International University of Sarajevo
Faculty of Engineering and Natural Sciences

2022 – 2023
Academic Year

Report

CS307 Operating Systems

15.12.2023.

Student: Nur Rustempašić 210302078,
Nedžla Šehović 210302011
Nejira Subašić 210302167

Contents

Introduction

The implemented shell provides a condensed but educational representation of a command-line interface, intended to provide users with a basic grasp of the fundamental elements required in shell programming. Users can enter commands into this shell, and the system will process and carry out these commands, giving users a practical introduction to how a simple shell environment functions. This shell's features and purposefully simplistic design act as an instructional aid, guiding users through the fundamentals of command execution, parsing, and integrating built-in capabilities.

The interpretation of commands and user interaction are crucial to the shell's architecture. Character by character, input is processed, and when a space or newline character is encountered, the shell recognizes that a command has ended. Although simple in nature, this parsing method establishes the foundation for more intricate parsing systems seen in sophisticated shells.

The shell supports a limited collection of built-in commands in the area of command execution: myls, mygrep, mypwd, and mycd. Using these commands, users can explore common functionalities such as listing directory contents, looking for patterns in files, finding the current working directory, and navigating the file system. These commands demonstrate basic interactions with the underlying operating system.

The shell is simple, but it represents a number of design choices. Because it only supports a small set of built-in commands, it is very easy to understand and a great place to start for those who are new to shell programming. Although it's not the most sophisticated approach, parsing commands character by character makes learning easier and paves the way for more complex parsing strategies.

Additionally, although convenient, using the system function for command execution raises security concerns. Users are encouraged to investigate alternate command execution techniques in situations where increased security is crucial by being aware of these trade-offs.

The goal of this shell is to give users a basic understanding of shell ideas, serving as a stepping stone. By delving into the inner workings of the shell, users are encouraged to investigate more intricacies like handling pipelines, redirection, and controlling background processes, which paves the way for a more extensive and adaptable command-line experience.

Basic Structure

- **Input Handling:** To read characters from the standard input, the shell employs the getchar() function. Until a space or newline appears, signaling the conclusion of a command, characters are processed.

- **Command Execution**: Checking for supported built-in commands using a switch statement. Spawns a child process for each command using fork(), and then uses exec() to replace the child process with the corresponding command's execution. The parent process waits for the child to terminate using wait(). The exit() system call is appropriately used in the child process to terminate its execution after the command is executed, ensuring proper resource management.

- **Built-in Commands:**
    - *myls:* This command runs the ls system command.
    - *mygrep:* Takes a user-supplied pattern, builds a grep command, and runs it on a test.txt sample file.
    - *mypwd:* This command runs the pwd system command.
    - *mycd:* Uses chdir to modify the current working directory after accepting a path from the user

Implemented Features

1. **Basic Command Parsing:** The shell uses a straightforward but efficient method for parsing commands character by character. Although not as complex as tokenization, this approach establishes the foundation for comprehending the analysis of command-line inputs. Commands are identified by whether or not they contain a space or a newline character. This gives a basic understanding of the parsing techniques used in more complex shell implementations.

2. **Built-in Commands:** To demonstrate the direct contact between the user and the operating system, the shell includes a set of built-in commands. Every built-in command demonstrates a certain system function:

    - *myls:* Allows users to list the contents of the current directory by using the ls command. Using this command, users can learn how to run system commands from within a shell.
    - *mygrep*: Constructs a grep command, allows user input for a search pattern, and runs it on a test.txt sample file. This feature highlights the ability to include user input into command execution in addition to demonstrating command parameterization.
    - *mypwd:* This command runs pwd and displays the current working directory. This built-in command demonstrates how a shell can retrieve and present system data to the user directly.
    - *mycd:* Accepts a path as input and allows users to modify the current working directory. The incorporation of chdir illustrates how the shell can modify the environment of a process, enabling users to traverse the file system.

3. **Interactive User Experience:** Users can engage in an engaging, hands-on experience with the shell. Users get real-time feedback on how each command is being executed as they enter it. Understanding command flow and the dynamic interaction between the user and the shell are fostered by this iterative process.

