PROJECT PHASE 1 REPORT

Course: Operating Systems
Project: Phase 1—myshell
Submission Date: 26 February 2026

Group Information

Name: Bernard Gharbin
NetID: bg2696

Name: Bismark Buernortey Buer
NetID: bb3621

Introduction

In this project, we designed and implemented a simplified Unix shell named *myshell*. The shell provides a command-line interface that reads user input, creates child processes using **fork**, and executes programs using **execvp**. It supports command arguments, input redirection (<), output redirection (>), error redirection (**2**>), and command pipelines using |, including multiple chained pipes.

The shell repeatedly displays the **$** prompt, processes each command entered by the user, executes it using proper process control and file descriptor manipulation, and then returns to the prompt. The implementation emphasizes correctness, robustness, and disciplined use of low-level operating system primitives such as **fork**, **pipe**, **dup2**, **open**, and **waitpid**. Particular attention was given to accurate file descriptor management and reliable error handling to ensure stable behavior under both valid and malformed input.

Architecture and Design

The shell was designed using a modular architecture that clearly separates input handling, parsing, and execution. This separation of concerns ensures that each component performs a single, well-defined role, making the system easier to test, debug, and maintain. By organizing the shell according to its natural processing stages, we achieve a clean control flow and reduce unnecessary coupling between components.

At a high level, the project is organized into the following files: **main.c**, **input.c/input.h**, **parse.c/parse.h**, **execute.c/execute.h**, and the **Makefile**. The file structure mirrors the logical workflow of a Unix shell: read user input, parse the command structure, and execute the resulting command pipeline.

The file **main.c** acts as the control layer of the application. It implements the shell loop, prints the **$** prompt, retrieves input from the input module, checks for termination conditions such as **exit**, invokes the parser, passes structured commands to the execution module, and then repeats. Importantly, **main.c** does not perform detailed parsing or low-level process management. This design decision keeps the control logic simple and ensures that responsibilities remain clearly separated.

Input handling is isolated in **input.c**. The implementation uses **fgets** with a fixed-size buffer to prevent uncontrolled input growth and reduce the risk of buffer overflow. After reading input, the trailing newline character is removed before passing the string to the parser. This guarantees consistent formatting and prevents unnecessary complexity during tokenization.

The parser, implemented in **parse.c**, converts raw command-line text into structured data representations. Each command is represented using a **Command** structure that contains:

- A fixed-size argument pointer array (**args[MAX_ARGS]**) with dynamically allocated argument strings
- An argument count (**argc**)
- Optional filenames for input, output, and error redirection

Multiple commands connected by **|** are represented using a **Pipeline** structure containing an array of **Command** structures and a command count. This structured representation allows the execution logic to operate directly on validated command data rather than repeatedly tokenizing raw strings. It also simplifies support for multi-command pipelines.

Parsing is implemented in clear stages. First, the parser validates basic pipe syntax to ensure there are no leading or trailing pipes and no invalid consecutive operators. Next, the input is split by **|** to identify pipeline segments. Each segment is then tokenized by whitespace. During tokenization, redirection operators (**<**, **>**, **2>**) are detected, and their associated filenames are stored in the corresponding command structure fields.

Memory ownership is managed explicitly. Since the parser dynamically allocates memory for argument strings and redirection filenames, a centralized cleanup function, **free_pipeline()**, is used to release all allocated memory after execution completes. This design prevents memory leaks and ensures predictable resource management.

Execution is implemented in **execute.c** using a deterministic pipeline algorithm. For a pipeline containing n commands, up to $n - 1$ pipes are created as required using **pipe**. Each command is executed in its own child process created with **fork**. Within each child process, **dup2** is used to connect pipe endpoints and apply any necessary redirections. After file descriptors are configured, **execvp** replaces the child process image with the requested program. The parent process closes all unused file descriptors and waits for child termination using **waitpid**.

A key design decision is the ordering of descriptor setup. Pipe wiring is established first within each child process, and command-specific redirections are applied afterward. This ensures predictable and consistent behavior when pipes and redirection operators appear together in the same command line.

Finally, the **Makefile** standardizes compilation using **make** and cleanup using **make clean**. This ensures reproducible builds across environments, including the course Linux server, and reinforces the modular structure of the project.

Overall, the architecture emphasizes modularity, clear control flow, structured command representation, and disciplined file descriptor management. This design supports single commands, argument passing, redirection, and multi-pipe execution while maintaining well-defined boundaries between components.

Implementation Highlights

The shell operates in a continuous loop implemented in **main.c**. After printing the **$** prompt, it reads user input. If the input is empty, the shell prompts again. If the user enters **exit** or signals end-of-file, the shell terminates gracefully.

The parser first validates pipe placement to ensure syntactic correctness. The input string is then split on |, and each segment is tokenized by whitespace. When redirection operators (<, >, **2**>) are encountered, the following token is recorded as the corresponding filename. Remaining tokens form the argument array passed to **execvp**.

For single commands, the shell creates one child process using **fork**. Inside the child, redirections are applied using **open** and **dup2**, after which **execvp** replaces the process image. If execution fails, **perror** reports the error and the child exits. The parent process waits for completion using **waitpid**.

For pipelines, the execution module creates the necessary pipes using **pipe** and forks a child process for each command. Each child duplicates appropriate file descriptors using **dup2** to ensure that the output of one command becomes the input of the next. After configuration, **execvp** executes the command. The parent closes unused file descriptors and waits for all children to terminate.

Edge cases and error handling were carefully implemented. Syntax errors detected by the parser prevent execution from occurring. Execution errors are reported using **perror**, and control always returns safely to the main loop. A centralized cleanup routine, **free_pipeline()**, ensures that dynamically allocated memory from the parser is properly released after execution.

Execution Instructions

To compile the program, navigate to the project directory and run:

make

This generates the executable **myshell**.

