

Operating Systems

Project Report - Phase 1

Date: February 27, 2025

Instructor: Dena Ahmed

Group Members:

Haad Mehboob (hm2957)

Ameen Vadakkekara (av2851)

Contents

1	Architecture and Design	3
1.1	System Architecture Overview	3
1.2	Design Principles and Rationale	3
1.3	Core Components and Their Interactions	4
1.3.1	Parser Module	4
1.3.2	Execution	4
1.3.3	I/O Redirection and Pipeline Subsystems	4
1.4	Data Flow Architecture	5
1.5	Data Structures	5
1.6	File Organization and Code Structure	6
1.7	Process Flow and Execution Model	6
1.8	Design Decisions and Alternatives Considered	8
2	Implementation Highlights	8
2.1	Shell Architecture Overview	8
2.2	Command Parsing Implementation	8
2.2.1	Memory Management	10
2.3	Command Execution	10
2.4	Redirection Implementation	11
2.5	Pipeline Implementation	12
2.5.1	Pipeline Data Flow	14
2.6	Error Handling	14
2.7	Main Shell Loop	15
2.8	Advanced Features and Edge Cases	16
2.8.1	Handling Complex Command Combinations	16
2.8.2	Whitespace Handling	17
2.9	Resource Management	17
3	Execution Instructions	18
3.1	Compilation	18
3.2	Running the Shell	18
3.3	Using the Shell	18
4	Testing	18
4.1	Basic Command Execution	18
4.2	Redirection Testing	19
4.3	Pipeline Testing	19
4.4	Combined Operations Testing	19
4.5	Error Handling Testing	20
5	Challenges	21
5.1	Pipe Implementation	21
5.2	Memory Management	21
5.3	Error Handling	21
5.4	Command Parsing	21
6	Division of Tasks	21

7	References	22
A	Appendix: Source Code	23
A.1	Header Files	23
A.1.1	error_handling.h	23
A.1.2	executor.h	23
A.1.3	parser.h	23
A.1.4	pipes.h	24
A.1.5	redirection.h	24
A.2	Implementation Files	25
A.2.1	error_handling.c	25
A.2.2	executor.c	25
A.2.3	main.c	27
A.2.4	parser.c	28
A.2.5	pipes.c	30
A.2.6	redirection.c	33
A.3	Makefile	34

1 Architecture and Design

1.1 System Architecture Overview

MyShell implements a modular architecture following the UNIX philosophy of having specialized components that perform distinct functions and work together cohesively. The design follows a layered architecture pattern with clear separation between parsing, execution, and I/O management.

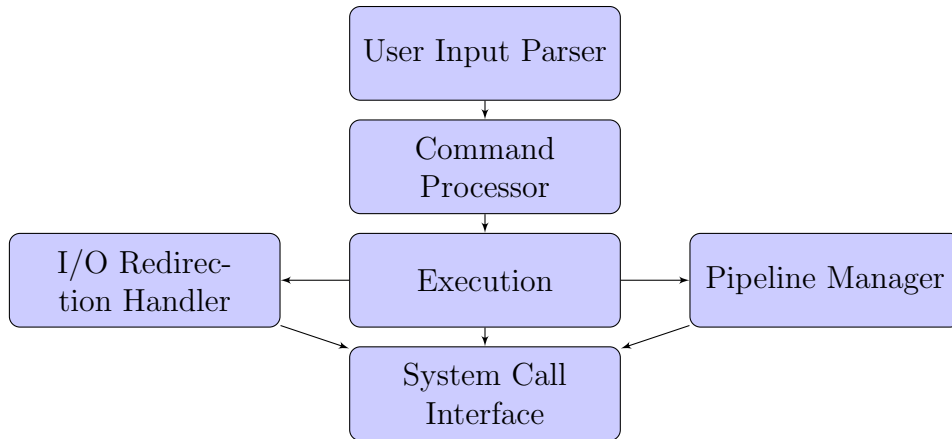


Figure 1: MyShell Architectural Overview

1.2 Design Principles and Rationale

The implementation of MyShell adheres to several key design principles that guided our architectural decisions:

1. **Modularity and Single Responsibility:** Each component of the shell is designed with a single, well-defined responsibility. This approach improves code maintainability and makes the system easier to debug and extend. For example, the parser module focuses exclusively on interpreting user input, while the execution module handles the process creation and management.
2. **Loose Coupling:** Modules interact through clearly defined interfaces, reducing interdependencies. This allows us to modify components with minimal impact on other parts of the system.
3. **Robust Error Handling:** Error detection, reporting, and recovery mechanisms are integrated at all levels, enhancing the reliability of the shell in various scenarios.
4. **Resource Management:** The design carefully manages system resources like file descriptors and memory to prevent leaks, especially important in long-running applications like shells.
5. **UNIX Programming Model:** The architecture embraces the UNIX paradigm where small, specialized tools are combined to perform complex tasks, particularly evident in our pipeline implementation.

1.3 Core Components and Their Interactions

1.3.1 Parser Module

The parser translates user input strings into structured command representations using a tokenization approach. We opted for a custom parsing algorithm rather than using tools like lex/yacc for several reasons:

- **Control and Flexibility:** Our custom parser gives us precise control over the interpretation of command syntax.
- **Performance Considerations:** A lightweight, purpose-built parser avoids the overhead of more general parsing frameworks.
- **Learning Opportunity:** Implementing the parser from scratch provided valuable insights into lexical analysis and command interpretation.

The parser implements a state-machine approach to track different parsing contexts (regular arguments, redirection targets, etc.) and produces a comprehensive command structure that encapsulates all necessary execution information.

1.3.2 Execution

The execution section is responsible for process creation and management. We implement the classic fork-exec-wait pattern with additional handling for redirections and pipelines. Key design considerations include:

- **Process Isolation:** Each command runs in its own process space, preventing interference between commands.
- **Parent-Child Relationship:** The main shell process maintains control by creating child processes for command execution and waiting for their completion.
- **Resource Cleanup:** Careful attention to releasing resources in both successful and error conditions.

1.3.3 I/O Redirection and Pipeline Subsystems

These critical subsystems manage the data flow between processes and between processes and files. Our design:

- Uses the file descriptor manipulation capabilities of UNIX systems (through `dup2()`)
- Handles all standard streams (`stdin`, `stdout`, `stderr`) independently
- Supports complex redirection combinations within pipeline contexts
- Implements n-way pipelines with systematic file descriptor management

1.4 Data Flow Architecture

MyShell implements a linear data flow pattern:

1. User input is captured in the main loop
2. The input string is parsed into a command structure
3. The command structure dictates process creation and I/O configuration
4. Child processes execute commands with appropriate redirections
5. The parent process collects results and presents the next prompt

This approach provides a clean separation between input processing, command execution, and output handling.

