Mini UNIX Shell

Technical Report

Saurabh Pandey (2025MCS), Maj Girish Singh Thakur (2025MCS2973)

December 2025

Abstract

This technical report documents the design, implementation, and testing of a UNIX-style command-line shell (myshell) developed in C. The shell implements core functionality including external program execution, argument parsing with quoted string support, I/O redirection, single-stage pipelines, background process execution, and built-in commands. The implementation demonstrates key operating system concepts such as process management, file descriptor manipulation, signal handling, and inter-process communication.

Contents

1 Introduction

1.1 Objectives

The primary objective of this project was to design and implement a functional UNIX shell that:

- Executes external programs using PATH resolution
- Parses command-line arguments including quoted strings
- Implements I/O redirection (stdin and stdout)
- Supports single-stage pipelines between commands
- Enables background process execution
- Provides essential built-in commands (cd, exit)
- Handles errors gracefully without crashing
- Manages signals appropriately (Ctrl-C, zombie processes)

1.2 System Requirements

- Operating System: UNIX-like (Linux, macOS, BSD)
- Compiler: GCC with C99 standard support
- Build System: GNU Make
- Standard Libraries: POSIX-compliant system calls

2 Architecture and Design

2.1 Overall Architecture

The shell follows a classic REPL (Read-Eval-Print Loop) architecture with the following major components:

1. **Input Handler:** Reads user input and performs initial preprocessing
2. **Tokenizer:** Breaks input into tokens (words, operators, quotes)
3. **Parser:** Constructs command structures from tokens
4. **Executor:** Forks processes and sets up execution environment
5. **Signal Manager:** Handles asynchronous signals (SIGINT, SIGCHLD)

2.2 Data Structures

2.2.1 Command Structure

```
typedef struct {
    char *argv[MAX_ARGV];     // Argument array
    char *infile;             // Input redirection file
    char *outfile;            // Output redirection file
    int background;           // Background flag
} Command;
```

This structure encapsulates all information needed to execute a single command, including arguments, I/O redirection targets, and execution mode.

2.3 Design Decisions

2.3.1 Tokenization Strategy

The tokenizer implements a state-machine approach that:

- Recognizes quoted strings (single and double quotes)
- Handles escape sequences within double quotes
- Treats operators (<, >, |, &, ;) as separate tokens
- Preserves whitespace within quoted strings
- Dynamically allocates token storage for flexibility

2.3.2 Process Management

Foreground Processes: The parent shell waits for foreground processes using `waitpid()`, ensuring synchronous execution and proper exit status collection.

Background Processes: For background jobs, the shell:

1. Prints the PID immediately after forking
2. Returns control to the user without waiting
3. Relies on SIGCHLD handler for zombie cleanup

2.3.3 Signal Handling

Two signal handlers are installed at startup:

- **SIGINT Handler:** Prevents Ctrl-C from terminating the shell while allowing it to interrupt foreground children
- **SIGCHLD Handler:** Automatically reaps zombie processes using non-blocking `waitpid()` with WNOHANG flag

3 Implementation Details

3.1 Core Functions

3.1.1 Tokenization (`tokenize_sb`)

The tokenizer scans the input string character by character:

```c
static int tokenize_sb(const char *line, char **tokens,
                       int max_tokens) {
    // Handles quotes, operators, and whitespace
    // Returns token count
    // Dynamically allocates each token
}
```

Key Features:

- Quote-aware parsing (preserves spaces in quoted strings)

- Escape sequence support in double quotes
- Operator detection (<, >, |, &)
- Whitespace handling

3.1.2 Command Parsing (parse_command_from_tokens_sb)

Converts token array into Command structure:

```
static int parse_command_from_tokens_sb(char **tokens,
    int start, int end, Command *cmd) {
    // Identifies redirection operators
    // Builds argv array
    // Sets background flag
    // Validates syntax
}
```

Error Detection:

- Missing filenames after redirection operators
- Misplaced ampersand (&) operator
- Excessive argument count

3.1.3 Single Command Execution (execute_single_sb)

Handles both built-in and external commands:

```
static int execute_single_sb(Command *cmd) {
    // Check for built-ins (cd, exit)
    // Fork for external commands
    // Setup I/O redirection in child
    // Execute via execvp()
    // Parent waits or returns (background)
}
```

I/O Redirection Setup:

1. Open file with appropriate flags
2. Use `dup2()` to redirect file descriptor
3. Close original file descriptor
4. Proceed with `execvp()`

3.1.4 Pipeline Execution (execute_pipe_sb)

Implements single-stage pipelines using unnamed pipes:

```
static int execute_pipe_sb(Command *left, Command *right) {
    int pipefd[2];
    pipe(pipefd);

    // Fork left command (producer)
    // Redirect stdout to pipe write end

    // Fork right command (consumer)
    // Redirect stdin to pipe read end
```

```c

    // Close pipe in parent
    // Wait for both children
}
```

Pipe Coordination:

- Left process writes to `pipefd[1]` (stdout)
- Right process reads from `pipefd[0]` (stdin)
- Parent closes both ends to avoid blocking
- Proper wait order prevents deadlocks

3.2 Built-in Commands

3.2.1 cd (Change Directory)

```c
if (strcmp(cmd->argv[0], "cd") == 0) {
    const char *dir = cmd->argv[1] ?
        cmd->argv[1] : getenv("HOME");
    if (chdir(dir) != 0)
        perror_continue("cd");
    return 0;
}
```

Rationale: Must be a built-in because changing directory in a child process doesn't affect the parent shell's working directory.

