Jaypee Institute of Information Technology

Department of Computer Science & Engineering and Information Technology

Project Title: ProSysMonitor: An Intelligent System Resource Monitoring and Optimization Tool

Submitted to:
Dr. Prashant Kaushik
Dr. Vivek Kumar Singh

Submitted By:

ALOKIK GARG 22103302
SHIVAM ARYAN 22103298
SIDDHARTH AGRAWAL 22103279
DAVID SHARMA 22103299

COURSE : Operating Systems and System Programming (15B17CI472)
ODD 2024

Abstract:

ProSysMonitor is an advanced system resource management and monitoring tool designed to provide comprehensive insights into system performance, resource utilization, and process management. The application offers a robust set of features that enable system administrators and users to monitor, analyze, and control various system resources in real-time. By leveraging low-level system interfaces and providing a user-friendly command-line interface, ProSysMonitor enables detailed system diagnostics, performance tracking, and proactive resource management.

Key capabilities include real-time CPU, memory, and disk usage monitoring, process profiling, network connection tracking, disk scheduling simulation, and system security assessment. The tool employs advanced algorithms like Exponential Moving Average (EMA) for smoothing resource usage metrics and implements various scheduling techniques such as Round Robin and Shortest Seek Time First (SSTF).

Scope of the Project

a. Functional Requirements

1. Real-time system resource monitoring (CPU, memory, disk)
2. Process control and management
3. Resource usage logging and alerting
4. Disk I/O scheduling simulation
5. Network usage monitoring
6. System performance analysis
7. Process tree visualization
8. File descriptor tracking
9. TCP connection monitoring
10. System security assessment
11. Detailed process resource usage analysis

b. Non-functional Requirements

1. Performance efficiency
2. Low system overhead

3. Accurate resource measurement
4. User-friendly interface
5. Robust error handling

c. Project Modules

1. **Resource Monitoring Module**
    a) Tracks CPU, memory, and disk usage
    b) Implements Exponential Moving Average (EMA)
    c) Generates real-time resource alerts
2. **Process Management Module**
    a) Controls process lifecycle
    b) Sets process priorities
    c) Monitors process resources
    d) Provides process profiling
3. **Disk Scheduling Module**
    a) Implements FCFS and SSTF algorithms
    b) Simulates disk head movement
    c) Calculates seek times
4. **Network and Security Module**
    a) Monitors network interfaces
    b) Tracks TCP connections
    c) Performs system security checks
5. **Performance Analysis Module**
    a) Generates system-wide performance reports
    b) Analyzes process resource consumption
    c) Provides detailed memory insights

Tools and Technologies Used

1. Programming Language: C++
2. Operating System: Linux
3. System Interfaces: /proc filesystem

4. System Calls: fork(), kill(), setpriority()
5. Multithreading: POSIX Threads (pthread)
6. Command-line Tools: top, free, df, iotop, netstat
7. Development Environment: GCC Compiler
8. Version Control: Git

Design of the Project

Architectural Components

1. Main Menu Interface
2. Resource Monitoring Engine
3. Process Control Subsystem
4. Scheduling Algorithms Implementation
5. Reporting and Logging Mechanism

Key Design Principles

1. Modular Architecture
2. Low-level System Access
3. Real-time Monitoring
4. Minimal Performance Overhead

Algorithmic Components

1. Exponential Moving Average (CPU Usage Smoothing)
2. Round Robin CPU Scheduling
3. Shortest Seek Time First (SSTF) Disk Scheduling

Implementation Details

Core Functionalities Implemented

1. Resource Monitoring

```
void monitor_resources() {
    printf("Monitoring system resources...\n");

    // Get real-time CPU usage using 'top' command
    FILE *fp = popen("top -b -n1 | grep 'Cpu(s)' | awk '{print $2 + $4}'", "r");  // CPU usage as a percentage
    if (fp == NULL) {
        printf("Failed to run top command\n");
        exit(1);
    }
    float current_cpu = 0;
    fscanf(fp, "%f", &current_cpu);
    pclose(fp);

    // Get real-time memory usage using 'free' command
    fp = popen("free | grep Mem | awk '{print $3/$2 * 100.0}'", "r");  // Memory usage as percentage
    if (fp == NULL) {
        printf("Failed to run free command\n");
        exit(1);
    }
    float current_memory = 0;
    fscanf(fp, "%f", &current_memory);
    pclose(fp);

    // Get real-time disk usage using 'df' command
    fp = popen("df / | grep / | awk '{ print $5 }'", "r");  // Disk usage as percentage
    if (fp == NULL) {
        printf("Failed to run df command\n");
        exit(1);
    }
    int current_disk_io = 0;
    fscanf(fp, "%d", &current_disk_io);
    pclose(fp);

    // Update CPU usage history and calculate EMA
    cpu_usage_history[history_index] = current_cpu;
    history_index = (history_index + 1) % HISTORY_SIZE;
    float smoothed_cpu = calculate_ema(cpu_usage_history, HISTORY_SIZE);

    printf("\nSmoothed CPU Usage (EMA): %.2f%%\n", smoothed_cpu);
    printf("Current CPU Usage: %.2f%%\n", current_cpu);
    printf("Current Memory Usage: %.2f%%\n", current_memory);
    printf("Current Disk Usage: %d%%\n", current_disk_io);
}
```

