Process Management Tool

Fall 2024

System Programming Lab

Submitted by:

Name : **Hassan Zaib Jadoon, Ahsan Raza, Mutahhar Fayyaz**

Reg no**. : 22PWCSE2144, 22PWCSE2099, 22PWCSE2176**

Class Section **: A**

Submitted to:

Engr. Abdullah Hamid

Department of Computer Systems Engineering

University of Engineering and Technology, Peshawar

Project Report: Process Management Tool

1. Introduction

- The **Process Management Tool** is a command-line utility designed to monitor and manage system processes on a Linux-based operating system. It provides a user-friendly interface for listing active processes, terminating processes, analyzing system load, tracking specific processes, and starting new processes. The program is written in C and leverages system calls and Linux's `/proc` filesystem to gather and display process-related information.

---

2. Features

The program offers the following features:

1. **List Active Processes**: Displays a list of all active processes with details such as PID, user, state, and command.
2. **Terminate Process**: Allows the user to terminate a process using either SIGTERM or SIGKILL.
3. **Monitor System Load**: Provides an analysis of system load averages and memory usage.
4. **Get Process Details**: Displays detailed information about a specific process, including its state, memory usage, and command.
5. **Start New Process**: Executes a user-specified command as a new process.
6. **Track Process**: Tracks specific processes and displays their runtime and state.
7. **Show Tracked Processes**: Lists all currently tracked processes and their status.

Track New Process
Allows tracking of specific processes
List Active Processes
Displays currently running processes with details
Start New Process
Enables starting new processes by executing commands
Terminate Process
Allows termination of processes using signals
Get Process Details
Provides detailed information about a process
Monitor System Load
Analyzes and shows system load and memory

Figure 1: Overview of Features of Process Management Tools

3. Technical Details

3.1 Data Structures

`ProcessInfo` **Structure:**

```
typedef struct {
    pid_t pid;
    char name[256];
    char user[256];
    long memory;
    char state;
    time_t start_time;
} ProcessInfo;
```

This structure stores information about tracked processes, including PID, name, user, memory usage, state, and start time.

3.2 Key Functions

```
list_processes()
```

- Lists all active processes by reading the `/proc` directory.
- Displays PID, user, state, and command for each process.

```
terminate_process(pid_t pid, const char* signal_type)
```

- Terminates a process using either `SIGTERM` or `SIGKILL`.

```
analyze_system_load()
```

- Reads `/proc/loadavg` and `/proc/meminfo` to display system load averages and memory usage.

```
get_process_details(pid_t pid)
```

- Retrieves detailed information about a specific process from `/proc/[pid]/status` and `/proc/[pid]/cmdline.`

```
start_process(const char* command)
```

- Executes a user-specified command using `fork()` and `execvp().`

```
track_process(pid_t pid)
```

- Tracks a process by storing its details in the `tracked_processes` array.

```
display_tracked_processes()
```

- Displays the status of all tracked processes, including their runtime and current state.

4. Implementation Details

4.1 Process Listing

- The program reads the `/proc` directory, which contains information about all running processes.
- For each process, it reads the `/proc/[pid]/status` file to extract details such as PID, state, and memory usage.

4.2 Process Termination

- The `kill()` system call is used to send either `SIGTERM` or `SIGKILL` to the specified process.

4.3 System Load Analysis

- The `/proc/loadavg` file provides system load averages for the past 1, 5, and 15 minutes.

- The `/proc/meminfo` file provides detailed memory usage information.

4.4 Process Tracking

- The program maintains an array of `ProcessInfo` structures to store details of tracked processes.
- It periodically checks the status of tracked processes by verifying the existence of the `/proc/[pid]` directory.

4.5 User Interface

- The program uses ANSI color codes to enhance the user interface.
- A menu-driven interface allows users to interact with the program.

---

5. Code Structure

Header Files

Standard C libraries and system headers are included for functionality such as file I/O, process management, and signal handling.

Global Variables

- `tracked_processes[MAX_PROCESS_COUNT]:` Array to store tracked processes.
- `tracked_count:` Counter for the number of tracked processes.

Main Function

- Displays a banner and menu.
- Handles user input and invokes the appropriate functions.

