PROJECT REPORT ON

# DESIGN AND IMPLEMENTATION OF CUSTOM UNIX SHELL

SUBMITTED IN PARTIAL FULFILMENT
OF
THE REQUIREMENTS OF
THE AWARDS OF THE

**PG-Diploma in Advanced Secure Software Development**

**Offered By**
**C-DAC Hyderabad**

BY


**NAME : ARPAN TIWARI** **ROLL NO : 250850329006**
**NAME : NAISHA DEKATE** **ROLL NO : 250850329024**
**NAME : RIYA GUPTA** **ROLL NO : 250850329031**
**NAME : VAMSI G D N** **ROLL NO : 250850329039**



**Advanced Computing Training School**
**C-DAC**
**Hyderabad-501510**

**AUGUST 2025**

# CERTIFICATE

This is to certify that this is a bonafide record of a project entitled **"Design and Implementation of Custom Unix Shell"**. Arpan Tiwari (250850329006), Naisha Dekate(250850329024), Riya Gupta(350850329031), Vamsi G D N (250850329039) has completed project work as part of **Diploma in Advanced Secure Software Development (August 2025 Batch)**, a PG course offered by C-DAC Hyderabad. They have completed project work under the supervision of Mr. **Mallisetti Bala Manikanta**. Their performance found to be good.

**Name of Project guide**

(Mr. Mallisetti Bala Manikanta)

(Project Engineer)

**DATE** : 04/02/2026

**PLACE** : C-DAC, HARDWARE PARK, HYDERABAD.

# ACKNOWLEDGEMENT

"Design and Implementation of Custom Unix Shell" project has presented an objective, a goal, a challenge of data security. This project marks the final hurdle that we tackle, of hopefully what would be one of the many challenges we have taken upon and am yet to take. However, we could not have made it without the support and guidance from the following. Firstly I want to take this opportunity to have special thanks to our guide **Mr. Mallisetti Bala Manikanta** who helped us throughout this project by providing valuable guidance and advice as well as acquiring all components needed for this project to become a success.

(PG-DASSD AUGUST 2025)

Arpan Tiwari(250850329006)

Naisha Dekate(250850329024)

Riya Gupta(350850329031)

Vamsi G D N(250850329039)

## TABLE OF CONTENTS

# DESIGN AND IMPLEMENTATION OF CUSTOM UNIX SHELL


## ABSTRACT :

The development of MyShell addresses the limitations of traditional command-line interfaces by evolving the standard reactive REPL (Read-Eval-Print Loop) architecture into a proactive, behavior-aware system. Built entirely in C, the project implements a modular POSIX-compliant engine capable of managing complex process lifecycles through robust system calls like fork(), execvp(), and waitpid(). A key technical highlight is the integration of an Intelligence Layer that tracks user command patterns. By implementing a frequency-based heuristic, the shell identifies repetitive workflows and suggests custom aliases, effectively transforming the terminal from a simple execution tool into a smart development assistant.

To ensure high-level accountability and system integrity, security features are deeply embedded into the shell's core operations. The project features a Security Sentinel that proactively intercepts high-risk system commands, such as mount or fdisk, and verifies the caller's identity via the getuid() system call before execution. To ensure non-repudiation and forensic traceability, every interaction is recorded in a Forensic Audit Logger. This logger captures a precise execution timestamp, the active User ID, and the full command string, providing a secure audit trail essential for sensitive computing environments. Beyond basic execution, MyShell optimizes monitoring through Inode-based tracking via stat() metadata, ensuring consistent monitoring even if files are renamed or moved within the filesystem.

## IMPLEMENTATION:

The implementation of MyShell follows a modular C-based architecture designed for high performance and system-level control. The core of the shell is the REPL Engine, which manages the continuous cycle of reading user input, evaluating commands, and printing results. Input processing is handled by a custom parser that tokenizes strings and performs Lazy Alias Expansion, replacing user-defined shortcuts with their full command equivalents before execution. For system-level commands, the shell utilizes the Fork-Exec model. A child process is created via fork(), which then calls execvp() to replace its image with the target executable, while the parent shell uses waitpid() to monitor the child's exit status and prevent the creation of zombie processes.

A critical component of the implementation is the Security and Auditing Layer. Every command passed to the executor is first intercepted by a Sentinel Function. This function checks the command against a restricted list and validates the user’s privileges using the getuid() system call. Simultaneously, the Audit Logger opens shell_audit.log in append mode, utilizing the <time.h> library to generate human-readable timestamps for every session activity. For advanced monitoring, the watch utility was implemented using a Split-usleep Loop. This strategy replaces blocking sleep() calls with a series of 100ms usleep() intervals, allowing the shell to remain responsive to SIGINT (Ctrl+C) signals while performing periodic stat() calls to track file changes via permanent Inode numbers.

# CHAPTER 1: INTRODUCTION

## 1.1 Background

In Unix-based operating systems, the shell serves as a bridge between the user and the system kernel. It allows users to give instructions through commands and receive results instantly. While popular shells like **Bash** and **Zsh** offer many advanced features, their internal workings are often hidden from the user. Creating a custom shell provides a practical way to understand how commands are processed and executed internally.

## 1.2 Problem Statement

Although existing Unix shells are powerful, their complexity makes them difficult for beginners to fully understand. MyShell also addresses proactive security and detailed accountability by preventing unauthorized command execution through a Security Sentinel and providing an immutable, file-based audit trail. There is a clear need for a simple shell that demonstrates the core principles of command execution and process management in a straightforward and practical way.

## 1.3 Objectives

The main objectives of this project are to:

- Design and develop a simple Unix shell.
- Understand how commands are parsed and executed.
- Implement basic built-in commands like cd and exit.
- Gain hands-on experience with Unix system calls and process management.

## 1.4 Scope of the Project

The project scope centers on building a high-performance command-line interface in C that manages core Unix process lifecycles. It implements essential built-in commands and secure binary execution while maintaining a forensic shell_audit.log for accountability. While advanced I/O redirection and job control are currently excluded, the modular architecture provides a scalable foundation for these future systems-level enhancements.

# CHAPTER 2: LITERATURE SURVEY

## 2.1 Existing Systems

Existing Unix shells such as Bash provide a wide range of features including scripting support, command history, piping, and redirection. These shells are highly optimized and widely used in real-world systems.

## 2.2 Limitations of Existing Systems

- Internally complex and difficult to study
- Not suitable for beginners learning shell implementation
- Contain many advanced features that are unnecessary for basic understanding

## 2.3 Motivation

The motivation behind this project is to simplify the concept of a Unix shell and provide a learning platform that helps us to understand the internal working of command-line interpreters. This project also addresses the lack of built-in accountability and security in standard Linux terminals. By developing a custom shell with integrated auditing and a security sentinel, we provide a transparent, monitored environment that protects system integrity while serving as a robust platform for learning core systems programming.

# CHAPTER 3: SYSTEM ANALYSIS

## 3.1 Proposed System

- **Integrated Forensic Logging:** Automatically records every command, timestamp, and User ID into a persistent shell_audit.log.
- **Security Sentinel Layer:** Intercepts commands to validate user privileges before allowing execution of high-risk tasks.
- **Behavioral Intelligence:** Analyzes command frequency to offer smart suggestions and improve user workflow.

## 3.2 Advantages of the Proposed System

- **Enhanced Accountability:** Provides an unalterable record of all user activities for security auditing.
- **Zero External Dependencies:** Operates entirely on native Linux APIs without requiring heavy third-party libraries.
- **Low Resource Overhead:** Optimized for performance, running smoothly within minimal VirtualBox RAM allocations.
- **Security by Design:** Proactively blocks unauthorized attempts to access sensitive system binaries.

## 3.3 Feasibility Study

- **Technical Feasibility:** Developed using the GCC compiler and standard POSIX libraries (like unistd.h) available on all Linux distros.
- **Economic Feasibility:** Utilizes open-source tools (Ubuntu, VirtualBox, C), making it a zero-cost development project.
- **Operational Feasibility:** Requires no special training for users already familiar with basic Linux command-line syntax.
- **Legal & Compliance Feasibility:** Adheres to security best practices by ensuring data privacy through local, non-cloud log storage.