2. Process Profiling

```c
void profile_process() {
    int pid;
    printf("Enter the PID of the process to profile: ");
    scanf("%d", &pid);

    // Get process statistics using /proc filesystem
    char stat_path[256];
    sprintf(stat_path, "/proc/%d/stat", pid);

    FILE *stat_file = fopen(stat_path, "r");
    if (stat_file == NULL) {
        printf("Unable to open process statistics. Check if PID exists.\n");
        return;
    }

    // Variables to store process statistics
    char comm[256];
    char state;
    unsigned long utime, stime, vsize;
    long rss;

    // Read process statistics
    fscanf(stat_file, "%*d (%[^)]) %c %*d %*d %*d %*d %*d %*u %*u %*u %*u %*u %lu %lu %*d %*d %*d %*d %*d %*d %*u %lu %ld",
           comm, &state, &utime, &stime, &vsize, &rss);
    fclose(stat_file);

    // Calculate CPU usage time in seconds
    float cpu_time = (utime + stime) / (float)sysconf(_SC_CLK_TCK);

    // Get memory information
    char status_path[256];
    sprintf(status_path, "/proc/%d/status", pid);
    FILE *status_file = fopen(status_path, "r");

    long vm_peak = 0, vm_size = 0, vm_rss = 0;
    char line[256];

    if (status_file != NULL) {
        while (fgets(line, sizeof(line), status_file)) {
            if (strncmp(line, "VmPeak:", 7) == 0) sscanf(line, "VmPeak: %ld", &vm_peak);
            if (strncmp(line, "VmSize:", 7) == 0) sscanf(line, "VmSize: %ld", &vm_size);
            if (strncmp(line, "VmRSS:", 6) == 0) sscanf(line, "VmRSS: %ld", &vm_rss);
        }
        fclose(status_file);
    }
```

```
Enter your choice: 6
Enter the PID of the process to profile: 5569

Process Profile for PID 5569:
Name: gnome-system-mo
State: S
CPU Time: 2.43 seconds
Virtual Memory Size: 1896419328 bytes
RSS (Resident Set Size): 39350 pages
Peak Virtual Memory: 1888168 kB
Current Virtual Memory: 1851972 kB
Current RSS: 157400 kB
```

