TECHNICAL REPORT
Mini UNIX Shell (AP_SHELL)

Assignment: Lab 1
Language: C (GCC Compiler)

Team: Abhinav Gupta (2025MCS2089)
Pratik Chaudhari (2025MCS2096)

---

1. Introduction

AP_SHELL is our implementation of a custom UNIX-style command-line interface, as per the requirement of lab 1 assignment. The shell operates as a **Read-Eval-Print Loop (REPL)**. It is capable of parsing user input, managing process creation, handling I/O redirection and executing system programs, in-build commands etc. It is implemented in C with a modular code structure, making low-level system calls to manage processes and memory directly.

2. Code Structure

The shell is built as a modular program, splitting functionality across multiple source files for maintainability. The core lifecycle of the shell program is as follows:

1. **Initialization:** we first setup signal handlers and memory structs (LRU History).
2. **Prompt:** the current working directory(color coded) is displayed as prompt for input.
3. **Input parsing:** we read raw input string and split (tokenize) it into arguments, handling quoted strings and various operators like | , &, && etc.
4. **Execution:** we determine if a command is a built-in or an external program and route it accordingly.
5. **Process management:** we are using **fork()** and **exec()** calls to run various commands.

File Structure

- **main.c**: It is the central entry point containing the REPL loop and tokenization logic.
- **apsh_module.h**: It is the main header file of our program defining all function prototypes and data structures used.
- **apsh_execute_pipeline.c**: It handles inter-process communication via pipes.
- **lru_history.c**: It implements our custom logic for history command as a **Least Recently Used (LRU) cache**.
- **built-in modules**: apsh_cd.c, apsh_exit.c, apsh_export.c, apsh_background.c.

3. Implementation

3.1 Input parsing & tokenization

Since standard strtok was insufficient for the requirement to handle **quoted strings**, a custom tokenizer (**tokenize_input**) has been implemented in main.c. It iterates through the input string using a pointer and handles following scenarios:

- **Whitespace handling:** we have skipped leading spaces and split arguments by standard whitespace delimiters (**\t\r\n\a**).

- **Quotes handling:** we detected double quotes (") and single quotes (') and treated the enclosed text as a single token, preserving internal spaces. This allowed commands like echo "Hello World" to be parsed correctly as two arguments than three.

3.2 Process creation & execution

External commands have been implemented using the **Fork-Exec-Wait** pattern:

1. **fork()**: It creates a child process (i.e. duplicate of the shell).
2. **execvp()**: It replaces the child's memory with the requested program. **execvp** has been used specifically to satisfy requirement of using PATH lookup.
3. **waitpid()**: The parent process waits for child to change state. The **WUNTRACED** flag is used to detect if a child has been stopped or not.

3.3 I/O redirection

Input(**<**) & output(**>**) redirection is handled inside the child process before **execvp** is called.

- The argument list is scanned for redirection symbols.
- **Output(>):** The target file is opened with **O_WRONLY | O_CREAT | O_TRUNC** flags. The **dup2(fd, STDOUT_FILENO)** system call replaces standard output stream with file descriptor.
- **Input(<):** The target file is opened with **O_RDONLY** flag and **dup2**(fd, STDIN_FILENO) replaces the standard input stream.

3.4 Pipelines

Pipe support (**|**) is implemented in **apsh_execute_pipeline.c**.

1. **Splitting:** The main loop finds the pipe symbol and splits the arguments into **left_args** and **right_args**.
2. **Pipe creation: pipe(pfd)** creates a unidirectional data channel.
3. **Process chaining:**
    a. **Child 1 (Left Command):** It connects STDOUT to write-end of the pipe (pfd[1]).
    b. **Child 2 (Right Command):** It connects STDIN to the read-end of the pipe (pfd[0]).
4. **Synchronization:** The parent then closes both pipe ends and waits for both children to finish, ensuring no orphaned processes.

3.5 Logical operator (&&)

We have also implemented the support for the logical AND operator (**&&**). The function **apsh_execute_with_and** recursively splits the commands. It executes first command and executes the second only if the first returns a success status.

3.6 Signal handling

- **SIGINT (for Ctrl-C):** It is caught by **handle_sigint**. Rather than terminating the shell, we are printing a new line and displaying the prompt as per standard shell behavior.

- **SIGCHLD:** It is caught by **handle_sigchld** to clean up zombie processes. It uses **waitpid(-1, NULL, WNOHANG)** to obtain background processes which have finished without blocking the main shell loop.

3.7 Built-in commands

Commands which required modifying the shell's own state have been implemented as built-ins:

- **cd:** It uses **chdir()** to change current working directory.
- **exit:** It returns 0 to terminate the main while loop.
- **export (advanced feature):** It uses **setenv()** to modify environment variables.

3.8 LRU history (Advanced Feature)

Instead of providing simple array history, it has been implemented as a Least Recently Used (LRU) cache using a doubly linked list.

- It stores the most recent commands (set to 20 in our implementation).
- **Logic:** when command is typed, it is added to the head. If it already exists, it is moved to the front. If capacity is exceeded, the tail (least recently used) is removed. It would ensure that the history remains relevant and memory-efficient.

4. Challenges & solutions

- **Zombie processes:** Initially, background processes (&) would remain as "defunct" in the system table. It has been solved by registering a SIGCHLD handler that asynchronously reaps children.
- **Parsing quotes:** strtok is unable to handle quotes. So, we have writen a manual pointer-based parser, allowing for complex arguments containing spaces.

5. Conclusion

AP_SHELL implements all mandatory functional requirements of the assignment, including process management, parsing, and pipelines. It further extends functionality with advanced features like environment variable export, logical operators (&&), and an LRU cache based history.