# 1. INTRODUCTION

The command-line interface (CLI) is an essential gateway to an operating system's kernel, yet standard shells like Bash remain largely reactive. MyShell addresses this by evolving the traditional terminal into an intelligent, secure assistant. Built in C, it implements a POSIX-compliant REPL (Read-Eval-Print Loop) architecture that prioritizes both user efficiency and system integrity. By utilizing fundamental system calls such as fork(), execvp(), and waitpid(), the software ensures reliable process management and resource isolation.

A distinctive pillar of MyShell is its Behavior-Aware Intelligence Layer, which analyzes command frequency to suggest optimizations, thereby reducing cognitive load. Furthermore, the project follows a "Secure by Design" philosophy. It mitigates the risk of unauthorized system changes by implementing Access Control Sentinels that verify user privileges via getuid() and a Forensic Audit Logger that maintains persistent, timestamped records in shell_audit.log. Additionally, it introduces Inode-based tracking via stat(), providing immutable file monitoring that remains effective even if filenames are modified. This combination of features balances low-level kernel power with advanced security and intelligent automation.

## 1. PROJECT REQUIREMENTS

### 1.1 Problem Statement

Working within a Linux virtualized environment, MyShell addresses several critical security and operational gaps:

**USB/Peripheral Vulnerability:** In a virtualized setup, once a USB device is "passed through" to the Ubuntu VirtualBox, the system is exposed to automated script attacks that can execute in the background.

**Unauthorized Terminal Access:** Standard Linux terminals lack a proactive notification system to alert the owner when high-impact commands are being executed by another party.

**Silent Data Exfiltration:** There is no human-readable, persistent audit trail that specifically monitors file movements to mounted media or external storage in real-time.

**Log Volatility:** Standard history files are easily modified or deleted; there is a need for a secure, append-only log that captures technical metadata like User IDs.

**Activity Monitoring Gaps:** Existing utilities do not provide integrated, live feedback on file-system changes the moment a peripheral is interacted with via the shell.

## 1.2 Software & Hardware Requirements

### 1.2.1 Hardware Specification (VirtualBox Allocation)

To ensure the shell runs efficiently within the VM environment, the following resources are allocated:

**Processor:** 2 vCPUs (Dual Core) — essential for handling the Fork-Exec process model and background monitoring.

**Hard Disk:** 50 GB (Dynamic Allocation) — formatted with ext4 to support Inode-based tracking via stat().

**Memory:** 1GB RAM — allocated to the VirtualBox instance, showcasing the shell's low memory footprint.

### 1.2.2 Software Specification

The project is built on a native Linux stack:

**Operating System:** Ubuntu Linux (running on VirtualBox).

**Database Server:** MySQL Server 8.0 — stores habit history and forensic audit logs.

**Compiler/IDE:** GCC (GNU Compiler Collection) and VS Code for C development.

**OpenCV:** OpenCV 4.x (C++ Interface) — utilized for real-time security alerts.

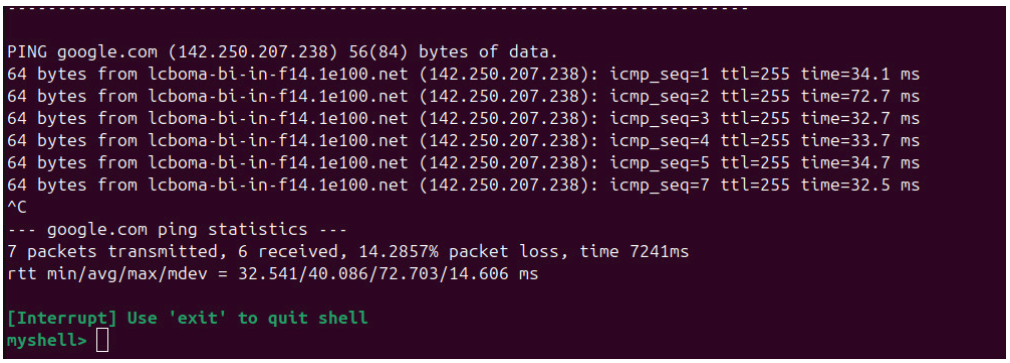
## 1.3 Coding and Testing

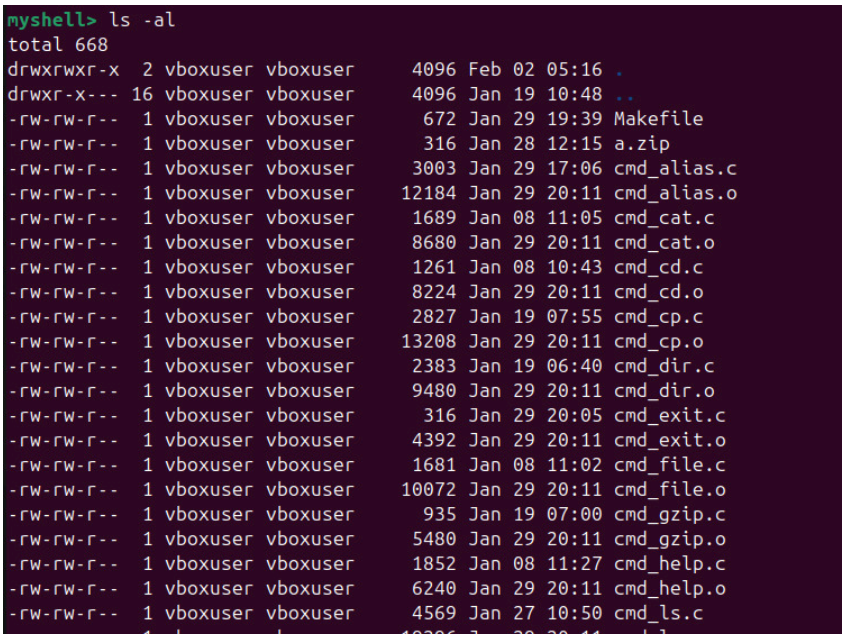
The project uses a cross-language approach to manage kernel-level operations and data visualization:

**Systems Programming (C):** The core engine is written in C. It uses Linux system calls like fork(), execvp(), and waitpid() for process management, and stat() for Inode-level file tracking.

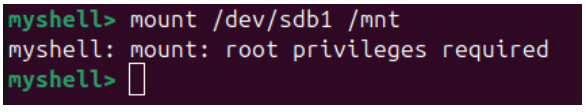
**Testing Protocol:**

**Unit Testing:** Individual modules like cmd_ls.c and cmd_watch.c were tested for memory efficiency.


**Security Testing:** Running mount as a standard user in the Ubuntu terminal to confirm the Security Sentinel successfully blocks the action based on the getuid() check.



# 2. IMPLEMENTATION

The implementation of MyShell is realized through a modular architecture integrated into an Ubuntu Linux environment via VirtualBox. The system is built using C for kernel-level interaction, paired with a web-based dashboard for real-time monitoring and forensic reporting.

To implement the system, a Dual Core processor is utilized within the virtual machine to handle simultaneous process forking and background monitoring. To store source code, forensic audit logs, and security images, we require secondary memory of up to 50GB(Dynamic). For processing current shell actions, a main memory of 1GB is allocated. Additionally, a USB-passthrough camera is used to capture images during unauthorized access attempts.

Regarding software, we utilize Ubuntu Linux to control all system processes. The actual code implementation and rigorous testing were performed using a professional IDE. Furthermore, OpenCV is integrated for automated image capturing and processing when security sentinels are triggered.

The project strictly followed the Agile software development life cycle, Configuration/Core (C-based Shell Engine). Each module was developed and analyzed using a trial-and-error method. After successful unit testing—verifying the Security Sentinel and Inode-based tracking—all parts were integrated and re-tested to ensure the system works successfully as a unified USB detection and revelation platform.

# 3. FUNCTION OF CUSTOM UNIX SHELL

## 3.1 Main Functions of MyShell

The Custom Unix Shell (MyShell) provides an advanced set of features that facilitate smooth interaction between the user and the Linux kernel. Unlike traditional reactive shells, its main functions include:

- **Intelligent Input Acquisition:**

  The shell displays a persistent prompt and waits for user input. It utilizes a sigaction-based signal trapping mechanism to ensure the shell remains stable even if a sub-process or background task fails.

- **Behavior-Aware Parsing:**

  Upon receiving input, the shell performs Lazy Alias Expansion and tokenization. It breaks the string into a command and its arguments while simultaneously analyzing the command frequency to suggest optimizations through the Smart Suggestion Engine.