4. **Dynamic Memory Management:** Dynamic memory management is required when using character arrays for command parsing, such as tmp and pattern. Good programming

techniques are encouraged by the null-termination of strings and the resetting of arrays following command execution, which guarantee correct memory handling.

5. **Error Handling (Partial):** Limited error handling for the mycd command is incorporated into the implementation. When a directory change fails, the perror function is used to give users an informative error message. Improving error handling for every built-in command would be a worthwhile endeavor.

6. **Security Considerations:** Security concerns are still of the utmost importance, even if the current design depends on spawning child processes. There are still potential vulnerabilities for command injection, which highlights the necessity of thorough input validation to stop malicious inputs. It is essential to follow secure command execution procedures, which include cautiously processing user-provided data. Because of the design's emphasis on security, users and developers are more likely to emphasize rigorous input validation and continuous development on safer shell programming techniques.

When taken as a whole, the implemented features give users a basic understanding of important shell programming ideas. The shell is a useful teaching tool for people who are just starting to learn about the complexities of command-line interface programming. It covers everything from basic command parsing to the execution of built-in commands and concerns for security and error handling. Users can learn more about the dynamic relationship between a user and a shell environment by navigating through these elements, which opens up new avenues for inquiry and comprehension of more sophisticated shell operations.

Not Implemented Features

1. **Advanced Command-Line Features:** The simplicity and foundational ideas are the main priorities of the present shell implementation. But it doesn't support sophisticated command-line options like these:

    - **Redirection:** The capacity to reroute input and output streams, giving users control over the origin of command input and the destination of output.
    - **Pipelines:** The ability to join several instructions together to establish a data flow pipeline.
    - **Background Processes:** The ability to run commands in the background while a process is running, freeing up the shell for more input.

2. **Customization Options:** Features that let consumers personalize their experience are absent from the shell. Typical possibilities for customisation are as follows:

    - **Custom Shell Prompt:** Enabling users to specify their own shell prompt, giving it a unique appearance and feel.
    - **Environment Variable Management:** Increasing the dynamic shell environment by allowing users to set, unset, and modify environment variables.

3. **Scripting Support:** The shell does not allow the execution of scripts, despite its good handling of interactive input. Expert shells frequently let users create and run shell scripts, which are collections of instructions saved in a file for automatic job execution.

4. **Job Control:** The shell's capacity to oversee and manage several active processes is hampered by the lack of task management tools. The user experience is improved overall by features like process monitoring, job suspension, and foreground and background process control.

5. **Input Editing and History:** The current implementation does not provide input editing or command history, in contrast to more sophisticated shells. User convenience and efficiency are increased by features like command history recall and command line editing (e.g., arrow key navigation).

There is much of space for improvement and growth, even if the existing shell offers a strong foundation for comprehending fundamental command-line interactions and built-in commands. Enhancing the shell's usefulness with additional features, customization choices, scripting capabilities, and security considerations would make it a more potent and versatile tool. These unfinished features offer chances for further development and let users learn more about the wider field of command-line interfaces and shell programming.

Partially Implemented Features

1. **Error Handling (Limited):** At the moment, the shell uses perror to handle basic errors for the mycd command. To increase user advice, it would be beneficial to customize error messages and apply this to all commands.

2. **User Input Validation:** The absence of strong input validation makes the shell vulnerable to erroneous inputs. Enhancements may comprise of applying input sanitization and providing more lucid user prompts.

3. **Command Execution Security:** Security issues can come up when performing instructions, especially when depending on system functioning. This implementation decides to spawn additional child processes in order to handle possible security vulnerabilities. To improve safety, strong input sanitization protocols must be prioritized. It becomes vital to investigate alternatives, such the exec family of functions. These features give users more control over how commands are executed, which lowers risks and makes the implementation more secure.

4. **Code Structure and Readability:** Both better readability and modularization of the code structure would be advantageous. This entails segmenting the code into functions and adding comments to improve understanding.

These improvements are intended to improve the shell's codebase's maintainability, strengthen security, and improve user experience. By addressing these issues, the instrument will become more reliable and easy to use for both practical application and pedagogical investigation.

Design Decisions

1. **Built-in Commands:** To maintain simplicity and an emphasis on education, the shell comes with a limited number of built-in commands.