3.2.2 exit (Terminate Shell)

```c
if (strcmp(cmd->argv[0], "exit") == 0) {
    exit(0);
}
```

Rationale: Built-in to ensure clean shell termination and avoid spawning unnecessary processes.

3.3 Memory Management

Token Lifecycle:

1. Allocated in `tokenize_sb()` using `malloc()`
2. Used during parsing phase
3. Freed in `free_tokens_sb()` after command execution

Command Lifecycle:

1. Command structure populated with `strdup()`'d strings
2. Used during execution
3. Freed in `free_command_sb()` after execution

4 Challenges and Solutions

4.1 Challenge 1: Zombie Process Accumulation

Problem: Background processes become zombies if not reaped.

Solution: Installed SIGCHLD handler that calls `waitpid(-1, NULL, WNOHANG)` in a loop to reap all terminated children without blocking.

4.2 Challenge 2: Pipe File Descriptor Leaks

Problem: Parent must close both pipe ends to avoid blocking children.

Solution: Explicitly close `pipefd[0]` and `pipefd[1]` in parent after both children are forked.

4.3 Challenge 3: Quote Parsing Complexity

Problem: Handling quotes while preserving whitespace and recognizing escape sequences.

Solution: State-based tokenizer that tracks quote context and handles escape characters specially within double quotes.

4.4 Challenge 4: Semicolon vs. Pipe Precedence

Problem: Determining whether semicolons or pipes should be parsed first.

Solution: Split on semicolons first (outer level), then check for pipes within each subcommand. This allows constructs like: `cmd1 | cmd2 ; cmd3`

4.5 Challenge 5: Signal Safety in Child Processes

Problem: Child processes inherit parent's signal handlers.

Solution: Reset SIGINT to default (`SIG_DFL`) in child processes before `execvp()` to allow Ctrl-C to work normally in foreground programs.

5 Testing

5.1 Test Methodology

A comprehensive automated test suite (`tests/run_tests.sh`) validates all functional requirements:

Test Category	Test Count
Basic execution	2
I/O redirection	3
Pipelines	2
Background jobs	1
Built-ins	1
Error handling	2
Command sequencing	1
Total	**12**

Table 1: Test Suite Coverage

5.2 Test Results

All 12 tests pass successfully, validating:

- Correct program execution and output
- Proper I/O redirection behavior
- Pipeline data flow
- Background process creation
- Built-in command functionality
- Appropriate error messages for malformed input

5.3 Edge Cases Tested

1. Empty input lines
2. Commands with only whitespace
3. Misplaced operators (pipe at start/end)
4. Missing redirection filenames
5. Multiple redirections in same command
6. Quoted arguments with embedded spaces

6 Performance Analysis

6.1 Time Complexity

- **Tokenization:** O(n) where n is input length
- **Parsing:** O(t) where t is token count
- **Execution:** O(1) for single commands, O(2) for pipes

6.2 Space Complexity

- **Token storage:** O(t) for t tokens
- **Command structure:** O(a) for a arguments
- Maximum of MAX_TOKENS (256) and MAX_ARGV (128) enforced

7 Limitations and Future Work

7.1 Current Limitations

1. No multi-stage pipelines (only cmd1 | cmd2 supported)
2. No append redirection (>>)
3. No here-documents (<<)
4. No environment variable expansion ($VAR)

5. No wildcard globbing (*,?,[ ])
6. No command history
7. No job control (fg, bg, jobs)

7.2 Proposed Enhancements

1. **Multi-stage pipelines:** Recursive pipeline parsing
2. **Command history:** Linked list of previous commands
3. **Job control:** Process group management and terminal control
4. **Tab completion:** PATH and filename completion
5. **Environment variables:** Export/unset commands with $VAR expansion
6. **Scripting support:** If/while/for constructs

8 Conclusion

This project successfully demonstrates the core concepts of UNIX shell implementation including process management, I/O redirection, inter-process communication via pipes, and signal handling. The modular design separates concerns effectively (tokenization, parsing, execution) making the codebase maintainable and extensible.

Key achievements:

- Robust tokenizer handling quoted strings and operators
- Proper process lifecycle management (fork/exec/wait)
- Correct I/O redirection via file descriptor manipulation
- Functional pipeline implementation using unnamed pipes
- Safe signal handling preventing shell termination
- Comprehensive error checking and reporting
- Complete test suite validating all requirements

The implementation serves as a solid foundation for understanding operating system internals and can be extended with advanced features as outlined in the future work section.

9 References

1. Stevens, W. R., & Rago, S. A. (2013). *Advanced Programming in the UNIX Environment* (3rd ed.). Addison-Wesley.
2. Kerrisk, M. (2010). *The Linux Programming Interface.* No Starch Press.
3. POSIX.1-2017 (IEEE Std 1003.1-2017). *The Open Group Base Specifications.*
4. `fork(2)`, `execvp(3)`, `pipe(2)`, `dup2(2)` - Linux manual pages.