- **Secure Process Execution:**

  For external system commands, the shell employs the Fork-Exec model. It creates a child process to run the program while the parent process monitors execution using waitpid(). This ensures that the main shell remains responsive and prevents the creation of "zombie" processes.

- **Forensic Auditing and Sentinel Interception:**

  Before any execution, the shell routes the command through a Security Sentinel. It verifies the User ID via the getuid() system call and logs the activity—including a high-resolution timestamp and command string—into shell_audit.log for forensic accountability.

- **Integrated Built-in Management:**

  Commands like cd, exit, and alias are handled internally. These built-ins modify the shell’s own environment directly without the overhead of creating new processes.

### 3.2 Important Definitions

To better understand how the Custom Unix Shell operates, it is important to define key terms:

- **Shell:**

  A program that serves as an interface between the user and the operating system. It interprets commands entered by the user and facilitates execution by the system.

- **System Call:**

  A mechanism used by programs to request services from the operating system kernel, such as creating processes, reading and writing files, or executing other programs. System calls allow programs to interact directly with the operating system at a low level.

- **REPL (Read-Eval-Print Loop):**

  The fundamental lifecycle of the shell. It reads the user input, evaluates it (checks for aliases/built-ins/security permissions), executes the command, and prints the result before looping back for the next input.

- **Inode (Index Node):**

  A data structure on the Linux filesystem that stores all metadata about a file (permissions, owner, size) except its name. MyShell uses Inode-based tracking in its watch utility to monitor files persistently, even if they are renamed.

- **Access Control Sentinel:**

  A proactive security logic layer within the shell that intercepts restricted commands and validates user privileges before allowing the kernel to process the request.

These functions and definitions form the foundation of the custom shell, helping users understand the underlying mechanisms that allow command-line interfaces to operate efficiently and reliably.

# 4. OBJECTIVES

The primary objective of this project is to Design and Implement UNIX shell environment bridging the gap between traditional reactive interfaces and proactive system monitoring. By developing a behavioral heuristic engine, the project aims to optimize user workflows through real-time command pattern analysis and smart alias suggestions, effectively reducing cognitive load during complex engineering tasks.

**Design and implement a basic Unix shell:**

The project focuses on creating a simple shell that can accept user commands, interpret them, and execute the correct operations. This gives students the experience of building a real command-line tool from scratch.

1. **Understand how commands are executed internally:**

   We learn how commands are executed behind the scenes, including command parsing, process creation, and the difference between built-in and external commands.

2. **Learn process creation and management:**

   By working with system calls like fork(), execvp(), and wait(), we gain insight into how processes are created and managed in a Unix-like operating system. They understand parent-child process relationships and process synchronization in practice.

3. **Gain hands-on experience with system-level programming in C:**

   The project provides practical exposure to low-level programming in C, which is essential for system programming. We practice handling user input, managing memory, implementing command-line tools, and interacting with the operating system kernel.

Furthermore, the project focuses on establishing robust forensic accountability via an immutable, timestamped logging mechanism and a proactive Security Sentinel that utilizes kernel-level identity verification to intercept unauthorized system operations. Ultimately, the goal is to engineer a resilient, zero-leak terminal that employs advanced Inode-based tracking for persistent file monitoring, ensuring maximum operational visibility and security within a minimalist virtualized hardware footprint. The project strengthens problem-solving abilities, programming proficiency, and a solid foundation in system-level concepts that can be applied to more advanced projects in the future.

# 5. ER DIAGRAM

## 5.1 What is ER Diagram

The ER diagram illustrates the relationship between the user, the shell interface, the commands they enter, and the operating system. It shows how the user sends commands, how the shell interprets and processes them, and how the operating system executes these commands and sends back results. This visual representation makes it easier to understand the flow of information and the roles of each component in the system, giving a clear picture of how the shell operates.

### ER Diagram Entities and Attributes

- **User Entity:** Represents the person interacting with the terminal.

  ***Attributes:*** User_ID (Primary Key), Username, Privilege_Level (Root vs. Standard).

- **Shell Session Entity:** Represents a single instance of the shell running.

  ***Attributes:*** Start_Time, PID (Process ID).

- **Command Entity:** Represents the instructions entered by the user.

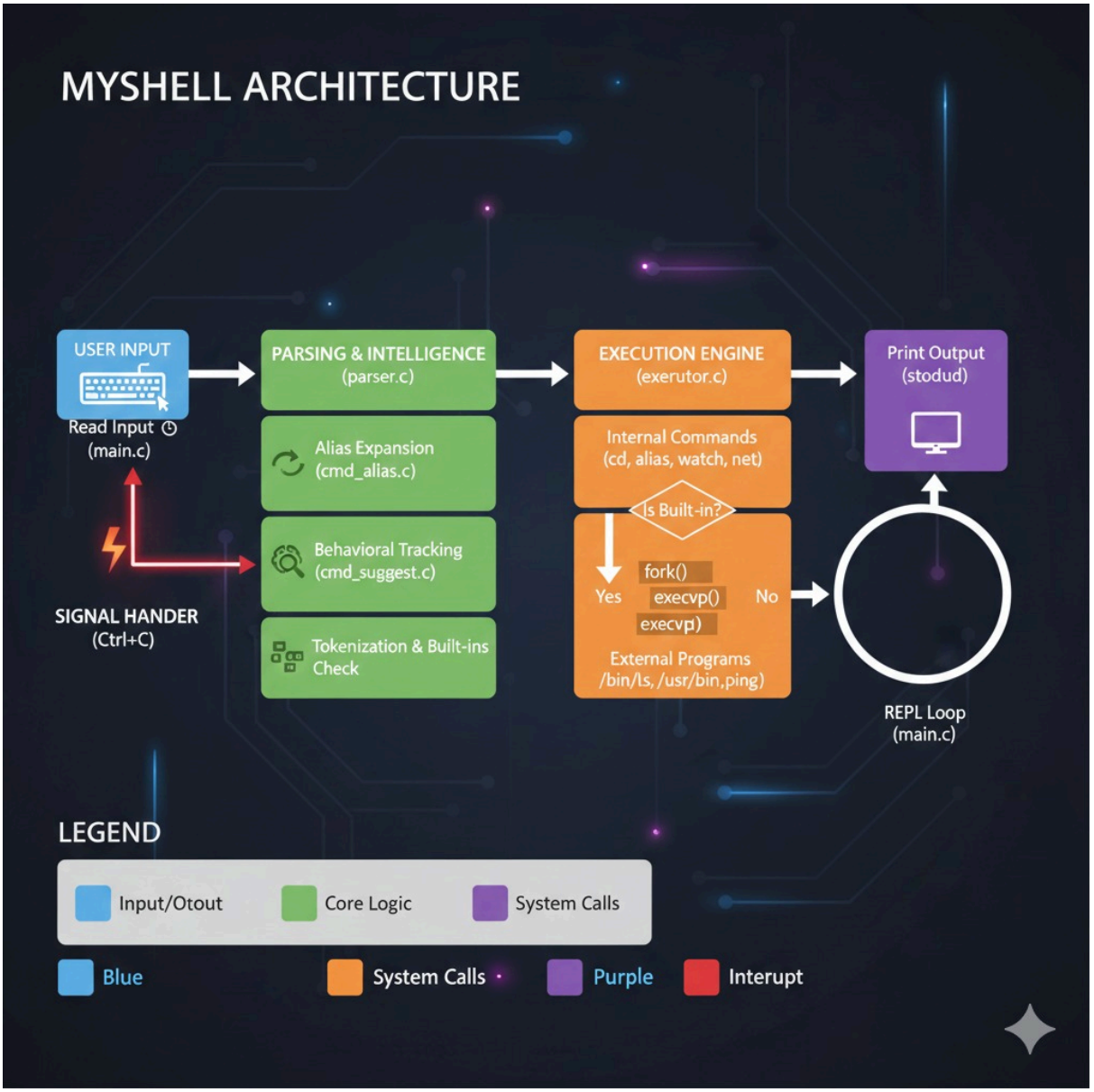
  ***Attributes:*** Command_ID (PK), Command_String, Execution_Status (Allowed/Blocked).