1.5 Data Structures

The central data structure is the `Command` structure, which serves as the bridge between parsing and execution:

```
1 typedef struct Command {
2     char **args;           // NULL-terminated array of arguments
3     char *input_file;      // input redirection file
4     char *output_file;     // output redirection file
5     char *error_file;      // error redirection file
6     int pipe_count;        // number of pipes detected
7 } Command;
```

This structure was chosen for its:

- **Completeness:** It captures all aspects of a command (arguments, redirections)
- **Simplicity:** It presents a clear interface between modules
- **Extensibility:** New command attributes can be added as the shell evolves
- **Memory Efficiency:** It uses dynamic allocation only where needed

For pipeline handling, we use arrays of file descriptors to manage the inter-process communication channels, arranged to minimize complexity while supporting n-way pipelines.

1.6 File Organization and Code Structure

The project follows a logical file organization that reflects the component architecture:

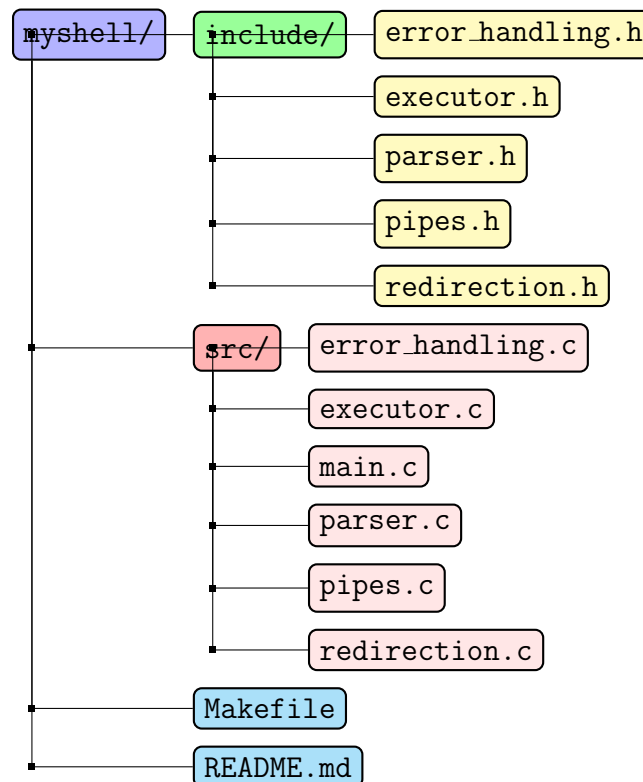


Figure 2: Shell Project Directory Structure

The file organization follows these principles:

- **Header-Source Separation:** Interface definitions are separated from implementations
- **Functional Grouping:** Files are organized by functional area rather than implementation details
- **Clear Dependencies:** Header files define the interfaces between modules

1.7 Process Flow and Execution Model

Our shell implements a standard read-evaluate-print loop (REPL) with process-based command execution:

The figure 2 illustrates the primary execution flow, emphasizing the fork-exec pattern and the clear separation between the main shell process and command execution processes. The parent process maintains shell state and provides the interactive interface, while child processes handle command execution in isolated environments.

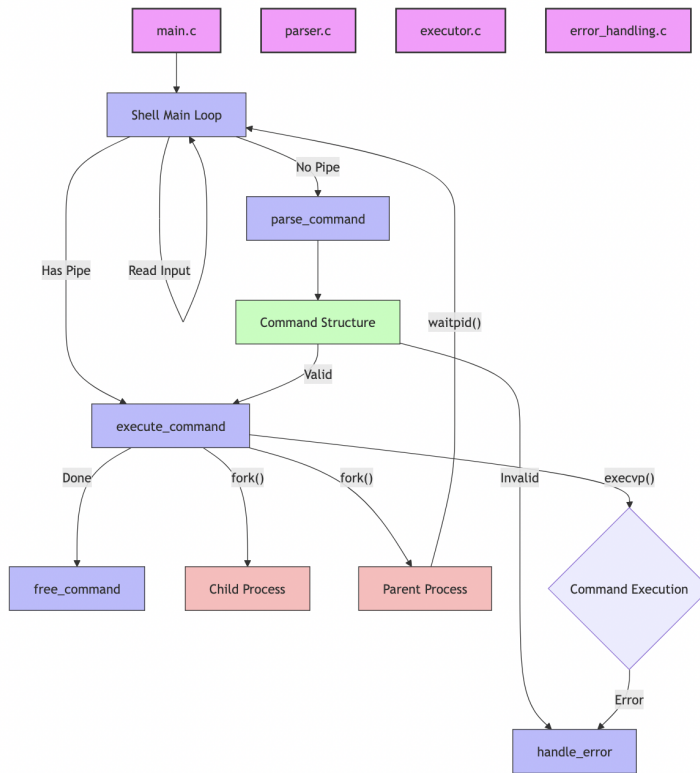


Figure 3: Main Flow and Execution

The pipeline and redirection architecture demonstrates (figure 3) how we handle complex data flows between processes and files using the UNIX I/O model. By carefully managing file descriptors and process relationships, we enable flexible command compositions while maintaining system resource integrity.

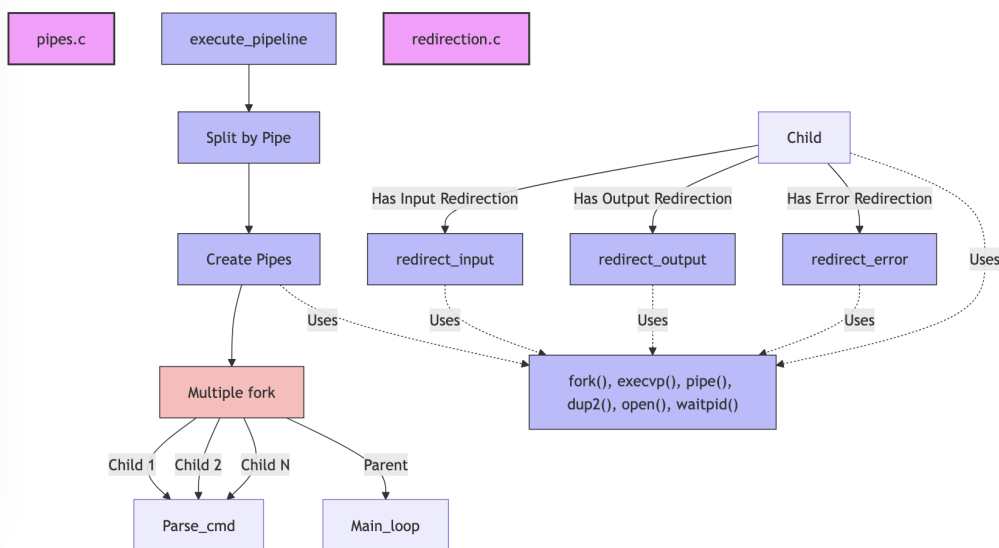


Figure 4: Pipeline, Redirection, and System Calls

1.8 Design Decisions and Alternatives Considered

Several key design decisions shaped our implementation:

1. **Command Representation:** We chose a structure-based approach over a more object-oriented design to maintain simplicity and performance. Alternative approaches like command objects with polymorphic execution methods would have added complexity without significant benefits in this context.
2. **Pipeline Implementation:** We implemented a multi-process pipeline using the standard UNIX pipe mechanism rather than using threads with shared memory buffers. This decision aligns with the UNIX philosophy and provides better isolation between command executions.
3. **Error Handling Strategy:** We opted for explicit error checking with specific error messages rather than using exceptions or error codes. This approach improves user experience by providing clear feedback about what went wrong.
4. **Memory Management:** We chose manual memory management with explicit allocation and deallocation rather than using garbage collection or reference counting. This gives us precise control over resource usage in the shell environment.

