Operating Systems: Project Phase 1

Sihyun Kim(sk11019)
Shota Matsumoto (sm11745)

February 26, 2026

Contents

1 Introduction

The objective of this project is to build a custom shell that allows users to interact with the operating system's services through a remote terminal. Specifically, the shell reads and parses user input, then executes the corresponding commands as child processes using specific system calls. The shell supports running commands both with and without arguments, manages parent and child processes appropriately, implements input, output, and error redirection, supports an arbitrary number of pipes to connect the output of one command to the input of another, and handles complex combinations of these features together.

2 Architecture and Design

The `main.c` file acts as the coordinator of the system. It runs the interactive loop, collects user input, invokes the parser, and delegates valid execution plans to the executor module.

The system is structured into two independent modules:

1. Parser Module (parser.c): Responsible for breaking down the user's input string into actionable data. It identifies pipe symbols (`|`) to count and separate command segments, then performs a second pass on each segment to identify arguments and redirection operators (`<`, `>`, `2>`.

2. Executor Module (executor.c): Handles the process lifecycle. It creates pipes using `pipe()`, spawns child processes via `fork()`, and manages file descriptor redirection using `dup2()` before calling `execvp()`.

Regarding data structures, the `ExecutionPlan` and `Command` structures in `common.h` act as the bridge between the two modules, storing the argument arrays and redirection filenames.

Essentially, the parser processes raw user input and produces a validated execution plan. The executor receives this plan and performs process creation and input/output redirection based on the parsed structure.

2.1 Data Flow

Here is the diagram that shows the overall execution flow:

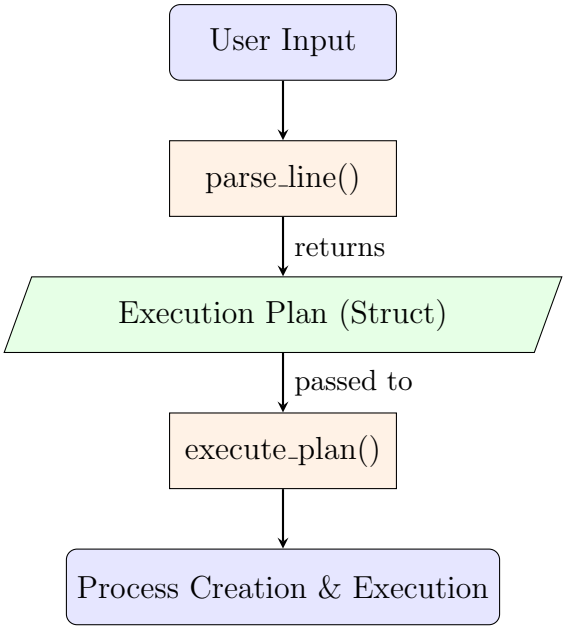


Figure 1: System Architecture Flow: From Input to Execution

The following components represent the logical flow of `myshell` from input to execution:

User Input The raw string entered by the user in the `main` loop. The shell captures this using `fgets()` and clears any trailing newline characters before processing.

`parse_line()` The core function in `parser.c` that tokenizes the input based on separators such as pipes (`|`), input redirection (`<`), and output redirection (`> or 2>`). It populates the necessary data structures to guide the executor.

Execution Plan Represented by the `ExecutionPlan` struct defined in `common.h`. It acts as a container that holds an array of `Command` objects and the total count of commands in the pipeline.

`execute_plan()` The function in `executor.c` that iterates through the `ExecutionPlan`. It is responsible for calculating the required number of pipes and setting up the file descriptor redirections using `dup2()`.

Process Creation The final stage where `fork()` is used to create a child process for each command. The child then calls `execvp()` to replace its process image with the intended program while the parent waits for completion.

2.2 Separation of Responsibilities

To ensure modularity and reduce interdependency, the system is strictly divided into two functional domains. This separation allowed for independent development and testing of each module.

- **The Parser (`parser.c`) performs:**
  - **Tokenization:** Breaking strings into discrete arguments for the `argv` array.

- **Syntax Validation:** Checking for valid input before attempting execution.
- **Pipeline Splitting:** Identifying the `|` symbol to count and separate command segments.
- **Redirection Detection:** Identifying `<`, `>`, and `2>` to extract target filenames.
- **Error Reporting:** Handling cases like missing redirection targets or empty pipes.