To run the shell:

./myshell

To remove compiled object files and the executable:

make clean

The program was compiled and tested on the designated remote Linux server to ensure compatibility with the required grading environment.

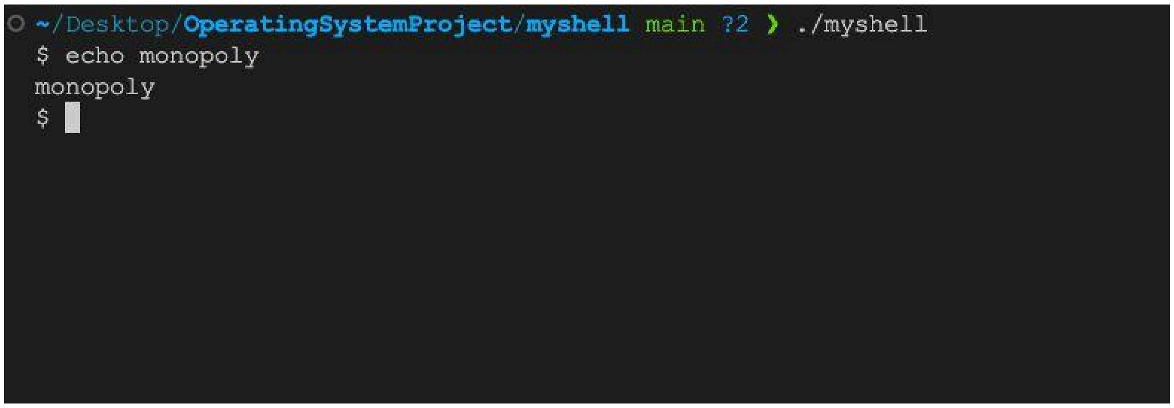
Testing

Comprehensive testing was performed to validate all required features. The following screenshots demonstrate successful execution and behavior under various scenarios.

Basic Commands Execution

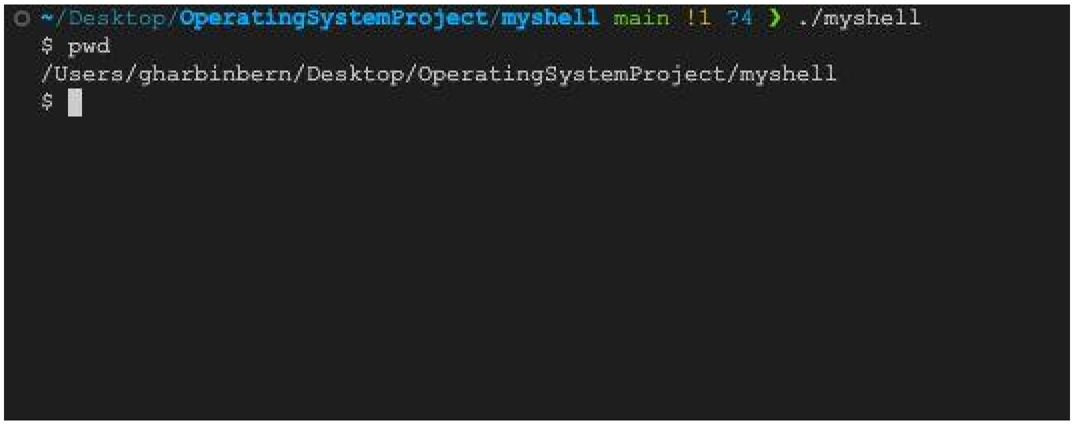
The first set of tests validates fundamental command execution within the shell. These tests confirm that the shell correctly parses simple commands, creates child processes, and executes external programs using **fork** and **execvp**.

The command **echo monopoly** verifies that the shell correctly parses command arguments and passes them to the executed program without modification. The expected output is the exact string *monopoly*, confirming proper argument handling and successful process execution.



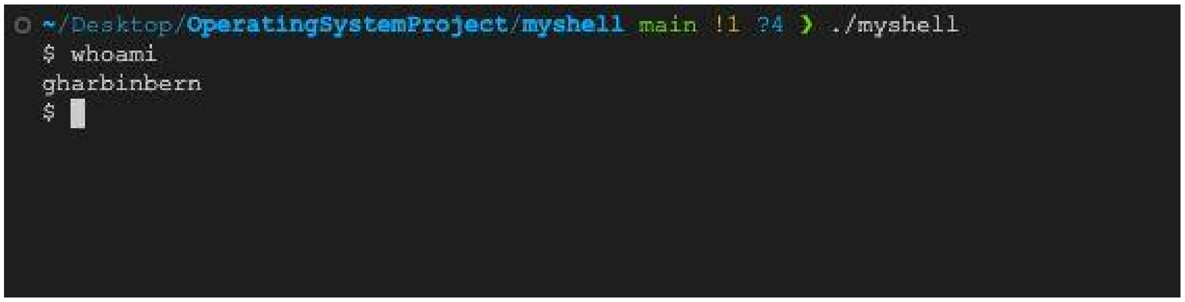
```
○ ~/Desktop/OperatingSystemProject/myshell main ?2 ❯ ./myshell
  $ echo monopoly
  monopoly
  $ 
```

The command **pwd** tests execution of a program that returns the current working directory. This confirms that the child process inherits the correct environment and that standard output is properly displayed in the terminal.