2 Implementation Highlights

2.1 Shell Architecture Overview

The myshell implementation is built around several key system calls from the POSIX API, primarily:

- `fork()`: Creates child processes to execute commands
- `execvp()`: Replaces the current process image with a new one
- `wait()/waitpid()`: Waits for child processes to complete
- `pipe()`: Creates a unidirectional data channel for process communication
- `dup2()`: Duplicates file descriptors for redirection
- `open()`: Opens files for redirection

2.2 Command Parsing Implementation

The parser transforms user input into a structured command representation. The parsing logic handles several complexity layers:

```
1 Command* parse_command(const char *input) {
2     // Initialize command structure
3     Command *cmd = malloc(sizeof(Command));
4     if (!cmd) {
5         perror("malloc");
6         return NULL;
7     }
```

```

8
9 // Initialize with null/default values
10 cmd->args = malloc(MAX_TOKENS * sizeof(char*));
11 cmd->input_file = NULL;
12 cmd->output_file = NULL;
13 cmd->error_file = NULL;
14 cmd->pipe_count = 0;
15
16 // Tokenize and process input
17 char *input_copy = strdup(input);
18 char *token = strtok(input_copy, " ");
19 int arg_index = 0;
20
21 // Parse tokens into command structure
22 while (token != NULL && arg_index < MAX_TOKENS - 1) {
23     // Handle redirection operators
24     if (strcmp(token, "<") == 0) {
25         token = strtok(NULL, " ");
26         if (!token) {
27             handle_error("Input file not specified");
28             free_command(cmd);
29             free(input_copy);
30             return NULL;
31         }
32         cmd->input_file = strdup(token);
33     }
34     else if (strcmp(token, ">") == 0) {
35         // Similar handling for output redirection
36         // ...
37     }
38     // Other cases...
39     else {
40         cmd->args[arg_index++] = strdup(token);
41     }
42     token = strtok(NULL, " ");
43 }
44
45 // Ensure NULL termination of args array
46 cmd->args[arg_index] = NULL;
47 free(input_copy);
48 return cmd;
49 }

```

Listing 1: Core parsing logic

Key implementation aspects:

- The parser creates a deep copy of the input string to avoid modifying the original
- It separates tokens by spaces and analyzes each token to identify commands, arguments, and operators
- Each redirection operator (<, >, 2>) causes the parser to store the following token as the respective redirection file
- For pipes, the parser increments a counter to track the number of pipe segments
- The parser implements robust error handling, checking for missing redirection targets and empty commands

2.2.1 Memory Management

Memory management is critical to prevent leaks and ensure proper cleanup:

```
1 void free_command(Command *cmd) {
2     if (!cmd) return;
3
4     // Free the argument array
5     if (cmd->args) {
6         for (int i = 0; cmd->args[i] != NULL; i++) {
7             free(cmd->args[i]);
8         }
9         free(cmd->args);
10    }
11
12    // Free redirection file paths
13    if (cmd->input_file) free(cmd->input_file);
14    if (cmd->output_file) free(cmd->output_file);
15    if (cmd->error_file) free(cmd->error_file);
16
17    // Free the command structure
18    free(cmd);
19 }
```

Listing 2: Command cleanup function

This function systematically frees all allocated memory, including:

- Each string in the args array
- The args array itself
- The redirection file paths
- The command structure

2.3 Command Execution

The executor handles command execution using the fork-exec pattern:

```
1 void execute_command(Command *cmd) {
2     pid_t pid = fork();
3
4     if (pid < 0) {
5         perror("fork");
6         return;
7     }
8
9     if (pid == 0) {
10        // Child process
11
12        // Handle redirections
13        if (cmd->input_file && redirect_input(cmd->input_file) != 0)
14            exit(EXIT_FAILURE);
15
16        if (cmd->output_file && redirect_output(cmd->output_file) != 0)
17            exit(EXIT_FAILURE);
18
19        if (cmd->error_file && redirect_error(cmd->error_file) != 0)
```

```

20         exit(EXIT_FAILURE);
21
22         // Execute the command
23         if (execvp(cmd->args[0], cmd->args) < 0) {
24             perror("execvp");
25             free_command(cmd);
26             exit(EXIT_FAILURE);
27         }
28     } else {
29         // Parent process: wait for child completion
30         int status;
31         waitpid(pid, &status, 0);
32     }
33 }

```

Listing 3: Command execution function

The execution process follows these steps:

1. Create a child process using `fork()`
2. In the child process:
 - Set up input/output/error redirections if specified
 - Use `execvp()` to replace the process image with the requested command
 - Handle execution failures with appropriate error messages
3. In the parent process:
 - Wait for the child to finish using `waitpid()`
 - Resume the shell loop after command completion

2.4 Redirection Implementation

File redirection is handled using the `dup2()` system call:

```

1 int redirect_input(const char *filename) {
2     // Open the input file for reading
3     int fd = open(filename, O_RDONLY);
4     if (fd < 0) {
5         perror("open input file");
6         return -1;
7     }
8
9     // Redirect stdin to the file
10    if (dup2(fd, STDIN_FILENO) < 0) {
11        perror("dup2 input");
12        close(fd);
13        return -1;
14    }
15
16    // Close the original file descriptor
17    close(fd);
18    return 0;
19 }

```

Listing 4: Input redirection implementation

The output and error redirection functions follow a similar pattern, but with different flags for `open()`:

```
1 int redirect_output(const char *filename) {
2     // Open for writing with create and truncate flags
3     int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
4     // Rest of implementation similar to redirect_input
5     // ...
6 }
```

Listing 5: Output redirection implementation

Key points about the redirection implementation:

- Each redirection function checks for errors during file opening and descriptor duplication
- For output and error redirection, files are created if they don't exist (with permissions 0644)
- Files are properly closed after redirection to prevent file descriptor leaks
- The redirection functions return error codes that are checked by the caller

