ELEC 377 Lab 2: Write a Simple Shell – Description of Solution

Section 003, Group 16 – Thursday Lab

Erhowvosere Otubu, #20293052

Ivan Samardzic, #20296563

Date of Submission: October 17, 2023

Problem Solution

The purpose of Lab 2 is to develop a simple shell that emulates the functionality of a command-line interface. To be more specific, the group will write a shell that implements a few simple built-in commands that will aid in developing an understanding of the memory model underlying programming low-level operating systems services and modules, and allow for practice of good coding style. The primary objectives of this lab include the following: using C function pointers, practice using lower-level C memory representation, and using Linux system calls to retrieve system information and start a process.

Solution Approach

The following sections will explain in-depth the process of the group's solution.

Command Parsing

The first task requires separating the user's input stored in the *commandBuffer* array into individual words, each work being separated by a space character. The array *args* is an array of character pointers that will point to each of the words in the *commandBuffer*. The functions *skipChar* and *splitCommandLine* are the necessary functions used in this task. The *skipChar* function is designed to skip a specific specified character in the *commandBuffer* array (passed as the second parameter). If the character to be skipped is a space (' '), the function returns a pointer to the first non-space character in the array. If the character is a null character, it returns the pointer unchanged. The *splitCommandLine* function is responsible for extracting individual words from the user's input. It takes as arguments the *commandBuffer* array, the *args* array, the maximum number of arguments allowed (the length of the *args* array), and returns the number of words (i.e., the number of pointers in the args array). The *skipCommandLine* may call on *skipChar* and create a pointer that points to the first character in the input and stores the pointer in the args array. It then finds the first space after the marked word and changes it to the null character. This process gets repeated for as many words as there were inputted into the *commandBuffer*, if it is within the set max number of arguments. If not, a printed error message is displayed to the user and *skipCommandLine* returns 0 so the shell does not try to execute the command.

Implementing Internal Commands

The second step of this lab is to implement the internal commands handler; doInternalCommads, and the specific functions for executing each command; *exit, pwd, ls, and cd*.

The *cdFunc* function handles the *cd* (Change Directory) command. This command is used to change the current working directory in a shell. The *cd* command allows the user to navigate the file system and switch to a different directory. When the *cd* command is entered, the *cdFunc* function checks the number of arguments to ensure it's at most two. This command expects a zero or one argument. If an argument is provided, it changes to the specified directory. Otherwise, if an argument isn't provided, it changes to the home directory.

The *lsFunc* function handles the *ls* (List) command. This command is used to list the contents of a directory, typically the current working directory. The *ls* command provides various options for listing files and directories, such as displaying hidden files or sorting the output. The *lsFunc* function checks the number of arguments to determine the behaviour. If an agreement is provided, it lists the non-hidden files in the current directory. Otherwise, if the '-a' argument is provided, it lists all files including the hidden ones. To list the contents of the directory, the function uses the *scandir* function. If the '-a' option is not provided, the *filterFunc* function is used to exclude files starting with a dot ('.'). The function iterates through the list of directory entries and prints their names to the standard output. After listing the contents, the memory allocated for the list of directory entries is freed.

The *pwdFunc* function handles the *pwd* (Print Working Directory) command. This command is used to display the current working directory in a Unix-like shell. The current working directory represents the directory in which the user is currently located within the file system. The *pwdFunc* function checks the number of arguments to ensure that the *pwd* command is used correctly. If there are no other arguments (i.e., 'nargs' is 1), the function calls the *getcwd* function to retrieve the current working directory. The *getcwd* function allocates memory for a buffer that contains the current working directory path. This path is stored in the *cwd* pointer.

The *exitFunc* function handles the *exit* command. This command is used to let the user exit the shell. If the '*exit*' command is provided with an optional argument, it can also specify an exit status for the shell process. If there are too many arguments (more than two), it prints an error message to the standard error stream (stderr). If there are exactly two arguments provided, it converts the second into an integer

using the *atoi* function. It then exits the shell with the provided integer as the exit status. If no additional argument is provided (i.e., nargs is 1), it exits with a status of 0.

Lastly, in the main program, the function *doInternalCommand* is included to check if the user input corresponds to any of the internal commands. If it does, it calls the appropriate command-handling function. If it doesn't match any internal commands, it returns 0.

Implementing External Commands

The *doCommand* function is responsible for searching and executing external programs within the shell. It attempts to find and execute a program based on the command entered by the user. It is an essential part of the shell program since it allows it to execute external programs, resolve paths, handle errors, manage parent-child processes, and retrieve exit statuses. Its main significance is in extending the shell's ability to interact with a wide range of external utilities and commands, hence increasing the shell's versatility and usefulness.

Special Features Used

Some special features of the C language demonstrated in the code include the following:

- Functions and Function pointers: functions like *main*, *splitCommandLine*, *doInternalCommand*
- Header Files and Standard Libraries: header files such as *<stdio.h>*, *<stdlib.h>*, *<string.h>*, and others to access functions and data types defined in those libraries
- Pointers and Dyanmic Memory Allocation: pointers used for iterating over strings, managing memory, and storing directory entries. Functions like *malloc* are used to allocate memory for strings and arrays while *free* is used to free-up memory.
- System Calls: the code used system calls like *stat*, *chdir*, *getcwd*, and *execv* for file and directory operations, process management, and program execution.
- Error Handling: The code uses error handling by checking the return values of system calls and functions and printing error messages to the standard error stream (stderr)