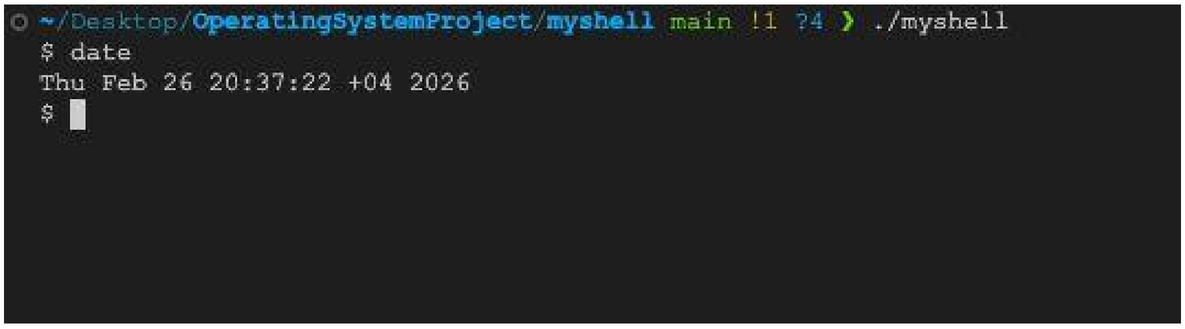
```
o ~/Desktop/OperatingSystemProject/myshell main !1 ?4 ❯ ./myshell
  $ pwd
  /Users/gharbinbern/Desktop/OperatingSystemProject/myshell
  $ 
```

The command **whoami** verifies correct execution of environment-related utilities. The username is displayed directly in the terminal, confirming that the child process executes successfully and that the shell preserves the user environment when launching external programs. This demonstrates that the shell does not interfere with user identity resolution and correctly forwards execution to the underlying system utility.



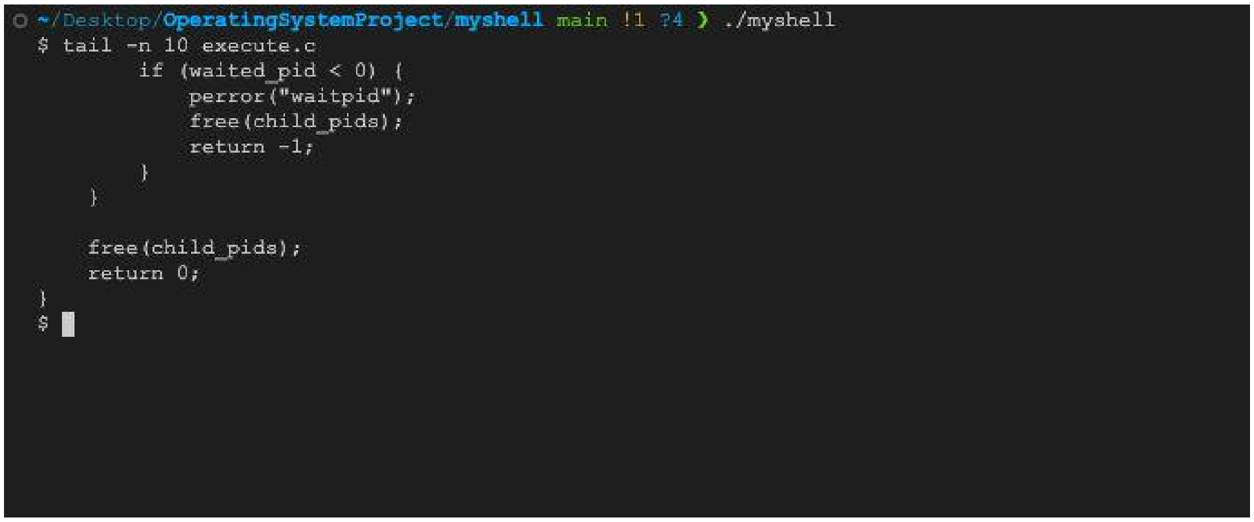
```
o ~/Desktop/OperatingSystemProject/myshell main !1 ?4 ❯ ./myshell
  $ whoami
  gharbinbern
  $ 
```

The command **date** demonstrates execution of a system utility that generates formatted output. The current system date and time are displayed directly in the terminal, confirming that the child process executes successfully and that standard output is correctly connected to the shell's terminal. This verifies proper handling of output streams and confirms that the shell does not interfere with time-sensitive system commands.



```
o ~/Desktop/OperatingSystemProject/myshell main !1 ?4 ❯ ./myshell
  $ date
  Thu Feb 26 20:37:22 +04 2026
  $ 
```

The command **tail -n 10 execute.c** provides a more rigorous test of argument parsing. This command includes a flag (**-n**), a numeric parameter (**10**), and a filename (**execute.c**). Successful execution confirms that the shell correctly constructs the argument array, preserves argument order, and passes multiple parameters accurately to **execvp**. The output displays the last ten lines of the specified file, verifying that file arguments are handled correctly.

