How to duplicate shell functionality in C

Alexander Berger, Christian Hummel

1 General

This document intends to give an overview of the underlying concepts of a Linux shell through code examples provided in the C programming language. A shell is an essential part of any operating system that is needed for performing certain tasks. In the Linux operating system, the shell is where the main part of the interaction of user and system takes place. The first part of the user input consists of special keywords, also known as commands. Additionally, options and modifiers can also be provided by the user, to be able to request a more specific output format.

2 Implementation

2.1 C Setup

This program has been created using the C17 standard of C. To generate the executable file needed for the program to start, switch to the directory where the shell.c file is located and enter

make

into the command line. A file with the name “shell” will be the result. Enter

./shell

in the same command line interface and the program will execute. You will now be able to interact with the shell via keyboard inputs.

2.1.1 Input Format

The program responds to Linux commands e.g. ls -a, which will display the files of the current directory, including hidden files. More than one option can also be provided, if it starts with a hyphen (-). There are two different modifiers, to store the output of the command in a file, to run a resulting process in the background or a combination of both. The overall format of a system command to be executed by the C program is as follows:

exec <command>

which can be modified by adding one or more flags like this:

exec <command> -<option1> -<option2>

If the output should be stored in a file, the **>** operator , marked in red for demonstration purposes, will follow the command or the last flag to be executed and after this operator the path to the file with the corresponding ending, e.g. **.txt** or **.md**, has to be specified. The general format of this operation is:

exec <command> -<option1> > <filepath>

A process can also be executed in the background, which has to be marked with the **&** modifier at the end of the command. Each input, if written in the right format, will create a process, can be redirected to the background to immediately continue with following command inputs:

exec <command> -<option1> *&*

Storing the output into a file is possible to be executed in the background, if the previously mentioned formats are correctly applied like so:

exec <command> -<option1> <filepath> *&*

At last, there are two types of keyboard inputs, which do not need a prior insertion of *exec*.

The first one is *globalusage* which displays information about the current version of the program and the names of the authors.

Secondly, quit can be typed in to exit the program.

Almost every other input, will raise an Error which tells the user that the following command could not be found, similarly to the Linux command line interface.

If nothing gets entered the program will crash due to a segmentation error, caused by the function strtok().

2.1.2 Input Parsing

User input has to be transformed into the right input, to execute system commands correctly by the Program. This part of the program is responsible to check the format of the command and pass it in the right format to the *exec* function. Inside the main while loop, an iterative while loop ,which compares the first part of the input with *exec*, is introduced to check the input for essential components like *exec*, >, *and &*. With strtok(), the input gets tokenized and separated by the delimiter, which is a blank space in this case. Flag variables got defined as global variables to only store the commands needed for the execvp() function in an pointer to an array, since the size of the array depends on the size of the user input. The counter variable only gets incremented if the strings are to be stored as parts of the argument in ***argument_array.*** At the end of this section, the global flags get reset and replaced with temporary ones, that will get used in the main section, to avoid consistency issues if an *exec* call fails.

2.2 Exec

Main Idea

This part of the program inhabits the core functionalities. A while loop is created to continuously listen to commands. After temporary flags have been set by the parsing section, a child process gets created with every iteration by the fork() function and if conditions are utilized to point to different ways of command execution.

Every system call by the shell is made by calling the C function *execvp()*, which accepts two arguments. The first parameter is the command itself, and the second parameter is a vector of additional arguments, terminated with NULL at the end. Since the command has to be also included in the argument_array, the first index of the array is used as first argument and the complete array as second argument. In the corresponding conditional for the parent process, when no flag is activated, the wait(NULL) function will make sure that the output of the system call in the child process will be shown in the standard output before the process id of the child process is provided by the parent process. The iteration of the while loops always ends in the parent process and the user can enter a new command to be executed.