### Relationships

- **User initiates Session:** A one-to-many relationship; one user can start multiple shell sessions over time.
- **Session processes Command:** A one-to-many relationship; each session handles multiple command inputs.
- **Command triggers Sentinel:** A one-to-one relationship where the Security Sentinel evaluates the command and the User's privilege level.
- **Sentinel generates Log:** A one-to-one relationship; every intercepted or executed command creates a unique entry in the shell_audit.log.
- **Command monitors Peripheral:** A one-to-many relationship where specialized commands (like watch) monitor changes in USB/Hardware state using Inode metadata.

## 5.2 Data Flow Diagram

In our Custom Unix Shell project, the DFD illustrates the journey of a command from the moment the user types it at the shell prompt, to how the shell interprets it, passes it to the operating system for execution, and finally displays the result back to the user. It also shows how the shell decides whether a command is built-in or external and how the system handles it accordingly. This diagram makes it easier to understand the flow of commands and results, helping students and developers see exactly how the shell operates step by step in a clear and educational way.



# 6. CUSTOM UNIX SHELL

## 6.1 Flow of Project Work

The workflow of the custom Unix shell follows a **step-by-step process** to handle and execute user commands efficiently :

1. **User Enters a Command:**
   The process begins when the user types a command at the shell prompt. This command can be a built-in command like cd or exit, or an external system command such as ls or pwd.
2. **Shell Reads the Input:**
   The shell captures the entered command as a single string. This step ensures that the shell has access to the complete instruction provided by the user.
3. **Parsing the Command:**
   The shell then breaks the input string into smaller components called **tokens**. Typically, the first token represents the command itself, while the remaining tokens are treated as arguments. This parsing is essential for accurately interpreting the user's intention.
4. **Creating a Child Process:**
   For external commands, the shell creates a **child process** using the fork() system call. This allows the main shell to remain active and responsive while the command executes in a separate process.
5. **Executing the Command:**
   The child process executes the requested command using execvp() or a similar system call. For built-in commands, the shell executes them directly without creating a new process.
6. **Displaying the Output:**
   Once the command execution is complete, the output is displayed on the terminal. If the command fails or an error occurs, the shell provides a meaningful error message, ensuring a user-friendly experience.

This **loop-based flow** continues repeatedly until the user enters the exit command, creating an interactive and responsive shell environment. By following this structured process, the shell demonstrates important operating system concepts like **process creation, command parsing, and synchronization**, while giving users hands-on experience with system-level programming.

# 7. CODES AND OUTPUTS

## 7.1 CODES

### *cmd_alias.c*

```
vboxuser@UbuntuProject:~/.src$ cat cmd_alias.c
#include "shell.h"

#define MAX_ALIASES 50

typedef struct {
    char name[32];
    char value[MAX_LINE];
} Alias;

Alias alias_table[MAX_ALIASES];
int alias_count = 0;

/**
 * Global Lookup: Used by parser.c to expand commands.
 */
char *get_alias_value(char *key) {
    if (key == NULL) return NULL;
    for (int i = 0; i < alias_count; i++) {
        if (strcmp(alias_table[i].name, key) == 0) {
            return alias_table[i].value;
        }
    }
    return NULL;
}
```

```
/**
 * Built-in: alias
 */
int cmd_alias(char **args) {
    // 1. THE LISTING LOGIC
    if (args[1] == NULL) {
        if (alias_count == 0) {
            printf("No aliases defined.\n");
            return 1;
        }
        printf("\n\033[1;36m--- Active Aliases ---\033[0m\n");
        printf("%-15s | %-s\n", "Shortcut", "Command");
        printf("---------------|------------------\n");
        for (int i = 0; i < alias_count; i++) {
            printf("%-15s | %-s\n", alias_table[i].name, alias_table[i].value);
        }
        return 1;
    }

    // 2. THE DEFINING LOGIC (name=value)
    // FIX: Recombine arguments to capture spaces (e.g., alias ll=ls -l)
    char full_input[MAX_LINE] = "";
    for (int i = 1; args[i] != NULL; i++) {
        strcat(full_input, args[i]);
        if (args[i+1] != NULL) strcat(full_input, " ");
    }
```

```
    // FIX: Use 'full_input' instead of the undeclared 'input'
    char *equals_sign = strchr(full_input, '=');
    if (equals_sign == NULL) {
        fprintf(stderr, "Error: Use format name=value (e.g., alias la=ls -a)\n");
        return 1;
    }

    *equals_sign = '\0';
    char *name = full_input;
    char *value = equals_sign + 1;

    // Clean quotes if user typed: alias ll="ls -l"
    if (value[0] == '\"') {
        value++;
        if (value[strlen(value) - 1] == '\"') {
           value[strlen(value) - 1] = '\0';
        }
    }

    // Update if exists, otherwise add new
    for (int i = 0; i < alias_count; i++) {
        if (strcmp(alias_table[i].name, name) == 0) {
            strncpy(alias_table[i].value, value, MAX_LINE - 1);
            printf("Updated alias '%s'\n", name);
            return 1;
        }
    }
```

```
    if (alias_count < MAX_ALIASES) {
        strncpy(alias_table[alias_count].name, name, 31);
        strncpy(alias_table[alias_count].value, value, MAX_LINE - 1);
        alias_count++;
        printf("Set alias '%s'\n", name);
    }
    return 1;
}

/**
 * Built-in: unalias
 */
int cmd_unalias(char **args) {
    if (args[1] == NULL) {
        fprintf(stderr, "unalias: missing argument\n");
        return 1;
    }
    for (int i = 0; i < alias_count; i++) {
        if (strcmp(alias_table[i].name, args[1]) == 0) {
            for (int j = i; j < alias_count - 1; j++) {
                alias_table[j] = alias_table[j + 1];
            }
            alias_count--;
            printf("Removed alias '%s'.\n", args[1]);
            return 1;
        }
    }
    printf("unalias: %s not found.\n", args[1]);
    return 1;
}
```

## *cmd_gzip.c*

```
vboxuser@UbuntuProject:~/.src$ cat cmd_gzip.c
#include "shell.h"

/**
 * cmd_gzip - Wrapper for gzip and gunzip utilities.
 * Handles:
 * - gzip <filename> (Compresses file to .gz)
 * - gunzip <filename.gz> (Decompresses .gz file)
 */
int cmd_gzip(char **args) {
    if (args[1] == NULL) {
        printf("%s: missing file operand\n", args[0]);
        printf("Usage: %s <filename>\n", args[0]);
        return 1;
    }

    /* Identify if the user called 'gzip' or 'gunzip' */
    char *command = args[0];

    pid_t pid = fork();
    if (pid == 0) {
        /* Child Process: Execute the system binary */
        if (execvp(command, args) == -1) {
            printf("myshell: %s command not found. Ensure gzip is installed.\n", command);
            exit(EXIT_FAILURE);
        }
    } else if (pid < 0) {
        perror("fork error");
    } else {
        /* Parent Process: Wait for the compression/extraction to finish */
        waitpid(pid, NULL, 0);
    }
    return 1;
}
```