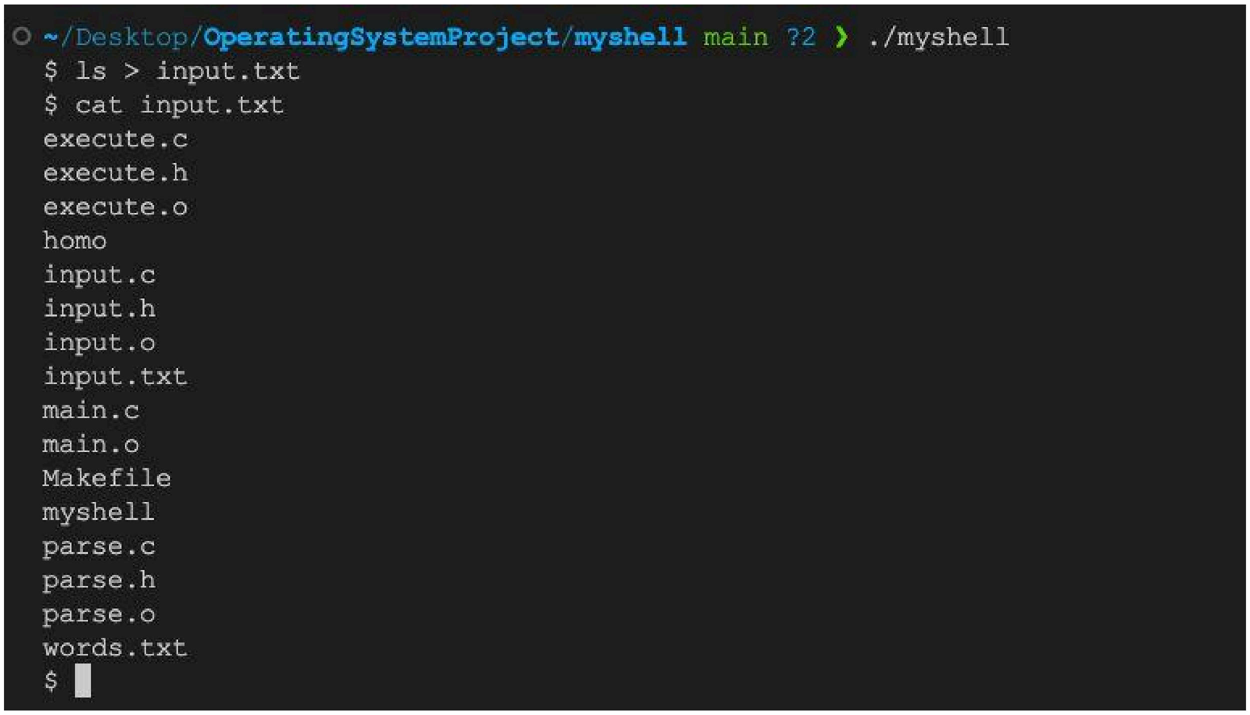


```
○ ~/Desktop/OperatingSystemProject/myshell main !1 ?4 ❯ ./myshell
  $ tail -n 10 execute.c
          if (waited_pid < 0) {
              perror("waitpid");
              free(child_pids);
              return -1;
          }
      }

      free(child_pids);
      return 0;
  }
  $ 
```

Together, these tests validate the core execution path of the shell: input reading, tokenization, argument array construction, process creation using **fork**, program execution using **execvp**, and synchronization using **waitpid**. Successful execution of these commands confirms that the basic command-processing pipeline of the shell is functioning correctly.

Output Redirection

This test verifies that output redirection works correctly in the shell. The command **ls > input.txt** was executed, and no directory listing appeared on the terminal. This shows that the output of **ls** was redirected to the file **input.txt** instead of being printed on the screen. To confirm that the redirection worked, the command **cat input.txt** was then executed. The terminal displayed the directory listing, proving that the output was successfully written into the file and that standard output redirection is functioning properly.



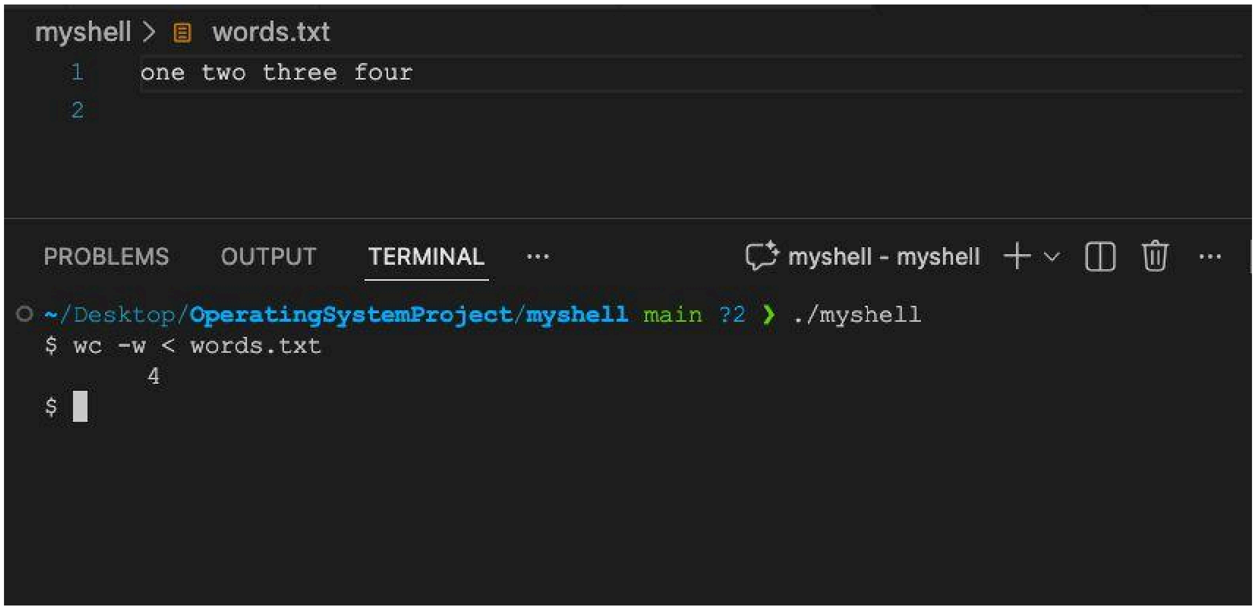
```
○ ~/Desktop/OperatingSystemProject/myshell main ?2 ❯ ./myshell
$ ls > input.txt
$ cat input.txt
execute.c
execute.h
execute.o
homo
input.c
input.h
input.o
input.txt
main.c
main.o
Makefile
myshell
parse.c
parse.h
parse.o
words.txt
$ 
```

Input Redirection

This screenshot demonstrates input redirection using the command **wc -w < words.txt**.

The output displays the correct computed word count.

This confirms that standard input is properly redirected from a file using **open** and **dup2**, and that the child process reads from the redirected file descriptor instead of the terminal.