2. **Input Parsing:** Character by character, the input is parsed, and commands are recognized by their use of space or newline separators. While the parsing algorithm is made simpler, the command input's flexibility might be compromised.

3. **Command Execution:** The shell runs tasks by starting new processes, which is done by forking and using the execlp tool. These steps are better than using the system function directly because they give you more control and privacy when running tasks. A built-in function, like myls, mygrep, mypwd, or mycd, is linked to a certain child process. This method makes things safer because it stops random orders from being directly run through the system function.

4. **User Interaction:** The shell communicates with the user by reading characters and giving them just a limited amount of feedback when a command is executed. Although this design may restrict the user experience, it is consistent with the shell's simplicity.

5. **Use of tmp[0] = '\0':** There here are two reasons why tmp[0] = '\0' was chosen over bzero(tmp, 100) as the way to initialize the temporary array. For starters, it follows normal C practice because setting tmp[0] to null ends the line with a null character, which means it is empty. A simple and usual way to clear the array before getting input is to do it this way. Second, this method fits with the shell's form, which stresses simplicity. Using standard C null-termination makes it easy to clear the array in a way that is clear and easy to understand. The result is the same with bzero(tmp, 100), but using tmp[0] = '\0' is more natural and keeps the style of writing uniform in C code.

6. **String Comparison with strcmp:** The code uses the strcmp function to compare words and check if a command typed by the user fits one of the built-in commands, like myls, mygrep, mypwd, or mycd. This built-in C library method lets you compare strings character by character, which makes it useful for checking if two commands are equal.

7. **Null-Termination with tmp[index] = '\0':** The code sets tmp[index] = '\0' to end the string saved in the tmp array with a null character. This method makes sure that the string ends correctly, so it can be used as an empty string. In C code, null-termination is very important because it marks the end of a string and makes it easier to restart the input buffer for the next command.

8. **Spawning Child Processes for Command Execution:** The code creates child processes so that commands can be run. The fork() system call starts a new process when a user types a command. The execlp function from the exec family then replaces the child process with the command execution that goes with it. The parent process uses the wait() system call to wait for the child process to finish. This method makes sure that orders don't mess with each other and keeps them separate.

9. **Use of mycd Command without Forking:** In contrast to some other tasks, the mycd command does not require starting a new process. The code instead changes the working path directly by using the chdir function in the present process. There is no need for process separation through forking because changing the working path does not involve running an outside command.

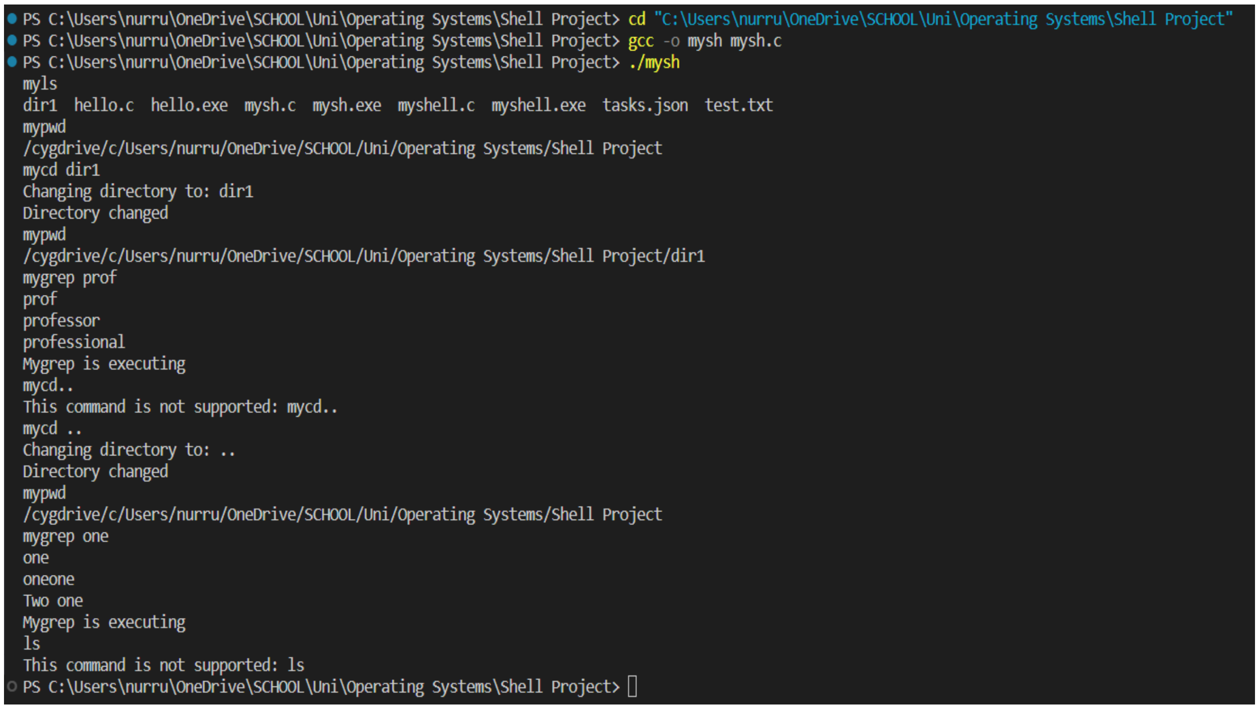
10. **Resetting tmp for the Next Command:** tmp is reset by setting tmp[0] = '\0' after each command is run. This ends the line with a null character, making sure the array is ready for the next user input. This reset makes sure that any characters left over from the previous command don't change how the new command is interpreted, which protects the accuracy of input processing.