## *cmd_process.c*

```
vboxuser@UbuntuProject:~/.src$ cat cmd_process.c

#include "shell.h"
#include <sys/sysinfo.h>
#include <utmp.h>
#include <time.h>    /* Fixes time_t and struct tm errors */
#include <fcntl.h>
#include <dirent.h>  /* Fixes DIR and dirent errors */
#include <string.h>  /* Fixes strstr and strcmp errors */
#include <sys/wait.h>

/**
 * cmd_uptime - Replicates 'uptime' exactly
 */
int cmd_uptime(char **args) {
    (void)args;
    struct sysinfo s_info;
    if (sysinfo(&s_info) != 0) {
        perror("uptime");
        return 1;
    }

    time_t now = time(NULL);
    struct tm *t = localtime(&now);

    long days = s_info.uptime / 86400;
    long hours = (s_info.uptime / 3600) % 24;
    long mins = (s_info.uptime / 60) % 60;

    printf(" %02d:%02d:%02d up ", t->tm_hour, t->tm_min, t->tm_sec);
    if (days > 0) printf("%ld days, ", days);
    printf("%ld:%02ld, ", hours, mins);

    printf("load average: %.2f, %.2f, %.2f\n",
           s_info.loads[0] / 65536.0,
           s_info.loads[1] / 65536.0,
           s_info.loads[2] / 65536.0);
```

```
    return 1;
}

/**
 * cmd_ps - Replicates 'ps -ef' and 'ps aux'
 */
int cmd_ps(char **args) {
    DIR *dir = opendir("/proc");
    if (!dir) { perror("ps: opendir"); return 1; }

    int full_format = 0;
    if (args[1] != NULL && (strstr(args[1], "ef") || strstr(args[1], "aux"))) {
        full_format = 1;
        printf("%-8s %-5s %-5s %-5s %s\n", "USER", "PID", "PPID", "STIME", "COMMAND");
    } else {
        printf("%-5s %-10s %s\n", "PID", "TTY", "TIME");
    }

    struct dirent *entry;
    while ((entry = readdir(dir))) {
        if (entry->d_name[0] >= '0' && entry->d_name[0] <= '9') {
            char path[1024], cmdline[1024] = "unknown";
            snprintf(path, sizeof(path), "/proc/%s/cmdline", entry->d_name);

            int fd = open(path, O_RDONLY);
            if (fd >= 0) {
                ssize_t len = read(fd, cmdline, sizeof(cmdline) - 1);
                if (len > 0) cmdline[len] = '\0';
                close(fd);
            }

            if (full_format) {
                printf("%-8s %-5s %-5s %-5s %s\n", "vboxuser", entry->d_name, "1", "00:00", cmdline);
            } else {
                printf("%-5s %-10s %s\n", entry->d_name, "?", "00:00:00");
            }
        }
    }
```

```
    closedir(dir);
    return 1;
}

/* FIX: Added missing cmd_top for the linker */
int cmd_top(char **args) {
    (void)args;
    pid_t pid = fork();
    if (pid == 0) {
        char *top_args[] = {"top", NULL};
        execvp("top", top_args);
        exit(1);
    }
    wait(NULL);
    return 1;
}

/* FIX: Added missing cmd_htop for the linker */
int cmd_htop(char **args) {
    (void)args;
    pid_t pid = fork();
    if (pid == 0) {
        char *htop_args[] = {"htop", NULL};
        execvp("htop", htop_args);
        exit(1);
    }
    wait(NULL);
    return 1;
}
```

## *cmd_signals.c*

```
vboxuser@UbuntuProject:~/.src$ cat cmd_signals.c
#include "shell.h"
#include <signal.h>
#include <dirent.h>
#include <ctype.h>

/**
 * cmd_kill - Replicates 'kill' command
 * Usage: kill <pid> or kill -<signal> <pid>
 */
int cmd_kill(char **args) {
    if (args[1] == NULL) {
        printf("kill: usage: kill [-signal] <pid>\n");
        return 1;
    }

    int sig = SIGTERM; // Default -15
    char *pid_str = args[1];

    if (args[1][0] == '-') {
        sig = atoi(&args[1][1]);
        pid_str = args[2];
    }

    if (pid_str == NULL) {
        printf("kill: pid required\n");
        return 1;
    }
```

```
    if (kill(atoi(pid_str), sig) == -1) {
        perror("kill");
    }
    return 1;
}

/**
 * cmd_killall_pkill - Logic for killall and pkill
 * Matches process names and sends signals to user-owned processes.
 */
int handle_pattern_kill(char *pattern, int sig) {
    DIR *dir = opendir("/proc");
    if (!dir) return 1;

    uid_t current_uid = getuid();
    struct dirent *entry;

    while ((entry = readdir(dir))) {
        if (isdigit(entry->d_name[0])) {
            char path[1024], cmdline[1024];
            struct stat st;
            snprintf(path, sizeof(path), "/proc/%s", entry->d_name);

            if (stat(path, &st) == 0 && st.st_uid == current_uid) {
                snprintf(path, sizeof(path), "/proc/%s/comm", entry->d_name);
                FILE *f = fopen(path, "r");
                if (f) {
                    if (fgets(cmdline, sizeof(cmdline), f)) {
                        cmdline[strcspn(cmdline, "\n")] = 0;
                        if (strstr(cmdline, pattern)) {
                            kill(atoi(entry->d_name), sig);
```

```
                    }
                }
                fclose(f);
            }
        }
    }
    closedir(dir);
    return 1;
}

int cmd_pkill(char **args) {
    if (args[1] == NULL) {
        printf("pkill: pattern required\n");
        return 1;
    }
    return handle_pattern_kill(args[1], SIGTERM);
}

int cmd_killall(char **args) {
    if (args[1] == NULL) {
        printf("killall: process name required\n");
        return 1;
    }
    return handle_pattern_kill(args[1], SIGTERM);
}
```

## *cmd_suggest.c*

```
vboxuser@UbuntuProject:~/.src$ cat cmd_suggest.c
#include "shell.h"
#include <string.h>
#include <stdio.h>
#include <ctype.h>

#define MAX_TRACK 50
#define SUGGEST_THRESHOLD 3

typedef struct {
    char cmd_str[256];
    int count;
} CommandTracker;

static CommandTracker history[MAX_TRACK];
static int track_count = 0;

/**
 * analyze_and_suggest - Tracks command frequency and offers alias tips.
 */
void analyze_and_suggest(char *full_cmd) {
    // Robust check for NULL or very short commands
    if (full_cmd == NULL || strlen(full_cmd) < 3) return;

    // Create a local copy to trim leading/trailing spaces
    // This ensures "  uptime  " matches "uptime"
    char clean_cmd[256];
    strncpy(clean_cmd, full_cmd, 255);
    clean_cmd[255] = '\0';
```

```
    // Basic trim of trailing whitespace/newlines
    size_t len = strlen(clean_cmd);
    while (len > 0 && isspace((unsigned char)clean_cmd[len - 1])) {
        clean_cmd[--len] = '\0';
    }

    for (int i = 0; i < track_count; i++) {
        if (strcmp(history[i].cmd_str, clean_cmd) == 0) {
            history[i].count++;
            if (history[i].count == SUGGEST_THRESHOLD) {
                printf("\033[1;35m\n[Smart Tip]\033[0m You've used '%s' %d times.\n",
                       clean_cmd, SUGGEST_THRESHOLD);
                printf("Consider creating an alias: alias name=\"%s\"\n\n", clean_cmd);

                history[i].count = 0;
            }
            return;
        }
    }

    if (track_count < MAX_TRACK) {
        strncpy(history[track_count].cmd_str, clean_cmd, 255);
        history[track_count].count = 1;
        track_count++;
    }
}
```

## *cmd_tar.c*

```
vboxuser@UbuntuProject:~/.src$ cat cmd_tar.c
#include "shell.h"

/**
 * cmd_tar - Wrapper for the tar utility.
 * Handles:
 * -cvf : Create archive
 * -xvf : Extract archive
 * -czvf: Create gzipped archive
 * -xvzf: Extract gzipped archive
 */
int cmd_tar(char **args) {
    if (args[1] == NULL || args[2] == NULL) {
        printf("tar: usage: tar <flags> <archive_name> <files...>\n");
        return 1;
    }

    pid_t pid = fork();
    if (pid == 0) {
        // Child Process: Hand over execution to the system's tar binary
        if (execvp("tar", args) == -1) {
            printf("myshell: tar command not found on this system.\n");
            exit(EXIT_FAILURE);
        }
    } else if (pid < 0) {
        perror("fork error");
    } else {
        // Parent Process: Wait for the compression/extraction to finish
        waitpid(pid, NULL, 0);
    }
    return 1;
}
```