- **The Executor (`executor.c`) performs:**
  - **Process Creation:** Spawning child processes for every command using `fork()`.
  - **Program Execution:** Using `execvp()` to search the PATH and run the commands.
  - **Pipe Setup:** Creating inter-process communication channels using `pipe()`.
  - **File Descriptor Redirection:** Mapping standard streams to files or pipes using `dup2()`.
  - **Process Synchronization:** Using `wait()` to ensure the shell prompts only after the last process terminates.

3 Implementation Highlights

3.1 Executor

1. **Process Isolation via fork/exec**
   Each command in the execution plan is executed within its own child process created using `fork()`. This ensures process isolation and mimics the behavior of a real Unix shell. After process creation, the child replaces its process image using `execvp()`, while the parebt process forks all child processes first and waits for all of them to complete after the loop.

   This design prevents the shell itself from being overwritten and allows multiple commands in a pipeline to execute concurrently.

2. **Dynamic Pipeline Construction**
   For a pipeline containing N commands, the executor dynamically computes the required number of pipes as $N - 1$. The pipe file descriptors are allocated using:

   ```
   int pipes= plan -> num_cmds-1;
   int pipefds[pipes > 0 ? 2 * pipes : 1];
   ```

   Each pipe contributes two file descriptors (read and write ends). This allocation allows the shell to support arbitrary-lengrh pipelines without hardcoding limits. The executor then iteratively connects adjacent commands using `dup2()`

3. **Unified File Descriptor Redirection**
   Input (`<`), output (`>`), and error (`2>`) redirections are implemented using the same

mechanism: `open()` followed by `dup2()`. Output files are opened with the flags `O_WRONLY`, `O_CREAT`, and `O_TRUNC` to ensure they are created or overwritten as needed. By treating files and pipes uniformly through file descriptors, the implementation leverages the Unix abstraction model where all I/O streams share the same interface.

4. **Robust Syntax and Runtime Error Handling**
   The parser ensures that malformed commands (e.g., missing filenames after redirection operators such as `<`, `>`, or `2>`) are rejected before execution. During runtime, system call failures such as `pipe()`, `fork()`, or `open()` are checked immediately and reported using `perror()`. This defensive approach prevents undefined behavior and improves overall system reliability.

5. **Resource Management**
   Proper resource management is critical in pipeline execution. After duplicating file descriptors with `dup2()`, the child process closes all unused pipe ends to prevent descriptor leaks and ensure correct EOF signaling. The parent process also closes pipe descriptors after `fork()` and uses `wait()` to avoid zombie processes. This careful handling of file descriptors prevents deadlocks and guarantees correct process termination.

3.2 Parser

1. **Pre-emptive Validation**
   Before we tokenize the input, we pre-check the user input using a for loop starting at line 44 to determine whether the input is valid. Specifically, we check for a pipe at the end with no destination command, as well as for empty segments between pipes. By doing so, we are able to provide immediate feedback for a missing command after a pipe and for empty commands between pipes.

2. **Ensuring Memory Efficiency Through Tokenization**
   The parser file uses two approaches to handle command arguments, resulting in improved memory efficiency. First, a helper function called `count_num_tokens()` iterates through the command line to count the number of executable commands, including the null terminator and excluding redirection operators and file names. This way, we can know exactly how many memory spaces we need to store them. Second, the actual strings are duplicated into the array, ensuring data integrity and memory efficiency.

3. **Effectively Separating Redirection Operators and Filenames**
   Our code effectively differentiates redirection operators and command arguments through conditional logic. Specifically, it identifies redirection operators using if statements, extracts the filenames that follow them, and stores them into the corresponding file segment of each command element in the array. It also identifies other commands and stores them into the corresponding argument element in the command array.

4. **Robust Error Reporting**
   Our code effectively identifies whether a command is missing after redirection operators. If a redirection operator is found but no filename follows, the parser halts

execution and displays an error message. This effectively prevents the shell from attempting to open null or invalid file paths.

4 Execution Instructions

Compilation

The project is compiled using the provided `Makefile`. From the project directory, run:

```
make
```

This command compiles all required source files and generates the executable `myshell`. If a clean rebuild is needed, the following commands can be used:

```
make clean
make
```

Running the Shell

After successful compilation, run the shell using:

```
./myshell
```

The program will display a prompt (e.g., `$`). The user can then enter commands, which will execute and return output before the prompt reappears.