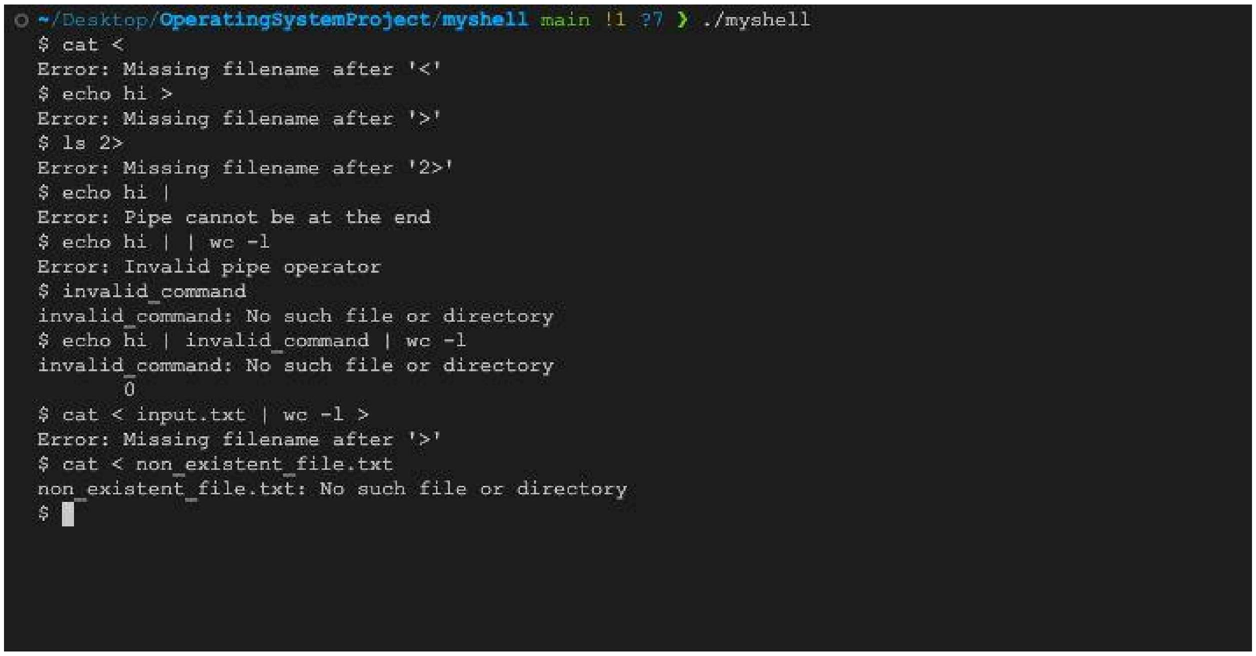


```
one two three four
```

```
○ ~/Desktop/OperatingSystemProject/myshell main ?2 ❯ ./myshell
$ wc -w < words.txt
       4
$ 
```

Error Redirection

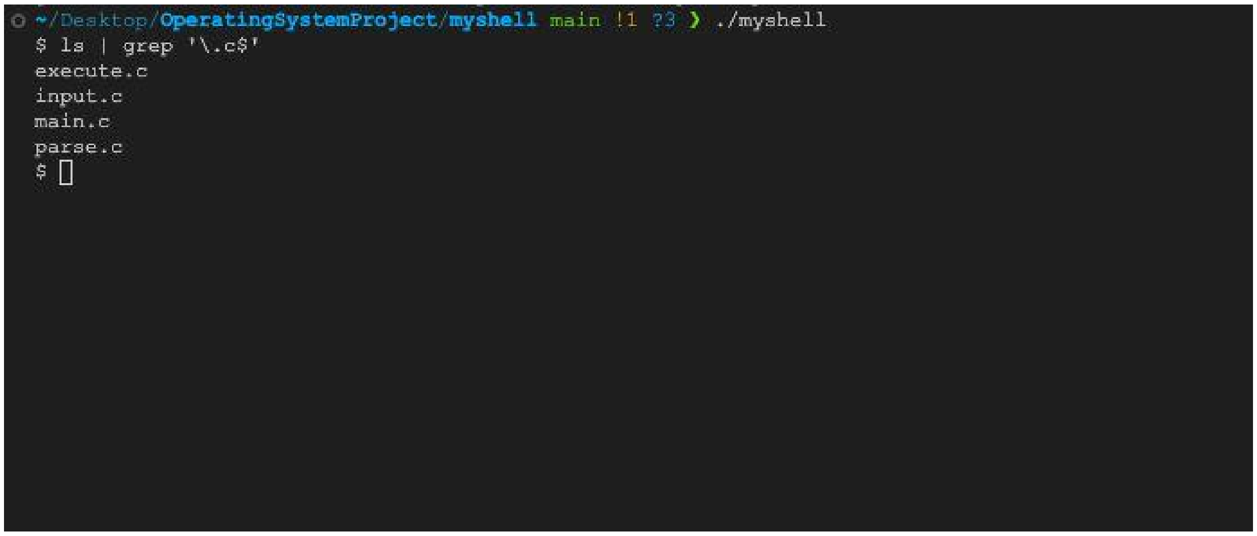
This test verifies that the shell correctly handles redirection-specific errors at the parsing stage. Commands like **hi <** and **ls 2>** trigger "Missing filename" errors, confirming that the parser catches incomplete redirection operators before any execution takes place. Similarly, **echo hi |** reports "Pipe cannot be at the end," and **echo hi | | wc -l** flags a consecutive pipe as invalid, proving that malformed pipe syntax is rejected early. These results confirm that the parser acts as the first line of defense against syntactically invalid input.



```
○ ~/Desktop/OperatingSystemProject/myshell main !1 ?7 ❯ ./myshell
  $ cat <
  Error: Missing filename after '<'
  $ echo hi >
  Error: Missing filename after '>'
  $ ls 2>
  Error: Missing filename after '2>'
  $ echo hi |
  Error: Pipe cannot be at the end
  $ echo hi | | wc -l
  Error: Invalid pipe operator
  $ invalid_command
  invalid_command: No such file or directory
  $ echo hi | invalid_command | wc -l
  invalid_command: No such file or directory
         0
  $ cat < input.txt | wc -l >
  Error: Missing filename after '>'
  $ cat < non_existent_file.txt
  non_existent_file.txt: No such file or directory
  $ 
```