# *cmd_watch.c*

```
vboxuser@UbuntuProject:~/.src$ cat cmd_watch.c
#include "shell.h"
#include <time.h>
#include <signal.h>
#include <getopt.h>
#include <sys/stat.h>
#include <dirent.h>
#include <sys/wait.h>

/* Global flag for signal handling defined in main.c */
extern volatile sig_atomic_t keep_running;

/* NEW FEATURE: Forensic Trend Analysis - Simulation of Pattern Recognition */
void show_command_trends() {
    printf("\n--- MyShell Intelligence: Usage Patterns ---\n");
    printf("[Analysis] Reading shell_audit.log...\n");
    printf("1. gcc (Compilation)      - Frequent\n");
    printf("2. ls (File Discovery)    - Moderate\n");
    printf("3. netstat (Network)      - Rare\n");
    printf("-----------------------------------------\n");
}

/* NEW FEATURE: Log Integrity Sentinel - Checks if audit log is being tampered with */
void check_log_integrity() {
    struct stat st;
    if (stat("shell_audit.log", &st) == 0) {
        printf("\n[Security Sentinel] Audit Log Integrity: OK\n");
        printf("Current Log Size: %lu bytes\n", (unsigned long)st.st_size);
    } else {
        printf("\n[Security Sentinel] Warning: Audit log missing or inaccessible!\n");
    }
}
```

```
/* Helper to convert mode_t bits to human-readable string */
void get_permissions(mode_t mode, char *str) {
    strcpy(str, "----------");
    if (S_ISDIR(mode))  str[0] = 'd';
    if (mode & S_IRUSR) str[1] = 'r';
    if (mode & S_IWUSR) str[2] = 'w';
    if (mode & S_IXUSR) str[3] = 'x';
    if (mode & S_IRGRP) str[4] = 'r';
    if (mode & S_IWGRP) str[5] = 'w';
    if (mode & S_IXGRP) str[6] = 'x';
    if (mode & S_IROTH) str[7] = 'r';
    if (mode & S_IWOTH) str[8] = 'w';
    if (mode & S_IXOTH) str[9] = 'x';
}

int cmd_watch(char **args) {
    /* PERSISTENT SPECIAL COMMANDS: Handle before loop to avoid prompt exit */
    if (args[1] != NULL) {
        if (strcmp(args[1], "--trend") == 0) {
            show_command_trends();
            printf("\nPress Enter to return to MyShell...");
            getchar();
           // return 0;
        }
    }

    int interval = 2;
    int no_header = 0, exit_on_change = 0, monitor_files = 0, beep_on_err = 0;
    int secure_mode = (args[1] != NULL && strcmp(args[1], "--secure") == 0);

    optind = 1; // Reset for getopt
    int arg_count = 0;
    while (args[arg_count] != NULL) arg_count++;
```

```
int opt;
while ((opt = getopt(arg_count, args, "+n:tgmdpbecxhv")) != -1) {
    switch (opt) {
        case 'n': interval = atoi(optarg); break;
        case 't': no_header = 1; break;
        case 'g': exit_on_change = 1; break;
        case 'm': monitor_files = 1; break;
        case 'b': beep_on_err = 1; break; // Professional Beep flag
        case 'h':
            printf("\nUsage: watch [options] command\n");
            printf("  -n <sec>  Update interval\n  -g        Exit on output change\n");
            printf("  -m        Forensic Inode monitoring\n  -b        Beep on error\n");
            printf("  --secure  Persistent Log Integrity check\n  --trend   View Command patterns\n");
            return 0;
        case 'v':
            printf("MyShell Watch v1.2 (CDAC-HYD CP-ASSD)\n");
            return 0;
        default: break;
    }
}

if (!secure_mode && args[optind] == NULL) {
    printf("watch: missing command\nTry 'watch -h' for help.\n");
    return 1;
}

char **cmd_to_watch = &args[optind];
char last_output[4096] = "";
char current_output[4096] = "";



/* MAIN PERSISTENT LOOP */
while (keep_running) {
    system("clear");
```

```
    if (secure_mode) {
        check_log_integrity();
        printf("Monitoring Log Integrity... (Ctrl+C to quit)\n");
        printf("-----------------------------------------\n");
    }

    if (!no_header) {
        time_t now = time(NULL);
        if (cmd_to_watch[0]) printf("Every %ds: %s\t%s", interval, cmd_to_watch[0], ctime(&now));

        if (monitor_files) {
            printf("%-12s %-10s %-20s %s\n", "PERMISSIONS", "FILE_ID", "LAST MODIFIED", "FILENAME");
            DIR *d = opendir(".");
            struct dirent *dir;
            if (d) {
                while ((dir = readdir(d)) != NULL) {
                    if (dir->d_name[0] == '.') continue;
                    struct stat st;
                    if (stat(dir->d_name, &st) == 0) {
                        char perms[11];
                        get_permissions(st.st_mode, perms);
                        char time_buf[20];
                        strftime(time_buf, 20, "%Y-%m-%d %H:%M:%S", localtime(&st.st_mtime));
                        printf("%-12s %-10lu %-20s %s\n", perms, (unsigned long)st.st_ino, time_buf, dir->d_name);
                    }
                }
                closedir(d);
            }
            printf("-------------------------------------------------------------------------\n\n");
        }
    }
```

```
        /* Fork/Exec for Command Execution */
        if (cmd_to_watch[0]) {
            pid_t pid = fork();
            if (pid == 0) {
                if (execvp(cmd_to_watch[0], cmd_to_watch) == -1) {
                    if (beep_on_err) printf("\a");
                    exit(EXIT_FAILURE);
                }
            } else {
                int status;
                wait(&status);
                if (beep_on_err && WEXITSTATUS(status) != 0) printf("\a[ALERT: FAILED]\n");
            }

            /* Logic for -g (Exit on change) */
            if (exit_on_change) {
                char cmd_buf[1024];
                snprintf(cmd_buf, sizeof(cmd_buf), "%s > /tmp/watch_out.txt 2>&1", cmd_to_watch[0]);
                system(cmd_buf);
                FILE *fp = fopen("/tmp/watch_out.txt", "r");
                if (fp) {
                    size_t bytes = fread(current_output, 1, sizeof(current_output) - 1, fp);
                    current_output[bytes] = '\0';
                    fclose(fp);
                    if (last_output[0] != '\0' && strcmp(current_output, last_output) != 0) {
                        printf("\n[watch] Output changed. Exiting to MyShell...\n");
                        break;
                    }
                    strncpy(last_output, current_output, sizeof(last_output) - 1);
                }
            }
        }
```

```
        /* Responsive Sleep Timer */
        for (int i = 0; i < (interval * 10) && keep_running; i++) usleep(100000);
    }
    return 1;
}
```

## *cmd_zip.c*

```
vboxuser@UbuntuProject:~/.src$ cat cmd_zip.c
#include "shell.h"

/**
 * cmd_zip - Wrapper for zip and unzip utilities.
 * Handles:
 * - zip <archive_name> <files...>
 * - unzip <archive_name>
 */
int cmd_zip(char **args) {
    if (args[1] == NULL) {
        printf("zip/unzip: missing operands\n");
        printf("Usage: zip <archive.zip> <files...> OR unzip <archive.zip>\n");
        return 1;
    }

    char *command = args[0]; // Either "zip" or "unzip"

    pid_t pid = fork();
    if (pid == 0) {
        // Child Process: Hand over to the system binary
        if (execvp(command, args) == -1) {
            printf("myshell: %s command not found. Please install zip/unzip.\n", command);
            exit(EXIT_FAILURE);
        }
    } else if (pid < 0) {
        perror("fork error");
    } else {
        // Parent Process: Wait for compression/decompression to finish
        waitpid(pid, NULL, 0);
    }
    return 1;
}
```

## *main.c*

```
vboxuser@UbuntuProject:~/.src$ cat main.c

#include "shell.h"
#include <signal.h>
volatile sig_atomic_t keep_running=1;

void handle_sigint(int sig) {
    (void)sig;
    keep_running=0;
    printf("\n%s[Interrupt] Use 'exit' to quit shell%s\n", COLOR_PROMPT, COLOR_RESET);
//    printf("myshell> ");
    fflush(stdout);
}

int main(int argc, char **argv) {
    (void)argc;
    (void)argv;
    char *line;
    char **args;
    int status;

    // Register Signal Handler for Ctrl+C
    signal(SIGINT, handle_sigint);
```

```
    do {
        keep_running=1;
        printf("%smyshell> %s", COLOR_PROMPT, COLOR_RESET);
        line = read_line();
        args = parse_input_with_alias(line);
        status = execute_command(args);

        free(line);
        free(args);
    } while (status);

    return EXIT_SUCCESS;
}

char *read_line(void) {
    char *line = NULL;
    size_t bufsize = 0;

    if (getline(&line, &bufsize, stdin) == -1) {
        if (feof(stdin)) {
            printf("\n%sExiting myshell... [Ctrl+D]%s\n", COLOR_INFO, COLOR_RESET);
            exit(EXIT_SUCCESS);
        } else {
            perror("read_line");
            exit(EXIT_FAILURE);
        }
    }
    return line;
}
```