11. **User Input Handling in mygrep:** When used with mygrep, the code asks the user to enter a pattern after the command is given. The pattern is kept in a different collection called char pattern[100]. When the grep command is run, this user-provided pattern is used as an argument. This dynamic pattern input makes the mygrep tool more flexible by letting users define search patterns as they type.

Test Plan

Testing all supported and unsupported commands

A test strategy has been created in order to fully verify the command-line interpreter's security and functioning. Both supported and unsupported commands are included in the following sample:

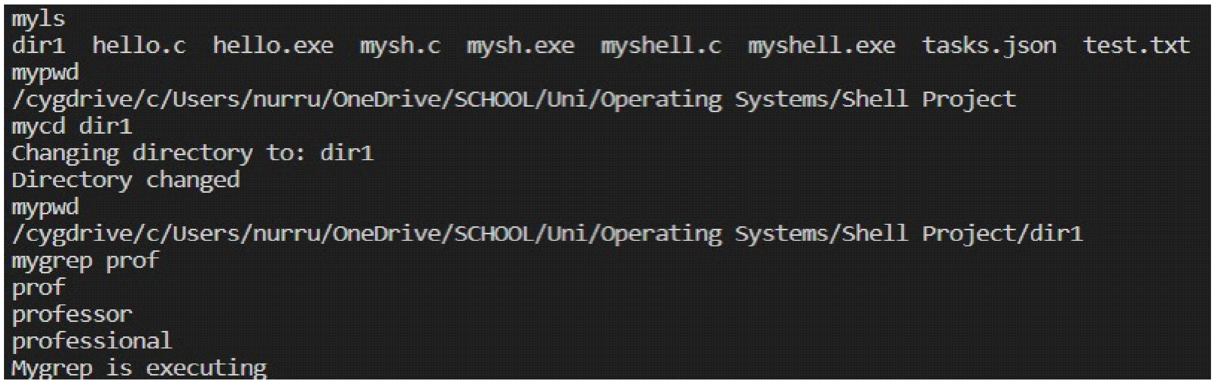


```
● PS C:\Users\nurru\OneDrive\SCHOOL\Uni\Operating Systems\Shell Project> cd "C:\Users\nurru\OneDrive\SCHOOL\Uni\Operating Systems\Shell Project"
● PS C:\Users\nurru\OneDrive\SCHOOL\Uni\Operating Systems\Shell Project> gcc -o mysh mysh.c
● PS C:\Users\nurru\OneDrive\SCHOOL\Uni\Operating Systems\Shell Project> ./mysh
  myls
  dir1  hello.c  hello.exe  mysh.c  mysh.exe  myshell.c  myshell.exe  tasks.json  test.txt
  mypwd
  /cygdrive/c/Users/nurru/OneDrive/SCHOOL/Uni/Operating Systems/Shell Project
  mycd dir1
  Changing directory to: dir1
  Directory changed
  mypwd
  /cygdrive/c/Users/nurru/OneDrive/SCHOOL/Uni/Operating Systems/Shell Project/dir1
  mygrep prof
  prof
  professor
  professional
  Mygrep is executing
  mycd..
  This command is not supported: mycd..
  mycd ..
  Changing directory to: ..
  Directory changed
  mypwd
  /cygdrive/c/Users/nurru/OneDrive/SCHOOL/Uni/Operating Systems/Shell Project
  mygrep one
  one
  oneone
  Two one
  Mygrep is executing
  ls
  This command is not supported: ls
○ PS C:\Users\nurru\OneDrive\SCHOOL\Uni\Operating Systems\Shell Project> 
```