3. **Analyse Process Usage**

CODE:

```
void analyze_process_resources() {
    printf("\nAnalyzing Process Resource Usage...\n");

    // Structure to hold process information
    struct ProcessInfo {
        pid_t pid;
        char name[256];
        float cpu_usage;
        long memory_usage;
        long read_bytes;
        long write_bytes;
    };

    // Arrays to store top and bottom processes
    ProcessInfo top_cpu[5];
    ProcessInfo bottom_cpu[5];
    ProcessInfo top_memory[5];
    ProcessInfo bottom_memory[5];

    // Initialize arrays
    for (int i = 0; i < 5; i++) {
        top_cpu[i].cpu_usage = -1;
        bottom_cpu[i].cpu_usage = 101;
        top_memory[i].memory_usage = -1;
        bottom_memory[i].memory_usage = LONG_MAX;
    }

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

    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
```

```
        // Check if the entry is a process directory (numeric name)
        if (entry->d_type == DT_DIR) {
            char *endptr;
            pid_t pid = strtol(entry->d_name, &endptr, 10);
            if (*endptr != '\0') continue;  // Not a number

            char stat_path[256];
            char status_path[256];
            char io_path[256];
            sprintf(stat_path, "/proc/%d/stat", pid);
            sprintf(status_path, "/proc/%d/status", pid);
            sprintf(io_path, "/proc/%d/io", pid);

            // Get process information
            FILE *stat_file = fopen(stat_path, "r");
            FILE *status_file = fopen(status_path, "r");
            FILE *io_file = fopen(io_path, "r");

            if (stat_file && status_file) {
                ProcessInfo current;
                current.pid = pid;

                // Read process name and CPU usage from stat
                char state;
                unsigned long utime, stime;
                fscanf(stat_file, "%*d (%[^)]) %c %*d %*d %*d %*d %*d %*u %*u %*u
%*u %*u %lu %lu",
                       current.name, &state, &utime, &stime);

                // Calculate CPU usage percentage
                current.cpu_usage = ((float)(utime + stime) / sysconf(_SC_CLK_TCK)) * 100;

                // Read memory usage from status
                char line[256];
                while (fgets(line, sizeof(line), status_file)) {
                    if (strncmp(line, "VmRSS:", 6) == 0) {
                        sscanf(line, "VmRSS: %ld", &current.memory_usage);
```

```
                break;
            }
        }

        // Read I/O statistics if available
        if (io_file) {
            while (fgets(line, sizeof(line), io_file)) {
                if (strncmp(line, "read_bytes:", 11) == 0)
                    sscanf(line, "read_bytes: %ld", &current.read_bytes);
                else if (strncmp(line, "write_bytes:", 12) == 0)
                    sscanf(line, "write_bytes: %ld", &current.write_bytes);
            }
            fclose(io_file);
        }

        // Update top and bottom CPU usage arrays
        for (int i = 0; i < 5; i++) {
            if (current.cpu_usage > top_cpu[i].cpu_usage) {
                memmove(&top_cpu[i + 1], &top_cpu[i], sizeof(ProcessInfo) * (4 - i));
                top_cpu[i] = current;
                break;
            }
            if (current.cpu_usage < bottom_cpu[i].cpu_usage && current.cpu_usage > 0) {
                memmove(&bottom_cpu[i + 1], &bottom_cpu[i], sizeof(ProcessInfo) * (4 - i));
                bottom_cpu[i] = current;
                break;
            }
        }

        // Update top and bottom memory usage arrays
        for (int i = 0; i < 5; i++) {
            if (current.memory_usage > top_memory[i].memory_usage) {
                memmove(&top_memory[i + 1], &top_memory[i], sizeof(ProcessInfo) * (4 - i));
                top_memory[i] = current;
```

```
                        break;
                    }
                    if (current.memory_usage < bottom_memory[i].memory_usage && current.memory_usage > 0) {
                        memmove(&bottom_memory[i + 1], &bottom_memory[i], sizeof(ProcessInfo) * (4 - i));
                        bottom_memory[i] = current;
                        break;
                    }
                }

                fclose(stat_file);
                fclose(status_file);
            }
        }
    }
    closedir(dir);
```

```
Analyzing Process Resource Usage...

Top 5 CPU-Intensive Processes:
PID     CPU%    MEM(KB) PROCESS
6409    21632.0 480036  brave
5945    21265.0 364852  brave
4544    5884.0  363312  code
3362    4828.0  316544  gnome-shell
3177    3179.0  118196  Xorg

Bottom 5 CPU-Usage Processes:
PID     CPU%    MEM(KB) PROCESS
24      1.0     14048   ksoftirqd/2
30      1.0     14048   ksoftirqd/4
36      1.0     14048   ksoftirqd/6
42      1.0     14048   ksoftirqd/8
48      1.0     14048   ksoftirqd/9

Top 5 Memory-Intensive Processes:
PID     MEM(KB) CPU%    PROCESS
6409    480036  21632.0 brave
7842    480036  0.0     kworker/8:2
5945    364852  21265.0 brave
4544    363312  5884.0  code
3362    316544  4828.0  gnome-shell

Bottom 5 Memory-Usage Processes:
PID     MEM(KB) CPU%    PROCESS
1901    768     0.0     run-cups-browse
1599    1280    11.0    run-cups-browse
1104    1292    0.0     avahi-daemon
1600    1408    19.0    run-cupsd
9713    1536    1.0     mainfile-1

System-wide Resource Usage Summary:
CPU Usage:
5.8%
Memory Usage:
45.1905%
Swap Usage:
0%
```