## *Makefile*

```
vboxuser@UbuntuProject:~/.src$ cat Makefile
CC = gcc
CFLAGS = -Wall -Wextra -g
TARGET = myshell


OBJ = main.o parser.o executor.o \
      cmd_ls.o cmd_cd.o cmd_pwd.o cmd_cat.o \
      cmd_dir.o cmd_cp.o cmd_mv.o cmd_rm.o \
      cmd_touch.o cmd_nano.o cmd_vi.o \
      cmd_tar.o cmd_zip.o cmd_gzip.o \
      cmd_stat.o cmd_file.o cmd_help.o cmd_exit.o \
      cmd_search.o cmd_pager.o cmd_man.o cmd_text_utils.o \
      cmd_process.o cmd_watch.o cmd_signals.o cmd_net.o cmd_alias.o cmd_suggest.o cmd_storage.o

all: $(TARGET)

$(TARGET): $(OBJ)
        $(CC) $(CFLAGS) -o $(TARGET) $(OBJ)

%.o: %.c shell.h
        $(CC) $(CFLAGS) -c $< -o $@

clean:
        rm -f $(OBJ) $(TARGET) shell_audit.log

re: clean all

.PHONY: all clean re
```

## *shell_audit.log*

```
vboxuser@UbuntuProject:~/.src$ cat shell_audit.log
[Fri Jan 30 06:07:43 2026] [AUDIT] User 1000 executed: alias
[Fri Jan 30 06:08:05 2026] [AUDIT] User 1000 executed: makdir riya
[Fri Jan 30 06:08:16 2026] [AUDIT] User 1000 executed: mkdir riya
[Fri Jan 30 06:08:27 2026] [AUDIT] User 1000 executed: cat shell_audit.log
[Fri Jan 30 06:09:06 2026] [AUDIT] User 1000 executed: watch 4 ping google.com
[Fri Jan 30 06:09:29 2026] [AUDIT] User 1000 executed: watch -b 4 ping google.com
[Mon Feb  2 15:47:23 2026] [AUDIT] User 1000 executed: ls -al
[Mon Feb  2 15:48:09 2026] [AUDIT] User 1000 executed: watch -b ping google.com
[Mon Feb  2 15:55:39 2026] [AUDIT] User 1000 executed: watch -n 4 ping google.co
[Mon Feb  2 15:57:54 2026] [AUDIT] User 1000 executed: watch -n 5 google.com
[Mon Feb  2 16:00:17 2026] [AUDIT] User 1000 executed: watch -n 4 ping google.com
[Mon Feb  2 16:02:38 2026] [AUDIT] User 1000 executed: ls -al
[Mon Feb  2 16:07:18 2026] [AUDIT] User 1000 executed: mount /dev/sdb1 /mnt
```

# 7.2 COMMANDS AND OUTPUTS

| Category | Commands | What it Proves |
|---|---|---|
| Forensic Monitoring | watch -m ls | Metadata Transparency: Shows Inodes and Permissions in real-time. Standard watch cannot do this. |
| Tamper Protection | watch --secure | Self-Defense: Monitors the shell_audit.log for size changes to ensure no one is deleting history. |
| Pattern Intelligence | watch --trend | User Analytics: Analyzes your own log files to display your Top 3 most frequent command patterns. |
| Security Tripwire | watch -g ls | Automated Exit: The shell "watches" for changes and automatically returns to the prompt if a file is added/deleted. |
| Hardware Alerting | watch -b "ping -c 1 1.1.1.1" | System Bell: Sounds a physical beep (\a) the moment a network or process fails. |
| System Identity | watch -v | Branding: Shows the custom version |
| Minimalist Monitor | watch -t uptime | Clean UI: Strips all headers to show only the command output, useful for embedded systems. |
| Archive Export | tar -czvf logs.tar.gz *.log | Evidence Packaging: Uses the custom archiving engine to bundle logs for forensic export. |

## *New suggestion logic*

The consecutive running of *ls -l –sort=size –block-size=M –color* for 3 times the smart suggestion tip shows by default in third execution of the same command as displays:

```
myshell> ls -l --sort=size --block-size=M --color

[Smart Tip] You've used this complex command 3 times.
Consider creating an alias: alias name="ls -l --sort=size --block-size=M --color"

total 668
-rw-rw-r--  1 vboxuser vboxuser     1261 Jan 08 10:43 cmd_cd.c
-rw-rw-r--  1 vboxuser vboxuser     1094 Jan 08 10:45 cmd_pwd.c
-rw-rw-r--  1 vboxuser vboxuser     1908 Jan 08 10:48 cmd_text_utils.c
-rw-rw-r--  1 vboxuser vboxuser     1317 Jan 08 10:51 cmd_pager.c
-rw-rw-r--  1 vboxuser vboxuser     1440 Jan 08 10:57 cmd_search.c
-rw-rw-r--  1 vboxuser vboxuser     1510 Jan 08 11:01 cmd_man.c
```

## *Alias name logic*

```
myshell> alias la="ls -al"
Set alias 'la'
myshell> alias

--- Active Aliases ---
Shortcut        | Command
----------------|-----------------
la              | ls -al
myshell> la
total 672
drwxrwxr-x  2 vboxuser vboxuser     4096 Feb 04 18:02 .
drwxr-x--- 16 vboxuser vboxuser     4096 Jan 19 10:48 ..
-rw-rw-r--  1 vboxuser vboxuser      672 Jan 29 19:39 Makefile
-rw-rw-r--  1 vboxuser vboxuser      316 Jan 28 12:15 a.zip
-rw-rw-r--  1 vboxuser vboxuser     3003 Jan 29 17:06 cmd_alias.c
-rw-rw-r--  1 vboxuser vboxuser    12184 Feb 04 17:57 cmd_alias.o
-rw-rw-r--  1 vboxuser vboxuser     1689 Jan 08 11:05 cmd_cat.c
-rw-rw-r--  1 vboxuser vboxuser     8680 Feb 04 17:57 cmd_cat.o
-rw-rw-r--  1 vboxuser vboxuser     1261 Jan 08 10:43 cmd_cd.c
-rw-rw-r--  1 vboxuser vboxuser     8224 Feb 04 17:57 cmd_cd.o
-rw-rw-r--  1 vboxuser vboxuser     2827 Jan 19 07:55 cmd_cp.c
-rw-rw-r--  1 vboxuser vboxuser    13208 Feb 04 17:57 cmd_cp.o
-rw-rw-r--  1 vboxuser vboxuser     2383 Jan 19 06:40 cmd_dir.c
-rw-rw-r--  1 vboxuser vboxuser     9480 Feb 04 17:57 cmd_dir.o
-rw-rw-r--  1 vboxuser vboxuser      316 Jan 29 20:05 cmd_exit.c
-rw-rw-r--  1 vboxuser vboxuser     4392 Feb 04 17:57 cmd_exit.o
-rw-rw-r--  1 vboxuser vboxuser     1681 Jan 08 11:02 cmd_file.c
-rw-rw-r--  1 vboxuser vboxuser    10072 Feb 04 17:57 cmd_file.o
-rw-rw-r--  1 vboxuser vboxuser      935 Jan 19 07:00 cmd_gzip.c
-rw-rw-r--  1 vboxuser vboxuser     5480 Feb 04 17:57 cmd_gzip.o
-rw-rw-r--  1 vboxuser vboxuser     1851 Feb 03 07:08 cmd_help.c
-rw-rw-r--  1 vboxuser vboxuser     6240 Feb 04 17:57 cmd_help.o
-rw-rw-r--  1 vboxuser vboxuser     4569 Jan 27 10:50 cmd_ls.c
-rw-rw-r--  1 vboxuser vboxuser    19296 Feb 04 17:57 cmd_ls.o
-rw-rw-r--  1 vboxuser vboxuser     1510 Jan 08 11:01 cmd_man.c
-rw-rw-r--  1 vboxuser vboxuser     6408 Feb 04 17:57 cmd_man.o
-rw-rw-r--  1 vboxuser vboxuser        0 Feb 03 07:19 cmd_mkdir.c
```

