Operating Systems Project Phase 1

Izah Sohail [is2587]
Sudiksha Kalepu [sk9806]

March 1, 2025

1 Architecture and Design

Our shell implementation follows a modular design with specialized functions for different command types. The architecture is structured around the following components:

1. **Command Parser**: In the main function, we parse user input into tokens and identify command types.
2. **Command Executors**: Separate functions handle different types of commands:
   - Simple commands
   - Input/output redirection
   - Pipe operations
   - Combined operations

1.1 Key Design Decisions

1. **Process-based Execution**: Each command executes in a separate forked process, allowing the parent shell to maintain control and continue operation after command execution.
2. **Function Specialization**: We implemented specialized handlers for different command types rather than a single complex function. This improves code readability and maintainability.
3. **Command Detection Logic**: We analyze the command string to detect pipes, redirections, and their combinations before deciding which handler to use.
4. **Error Handling**: Comprehensive error handling is implemented for all command types, with specific error messages for different failures.

1.2 Code Organization

The myshell.c file is structured into the following sections:

- Header includes and definitions
- Command handler functions
- Main function with command parsing and routing logic

2 Implementation Highlights

2.1 Core Functionalities

1. **Simple Command Execution** (`noArgCommand`):
   - Handles basic commands with/without arguments using fork() and execvp()
   - Parent process waits for child completion
2. **Output Redirection** (`handleRedirect`):
   - Supports `>` for stdout and `2>` for stderr redirection
   - Handles `2>&1` to redirect stderr to stdout
3. **Input Redirection** (`inputRedirect`):
   - Redirects stdin from a file using `<` operator
4. **Pipes** (`handlePipes`, `handleMultiplePipes`):
   - Single pipe implementation connects stdout of first command to stdin of second
   - Multiple pipes support n commands in sequence
5. **Combined Operations** (`handleCombinedRedirect`, `handlePipeRedirect`):
   - Supports combinations like `cmd < in.txt > out.txt`
   - Handles complex cases like `cmd1 < in.txt | cmd2 > out.txt | cmd3 2> err.log`

2.2 Key Implementation Logic

```c
// Pipe implementation using pipe(), fork(), and dup2()
int fd[2];
pipe(fd);

pid_t pid1 = fork();
if (pid1 == 0) {
    close(fd[0]);
    dup2(fd[1], STDOUT_FILENO);
    close(fd[1]);
    // Execute first command
}

pid_t pid2 = fork();
if (pid2 == 0) {
    close(fd[1]);
    dup2(fd[0], STDIN_FILENO);
    close(fd[0]);
    // Execute second command
}
```

Listing 1: Pipe implementation using pipe()

2.3 Error Handling

We implemented comprehensive error handling for various cases:

- Missing input/output files after redirection symbols
- Invalid or non-existent commands
- Empty commands between pipes
- Missing commands after pipes
- File opening failures

```c
if (command[i+1] == NULL) {
    fprintf(stderr, "Error: Input file not specified.\n");
    exit(1);
}
in_fd = open(command[i+1], O_RDONLY);
if (in_fd < 0) {
    fprintf(stderr, "Error: Cannot open input file '%s': %s\n",
            command[i+1], strerror(errno));
    exit(1);
}
```

Listing 2: Example error handling

3 Execution Instructions

3.1 Compilation

1. Make sure you have gcc installed
2. Navigate to the directory containing `myshell.c` and Makefile

3. Run the make command:

```
$ make
```

3.2 Running the shell

```
$ ./myshell
```

3.3 Basic Usage

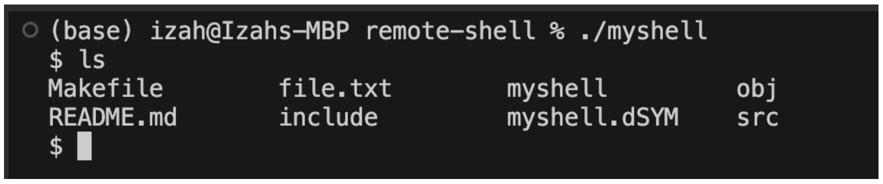
- Simple commands: `ls -l`, `ps aux`
- Redirection: `ls > output.txt`, `ls 2> error.log`
- Pipes: `ls | grep .c`, `cat file.txt | grep pattern | wc -l`
- Combined: `cat < input.txt > output.txt`, `cmd1 < in.txt | cmd2 > out.txt`
- Exit: Type `exit` to quit the shell

