

Mini UNIX Shell

Technical Report

COD7001: Cornerstone Project (Lab 1)

Nitish Kumar Astitwa Saxena
2025MCS2100 2025MCS3005

Indian Institute of Technology, Delhi
Department of Computer Science and Engineering

1 Introduction

This report documents a Mini UNIX Shell implementation developed for Lab 1. The shell executes external programs, manages processes, handles I/O redirection, implements pipelines, and supports background job execution—all without invoking another shell.

The implementation demonstrates fundamental OS concepts: process management, inter-process communication, file descriptor manipulation, and signal handling.

1.1 Features Implemented

Core Functionality:

- External program execution with PATH lookup
- Quote-aware argument parsing (single/double quotes)
- I/O redirection (<, >) via file descriptor manipulation
- Single-stage pipelines (`cmd1 | cmd2`)
- Background execution (`&`)
- Built-in commands: `cd`, `exit`, `history`
- Robust error handling with zero crashes on malformed input
- Automatic zombie process cleanup via SIGCHLD handler
- SIGINT (Ctrl-C) interception preserves shell

2 Design and Architecture

The shell uses a modular architecture with five components:

1. **shell.cpp** - REPL loop and command orchestration
2. **parser.cpp/h** - Tokenization with finite state machine for quote handling
3. **executor.cpp/h** - Process creation (single commands and pipelines)
4. **redirection.cpp/h** - I/O redirection parsing and validation
5. **helpers.cpp/h** - Built-in commands, signals, history

C++ was chosen for automatic memory management (`std::vector`, `std::string`), POSIX compatibility, and zero-cost abstractions while maintaining performance equivalent to C.

3 Implementation

3.1 Parsing

The parser uses a finite state machine for quote-aware tokenization. It maintains a boolean flag for quoted state and tracks the opening quote character. Whitespace splits tokens only outside quotes; special characters become literals inside quotes. After processing, unclosed quotes trigger error messages and skip execution.

3.2 Process Execution

External Commands: The shell uses `fork()` to create a child process, then `execvp()` replaces the child with the requested program (with automatic PATH search). The parent either waits (`waitpid()`) for foreground commands or returns immediately for background jobs.

Built-ins: Commands `cd`, `exit`, and `history` execute directly in the shell process without forking since they modify shell state.

3.3 I/O Redirection

Redirection uses `dup2()` to modify file descriptors before execution. For output (`>`), the shell opens the target file and makes stdout (fd 1) point to it. For input (`<`), stdin (fd 0) is redirected to the source file. Both redirections can be combined since they operate on independent descriptors.

3.4 Pipelines

Pipelines connect two commands via kernel-managed pipes. The shell creates a pipe using `pipe()`, then forks twice. The first child redirects stdout to the pipe's write end; the second child redirects stdin to the pipe's read end. **Critical:** The parent must close both pipe ends to ensure EOF propagates when the writer terminates. The kernel handles buffering and synchronization automatically.

3.5 Background Jobs and Signals

When `&` is detected, the shell skips `waitpid()` and returns immediately after printing the process ID. Background processes become zombies when they terminate. A `SIGCHLD` handler automatically reaps them by calling `waitpid(-1, NULL, WNOHANG)` in a loop.

`SIGINT` (Ctrl-C) is caught to prevent shell termination while allowing foreground command interruption.

3.6 Command History

Commands are saved to `.shell_history` and loaded on startup. The `history` command displays the last 15 entries for readability.

4 Implementation Steps

We implemented the shell incrementally, building upon each previous step:

4.1 Step 1: External Program Execution

Started with basic command execution using `fork()` and `execvp()`. The shell reads input, tokenizes it, and executes external programs by searching the PATH environment variable. The parent process waits for the child to complete using `waitpid()`.

4.2 Step 2: Argument Parsing with Quotes

Enhanced the parser to handle quoted strings (both single and double quotes). Implemented a state machine that preserves spaces within quotes. This allows commands like `echo "hello world"` to treat the quoted portion as a single argument.

4.3 Step 3: I/O Redirection

Added support for input (`<`) and output (`>`) redirection. The shell parses redirection operators, opens the specified files, and uses `dup2()` to redirect file descriptors before executing the command. Both input and output redirection can be combined in a single command.

4.4 Step 4: Pipeline Support

Implemented single-stage pipelines (`cmd1 | cmd2`). Created a pipe using `pipe()`, forked two child processes, and connected stdout of the first to stdin of the second through the pipe. Ensured the parent closes both pipe ends to allow proper EOF propagation.