2.5 Pipeline Implementation

Pipeline handling is one of the most complex parts of the shell:

```
1 void execute_pipeline(const char *input) {
2     // Split input into command segments
3     char *input_copy = strdup(input);
4     char *commands[MAX_COMMANDS];
5     int num_commands = 0;
6
7     // Tokenize on pipe symbol
8     char *token = strtok(input_copy, "|");
9     while (token != NULL && num_commands < MAX_COMMANDS) {
10         commands[num_commands++] = trim_whitespace(token);
11         token = strtok(NULL, "|");
12     }
13
14     // Create pipes
15     int pipefds[2 * (num_commands - 1)];
16     for (int i = 0; i < num_commands - 1; i++) {
17         if (pipe(pipefds + i * 2) < 0) {
18             perror("pipe");
19             free(input_copy);
20             return;
21         }
22     }
23
24     // Create processes for each command
25     for (int i = 0; i < num_commands; i++) {
26         pid_t pid = fork();
27         if (pid == 0) {
28             // Child process
29
30             // Set up pipe connections
```

```

31         if (i > 0) {
32             // Connect to previous command's output
33             dup2(pipefds[(i - 1) * 2], STDIN_FILENO);
34         }
35
36         if (i < num_commands - 1) {
37             // Connect to next command's input
38             dup2(pipefds[i * 2 + 1], STDOUT_FILENO);
39         }
40
41         // Close all pipe file descriptors
42         for (int j = 0; j < 2 * (num_commands - 1); j++) {
43             close(pipefds[j]);
44         }
45
46         // Parse and execute the command
47         Command *cmd = parse_command(commands[i]);
48         // ... execution code ...
49     }
50 }
51
52 // Parent: close pipes and wait for children
53 for (int i = 0; i < 2 * (num_commands - 1); i++) {
54     close(pipefds[i]);
55 }
56
57 for (int i = 0; i < num_commands; i++) {
58     wait(NULL);
59 }
60
61 free(input_copy);
62 }

```

Listing 6: Pipeline execution core logic

The pipeline implementation involves these critical steps:

1. Split the input string on pipe symbols to identify separate commands
2. Create the necessary pipes using the `pipe()` system call
3. For each command in the pipeline:
 - Create a child process
 - Configure stdin to read from the previous pipe (if not the first command)
 - Configure stdout to write to the next pipe (if not the last command)
 - Close all unnecessary pipe file descriptors
 - Parse and execute the command
4. In the parent process:
 - Close all pipe file descriptors
 - Wait for all child processes to complete

2.5.1 Pipeline Data Flow

For a pipeline like `cmd1 | cmd2 | cmd3`, the data flow is managed as follows:

```
1 // For cmd1 (i=0):
2 if (i < num_commands - 1) {
3     // Connect stdout to pipe[1]
4     dup2(pipefds[0*2 + 1], STDOUT_FILENO);
5 }
6
7 // For cmd2 (i=1):
8 if (i > 0) {
9     // Connect stdin to pipe[0]
10    dup2(pipefds[(1-1)*2], STDIN_FILENO);
11 }
12 if (i < num_commands - 1) {
13     // Connect stdout to pipe[3]
14     dup2(pipefds[1*2 + 1], STDOUT_FILENO);
15 }
16
17 // For cmd3 (i=2):
18 if (i > 0) {
19     // Connect stdin to pipe[2]
20     dup2(pipefds[(2-1)*2], STDIN_FILENO);
21 }
```

Listing 7: Pipeline data flow implementation

2.6 Error Handling

Error handling is implemented at multiple levels:

```
1 void handle_error(const char *msg) {
2     fprintf(stderr, "Error: %s\n", msg);
3 }
```

Listing 8: Error handling function

We handle various error cases throughout the shell:

- **System call errors:** All system calls (`fork`, `execvp`, `pipe`, etc.) are checked for errors using `perror()`
- **Missing redirection targets:** We verify that redirection operators are followed by valid filenames
- **Command not found:** The `execvp` failure is properly reported
- **Invalid pipe structures:** We check for empty commands between pipes
- **Memory allocation failures:** All `malloc` and `strdup` calls are checked for NULL returns

For example, when handling input redirection parsing:

```

1 if (strcmp(token, "<") == 0) {
2     token = strtok(NULL, " ");
3     if (!token) {
4         handle_error("Input file not specified");
5         free_command(cmd);
6         free(input_copy);
7         return NULL;
8     }
9     cmd->input_file = strdup(token);
10 }

```

Listing 9: Error handling for missing redirection target

2.7 Main Shell Loop

The main shell loop ties everything together:

```

1 int main() {
2     char input[MAX_INPUT_SIZE];
3
4     while (1) {
5         // Print prompt
6         printf("$ ");
7
8         // Get user input
9         if (!fgets(input, sizeof(input), stdin)) {
10             break; // EOF or error
11         }
12
13         // Remove trailing newline
14         input[strcspn(input, "\n")] = 0;
15
16         // Check for exit command
17         if (strcmp(input, "exit") == 0) {
18             break;
19         }
20
21         // Check for empty input
22         if (strlen(input) == 0) {
23             continue;
24         }
25
26         // Check for pipelines
27         if (strchr(input, '|')) {
28             execute_pipeline(input);
29         } else {
30             // Parse and execute a single command
31             Command *cmd = parse_command(input);
32             if (cmd) {
33                 execute_command(cmd);
34                 free_command(cmd);
35             }
36         }
37     }
38
39     return 0;
40 }

```

Listing 10: Main shell loop

The main loop provides the core interaction flow:

1. Display the shell prompt
2. Read a line of user input
3. Process special cases (exit command, empty input)
4. Determine if the command contains pipes
5. Execute the command appropriately (pipeline or single command)
6. Loop back for the next command

2.8 Advanced Features and Edge Cases

2.8.1 Handling Complex Command Combinations

For complex commands like `cmd1 < input.txt | cmd2 > output.txt`, we handle the redirections at different levels:

- Input redirection is handled by the first command in the pipeline
- Output redirection is handled by the last command
- Error redirection can be applied to any command in the pipeline

This is achieved by parsing each segment separately and applying the appropriate redirections:

```
1 // Inside the pipeline execution function
2 Command *cmd = parse_command(commands[i]);
3 if (!cmd) {
4     fprintf(stderr, "Parsing error in pipeline command.\n");
5     exit(EXIT_FAILURE);
6 }
7
8 // Handle redirects for this segment
9 if (cmd->input_file && i == 0) {
10     // Only apply input redirection for first command
11     if (redirect_input(cmd->input_file) != 0)
12         exit(EXIT_FAILURE);
13 }
14
15 if (cmd->output_file && i == num_commands - 1) {
16     // Only apply output redirection for last command
17     if (redirect_output(cmd->output_file) != 0)
18         exit(EXIT_FAILURE);
19 }
20
21 // Error redirection can be applied to any command
22 if (cmd->error_file) {
23     if (redirect_error(cmd->error_file) != 0)
24         exit(EXIT_FAILURE);
25 }
```

Listing 11: Handling complex redirection in pipelines

2.8.2 Whitespace Handling

To handle varying amounts of whitespace in commands, we implemented a helper function:

```
1 char *trim_whitespace(char *str) {
2     if (!str || *str == '\0') return str;
3
4     // Skip leading spaces
5     while (*str == ' ') str++;
6
7     // Trim trailing spaces
8     char *end = str + strlen(str) - 1;
9     while (end > str && (*end == ' ' || *end == '\n')) {
10         *end = '\0';
11         end--;
12     }
13
14     return str;
15 }
```

Listing 12: Whitespace trimming function

This function ensures that commands are correctly parsed regardless of extra spaces around operators.