4 Testing

We performed the following tests to verify the functionality of our shell:

4.1 Simple Commands

- Test: `ls`
- Expected: Displays files in current directory
- Actual Output:

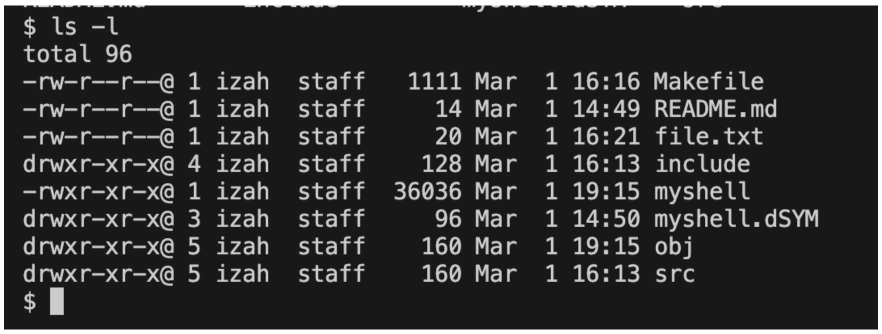


```
(base) izah@Izahs-MBP remote-shell % ./myshell
$ ls
Makefile        file.txt        myshell         obj
README.md       include         myshell.dSYM    src
$
```

Figure 1: Output of `ls` command

4.2 Commands with Arguments

- Test: `ls -l`
- Expected: Displays detailed file listing
- Actual Output:

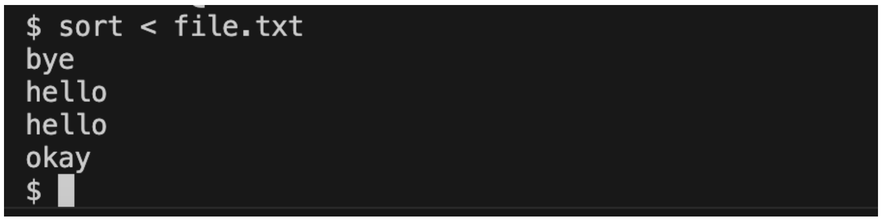


```
$ ls -l
total 96
-rw-r--r--@ 1 izah  staff   1111 Mar  1 16:16 Makefile
-rw-r--r--@ 1 izah  staff     14 Mar  1 14:49 README.md
-rw-r--r--@ 1 izah  staff     20 Mar  1 16:21 file.txt
drwxr-xr-x@ 4 izah  staff    128 Mar  1 16:13 include
-rwxr-xr-x@ 1 izah  staff  36036 Mar  1 19:15 myshell
drwxr-xr-x@ 3 izah  staff     96 Mar  1 14:50 myshell.dSYM
drwxr-xr-x@ 5 izah  staff    160 Mar  1 19:15 obj
drwxr-xr-x@ 5 izah  staff    160 Mar  1 16:13 src
$ 
```

Figure 2: Output of `ls -l` command

4.3 Input Redirection

- Test: `sort < file.txt`
- Expected: Sorts contents of `file.txt`
- Actual Output:

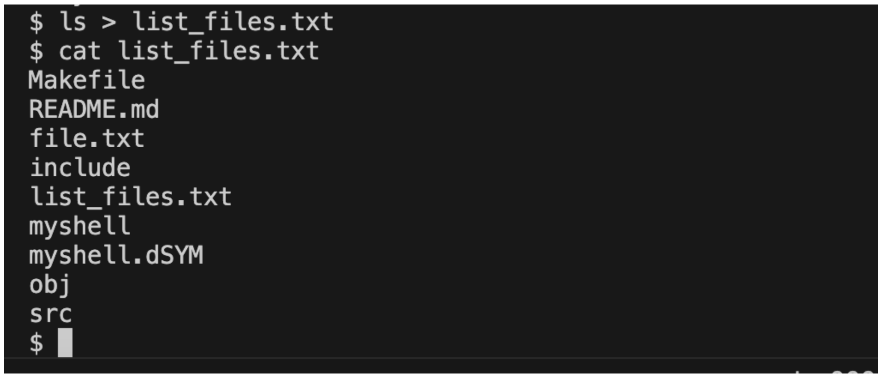


```
$ sort < file.txt
bye
hello
hello
okay
$ 
```