Single Pipe Execution

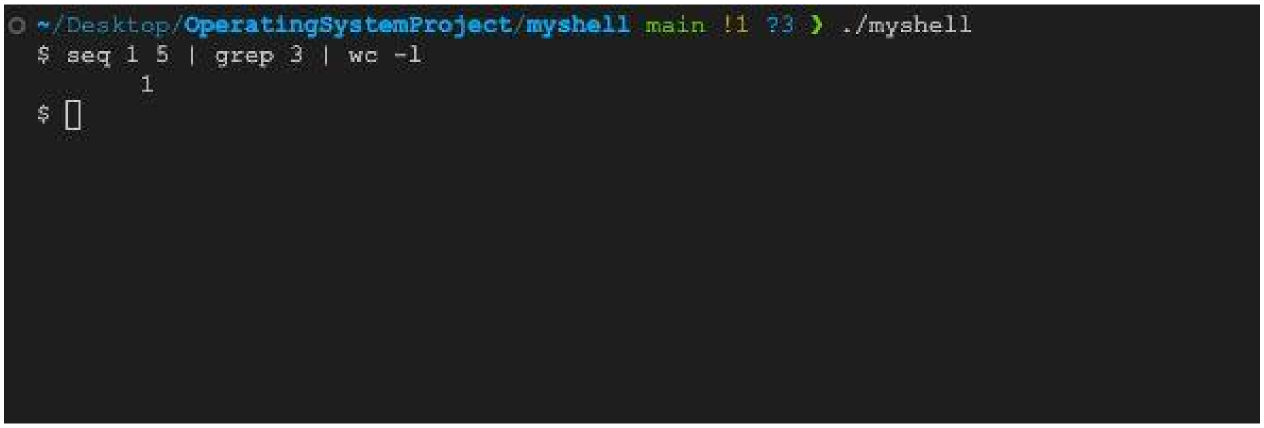
This test demonstrates single pipe execution. The command **ls | grep '\.c$'** pipes the output of **ls** into **grep** to filter only files ending with .c. The result correctly lists **execute.c, input.c, main.c**, and **parse.c**, confirming that the shell properly connects the output of one command to the input of another through a single pipe.



```
○ ~/Desktop/OperatingSystemProject/myshell main !1 ?3 ❯ ./myshell
  $ ls | grep '\.c$'
  execute.c
  input.c
  main.c
  parse.c
  $ 
```

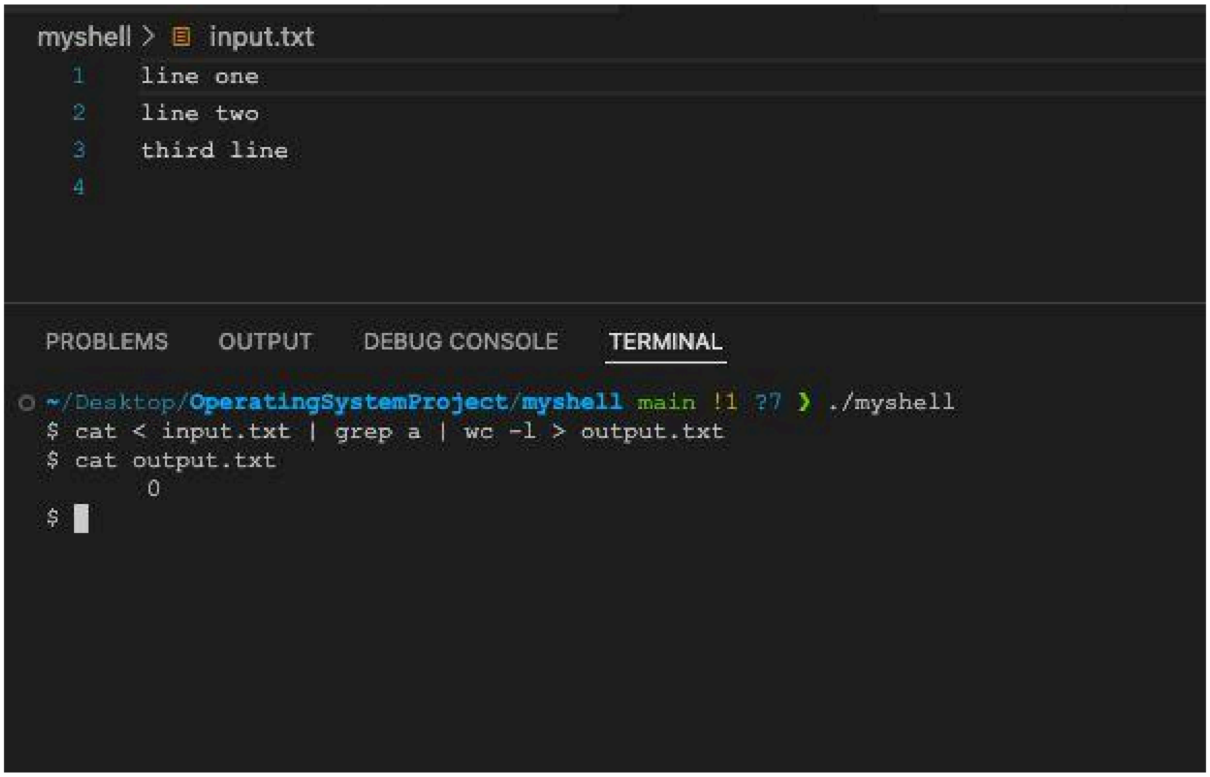
Multiple Pipe Execution

In the first test, **seq 1 5 | grep 3 | wc -l** generates numbers 1 to 5, filters for lines containing "3", and counts them. The result is **1**, correctly reflecting that only the number 3 matched. This confirms the shell handles a chain of multiple pipes, passing output correctly between each process in the pipeline.



```
○ ~/Desktop/OperatingSystemProject/myshell main !1 ?3 ❯ ./myshell
  $ seq 1 5 | grep 3 | wc -l
         1
  $ 
```

In the second test, **cat < input.txt | grep a | wc -l > output.txt** reads from **input.txt**, pipes it through **grep a** to filter lines containing the letter "a", then counts the matching lines with **wc -l**, and writes the result to **output.txt**. Since none of the lines in **input.txt** contain the letter "a", the output is **0**, which is correct.