---

6. Sample Output

List Active Processes:

```
ACTIVE PROCESSES:
PID      USER           STATE        COMMAND
----------------------------------------
1        root           S            init
2        root           S            kthreadd
```

Terminate Process:

```
Enter PID to terminate: 1234
Enter signal (SIGTERM/SIGKILL): SIGKILL
Process 1234 terminated successfully with SIGKILL
```

Monitor System Load:

```
SYSTEM LOAD ANALYSIS:
Load Averages: 0.25 (1m), 0.30 (5m), 0.35 (15m)

Memory Information:
MemTotal: 16384000 kB
MemFree: 12000000 kB
MemAvailable: 14000000 kB
```

Track Process:

```
Enter PID to track: 5678
Now tracking process 5678 (bash)
```

Show Tracked Processes:

```
TRACKED PROCESSES:
PID      NAME           STATE       RUNTIME(s)
------------------------------------------------
5678     bash           S           120
```

---

7. Limitations

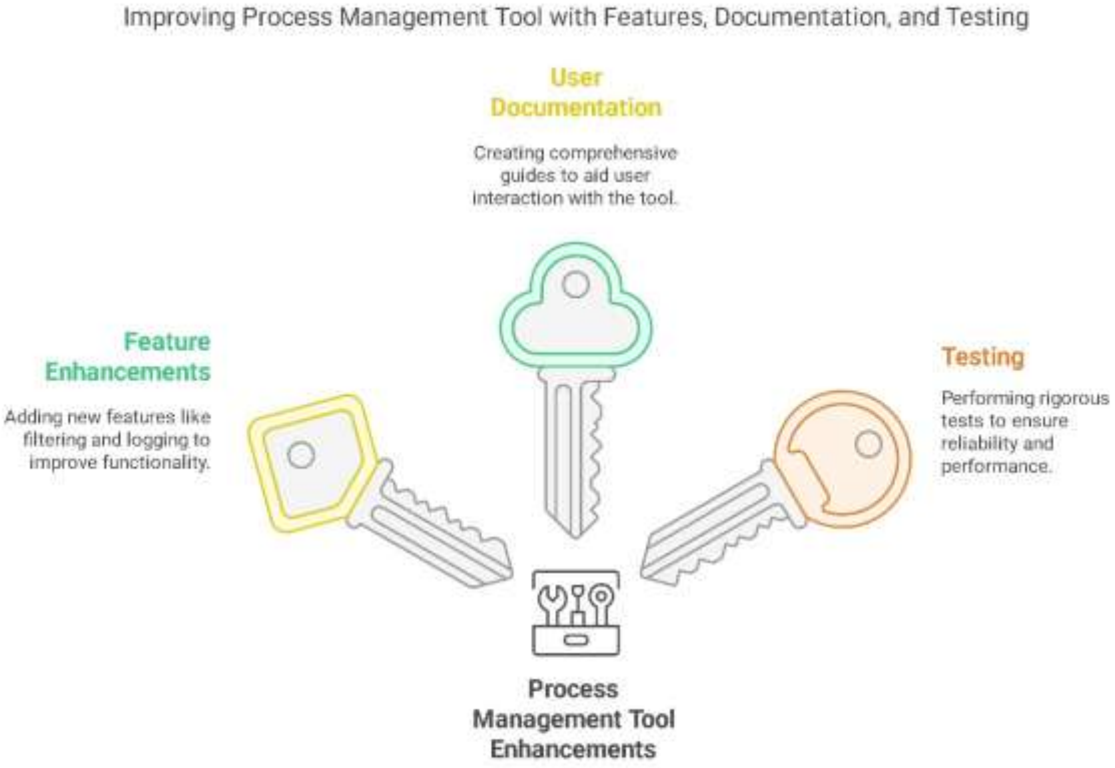
1. The program is designed for Linux systems and relies on the `/proc`

filesystem.

2. It can track a maximum of 1024 processes simultaneously.
3. Error handling is minimal, and some edge cases may not be handled gracefully.