Figure 3: Output of `sort < file.txt` command

4.4 Output Redirection

- Test: `ls > list_files.txt`
- Expected: Creates `list_files.txt` with directory contents
- Actual Output:

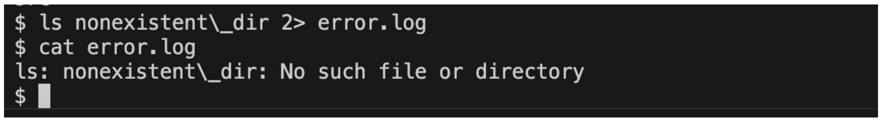


```
$ ls > list_files.txt
$ cat list_files.txt
Makefile
README.md
file.txt
include
list_files.txt
myshell
myshell.dSYM
obj
src
$ 
```

Figure 4: Output of `ls > list_files.txt` command

4.5 Error Redirection

- Test: `ls nonexistent_dir 2> error.log`
- Expected: Redirects error message to `error.log`
- Actual Output:

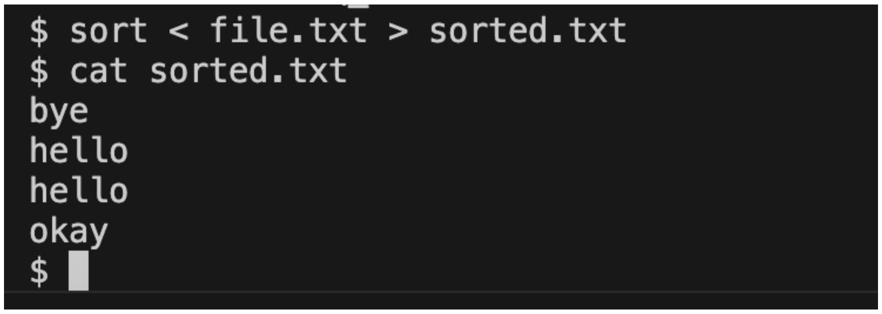


```
$ ls nonexistent\_dir 2> error.log
$ cat error.log
ls: nonexistent\_dir: No such file or directory
$ 
```

Figure 5: Output of `ls nonexistent_dir 2> error.log` command

4.6 Combined Redirections

- Test: `sort < file.txt > sorted.txt`
- Expected: Reads from `file.txt`, writes sorted output to `sorted.txt`
- Actual Output:

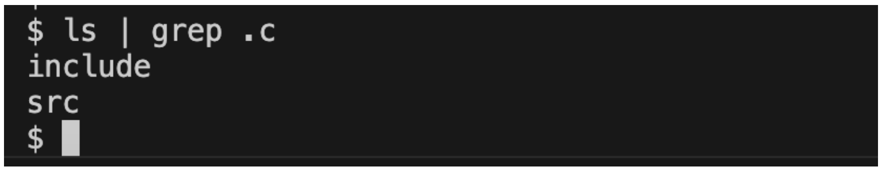


```
$ sort < file.txt > sorted.txt
$ cat sorted.txt
bye
hello
hello
okay
$
```

Figure 6: Output of `sort < file.txt > sorted.txt` command

4.7 Single Pipe

- Test: `ls | grep .c`
- Expected: Lists filenames containing any character followed by 'c'
- Actual Output:

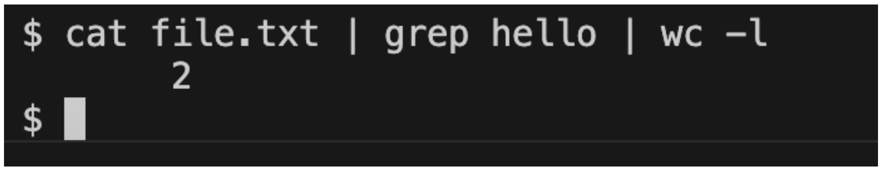


```
$ ls | grep .c
include
src
$
```

Figure 7: Output of `ls | grep .c` command

4.8 Multiple Pipes

- Test: `cat file.txt | grep hello | wc -l`
- Expected: Counts lines containing "hello" in file.txt
- Actual Output:

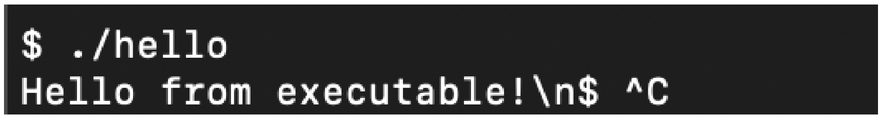


```
$ cat file.txt | grep hello | wc -l
       2
$ █
```

Figure 8: Output of `cat file.txt | grep hello | wc -l` command

4.9 Execute existing executable file

- Test: `./hello`
- Expected: Executes the file and prints hello
- Actual Output:

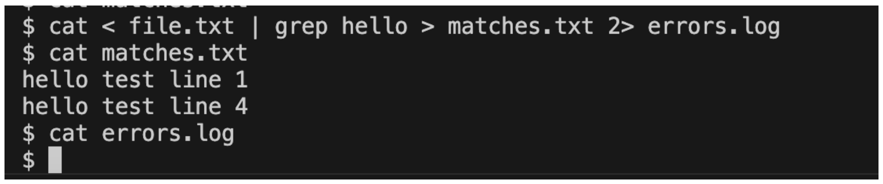


```
$ ./hello
Hello from executable!\n$ ^C
```

Figure 9: Output of `./hello` command

4.10 Complex Commands

- Test: `cat < file.txt | grep hello > matches.txt 2> errors.log`
- Expected: Reads from `file.txt`, finds lines with "hello", saves to `matches.txt`, redirects errors to `errors.log`.
- Actual output:



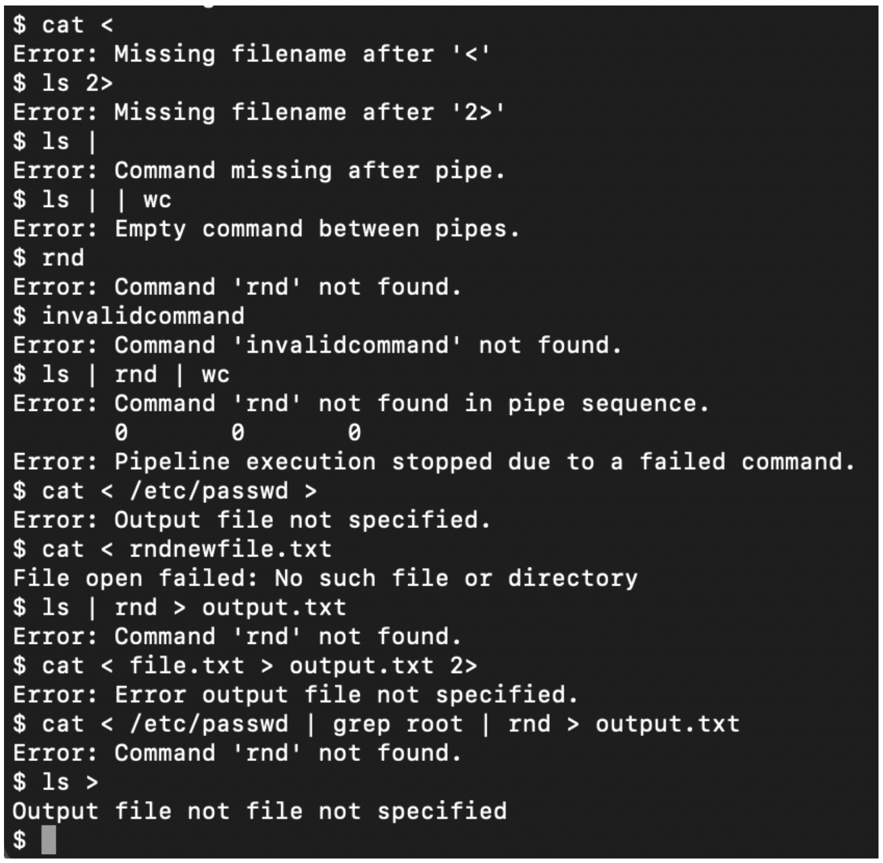
```
$ cat < file.txt | grep hello > matches.txt 2> errors.log
$ cat matches.txt
hello test line 1
hello test line 4
$ cat errors.log
$ 
```