2.9 Resource Management

To prevent resource leaks, we implemented careful file descriptor management:

```
1 // Close all pipe file descriptors
2 for (int j = 0; j < 2 * (num_commands - 1); j++) {
3     close(pipefds[j]);
4 }
```

Listing 13: File descriptor cleanup

This cleanup is performed in both the parent and child processes to ensure that file descriptors are properly released.

In summary, our implementation provides a robust shell with support for basic commands, redirections, and pipelines, with careful attention to error handling, resource management, and complex command combinations.

3 Execution Instructions

3.1 Compilation

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

```
make
```

This will compile all the source files and create the executable `myshell`.

3.2 Running the Shell

To start the shell, run:

```
./myshell
```

This will display a prompt (\$) where you can enter commands.

3.3 Using the Shell

The shell supports the following features:

- Basic commands: `ls`, `ps`, etc.
- Commands with arguments: `ls -l`, `ps aux`, etc.
- Input redirection: `command < input.txt`
- Output redirection: `command > output.txt`
- Error redirection: `command 2> error.log`
- Command pipelines: `command1 | command2`
- Combined operations: `command1 < input.txt | command2 > output.txt`

To exit the shell, type `exit`.

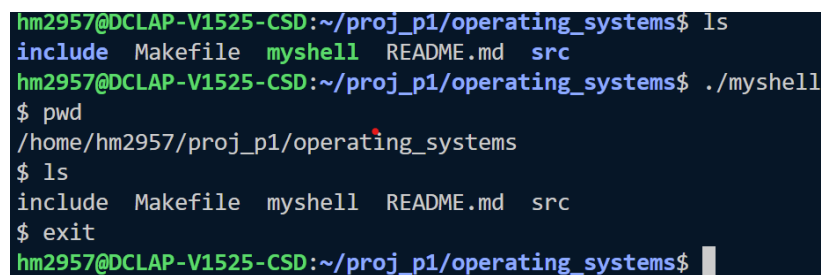
4 Testing

4.1 Basic Command Execution

Test Case: Simple command execution

```
$ ls
```

Output:

A terminal window with a dark background and light-colored text. The prompt is 'hm2957@DCLAP-V1525-CSD:~/proj_p1/operating_systems\$'. The user enters 'ls', and the output is 'include Makefile myshell README.md src'. The user enters './myshell', and the prompt changes to '\$'. The user enters 'pwd', and the output is '/home/hm2957/proj_p1/operating_systems'. The user enters 'ls', and the output is 'include Makefile myshell README.md src'. The user enters 'exit', and the prompt returns to 'hm2957@DCLAP-V1525-CSD:~/proj_p1/operating_systems\$'.

```
hm2957@DCLAP-V1525-CSD:~/proj_p1/operating_systems$ ls
include Makefile myshell README.md src
hm2957@DCLAP-V1525-CSD:~/proj_p1/operating_systems$ ./myshell
$ pwd
/home/hm2957/proj_p1/operating_systems
$ ls
include Makefile myshell README.md src
$ exit
hm2957@DCLAP-V1525-CSD:~/proj_p1/operating_systems$
```

Figure 5: Terminal output

4.2 Redirection Testing

Test Case: Output redirection

```
$ ls > output.txt  
$ cat output.txt
```

Expected Output: Directory listing saved in output.txt

Test Case: Input redirection

```
$ cat < output.txt
```

Output:

Test Case: Error redirection

```
$ ls /nonexistent 2> error.log  
$ cat error.log
```

Output:

4.3 Pipeline Testing

Test Case: Simple pipeline

```
$ ls | grep .c
```

Output:

Test Case: Multiple pipes

```
$ ls | grep .c | wc -l
```

Output:

4.4 Combined Operations Testing

Test Case: Input redirection with pipeline

```
$ cat < input.txt | grep keyword
```

Output:

Test Case: Pipeline with output redirection

```
$ ls | grep .c > c_files.txt
```

Output:

Test Case: Complex command

```
$ cat < input.txt | grep keyword | sort > sorted_output.txt
```

Output:

4.5 Error Handling Testing

Test Case: Invalid command

```
$ nonexistentcommand
```

Output:

Test Case: Missing redirection file

```
$ cat <
```

Output:

Test Case: Missing pipe command

```
$ ls |
```

Output:

5 Challenges

During the development of MyShell, we encountered several challenges:

5.1 Pipe Implementation

Challenge: Implementing a pipeline that supports any number of commands was complex, especially ensuring that file descriptors were properly set up and closed.

Resolution: We adopted a systematic approach, creating an array of pipe file descriptors and carefully managing their setup and cleanup. We also used a helper function to trim whitespace from commands to handle potential spaces around pipe symbols.

5.2 Memory Management

Challenge: Ensuring proper memory allocation and deallocation, especially for dynamically allocated command structures and string arrays.

Resolution: We implemented a comprehensive `free_command()` function to clean up all allocated resources and made sure to call it at appropriate points in the code to prevent memory leaks.

5.3 Error Handling

Challenge: Handling various error conditions, especially for combined operations like pipelines with redirections.

Resolution: We added error checking for all system calls and implemented specific error messages for different failure scenarios. We also created a dedicated error handling module to centralize error reporting.

5.4 Command Parsing

Challenge: Parsing complex command lines with multiple redirections and pipes was difficult to get right.

Resolution: We used a token-based approach with careful state tracking to properly parse commands. We also implemented extensive error checking to catch malformed commands.

6 Division of Tasks

The project responsibilities were divided as follows:

Team Member 1:

- Main shell loop implementation
- Command parsing
- Basic command execution
- Input/output/error redirection

Team Member 2:

- Pipeline implementation
- Error handling
- Testing and Report

We collaborated closely on the overall architecture and design decisions, and helped each other debug issues throughout the development process.