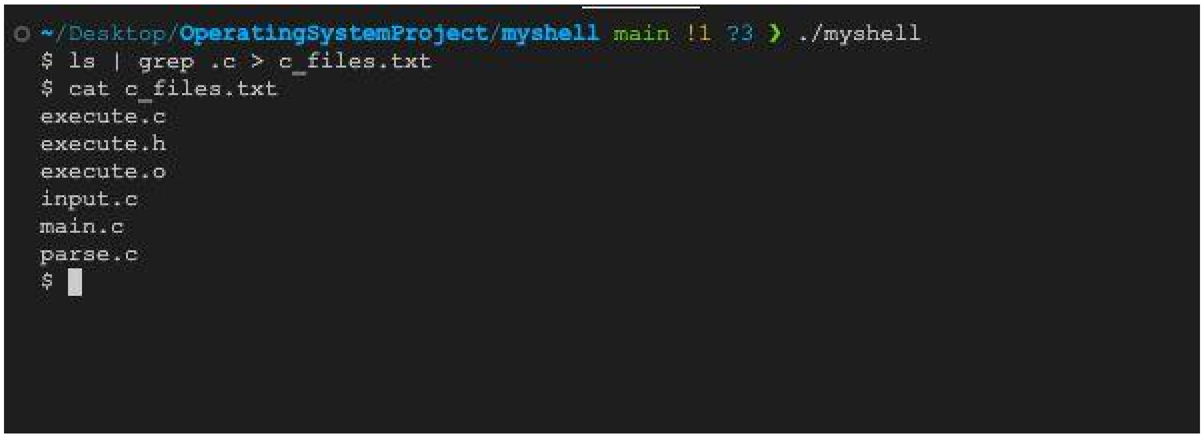
```
line one
line two
third line

```

```
~/Desktop/OperatingSystemProject/myshell main !1 ?7 ❯ ./myshell
$ cat < input.txt | grep a | wc -l > output.txt
$ cat output.txt
       0
$ 
```

Combined Pipes and Redirection

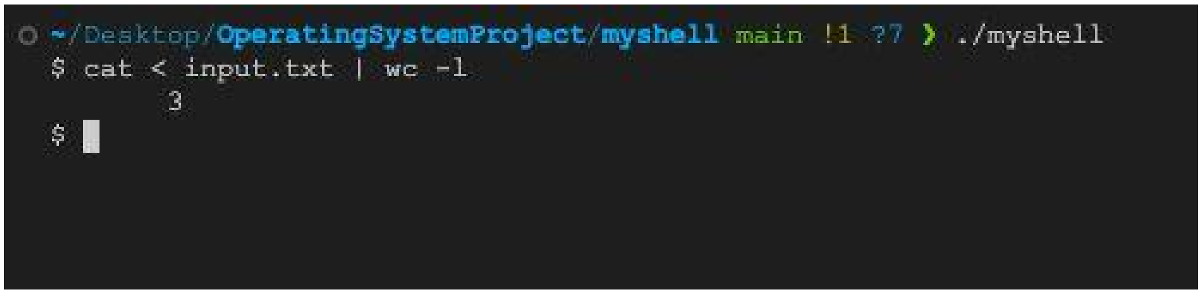
These tests verify that the shell correctly handles combinations of pipes and redirection together.

In the first test, **ls | grep .c > c_files.txt** pipes the directory listing into **grep** to filter .c-related files and writes the output to **c_files.txt**. Running **cat c_files.txt** correctly lists **execute.c**, **execute.h**, **execute.o**, **input.c**, **main.c**, and **parse.c**.



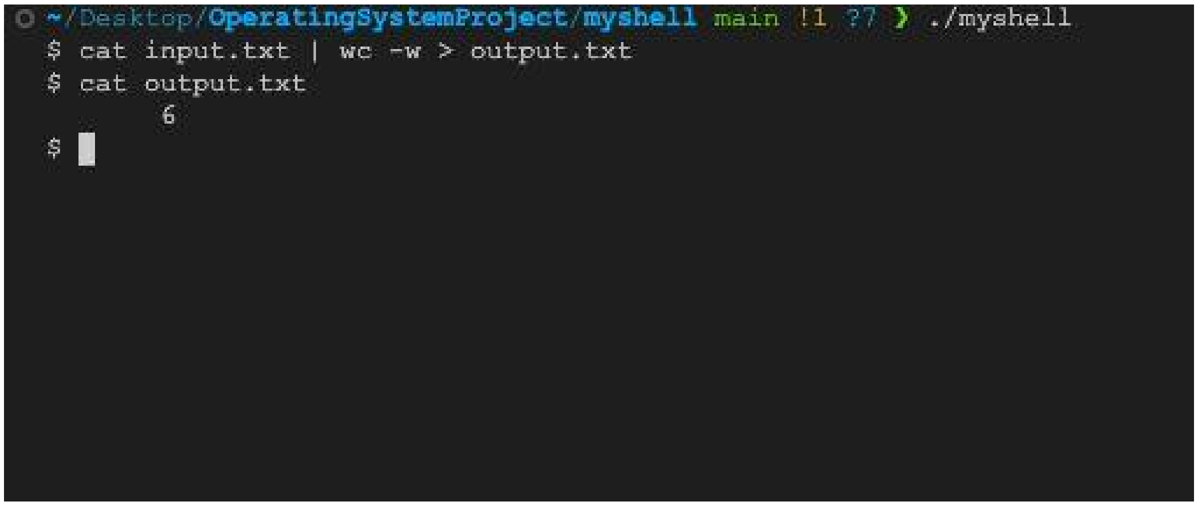
```
~/Desktop/OperatingSystemProject/myshell main !1 ?3 ❯ ./myshell
$ ls | grep .c > c_files.txt
$ cat c_files.txt
execute.c
execute.h
execute.o
input.c
main.c
parse.c
$ 
```