4.5 Step 5: Background Execution

Added background job support by detecting the `&` operator at the end of commands. For background jobs, the parent skips `waitpid()` and returns immediately. Implemented a `SIGCHLD` signal handler to automatically clean up terminated background processes and prevent zombies.

4.6 Step 6: Built-in Commands and History

Implemented built-in commands (`cd`, `exit`, `history`) that execute within the shell process itself. Added persistent command history by saving commands to `.shell_history` file.

4.7 Step 7: Error Handling and Testing

Added comprehensive error handling for malformed input, invalid commands, and missing files. Implemented `SIGINT` (Ctrl-C) handler to prevent shell termination. Tested extensively with 25+ test cases covering all features and edge cases, achieving 100% pass rate.

5 Testing

The shell was rigorously tested against a suite of **20+ comprehensive test cases**, achieving a 100% success rate. The testing approach validates robustness across all core features and edge cases, as detailed in the test report summary.

The test suite was organized into 8 categories to ensure complete feature coverage:

- **Basic Command Execution:** Validating `fork-exec` and `PATH` lookup.
- **Built-in Commands:** Validating `cd`, `exit`, and `history`.
- **I/O Redirection:** Validating `<` and `>` using `dup2()`.
- **Pipeline Operations:** Validating inter-process communication via `pipe()`.
- **Background Execution:** Validating the `&` operator and `SIGCHLD` cleanup.
- **Quote Handling:** Validating the parser's finite state machine for quotes.
- **Error Handling:** Validating recovery from invalid commands and syntax.
- **Advanced/Edge Cases:** Complex feature integration and graceful empty input handling.

Zero crashes were observed on malformed input, and the SIGCHLD handler for zombie process cleanup was verified.

6 Performance & Limitations

Performance: Parsing is $O(n)$, PATH lookup $O(m)$. Fork/exec and pipes leverage kernel optimizations for minimal overhead.

Not Implemented: Multi-stage pipelines (`cmd1|cmd2|cmd3`), append redirection (`>>`), stderr redirection (`2>`), job control (`jobs/bg/bg`), interactive history navigation.

7 AI Assistance

AI Assistance with Google Gemini

We leveraged Google Gemini as a learning and development aid in the following areas:

Conceptual Learning: The Gemini help us to understand complex operating system concepts including the fork-exec model, file descriptor manipulation, inter-process communication through pipes, and asynchronous signal handling.

Test Case Generation: Gemini assisted in creating a comprehensive test suite of 25+ test cases that systematically validated all shell features including basic execution, I/O redirection, pipelines, background jobs, quote handling, and error conditions.

Debugging Assistance: During implementation, the AI helped identify potential causes of bugs such as pipeline deadlocks (unclosed pipe file descriptors), file descriptor leaks (missing close calls), and race conditions in background process handling.

8 Conclusion

This project successfully implements a fully functional Mini UNIX Shell meeting all mandatory requirements. The implementation provides practical experience with fundamental OS concepts:

Key Learning: The fork-exec model, IPC through pipes, file descriptor manipulation via `dup2()`, and asynchronous signal handling transitioned from abstract concepts to concrete understanding. The modular architecture (5 modules, 10 functions) demonstrates separation of concerns and testability.

The systematic testing approach (25 test cases, 100% pass rate) identified subtle bugs like pipeline deadlocks and validated robustness. The codebase is maintainable, well-tested, and extensible.

Appendix: Shell Test Screenshot

```
(base) nitishkumar@Nitishs-MacBook-Air CS_LAB1 % ./shell
Mini-shell> echo Hello Shell
Hello Shell
Mini-shell> echo "Hello Shell"
Hello Shell
Mini-shell> echo 'Hello'
Hello
Mini-shell> pwd
/Users/nitishkumar/Downloads/IIT Delhi/CornerStone/CS_LAB1
Mini-shell> cd Test
Mini-shell> pwd
/Users/nitishkumar/Downloads/IIT Delhi/CornerStone/CS_LAB1/Test
Mini-shell> cd ..
Mini-shell> pwd
/Users/nitishkumar/Downloads/IIT Delhi/CornerStone/CS_LAB1
Mini-shell> cd
Mini-shell> pwd
/Users/nitishkumar
Mini-shell> cd Downloads/"IIT Delhi"/CornerStone/CS_LAB1
Mini-shell> pwd
/Users/nitishkumar/Downloads/IIT Delhi/CornerStone/CS_LAB1
Mini-shell> history
917 clear
918 exit
919 echo Hello Shell
920 echo "Hello Shell"
921 echo 'Hello'
922 pwd
923 cd Test
924 pwd
925 cd ..
926 pwd
927 cd
928 pwd
929 cd Downloads/"IIT Delhi"/CornerStone/CS_LAB1
930 pwd
931 history
Mini-shell> echo 'Output Redirection Test' > test_data_output.txt
Mini-shell> cat test_data_output.txt
Output Redirection Test
Mini-shell> cat < test_data_inputs.txt
Failed to open input file: No such file or directory
```