7 References

1. Stevens, W. R., & Rago, S. A. (2013). *Advanced Programming in the UNIX Environment*. Addison-Wesley. Available at: <https://xesoa.com/wp-content/uploads/2014/04/APUE-3rd.pdf>
2. GNU C Library Documentation. Available at: <https://www.gnu.org/software/libc/manual/pdf/libc.pdf>
3. Operating Systems Concepts, 10th Edition, by Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne
4. Stack Overflow discussion on pipes, dup2, and exec. Available at: <https://stackoverflow.com/questions/33884291/pipes-dup2-and-exec>

A Appendix: Source Code

This appendix contains the complete source code for the myshell implementation with detailed comments explaining each file's purpose and functionality.

A.1 Header Files

A.1.1 error_handling.h

```
1 #ifndef ERROR_HANDLING_H
2 #define ERROR_HANDLING_H
3
4 /*
5  * Function to handle and display error messages to the user.
6  * This provides a consistent error reporting mechanism across the
7  * shell.
8  *
9  * @param msg The error message to be displayed
10 */
11 void handle_error(const char *msg);
12 #endif // ERROR_HANDLING_H
```

Listing 14: error_handling.h: Error handling function declarations

A.1.2 executor.h

```
1 #ifndef EXECUTOR_H
2 #define EXECUTOR_H
3
4 #include "parser.h"
5
6 /*
7  * Executes a single command with its arguments and any specified
8  * redirections.
9  * Creates a child process to run the command while the parent waits
10  * for completion.
11  *
12  * @param cmd Pointer to the Command structure containing the command
13  * details
14  */
15 void execute_command(Command *cmd);
16 #endif // EXECUTOR_H
```

Listing 15: executor.h: Command execution interface

A.1.3 parser.h

```
1 #ifndef PARSER_H
2 #define PARSER_H
3
4 /*
5  * Command structure to store the parsed command information.
6  * This structure holds all necessary information about a command,
```



```

7  * including its arguments and any redirection specifications.
8  */
9  typedef struct Command {
10     char **args;           // NULL-terminated array of command
    arguments
11     char *input_file;      // Input redirection file (for < operator)
12     char *output_file;     // Output redirection file (for > operator)
13     char *error_file;      // Error redirection file (for 2> operator)
14     int pipe_count;        // Number of pipes detected in the command
15 } Command;
16
17 /*
18  * Parses a command string into a Command structure.
19  * Handles command arguments and redirection operators.
20  *
21  * @param input The command string to parse
22  * @return Pointer to an allocated Command structure or NULL on error
23  */
24 Command* parse_command(const char *input);
25
26 /*
27  * Frees all memory allocated for a Command structure.
28  * This includes the argument array and redirection file strings.
29  *
30  * @param cmd Pointer to the Command structure to free
31  */
32 void free_command(Command *cmd);
33
34 #endif // PARSER_H

```

Listing 16: parser.h: Command parsing structures and functions

A.1.4 pipes.h

```

1  #ifndef PIPES_H
2  #define PIPES_H
3
4  /*
5   * Executes a pipeline of commands connected with pipe operators.
6   * This function handles the parsing and execution of the entire
    pipeline,
7   * properly connecting the standard output of each command to the
8   * standard input of the next command in the pipeline.
9   *
10  * @param input The complete pipeline command string
11  */
12 void execute_pipeline(const char *input);
13
14 #endif // PIPES_H

```

Listing 17: pipes.h: Pipeline execution interface

A.1.5 redirection.h

```

1  #ifndef REDIRECTION_H
2  #define REDIRECTION_H

```

```

3
4 /*
5  * Redirects standard input from a specified file.
6  * Used to implement the < operator in the shell.
7  *
8  * @param filename The name of the file to redirect input from
9  * @return 0 on success, -1 on error
10 */
11 int redirect_input(const char *filename);
12
13 /*
14  * Redirects standard output to a specified file.
15  * Used to implement the > operator in the shell.
16  *
17  * @param filename The name of the file to redirect output to
18  * @return 0 on success, -1 on error
19 */
20 int redirect_output(const char *filename);
21
22 /*
23  * Redirects standard error to a specified file.
24  * Used to implement the 2> operator in the shell.
25  *
26  * @param filename The name of the file to redirect error output to
27  * @return 0 on success, -1 on error
28 */
29 int redirect_error(const char *filename);
30
31 #endif // REDIRECTION_H

```

Listing 18: redirection.h: I/O redirection functions

A.2 Implementation Files

A.2.1 error_handling.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "error_handling.h"
4
5 /*
6  * Implementation of the error handling function.
7  * Displays error messages to the user via standard error output.
8  *
9  * This simple implementation provides a consistent way to display
10 * error messages throughout the application. Could be expanded
11 * to include error codes, logging, etc.
12 */
13 void handle_error(const char *msg) {
14     fprintf(stderr, "Error: %s\n", msg);
15 }

```

Listing 19: error_handling.c: Error handling implementation

A.2.2 executor.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5 #include "executor.h"
6 #include "redirection.h"
7
8 /*
9  * Executes a single command with its arguments and redirections.
10  *
11  * This function handles:
12  * 1. Process creation via fork()
13  * 2. Setting up redirections in the child process
14  * 3. Executing the command using execvp()
15  * 4. Parent waiting for child completion
16  *
17  * The parent-child relationship ensures that the shell continues
18  * running while commands execute in separate processes.
19  */
20 void execute_command(Command *cmd) {
21     pid_t pid = fork();
22     if (pid < 0) {
23         // Error in fork() call
24         perror("fork");
25         return;
26     }
27
28     if (pid == 0) {
29         // Child process: set up redirections if specified
30         if (cmd->input_file) {
31             // Handle input redirection (< operator)
32             if (redirect_input(cmd->input_file) != 0)
33                 exit(EXIT_FAILURE);
34         }
35
36         if (cmd->output_file) {
37             // Handle output redirection (> operator)
38             if (redirect_output(cmd->output_file) != 0)
39                 exit(EXIT_FAILURE);
40         }
41
42         if (cmd->error_file) {
43             // Handle error redirection (2> operator)
44             if (redirect_error(cmd->error_file) != 0)
45                 exit(EXIT_FAILURE);
46         }
47
48         // Execute the command with its arguments
49         // execvp searches for the command in the PATH
50         if (execvp(cmd->args[0], cmd->args) < 0) {
51             perror("execvp");
52             free_command(cmd);
53             exit(EXIT_FAILURE);
54         }
55     } else {
56         // Parent process: wait for the child to finish
57         int status;
58         waitpid(pid, &status, 0);