8. Future Enhancements

1. **Enhanced Error Handling**: Improve error handling for invalid inputs and edge cases.
2. **Real-Time Monitoring**: Implement real-time monitoring of system load and processes.
3. **Graphical Interface**: Develop a graphical user interface (GUI) for better usability.
4. **Logging**: Add logging functionality to record process activities and system events.



Improving Process Management Tool with Features, Documentation, and Testing
User Documentation
Creating comprehensive guides to aid user interaction with the tool.
Feature Enhancements
Adding new features like filtering and logging to improve functionality.
Testing
Performing rigorous tests to ensure reliability and performance.
Process Management Tool Enhancements

9. Conclusion

The **Process Management Tool** is a powerful tool for system administrators and developers to monitor and manage processes on a Linux system. Its modular design and user-friendly interface make it easy to extend and customize for specific use cases. With further enhancements, it can become an indispensable tool for system monitoring and process management.

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/resource.h>
#include <ctype.h>
#include <time.h>
#include <pwd.h>

// ANSI Color Codes
#define COLOR_RED     "\x1b[31m"
#define COLOR_GREEN   "\x1b[32m"
#define COLOR_YELLOW  "\x1b[33m"
#define COLOR_BLUE    "\x1b[34m"
#define COLOR_MAGENTA "\x1b[35m"
#define COLOR_CYAN    "\x1b[36m"
#define COLOR_RESET   "\x1b[0m"

#define MAX_COMMAND_LENGTH 256
#define MAX_PROCESS_COUNT 1024
#define PATH_MAX 4096

typedef struct {
    pid_t pid;
    char name[256];
    char user[256];
    long memory;
    char state;
    time_t start_time;
} ProcessInfo;

ProcessInfo tracked_processes[MAX_PROCESS_COUNT];
int tracked_count = 0;

// Added the missing functions
int terminate_process(pid_t pid, const char* signal_type) {
    int sig;
    if (strcmp(signal_type, "SIGKILL") == 0) {
        sig = SIGKILL;
    } else {
        sig = SIGTERM;
    }

    if (kill(pid, sig) == 0) {
        printf(COLOR_GREEN "Process %d terminated successfully with %s\n" COLOR_RESET, pid, signal_type);
        return 0;
    } else {
        printf(COLOR_RED "Error terminating process %d\n" COLOR_RESET, pid);
        return -1;
    }
}

void get_process_details(pid_t pid) {
    char path[PATH_MAX], line[256];
    FILE *fp;

    // Status file
    snprintf(path, sizeof(path), "/proc/%d/status", pid);
    fp = fopen(path, "r");
    if (!fp) {
        printf(COLOR_RED "Unable to get details for process %d\n" COLOR_RESET, pid);
        return;
    }

    printf(COLOR_YELLOW "\nProcess Details for PID %d:\n" COLOR_RESET, pid);
    printf("----------------------------------------\n");

    while (fgets(line, sizeof(line), fp)) {
        // Print important process information
        if (strncmp(line, "Name:", 5) == 0 ||
            strncmp(line, "State:", 6) == 0 ||
            strncmp(line, "Pid:", 4) == 0 ||
            strncmp(line, "PPid:", 5) == 0 ||
            strncmp(line, "VmSize:", 7) == 0 ||
            strncmp(line, "VmRSS:", 6) == 0 ||
            strncmp(line, "Threads:", 8) == 0) {
            printf("%s", line);
        }
    }
    fclose(fp);

    // Cmdline
    snprintf(path, sizeof(path), "/proc/%d/cmdline", pid);
    fp = fopen(path, "r");
    if (fp) {
        if (fgets(line, sizeof(line), fp)) {
            printf("Command: %s\n", line);
        }
        fclose(fp);
    }
}

int start_process(const char* command) {
    pid_t pid = fork();

    if (pid < 0) {
        printf(COLOR_RED "Error: Fork failed\n" COLOR_RESET);
        return -1;
    } else if (pid == 0) {
        // Child process
        char* args[] = {"/bin/sh", "-c", (char*)command, NULL};
        execvp("/bin/sh", args);
        printf(COLOR_RED "Error: Command execution failed\n" COLOR_RESET);
        exit(1);
    } else {
        // Parent process
        int status;
        waitpid(pid, &status, 0);
        if (WIFEXITED(status)) {
            printf(COLOR_GREEN "Process completed with status %d\n" COLOR_RESET, WEXITSTATUS(status));
            return 0;
        } else {
            printf(COLOR_RED "Process terminated abnormally\n" COLOR_RESET);
            return -1;
        }
    }
}