Exiting the Shell

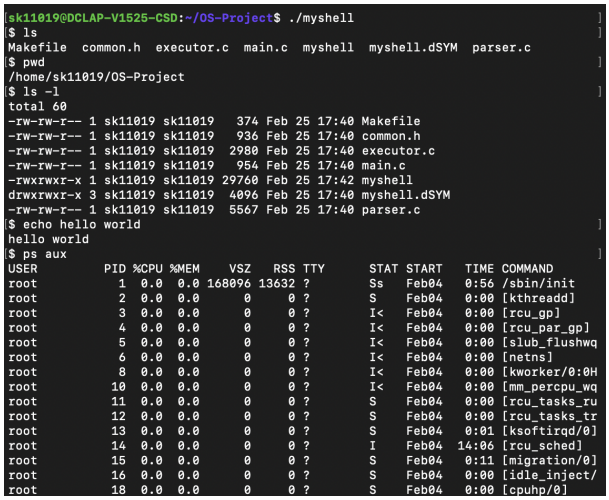
To terminate the shell session, enter:

```
exit
```

The shell will then terminate gracefully and return control to the system terminal.

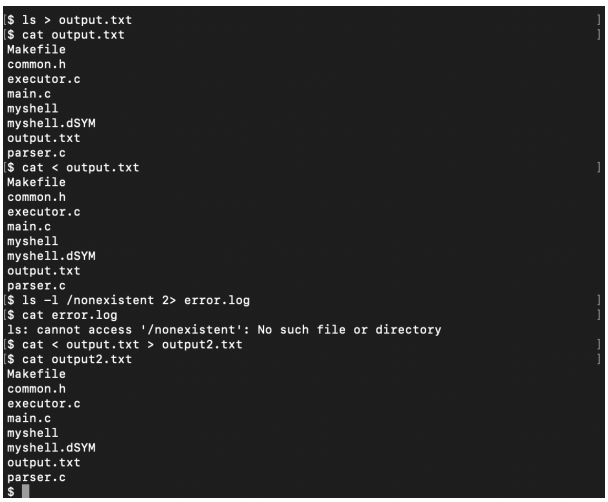
5 Testing

5.1 Single Commands



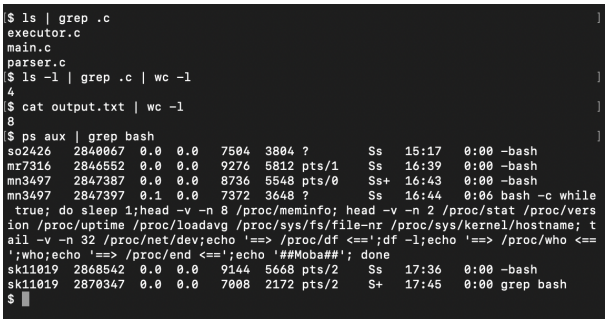
The screenshot above demonstrates basic command execution. We tested `ls`, `pwd`,`ls -l`, `ps -aux`, and `echo hello world`. All commands executed correctly in individual child processes, with output displayed as expected.

5.2 Input, Output, and Error Redirection



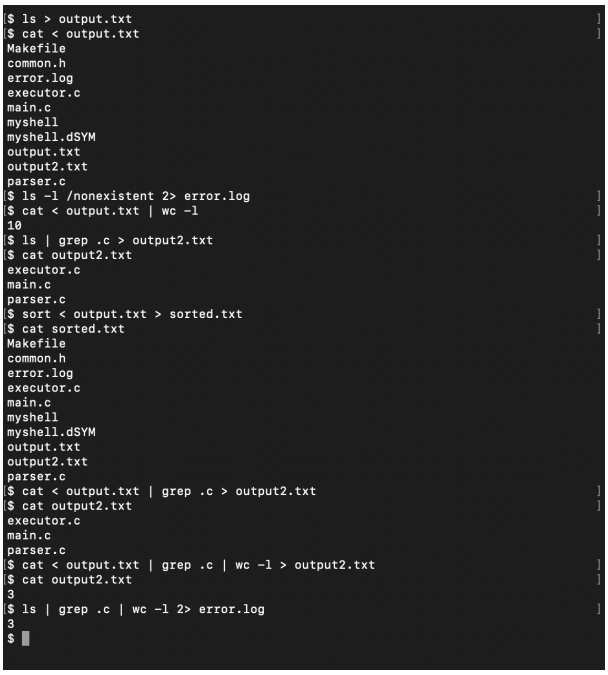
The screenshot above demonstrates all three redirection types. `ls > output.txt` correctly redirected stdout to a file, `cat < output.txt` correctly read from the file as stdin, and `ls -l /nonexistent 2> error.log` correctly captured the error message into the log file. We verified each redirection by reading the resulting files with `cat`.