Figure 10: Output of `cat < file.txt | grep hello > matches.txt 2> errors.log` command

4.11 Error Handling

- Test: `cat <`
  Expected: `Error:  Missing filename after '<'`
- Test: `ls 2>`
  Expected: `Error:  Missing filename after '2>'`
- Test: `ls |`
  Expected: `Error:  Command missing after pipe.`
- Test: `ls | | wc`
  Expected: `Error:  Empty command between pipes.`
- Test: `rnd`
  Expected: `Error:  Command 'rnd' not found.`
- Test: `ls | rnd | wc`
  Expected: `Error:  Command 'rnd' not found in pipe sequence.`
- Test: `cat < /etc/passwd >`
  Expected: `Error:  Output file not specified.`
- Test: `cat < rndnewfile.txt`
  Expected: `File open failed:  No such file or directory`
- Test: `ls | rnd > output.txt`
  Expected: `Error:  Command 'rnd' not found.`

- Test: `cat < file.txt > output.txt 2>`
  Expected: `Error:  Error output file not specified.`
- Test: `cat < /etc/passwd | grep root | rnd > output.txt`
  Expected: `Error:  Command ’rnd’ not found.`
- Test: `ls >`
  Expected: `Output file not specified`
- Actual Output:



```
$ cat <
Error: Missing filename after '<'
$ ls 2>
Error: Missing filename after '2>'
$ ls |
Error: Command missing after pipe.
$ ls | | wc
Error: Empty command between pipes.
$ rnd
Error: Command 'rnd' not found.
$ invalidcommand
Error: Command 'invalidcommand' not found.
$ ls | rnd | wc
Error: Command 'rnd' not found in pipe sequence.
       0       0       0
Error: Pipeline execution stopped due to a failed command.
$ cat < /etc/passwd >
Error: Output file not specified.
$ cat < rndnewfile.txt
File open failed: No such file or directory
$ ls | rnd > output.txt
Error: Command 'rnd' not found.
$ cat < file.txt > output.txt 2>
Error: Error output file not specified.
$ cat < /etc/passwd | grep root | rnd > output.txt
Error: Command 'rnd' not found.
$ ls >
Output file not file not specified
$ 
```

Figure 11: Error Handling

5 Challenges

During development, we encountered several challenges:

5.1 Multiple Pipe Implementation

- **Challenge**: Creating a dynamic number of pipes and properly connecting them between processes
- **Solution**: We implemented a loop that creates all pipes first, then forks processes and makes appropriate connections

5.2 Combined Redirections and Pipes

- **Challenge**: Handling complex combinations like `cmd1 < in.txt | cmd2 > out.txt`
- **Solution**: Created a specialized handler that processes each command segment separately, handling both pipe connections and file redirections

5.3 Error Handling

- **Challenge**: Detecting and responding to all possible error conditions
- **Solution**: Implemented comprehensive checking for missing files, invalid commands, and proper cleanup after errors

5.4 Command Detection Logic

- **Challenge**: Correctly identifying which handler to use for a given command
- **Solution**: Added a detection system that analyzes the command for pipes and redirection symbols before routing to the appropriate handler

6 Division of Tasks

6.1 Izah

- Single commands (with and without arguments)
- Input, output and error redirection
- Pipe implementation (single)
- Testing and debugging

6.2 Sudiksha

- Pipe implementation (multiple)
- Program to execute
- Composed compound commands
- Error handling

7 References

1. Linux Programmer's Manual: fork, exec, pipe, dup2
2. CS Course Materials and Lab Examples