```

```

59     }
60 }

```

Listing 20: executor.c: Command execution implementation

A.2.3 main.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "parser.h"
5  #include "executor.h"
6  #include "pipes.h"
7
8  #define MAX_INPUT_SIZE 1024
9
10 /*
11  * Main function that implements the shell loop:
12  * 1. Displays prompt
13  * 2. Reads user input
14  * 3. Parses and executes commands
15  * 4. Repeats until 'exit' command
16  *
17  * The shell handles both simple commands and pipelines.
18  */
19 int main() {
20     char input[MAX_INPUT_SIZE];
21
22     while (1) {
23         // Display shell prompt
24         printf("$ ");
25
26         // Read user input
27         if (!fgets(input, sizeof(input), stdin)) {
28             break; // Handle EOF (Ctrl+D)
29         }
30
31         // Remove trailing newline
32         input[strcspn(input, "\n")] = 0;
33
34         // Check for exit command
35         if (strcmp(input, "exit") == 0) {
36             break;
37         }
38
39         // If the input contains a pipe, use the pipeline executor
40         if (strchr(input, '|')) {
41             execute_pipeline(input);
42             continue;
43         }
44
45         // Otherwise, parse and execute a single command
46         Command *cmd = parse_command(input);
47         if (!cmd) {
48             fprintf(stderr, "Parsing error.\n");
49             continue;
50         }

```

```

51
52     // Execute the parsed command
53     execute_command(cmd);
54
55     // Free the command structure
56     free_command(cmd);
57 }
58
59 return 0;
60 }

```

Listing 21: main.c: Main shell loop implementation

A.2.4 parser.c

```

1  #define _POSIX_C_SOURCE 200809L
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include "parser.h"
6
7  #define MAX_TOKENS 64
8
9  /*
10 * Parses a command string into a Command structure.
11 *
12 * This function:
13 * 1. Tokenizes the input string on spaces
14 * 2. Identifies redirection operators (<, >, 2>) and their targets
15 * 3. Counts pipe operators (|)
16 * 4. Builds an argument array for the command
17 *
18 * The resulting Command structure contains all information needed
19 * to execute the command with the correct redirections.
20 */
21 Command* parse_command(const char *input) {
22     // Allocate the Command structure
23     Command *cmd = malloc(sizeof(Command));
24     if (!cmd) {
25         perror("malloc");
26         return NULL;
27     }
28
29     // Allocate array for command arguments
30     cmd->args = malloc(MAX_TOKENS * sizeof(char*));
31     if (!cmd->args) {
32         perror("malloc");
33         free(cmd);
34         return NULL;
35     }
36
37     // Initialize Command fields
38     cmd->input_file = NULL;
39     cmd->output_file = NULL;
40     cmd->error_file = NULL;
41     cmd->pipe_count = 0;
42

```

```

43 // Make a copy of the input string for tokenization
44 char *input_copy = strdup(input);
45 char *token = strtok(input_copy, " ");
46 int arg_index = 0;
47
48 // Process each token
49 while (token != NULL && arg_index < MAX_TOKENS - 1) {
50     if (strcmp(token, "<") == 0) {
51         // Input redirection: read the next token as the filename
52         token = strtok(NULL, " ");
53         if (token)
54             cmd->input_file = strdup(token);
55     } else if (strcmp(token, ">") == 0) {
56         // Output redirection: read the next token as the filename
57         token = strtok(NULL, " ");
58         if (token)
59             cmd->output_file = strdup(token);
60     } else if (strcmp(token, ">>") == 0) {
61         // Error redirection: read the next token as the filename
62         token = strtok(NULL, " ");
63         if (token)
64             cmd->error_file = strdup(token);
65     } else if (strcmp(token, "|") == 0) {
66         // Pipe operator: increment the pipe counter
67         cmd->pipe_count++;
68     } else {
69         // Normal argument: add to the args array
70         cmd->args[arg_index++] = strdup(token);
71     }
72     token = strtok(NULL, " ");
73 }
74
75 // Ensure there's a command specified
76 if (arg_index == 0) {
77     // No command specified
78     fprintf(stderr, "Error: No command specified.\n");
79     free_command(cmd);
80     return NULL;
81 }
82
83 // Terminate the arguments array with NULL
84 cmd->args[arg_index] = NULL;
85
86 // Free the temporary copy of the input
87 free(input_copy);
88 return cmd;
89 }
90
91 /*
92 * Frees all memory allocated for a Command structure.
93 *
94 * This includes:
95 * - Each string in the arguments array
96 * - The arguments array itself
97 * - The redirection file strings
98 * - The Command structure itself
99 */
100 void free_command(Command *cmd) {

```

```

101     if (!cmd) return;
102
103     // Free the argument strings and the arguments array
104     if (cmd->args) {
105         for (int i = 0; cmd->args[i] != NULL; i++) {
106             free(cmd->args[i]);
107         }
108         free(cmd->args);
109     }
110
111     // Free the redirection file strings
112     if (cmd->input_file) free(cmd->input_file);
113     if (cmd->output_file) free(cmd->output_file);
114     if (cmd->error_file) free(cmd->error_file);
115
116     // Free the Command structure itself
117     free(cmd);
118 }

```

Listing 22: parser.c: Command parsing implementation

A.2.5 pipes.c

```

1  #define _POSIX_C_SOURCE 200809L
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <unistd.h>
6  #include <sys/wait.h>
7  #include "pipes.h"
8  #include "parser.h"
9  #include "redirection.h"
10
11 // Maximum number of commands in a pipeline
12 #define MAX_COMMANDS 10
13
14 /*
15  * Helper function to trim leading and trailing whitespace from a
16  * string.
17  * @param str The string to trim
18  * @return Pointer to the trimmed string
19  */
20 char *trim_whitespace(char *str) {
21     if (!str || *str == '\0') return str; // Check for empty string
22
23     // Skip leading whitespace
24     while(*str == ' ') str++;
25
26     // Trim trailing whitespace
27     char *end = str + strlen(str) - 1;
28     while(end > str && (*end == ' ' || *end == '\n')) {
29         *end = '\0';
30         end--;
31     }
32
33     return str;

```