In the second test, **cat < input.txt | wc -l** uses input redirection to feed **input.txt** into **cat**, then pipes the output to **wc -l** to count the number of lines. The result is **3**, correctly matching the three lines in the file.



```
○ ~/Desktop/OperatingSystemProject/myshell main !1 ?7 ❯ ./myshell
  $ cat < input.txt | wc -l
         3
  $ 
```

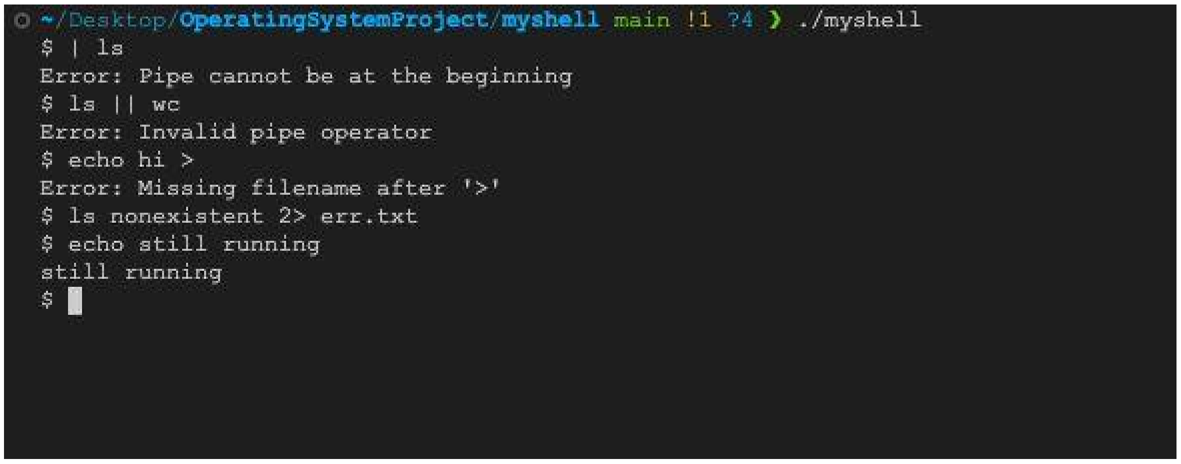
In the third test, **cat input.txt | wc -w > output.txt** pipes the contents of **input.txt** into **wc -w** to count the total number of words, then redirects the result into **output.txt**. Running **cat output.txt** shows **6**, which is correct based on the file's contents.



```
○ ~/Desktop/OperatingSystemProject/myshell main !1 ?7 ❯ ./myshell
  $ cat input.txt | wc -w > output.txt
  $ cat output.txt
         6
  $ 
```

Error Handling and Stability

This test focuses on runtime behavior when invalid commands are executed. **invalid_command** returns "No such file or directory," showing that when **execvp** fails, the shell correctly reports the error using **perror** without terminating. **cat < non_existent_file.txt** confirms that input redirection failures are properly surfaced to the user. In all cases, the shell prints a clear error message and safely returns to the **$** prompt, demonstrating that execution failures are handled gracefully and that the shell remains stable throughout.



```
○ ~/Desktop/OperatingSystemProject/myshell main !1 ?4 ❯ ./myshell
  $ | ls
  Error: Pipe cannot be at the beginning
  $ ls || wc
  Error: Invalid pipe operator
  $ echo hi >
  Error: Missing filename after '>'
  $ ls nonexistent 2> err.txt
  $ echo still running
  still running
  $ █
```

Each test case confirmed correct behavior of process creation, inter-process communication, redirection handling, and error reporting.

Challenges

One major challenge was managing pipe file descriptors correctly. It is easy to leak descriptors or connect incorrect endpoints when multiple pipes are involved. The solution was to implement a strict closing policy in both parent and child processes. After performing **dup2**, all unused file descriptors are immediately closed. This prevented descriptor leaks and avoided unintended blocking behavior.

Another challenge was combining pipes and redirection. Determining correct input and output behavior when both mechanisms are present required careful ordering. Pipe connections are established first, and command-specific redirections are then applied within each child process. This ensures predictable and correct I/O behavior.

Parser memory management presented another difficulty. The parser allocates many small strings while tokenizing commands. Without systematic cleanup, this would result in memory leaks. To address this, a centralized cleanup function, **free_pipeline()**, was implemented to release all dynamically allocated memory after execution.

Maintaining shell stability under malformed input was also critical. Malformed commands should not terminate the shell. The parser detects syntax errors early and reports them without invoking execution. Execution failures are reported using **perror**, and control always returns safely to the main loop.

Division of Tasks

Development responsibilities were divided to ensure balanced contribution.

Bernard Gharbin primarily implemented **input.c**, **parse.c**, and syntax validation logic. Bismark Buernortey Buer focused on **execute.c**, including process creation, pipe handling, redirection, and descriptor management.

Both team members collaborated on **main.c**, integration, debugging, testing, the creation of the **Makefile**, and report preparation.

The project was developed using GitHub with a structured branching workflow. Each member worked in separate branches, pushed changes regularly, and submitted pull requests for review before merging into the main branch. This ensured code quality, traceability, and equal participation.

References

Linux manual pages for **fork**, **execvp**, **waitpid**, **pipe**, **dup2**, and **open** were referenced during development. Course lecture materials and laboratory examples were also consulted to ensure correct system call usage and expected shell semantics.