// Rest of the existing functions remain the same
void display_banner() {
    printf(COLOR_CYAN);
    printf(
        "╔════════════════════════════════════════════╗\n"
        "║     🖥  Advanced Process Manager 2.0 🖥     ║\n"
        "╠════════════════════════════════════════════╣\n"
        "║     System Monitoring & Control Center     ║\n"
        "╚════════════════════════════════════════════╝\n"
        COLOR_RESET);
}

void list_processes() {
    DIR *dir;
    struct dirent *entry;
    char path[PATH_MAX], line[256], user[256];
    FILE *fp;
    struct passwd *pw;

    printf(COLOR_GREEN "\nACTIVE PROCESSES:\n" COLOR_RESET);
    printf("%-8s %-15s %-12s %-8s\n", "PID", "USER", "STATE", "COMMAND");
    printf("----------------------------------------\n");

    dir = opendir("/proc");
    if (!dir) {
        perror(COLOR_RED "Failed to open /proc" COLOR_RESET);
        return;
    }

    while ((entry = readdir(dir))) {
        if (!isdigit(*entry->d_name))
            continue;

        pid_t pid = atoi(entry->d_name);
        snprintf(path, sizeof(path), "/proc/%d/status", pid);

        fp = fopen(path, "r");
        if (!fp) continue;

        char state = '?';
        uid_t uid = 0;
        char command[256] = "unknown";

        while (fgets(line, sizeof(line), fp)) {
            if (strncmp(line, "State:", 6) == 0) {
                sscanf(line, "State: %c", &state);
            } else if (strncmp(line, "Uid:", 4) == 0) {
                sscanf(line, "Uid: %d", &uid);
            } else if (strncmp(line, "Name:", 5) == 0) {
                sscanf(line, "Name: %255s", command);
            }
        }
        fclose(fp);

        pw = getpwuid(uid);
        strncpy(user, pw ? pw->pw_name : "unknown", sizeof(user)-1);

        printf("%-8d %-15s %-12c %-8s\n",
            pid, user, state, command);
    }
    closedir(dir);
}

void analyze_system_load() {
    FILE *fp;
    char line[256];
    double loads[3];

    printf(COLOR_MAGENTA "\nSYSTEM LOAD ANALYSIS:\n" COLOR_RESET);

    // CPU Load
    fp = fopen("/proc/loadavg", "r");
    if (fp) {
        if (fscanf(fp, "%lf %lf %lf", &loads[0], &loads[1], &loads[2]) == 3) {
            printf("Load Averages: %.2f (1m), %.2f (5m), %.2f (15m)\n",
                loads[0], loads[1], loads[2]);
        }
        fclose(fp);
    }

    // Memory Info
    fp = fopen("/proc/meminfo", "r");
    if (fp) {
        printf("\nMemory Information:\n");
        int count = 0;
        while (fgets(line, sizeof(line), fp) && count < 3) {
            printf("%s", line);
            count++;
        }
        fclose(fp);
    }
}

void track_process(pid_t pid) {
    if (tracked_count >= MAX_PROCESS_COUNT) {
        printf(COLOR_RED "Maximum tracking limit reached\n" COLOR_RESET);
        return;
    }

    char path[PATH_MAX], line[256];
    snprintf(path, sizeof(path), "/proc/%d/status", pid);

    FILE *fp = fopen(path, "r");
```

```c
    if (!fp) {
        printf(COLOR_RED "Process %d not found\n" COLOR_RESET, pid);
        return;
    }