2.3 Modifiers

With the file modifier > and the background modifier &, the output of a command will be redirected through additional functionality of the shell, which is designed to duplicate the behavior of the same modifiers that can be used in a Linux shell.

2.3.1 File Modifier >

The parsing section will compare every token of the input string with this character and if it finds a match, the ***sfile*** flag will be flipped to indicate the child process that the output of a command has to be stored in a file. The filepath must be specified next to the modifier and will be stored in a character array.

In the child process, the file gets opened with the open function and if it does not exist, it will be created. If the file already exists and its path is correctly inserted into the command line, the program will append the output of the execvp() call at the end of the file. If the file cannot be created an error code of 1 will be returned. With dup2(), the standard output, which would be normally referred to by the execvp() function, gets redirected to the contents of the variable ***filepath***. The parent process will wait for the child process to finish and print the process id to the console.

2.3.2 Background Modifier &

Every child process sends a signal to the system, as soon as they finish executing. With a call of wait() in the parent process, the status of the child process can be extracted. It is also necessary to wait for child processes to prevent them becoming zombies, which are not active any more, but still occupy entries in the process table. To be able to track the termination of background processes, a signal handler for SIGCHLD is created, which checks for running processes in a loop and waits for their termination with a parametrized call of waitpid().

The parsing section will explicitly search for this modifier in an if condition, that also checks for this token to be in the very last position of the input string. The conditions in the forking section are comparing the flags for two possible cases.

In the first case, only the flag ***bg*** is active and **sfile** is not. The child process switches the process group with setpgid(), in order to enable this task to be continued in the background, and executes the system command with a call to execvp. In the parent process, there will be no call of the wait() or waitpid() function, the id of the child process will get appended to an array, which got initialized as a pointer to an array, and the parent process will terminate. The termination of the child process will be spotted by the signal handler function and after the output of the command, the process id will be printed by the handler in a new line.

In the second case, the flags ***bg*** and ***sfile*** are both active. The functionalities of the **>** and the **&** modifier will be combined to redirect the output of the execvp() call to a file and change the process group of the child process. Both cases refer to the same routine of the parent process, because in both cases the process id of the background task will get appended to the array ***bg_processes***, which keeps track of unfinished processes. The process id of the task will be printed out via standard output as soon as it terminates.

2.4 Quit

If a user enters

quit

the parsing section will compare that value to a constant and try to break the main loop to exit the program. If there are running processes, meaning the length of the array ***bg_processes*** is greater than 0, then the user will be asked if the program should exit and each running process will be shown in a new line. If the user enters

y or *Y*

every running process will be terminated by a call of kill() with the signal SIGKILL to forcefully terminate a process and the program will end. If the user provides an invalid input or inserts

n or *N*

to the console, the conditional will be exited and the while loop will be continued to ask for user commands.

3 Conclusion

This program is version 1.1, the first iteration of the development was concerned with a working implementation of the shell, and in the second iteration, the code got restructured to be more compact.

A different type of problems occurred in the process, as we tried to ensure best practices for better compatibility but it did not work out so far. This includes the way of initializing the function for signal handling and the set up in the main function for it. Sigaction should be preferred over the standard signal() call and it is also not safe to use printf() in a signal handler function. The compiler did not understand the proposed measures of defining the necessary struct data type for sigaction and the alternatively used write() instead of printf() inside of the ***handle_sigchld()*** function.

In relation to the implementation of the functions, the proposed functionality for the *quit* command is merely a concept, as the parent process for background tasks adds the process ids to an array, but the program does not update this array by itself after the signal handler catches the SIGCHLD signal for a terminated background process. Also, the implementation of creating background tasks might be flawed, due to a moderate complexity of the underlying concepts of signals and their correct application in C.

At last, the code structure can be further refined, e.g. through modularization, as this implementation is heavily focused on the correct implementation, which resulted in a tradeoff between functionality and cohesion of code. Overall, we still believe that this version represents a good foundation for further improvements.