In this test, all commands, including myls, mygrep, mypwd, and mycd, have been successfully executed within the shell. The comprehensive testing validates the proper functionality of each command.

Valid command

A number of tests have been carried out to confirm that the supported commands in the command-line interpreter work as intended. The expected results for the supported commands are described as follows:

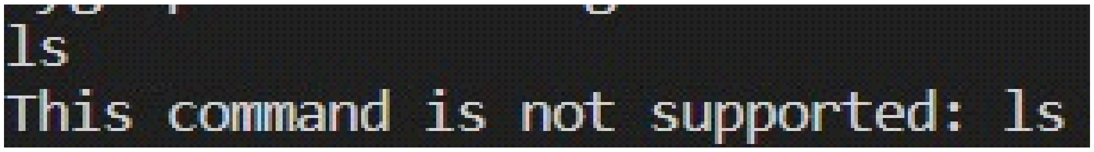


```
myls
dir1  hello.c  hello.exe  mysh.c  mysh.exe  myshell.c  myshell.exe  tasks.json  test.txt
mypwd
/cygdrive/c/Users/nurru/OneDrive/SCHOOL/Uni/Operating Systems/Shell Project
mycd dir1
Changing directory to: dir1
Directory changed
mypwd
/cygdrive/c/Users/nurru/OneDrive/SCHOOL/Uni/Operating Systems/Shell Project/dir1
mygrep prof
prof
professor
professional
Mygrep is executing
```

The screenshot demonstrates successful testing of specific commands within the shell. Commands such as mypwd, mycd, and mygrep have been validated and executed as intended, showcasing the functional capabilities of these built-in commands.

Invalid command

Several tests have been run to evaluate how well the command-line interpreter handles unsupported commands. The expected results for the tests involving unsupported commands are as follows:



```
ls
This command is not supported: ls
```

This test involves attempting to execute the unsupported command 'ls.' The shell appropriately identifies the unsupported command and provides feedback, illustrating the effective handling of unsupported inputs.

Conclusion

In this project, we've created a basic C command-line interpreter that supports a number of built-in commands and implements fundamental shell functions. In a loop, the program receives input commands, interprets them, and then performs the appropriate actions. The implementation of built-in commands, a loop for continuous command processing, and initialization comprise the three primary parts of the code's structure.

Setting up the required variables, such as an array (tmp) to hold the input command, a character (c) to read individual characters, and an index to record the current place in the command array, are all part of the setup process.

The while loop, which reads user input continually, is the heart of the program. The loop takes care of command parsing, determines which built-in commands (myls, mypwd, mygrep, and mycd) are supported and which are not, and then carries out the necessary operations. When a command is supported, the software uses fork to establish child processes and execlp to run the relevant system commands (grep, ls, and pwd). Before continuing with its execution, the parent process pauses until the child process has finished.

With the ability to list files, change directories, print the current working directory, and look for patterns in files, the code illustrates a functioning shell-like interface. The architecture includes a basic error-checking system to gracefully handle commands that are not supported.

To sum up, even though this command-line interpreter is a simple implementation, it can be used as a starting point for more intricate shell projects. Upcoming improvements might bring greater functionality, better error management, and support for more commands, transforming it into a useful and adaptable tool for users engaging with a command-line environment.