    ProcessInfo *proc = &tracked_processes[tracked_count];
    proc->pid = pid;
    proc->start_time = time(NULL);

    while (fgets(line, sizeof(line), fp)) {
        if (strncmp(line, "Name:", 5) == 0) {
            sscanf(line, "Name: %255s", proc->name);
        } else if (strncmp(line, "State:", 6) == 0) {
            sscanf(line, "State: %c", &proc->state);
        }
    }
    fclose(fp);

    tracked_count++;
    printf(COLOR_GREEN "Now tracking process %d (%s)\n" COLOR_RESET,
        pid, proc->name);
}

void display_tracked_processes() {
    if (tracked_count == 0) {
        printf(COLOR_YELLOW "No processes being tracked\n" COLOR_RESET);
        return;
    }

    printf(COLOR_GREEN "\nTRACKED PROCESSES:\n" COLOR_RESET);
    printf("%-8s %-15s %-10s %-15s\n", "PID", "NAME", "STATE", "RUNTIME(s)");
    printf("------------------------------------------------\n");

    time_t now = time(NULL);
    for (int i = 0; i < tracked_count; i++) {
        ProcessInfo *proc = &tracked_processes[i];
        long runtime = now - proc->start_time;

        // Verify if process still exists
        char path[PATH_MAX];
        snprintf(path, sizeof(path), "/proc/%d", proc->pid);
        if (access(path, F_OK) != -1) {
            printf("%-8d %-15s %-10c %-15ld\n",
                proc->pid, proc->name, proc->state, runtime);
        } else {
            printf("%-8d %-15s %-10s %-15s\n",
                proc->pid, proc->name, "ENDED", "-");
        }
    }
}

void display_menu() {
    printf("\n" COLOR_BLUE);
    printf("╔══════════════════════════════════════════════╗\n");
    printf("║                 MENU OPTIONS                 ║\n");
    printf("╠══════════════════════════════════════════════╣\n");
    printf("║  1. " COLOR_CYAN "List Active Processes            " COLOR_BLUE "║\n");
    printf("║  2. " COLOR_CYAN "Terminate Process                " COLOR_BLUE "║\n");
    printf("║  3. " COLOR_CYAN "Monitor System Load              " COLOR_BLUE "║\n");
    printf("║  4. " COLOR_CYAN "Get Process Details              " COLOR_BLUE "║\n");
    printf("║  5. " COLOR_CYAN "Start New Process                " COLOR_BLUE "║\n");
    printf("║  6. " COLOR_CYAN "Track New Process                " COLOR_BLUE "║\n");
    printf("║  7. " COLOR_CYAN "Show Tracked Processes           " COLOR_BLUE "║\n");
    printf("║  8. " COLOR_RED "Exit                              " COLOR_BLUE "║\n");
    printf("╚══════════════════════════════════════════════╝\n");
    printf(COLOR_GREEN "Enter your choice: " COLOR_RESET);
}

int main() {
    int choice;
    char input[256];
    pid_t pid;

    display_banner();

    while (1) {
        display_menu();
        if (scanf("%d", &choice) != 1) {
            while (getchar() != '\n'); // Clear input buffer
            printf(COLOR_RED "Invalid input. Please enter a number.\n" COLOR_RESET);
            continue;
        }
        while (getchar() != '\n'); // Clear input buffer

        switch(choice) {
            case 1:
                list_processes();
                break;
            case 2:
                printf("Enter PID to terminate: ");
                if (scanf("%d", &pid) == 1) {
                    printf("Enter signal (SIGTERM/SIGKILL): ");
                    scanf("%s", input);
                    terminate_process(pid, input);
                }
                while (getchar() != '\n'); // Clear input buffer
                break;
            case 3:
                analyze_system_load();
                break;
            case 4:
                printf("Enter PID for details: ");
                if (scanf("%d", &pid) == 1) {
                    get_process_details(pid);
                }
                while (getchar() != '\n');
                break;
            case 5:
                printf("Enter command to execute: ");
                fgets(input, sizeof(input), stdin);
                input[strcspn(input, "\n")] = 0;
                start_process(input);
                break;
            case 6:
                printf("Enter PID to track: ");
                if (scanf("%d", &pid) == 1) {
                    track_process(pid);
                }
                while (getchar() != '\n');
                break;
            case 7:
                display_tracked_processes();
                break;
            case 8:
                printf(COLOR_RED "Exiting Process Manager.\n" COLOR_RESET);
                return 0;
            default:
                printf(COLOR_RED "Invalid choice. Please try again.\n" COLOR_RESET);
        }
    }
    return 0;
}
```