on its command line. This batch file contains the list of commands (on separate lines) that the user wants to execute. In batch mode, you should not display a prompt, however, you will need to echo each line you read from the batch file (print it) before executing it. This feature in your program is to help debugging and testing your code. Your shell terminates when the end of the batch file is reached or the user types `Ctrl-D`.

Commands submitted by the user may be executed in either the **foreground** or the **background**. User appends an `&` character at the end of the command to indicate that your shell should execute it in the background. When a command is executed in the foreground, your shell waits until the execution of this command completes before it proceeds and displays the next prompt. When a command is executed in the background, your shell starts executing the command, and then immediately returns and displays the next prompt while the command is still running in the background.

Program Details

Your C program must be invoked exactly as follows:
`Shell [batchFile]`
where the `batchFile` is an optional argument to your shell program. If it is present, your shell will read each line of the batchFile for commands to be executed. If not present, your shell will run in interactive mode by printing a prompt to the user at stdout and reading the command from stdin.

For example, if you type the following `Shell /home/cs333/testCase1.txt`
your program will read commands from `/home/cs333/testCase1.txt` until `exit` command is read or end of file is reached.

Your shell should handle errors in a decent way. Your C program should not core dump, hang indefinitely, or prematurely terminate. Your program should check for errors and handle them by printing an understandable error message and either continue processing or exit, depending upon the situation.

The following cases are considered errors and you need to handle them in your program:

- An incorrect number of command line arguments to your shell program.
- The batch file does not exist or cannot be opened.

In the following cases, you should print a message to the user (stderr) and continue reading the following commands:

- A command does not exist or cannot be executed.
- A very long command line (over 512 characters).

These cases are not errors, however, you still need to handle them in your shell program:

- An empty command line.
- Multiple white spaces on a command line.
- White space before or after the `&` character.
- The input batch file ends without and `exit` command or the user types `Ctrl-D` as command in the interactive mode.
- If the `&` character appears in the middle of a line, then the job should not be placed in the background; instead, the `&` character is treated as one of the job arguments.

Hints

Your shell should be implemented as a loop that does the following at each iteration:

- In iterative mode, print a prompt and wait for user input. For example: `Shell>`
- Read the command line. Command lines typically have the following form:
  command [arg1] [arg2] ... [argN]
- Parse the command to identify the command name and its parameters.
- Execute the command. You need to do this by creating a new child process for the command. Creating a child process to execute the command has the following advantages: (1) it protects the main shell process from any errors that occur in the command that is being executed and (2) it allows easy concurrency; that is, multiple commands can be started and allowed to execute simultaneously.
- If the command is executed in the foreground, your shell program should wait for the child process, which is executing the command to finish.

You should keep a log file for your shell program such that whenever a child process terminates, the shell program appends the line `'Child process was terminated'` to the log file. To do this, you have to write a signal handler that appends the line to the log file when the SIGCHLD signal is received.

For writing your shell program, you might need the following hints:

- For reading input lines, check `fgets()`.
- For reading input files, check `fopen()`.
- Do not forget to check the return codes of routines for any errors. `perror()` can be useful to display the error.
- Use `fork()` to create a new child process.
- Use `execv()` to execute a command. You will need to pass the command and its parameters to it. The first argument is the path of the program to be executed. The second argument is an array of char* that represents the list of parameters of the command. Note that the array entry after the last parameter need to be set to `NULL` Note that `execv()` only returns if an error has occurred. The return value is -1, and `errno` is set to indicate the error.
- use `wait()` or `waitpid()` to force the parent process (i.e., your shell program) to wait for its child process to finish.

Testing

- Make sure that your code will run on the lab machines.
- Use our sample test cases to test your program. Try several test cases. First test that your program can start up simple programs (e.g. a calculator, a text editor). Then test it by running standard UNIX/Linux utilities like `ls`, `cat`, `cp`, and `rm`. Next, test passing parameters to them these commands. We will have many other test cases to test your program.
- Use a process monitor package to monitor your processes. Provide a screenshot for your shell parent process and some child processes spawned as background processes. Suggested packages: KSysguard or GnomeSystemMonitor

Deliverables

- Complete source code in **C**, commented thoroughly and clearly. We also need to submit a makefile so we can use it to compile your code.
- A report that describes the following: (1) how your code is organized, (2) its main functions, and (3) how to compile and run your code.
- Sample runs.
- Screenshots for the processes hierarchy in KSysguard (or any similar package) during the execution of your shell program.
- All deliverables are to be put in one directory named lab1_XX, where XX is your ID and then zipped.