```

34 }
35
36 /*
37  * Executes a pipeline of commands connected with pipe operators.
38  *
39  * This function:
40  * 1. Splits the input into individual commands at pipe symbols
41  * 2. Creates necessary pipes to connect commands
42  * 3. Forks a process for each command
43  * 4. Sets up input/output redirections for each process
44  * 5. Waits for all processes to complete
45  *
46  * The implementation supports n number of commands in the pipeline
47  * and handles redirections within each command.
48  */
49 void execute_pipeline(const char *input) {
50     // Duplicate input to avoid modifying the original string
51     char *input_copy = strdup(input);
52     char *commands[MAX_COMMANDS];
53     int num_commands = 0;
54
55     // Split input on the pipe symbol
56     char *token = strtok(input_copy, "|");
57     while (token != NULL && num_commands < MAX_COMMANDS) {
58         commands[num_commands++] = trim_whitespace(token);
59         token = strtok(NULL, "|");
60     }
61
62     // Create pipes (num_commands-1 pipes needed for num_commands
63     // processes)
64     int pipefds[2 * (num_commands - 1)];
65     for (int i = 0; i < num_commands - 1; i++) {
66         // Each pipe has two file descriptors: read (2*i) and write (2*
67         // i+1)
68         if (pipe(pipefds + i * 2) < 0) {
69             perror("pipe");
70             free(input_copy);
71             return;
72         }
73     }
74
75     // Create a process for each command
76     for (int i = 0; i < num_commands; i++) {
77         pid_t pid = fork();
78         if (pid < 0) {
79             perror("fork");
80             free(input_copy);
81             return;
82         }
83
84         if (pid == 0) {
85             // Child process: set up pipe redirections
86
87             // If not the first command, redirect stdin from the
88             // previous pipe
89             if (i > 0) {
90                 if (dup2(pipefds[(i - 1) * 2], 0) < 0) {
91                     perror("dup2 stdin");
92                 }
93             }
94
95             // Execute the command
96             if (execvp(commands[i], &argv) < 0) {
97                 perror("execvp");
98                 _exit(1);
99             }
100         }
101     }
102
103     // Wait for all processes to complete
104     for (int i = 0; i < num_commands; i++) {
105         waitpid(-1, 0, 0);
106     }
107
108     // Free the input copy
109     free(input_copy);
110 }

```



```

89         exit(EXIT_FAILURE);
90     }
91 }
92
93 // If not the last command, redirect stdout to the current
pipe
94 if (i < num_commands - 1) {
95     if (dup2(pipefds[i * 2 + 1], 1) < 0) {
96         perror("dup2 stdout");
97         exit(EXIT_FAILURE);
98     }
99 }
100
101 // Close all pipe file descriptors in the child
102 for (int j = 0; j < 2 * (num_commands - 1); j++) {
103     close(pipefds[j]);
104 }
105
106 // Parse the individual command with its arguments and
redirections
107 Command *cmd = parse_command(commands[i]);
108 if (!cmd) {
109     fprintf(stderr, "Parsing error in pipeline command.\n");
110     exit(EXIT_FAILURE);
111 }
112
113 // Set up redirections for this command
114 if (cmd->input_file) {
115     if (redirect_input(cmd->input_file) != 0)
116         exit(EXIT_FAILURE);
117 }
118 if (cmd->output_file) {
119     if (redirect_output(cmd->output_file) != 0)
120         exit(EXIT_FAILURE);
121 }
122 if (cmd->error_file) {
123     if (redirect_error(cmd->error_file) != 0)
124         exit(EXIT_FAILURE);
125 }
126
127 // Execute the command
128 if (execvp(cmd->args[0], cmd->args) < 0) {
129     perror("execvp");
130     free_command(cmd);
131     exit(EXIT_FAILURE);
132 }
133 }
134 }
135
136 // Parent process: close all pipe file descriptors
137 for (int i = 0; i < 2 * (num_commands - 1); i++) {
138     close(pipefds[i]);
139 }
140
141 // Wait for all child processes to complete
142 for (int i = 0; i < num_commands; i++) {
143     wait(NULL);

```

```

144     }
145
146     // Clean up
147     free(input_copy);
148 }

```

Listing 23: pipes.c: Pipeline execution implementation

A.2.6 redirection.c

```

1  #include <stdio.h>
2  #include <fcntl.h>
3  #include <unistd.h>
4  #include "redirection.h"
5
6  /*
7   * Redirects standard input from a specified file.
8   *
9   * Implementation for the < operator:
10  * 1. Opens the specified file for reading
11  * 2. Duplicates the file descriptor to stdin (file descriptor 0)
12  * 3. Closes the original file descriptor
13  *
14  * @param filename The name of the file to redirect input from
15  * @return 0 on success, -1 on error
16  */
17 int redirect_input(const char *filename) {
18     // Open the file for reading only
19     int fd = open(filename, O_RDONLY);
20     if (fd < 0) {
21         perror("open input file");
22         return -1;
23     }
24
25     // Replace stdin with the opened file
26     if (dup2(fd, STDIN_FILENO) < 0) {
27         perror("dup2 input");
28         return -1;
29     }
30
31     // Close the original file descriptor (no longer needed)
32     close(fd);
33     return 0;
34 }
35
36 /*
37 * Redirects standard output to a specified file.
38 *
39 * Implementation for the > operator:
40 * 1. Opens/creates the specified file for writing
41 * 2. Duplicates the file descriptor to stdout (file descriptor 1)
42 * 3. Closes the original file descriptor
43 *
44 * @param filename The name of the file to redirect output to
45 * @return 0 on success, -1 on error
46 */
47 int redirect_output(const char *filename) {

```

```

48 // Open/create the file for writing
49 // O_TRUNC truncates the file if it exists
50 // 0644 sets read/write permissions for owner, read for group and
   others
51 int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
52 if (fd < 0) {
53     perror("open output file");
54     return -1;
55 }
56
57 // Replace stdout with the opened file
58 if (dup2(fd, STDOUT_FILENO) < 0) {
59     perror("dup2 output");
60     return -1;
61 }
62
63 // Close the original file descriptor
64 close(fd);
65 return 0;
66 }
67
68 /*
69 * Redirects standard error to a specified file.
70 *
71 * Implementation for the 2> operator:
72 * 1. Opens/creates the specified file for writing
73 * 2. Duplicates the file descriptor to stderr (file descriptor 2)
74 * 3. Closes the original file descriptor
75 *
76 * @param filename The name of the file to redirect error output to
77 * @return 0 on success, -1 on error
78 */
79 int redirect_error(const char *filename) {
80     // Same as redirect_output but for stderr
81     int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
82     if (fd < 0) {
83         perror("open error file");
84         return -1;
85     }
86
87     // Replace stderr with the opened file
88     if (dup2(fd, STDERR_FILENO) < 0) {
89         perror("dup2 error");
90         return -1;
91     }
92
93     // Close the original file descriptor
94     close(fd);
95     return 0;
96 }

```

Listing 24: redirection.c: I/O redirection implementation

A.3 Makefile

The project uses the following Makefile:

```

1 CC = gcc

```

```
2 CFLAGS = -Wall -Wextra -std=c99 -Iinclude
3 SRC = src/main.c src/parser.c src/executor.c src/redirection.c src/
  pipes.c src/error_handling.c
4 OBJ = $(SRC:.c=.o)
5 TARGET = myshell
6
7 all: $(TARGET)
8
9 $(TARGET): $(OBJ)
10  $(CC) $(CFLAGS) -o $(TARGET) $(OBJ)
11
12 src/%.o: src/%.c
13  $(CC) $(CFLAGS) -c $< -o $@
14
15 clean:
16  rm -f $(OBJ) $(TARGET)
```