4. **System Security Check**

CODE:

```
oid check_system_security() {
    printf("\nSystem Security Check:\n");

    // Check for ASLR (Address Space Layout Randomization)
    FILE *aslr = fopen("/proc/sys/kernel/randomize_va_space", "r");
    if (aslr) {
        char value;
        fscanf(aslr, "%c", &value);
        printf("ASLR Status: %s (value=%c)\n",
               value != '0' ? "Enabled" : "Disabled", value);
        fclose(aslr);
    }

    // Check core dump settings
    FILE *core = fopen("/proc/sys/kernel/core_pattern", "r");
    if (core) {
        char pattern[256];
        fgets(pattern, sizeof(pattern), core);
        printf("Core dump pattern: %s", pattern);
        fclose(core);
    }

    // Check for running security services
    printf("\nSecurity Services Status:\n");
    system("systemctl is-active --quiet apparmor && "
           "echo 'AppArmor: Running' || echo 'AppArmor: Not Running'");
    system("systemctl is-active --quiet selinux && "
           "echo 'SELinux: Running' || echo 'SELinux: Not Running'");
    system("systemctl is-active --quiet ufw && "
           "echo 'UFW Firewall: Running' || echo 'UFW Firewall: Not Running'");

    // Check for open ports
    printf("\nOpen Ports:\n");
    system("ss -tuln | grep LISTEN");
}
```

```
System Security Check:
ASLR Status: Enabled (value=2)
Core dump pattern: |/usr/share/apport/apport -p%p -s%s -c%c -d%d -P%P -u%u -g%g -- %E

Security Services Status:
AppArmor: Running
SELinux: Not Running
UFW Firewall: Running

Open Ports:
tcp   LISTEN 0      4096      127.0.0.54:53         0.0.0.0:*
tcp   LISTEN 0      4096       127.0.0.1:631        0.0.0.0:*
tcp   LISTEN 0      4096   127.0.0.53%lo:53         0.0.0.0:*
tcp   LISTEN 0      4096           [::1]:631           [::]:*
```

5. **Monitor TCP Connection**

CODE:

```
void monitor_tcp_connections() {
    FILE *fp = popen("netstat -tn", "r");
    if (!fp) {
        printf("Failed to run netstat command\n");
        return;
    }

    printf("\nActive TCP Connections:\n");
    printf("Proto Local Address           Foreign Address         State\n");

    char line[256];
    int connection_count = 0;
    // Skip header lines
    for (int i = 0; i < 2; i++) {
        fgets(line, sizeof(line), fp);
    }

    while (fgets(line, sizeof(line), fp)) {
        if (strstr(line, "tcp")) {
            printf("%s", line);
            connection_count++;
        }
    }
```

```
    printf("\nTotal TCP connections: %d\n", connection_count);
    pclose(fp);
}
```

```
Active TCP Connections:
Proto Local Address           Foreign Address         State
sh: 1: netstat: not found

Total TCP connections: 0
```

6. **Analyze Memory Usage**

CODE:

```
void analyze_memory_details() {
    FILE *meminfo = fopen("/proc/meminfo", "r");
    if (!meminfo) {
        printf("Cannot open memory information\n");
        return;
    }

    printf("\nDetailed Memory Analysis:\n");
    char line[256];
    while (fgets(line, sizeof(line), meminfo)) {
        // Print specific memory metrics
        if (strstr(line, "MemTotal") ||
            strstr(line, "MemFree") ||
            strstr(line, "MemAvailable") ||
            strstr(line, "Buffers") ||
            strstr(line, "Cached") ||
            strstr(line, "SwapTotal") ||
            strstr(line, "SwapFree") ||
            strstr(line, "Dirty") ||
            strstr(line, "Writeback")) {
            printf("%s", line);
        }
    }
    fclose(meminfo);

    // Get memory page size
```

```
    long page_size = sysconf(_SC_PAGESIZE);
    printf("Memory Page Size: %ld bytes\n", page_size);
}
```

```
Detailed Memory Analysis:
MemTotal:         7793496 kB
MemFree:          1904408 kB
MemAvailable:     4194528 kB
Buffers:           141768 kB
Cached:           2864100 kB
SwapCached:             0 kB
SwapTotal:       15625212 kB
SwapFree:        15625212 kB
Dirty:                740 kB
Writeback:              0 kB
WritebackTmp:           0 kB
Memory Page Size: 4096 bytes
```

Testing Details

1. Resource Monitoring Accuracy Test
2. Process Management Verification
3. Disk Scheduling Algorithm Validation
4. Network Connection Tracking Test
5. System Security Assessment Test

References

1. *Linux System Programming by Robert Love*
2. *Advanced Programming in the UNIX Environment by W. Richard Stevens*
3. *The Linux Kernel Documentation*
4. *POSIX Threads Programming Guide*

Conclusion

ProSysMonitor demonstrates a comprehensive approach to system resource management, providing administrators with powerful tools for monitoring, analyzing, and controlling system resources efficiently.

Recommendations for Future Enhancements:

1. Graphical User Interface (GUI)
2. Remote monitoring capabilities
3. Advanced machine learning-based predictive analysis
4. Enhanced visualization of resource trends