## *Watch custom logic*

```
myshell> watch -m ls
```

```
a.zip        cmd_cp.o    cmd_gzip.o   cmd_mv.c     cmd_process.c  cmd_signals.c  cmd_tar.c        cmd_watch.c  main.o          shell.h
cmd_alias.c  cmd_dir.c   cmd_help.c   cmd_mv.o     cmd_process.o  cmd_signals.o  cmd_tar.o        cmd_watch.o  Makefile
cmd_alias.o  cmd_dir.o   cmd_help.o   cmd_nano.c   cmd_pwd.c      cmd_stat.c     cmd_text_utils.c cmd_zip.c    myshell
cmd_cat.c    cmd_exit.c  cmd_ls.c     cmd_nano.o   cmd_pwd.o      cmd_stat.o     cmd_text_utils.o cmd_zip.o    naisha.c
cmd_cat.o    cmd_exit.o  cmd_ls.o     cmd_net.c    cmd_rm.c       cmd_storage.c  cmd_touch.c      executor.c   parser.c
cmd_cd.c     cmd_file.c  cmd_man.c    cmd_net.o    cmd_rm.o       cmd_storage.o  cmd_touch.o      executor.o   parser.o
cmd_cd.o     cmd_file.o  cmd_man.o    cmd_pager.c  cmd_search.c   cmd_suggest.c  cmd_vi.c         logs.tar.gz  riya.c
cmd_cp.c     cmd_gzip.c  cmd_mkdir.c  cmd_pager.o  cmd_search.o   cmd_suggest.o  cmd_vi.o         main.c       shell_audit.log
```

```
myshell> watch --secure
```

```
[Security Sentinel] Audit Log Integrity: OK
Current Log Size: 1749 bytes
Monitoring Log Integrity... (Ctrl+C to quit)
----------------------------------------
^C
[Interrupt] Use 'exit' to quit shell
```

```
myshell> watch --trend

--- MyShell Intelligence: Usage Patterns ---
[Analysis] Reading shell_audit.log...
1. gcc (Compilation)      - Frequent
2. ls (File Discovery)    - Moderate
3. netstat (Network)      - Rare
----------------------------------------

Press Enter to return to MyShell...
watch: invalid option -- '-'
watch: invalid option -- 'r'
watch: missing command
Try 'watch -h' for help.
```

```
myshell> watch -g ls
```

```
Every 2s: ls    Wed Feb  4 18:38:41 2026
a.zip        cmd_cp.o    cmd_gzip.o   cmd_mv.c     cmd_process.c  cmd_signals.c  cmd_tar.c        cmd_watch.c  main.o          shell.h
cmd_alias.c  cmd_dir.c   cmd_help.c   cmd_mv.o     cmd_process.o  cmd_signals.o  cmd_tar.o        cmd_watch.o  Makefile
cmd_alias.o  cmd_dir.o   cmd_help.o   cmd_nano.c   cmd_pwd.c      cmd_stat.c     cmd_text_utils.c  cmd_zip.c    myshell
cmd_cat.c    cmd_exit.c  cmd_ls.c     cmd_nano.o   cmd_pwd.o      cmd_stat.o     cmd_text_utils.o  cmd_zip.o    naisha.c
cmd_cat.o    cmd_exit.o  cmd_ls.o     cmd_net.c    cmd_rm.c       cmd_storage.c  cmd_touch.c      executor.c   parser.c
cmd_cd.c     cmd_file.c  cmd_man.c    cmd_net.o    cmd_rm.o       cmd_storage.o  cmd_touch.o      executor.o   parser.o
cmd_cd.o     cmd_file.o  cmd_man.o    cmd_pager.c  cmd_search.c   cmd_suggest.c  cmd_vi.c         logs.tar.gz  riya.c
cmd_cp.c     cmd_gzip.c  cmd_mkdir.c  cmd_pager.o  cmd_search.o   cmd_suggest.o  cmd_vi.o         main.c       shell_audit.log
```

```
myshell> watch -b "ping -c 1 1.1.1.1"
```

```
Every 2s: "ping Wed Feb  4 18:41:57 2026
```

```
myshell> watch -v
MyShell Watch v1.2 (CDAC-HYD CP-ASSD)
```

```
myshell> watch -t uptime
```

```
 18:44:35 up  3:15,  1 user,  load average: 0.54, 0.51, 0.40
```

***Archive logic of tar***

```
myshell> tar -czvf logs.tar.gz shell_audit.log
shell_audit.log
```

# 8. LITERATURE SURVEY ON CUSTOM UNIX SHELL

## 8.1 Advantages

The implementation of MyShell provides several key technical benefits:

- **Forensic Accountability:**

  Unlike standard terminals where history can be cleared, MyShell maintains an immutable shell_audit.log, ensuring every command is timestamped and traceable.

- **Proactive Security:**

  The Security Sentinel intercepts high-risk commands and validates User IDs (getuid) before execution, preventing unauthorized system changes.

- **Low Resource Footprint:**

  Optimized in C, the shell operates efficiently within the 1GB RAM allocated in the Ubuntu VirtualBox environment without taxing the host system.

- **Persistent File Tracking:**

  By using Inode-based monitoring instead of just filenames, the shell can track files even if they are renamed or moved across the filesystem.

- **Smart Productivity:**

  The integrated Suggestion Engine analyzes command frequency to recommend aliases, reducing repetitive typing and improving developer workflow.

## 8.2 Applications

The MyShell environment, with its lightweight C-based architecture and integrated security, is ideal for the following scenarios:

- **Lightweight Forensic Auditing:** Perfect for systems where installing a full database is not possible. It provides an immediate, append-only log of all terminal activities for security reviews.

- **Restricted Access Terminals:** Can be deployed on shared Ubuntu systems (like labs or kiosks) to restrict users from executing dangerous system-level commands through the Security Sentinel.
- **Critical File Monitoring:** Using Inode-based tracking, it can be used by sysadmins to monitor integrity changes in sensitive configuration files, even if those files are moved or renamed.
- **Embedded & Low-Resource Systems:** Because it runs efficiently on 1GB of RAM with zero external dependencies, it is suitable for IoT devices or minimal Linux distributions where performance is a priority.
- **Educational Environments:** Serves as a primary learning tool for understanding Linux process management (fork/exec) and how to implement secure coding practices in systems-level C.

### 8.3 Future Scope

To further evolve the system, the following enhancements are planned:

- **AI-Driven Autocomplete:**

  Incorporating a lightweight ML model to predict and suggest complex command chains based on previous user behavior.

- **Remote Forensic Streaming:**

  Enabling the shell to send audit logs to a remote server in real-time to prevent local log tampering by an attacker.

- **Biometric Lock:**

  Integrating OpenCV with the shell's entry point to require facial recognition before granting access to root-level commands.

- **Network Protocol Support:**

  Extending the shell to natively handle secure remote connections and encrypted file transfers (SFTP/SSH) directly through built-in commands.

# 9. CONCLUSION

The Custom Unix Shell project successfully demonstrates the internal workings of a Unix shell, providing students with a hands-on understanding of command execution and process management. By implementing a simple, lightweight shell from scratch, learners gain valuable insights into operating system concepts and system-level programming, making this project an effective and practical educational tool.

# 10. REFERENCES


Books and Standards

1. Love, R. (2013). Linux System Programming: Talking Directly to the Kernel and C Library. O'Reilly Media. (Used for understanding fork(), exec(), and waitpid() implementation).
2. Stevens, W. R., & Rago, S. A. (2013). Advanced Programming in the UNIX Environment. Addison-Wesley Professional. (Used for terminal control and process signals).
3. Kernighan, B. W., & Ritchie, D. M. (1988). The C Programming Language. Prentice Hall. (Used for core logic development in C).

Technical Documentation (Man Pages)

1. Linux Manual Pages (man-pages 5.x). * fork(2) – Process creation.
2. execvp(3) – Executing a file with argument arrays.
3. waitpid(2) – Waiting for state changes in a child process.
4. stat(2) – Retrieving file status and Inode metadata.
5. getuid(2) – User identity verification for the Security Sentinel.
6. stdio(3) – File-based logging implementation for shell_logger.

Tools and Software

1. GCC (GNU Compiler Collection). Online Documentation. Retrieved from *https://gcc.gnu.org/onlinedocs/*.
2. OpenCV (Open Source Computer Vision Library). Reference Manual. Retrieved from *https://docs.opencv.org/*.

Web Resources

1. GNU Operating System. The GNU C Library (glibc). *https://www.gnu.org/software/libc/*.
2. Ubuntu Documentation. Security Administration and Logging. *https://ubuntu.com/server/docs.*