5.3 Pipes



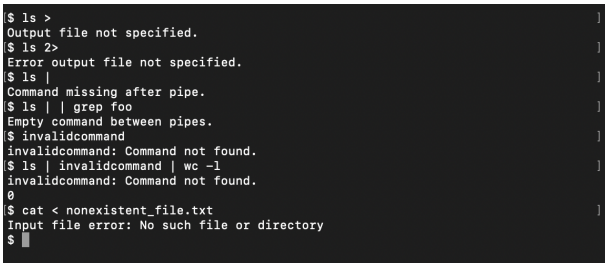
The screenshot aboved emonstrates pipe functionality. We tested a basic two-command pipe (`ls | grep .c`), a three-command chain (`ls -l | grep .c | wc -l`), and two additional pipe combinations. All commands communicated correctly through the pipe file descriptors.

5.4 Compound Commands



The screenshot above demonstrates combinations of pipes and redirections together. We tested all required compound combinations from the spec, including input redirection feeding into a pipe, pipes writing into output files, and chains involving all three redirection types simultaneously. All produced the correct output.

5.5 Error Handling



The screenshot above demonstrates all required error cases. Missing filenames after `<`, `>`, and `2>` produced clear error messages. A trailing pipe (`ls |`) and empty pipe (`ls | | grep foo`) were detected before execution. An invalid command printed a command not found message. A nonexistent input file produced a file error. In all cases the shell continued running normally after the error and displayed the `$` prompt again.

5.6 Exit

The screenshot above demonstrates the `exit` command terminating the shell cleanly and returning to the system prompt.

6 Challenges

While implementing the executor, we initially applied file redirections before setting up pipe connections, which caused compound commands like `cat < input.txt | wc -l` to not work correctly. We realized that pipe `dup2()` calls must happen first, and file redirection `dup2()` calls must come after, so that when both are present, the explicit file redirection takes priority and correctly overrides the pipe connection.

Moreover, when only one command was entered with no pipes, `pipes = 0` caused a zero-length variable-length array `pipefds[0]`, which is undefined behavior in C. Although this did not cause a visible crash on our machine, it could behave differently on another system. We fixed this by using a conditional size `pipes > 0 ? 2 * pipes : 1` to guarantee the array always has at least one element.

The challenge we faced during the implementation of the parser.c file was correctly calculating the size of the command's argument list when redirection operators were mixed into the command. It was especially difficult because strtok() treats every word as a token, but our shell must treat ls as an argument and the filename as a file destination. If we allocate memory space for the redirection operators as well as the filenames, the program will result in a segmentation fault when execvp() tries to read garbage data from the pointer array. Therefore, we needed to create a helper function to correctly allocate memory space only for the command arguments.

Another problem we encountered was that strtok() treats invalid command lines as valid, especially when it skips empty spaces between pipes. It essentially treats them as a single pipe. Thus, we had to create a for loop to manually check whether there is a space between the pipes to avoid accepting these invalid commands.

7 Division of Tasks

We divided the development of the shell evenly between the two of us to ensure efficient collaboration. Shota Matsumoto implemented `main.c` and `parser.c` to create a reliable interface for reading and parsing command strings into executable data. Sihyun Kim was responsible for developing `executor.c` and `common.h` to manage the process lifecycle, handle file descriptor redirection through `dup2()`, and manage inter-process communication with `pipe()`. We both tested the system comprehensively and contributed equally to the report.

8 References

1. Course Lecture and Lab Slides, *Operating Systems*, NYU Abu Dhabi, Spring 2026. Used as the primary reference for the implementation of core system calls including `fork()`, `execvp()`, `dup2()`, `pipe()`, and `wait()`.

2. Anthropic, *Claude AI Assistant*, claude.ai, 2026. Used for debugging assistance and code review during development.