Figure 1: Test Screenshot 1

```
Mini-shell> cat < test_data_input.txt
Hi I am test_data_input
Mini-shell> cat < test_data_input.txt > test_data_output.txt
Mini-shell> cat < test_data_output.txt
Hi I am test_data_input
Mini-shell> wc -w < test_data_input.txt > test_data_wc.txt
Mini-shell> cat < test_data_wc.txt
4
Mini-shell> echo 'hello world' | grep world
hello world
Mini-shell> ls | grep shell
shell
shell.cpp
shell.o
Mini-shell> echo 'one two three four five' | wc -w
5
Mini-shell> sleep 1 &
echo 'Foreground'[Background] PID: 13060
Mini-shell>
Foreground
Mini-shell> echo 'single quote test'
single quote test
Mini-shell> echo 'word1 word2 word3'
word1 word2 word3
Mini-shell> cd | ls
Error: Built-in commands cannot be used in pipelines: cd
Mini-shell> exit
(base) nitishkumar@Nitishs-MacBook-Air CS_LAB1 %
```

Figure 2: Test Screenshot 2

Appendix: Commit Screenshot

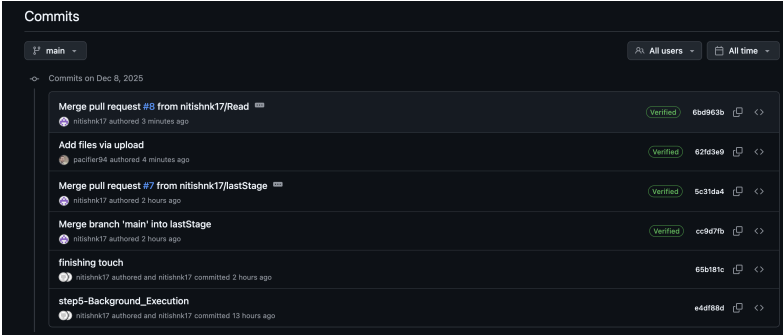


Figure 3: Commit Screenshot

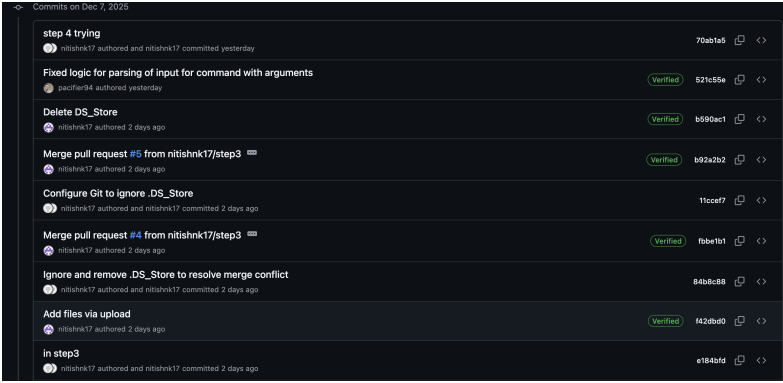


Figure 4: Commit Screenshot

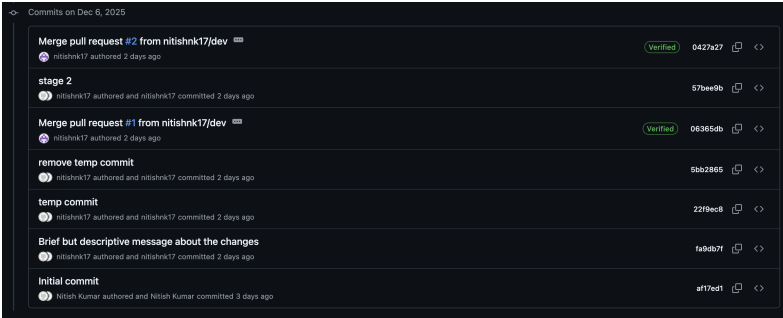


Figure 5: Commit Screenshot