SYSTEM PROGRAMMING

GROUP MEMBER

Name:
MOHAMAD SYAHMI BIN SORIA

Matrix:
84553

Name:
CHRIS HEZRON ANAK WILFRED CALVYN

Matrix:
83498

Name:
MOHD RUZAINI BIN ABU BAKAR

Matrix:
86650

Name:
THOMAS SONG SHYANG FA

Matrix:
87054

INTRODUCTTION

SCENARIO

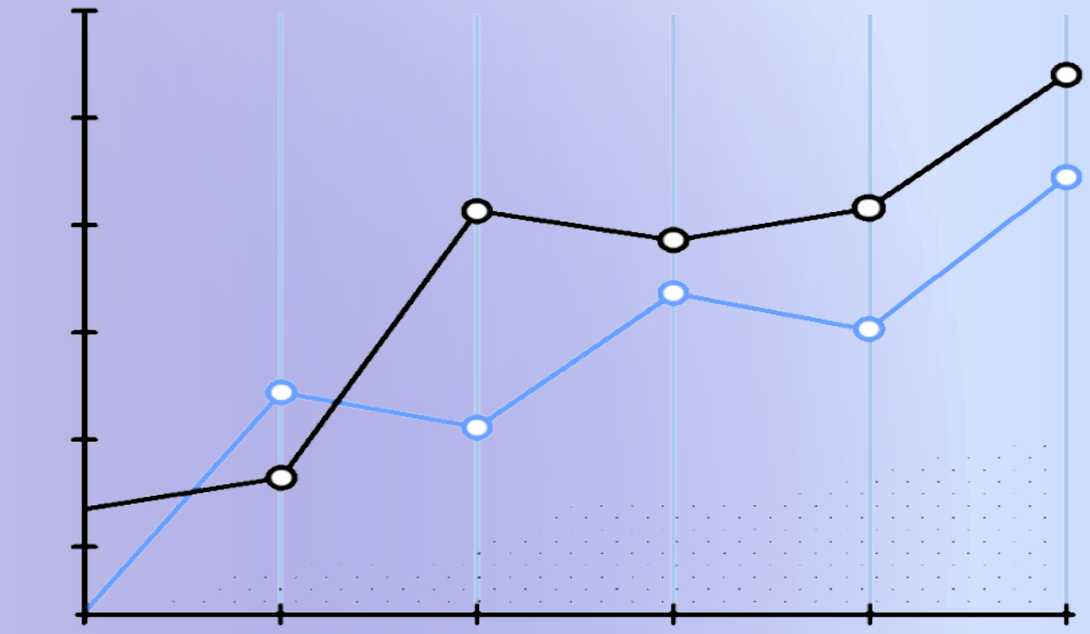
Our servers are running slow, and I need a simple monitoring tool that shows real-time CPU, memory, and process usage. I don’t want to install heavy tools like htop, I want something written in C using system calls. Can you build that?

SOLUTION

- A simple and lightweight monitoring tool
- Written entirely in C programming
- Uses native Linux system call to minimize overhead
- Interact directly with the /proc filesysten

PART A
CODING TASK

EXPLANATION OF MODULE



A. system resource module

- CPU: Measures active vs. idle time for real-time utilization.
- Memory: Reports Total/Used/Free RAM in Megabytes (MB).
- Processes: Identifies and ranks the top 5 resource-heavy tasks.

B. utility & infrastructure module

- Signal Handling: Ensures safe exit (Ctrl+C) without data loss.
- Logging: Saves timestamped activity to a syslog.txt file.
- User Interface: Provides an easy-to-use, error-checked menu.

SOURCE CODE SNIPPETS

CPU Stats Parsing

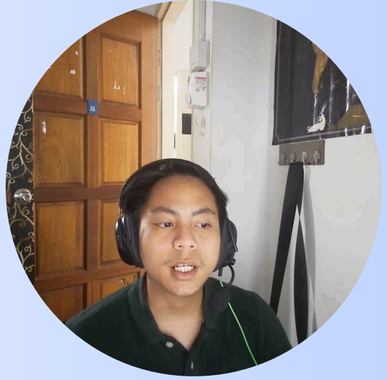
```c
void read_cpu_stats(CPUStats *stats) {
    int fd = open("/proc/stat", O_RDONLY);
    char buffer[BUFFER_SIZE];
    ssize_t bytes_read = read(fd, buffer, sizeof(buffer) - 1);
    if (bytes_read > 0) {
        buffer[bytes_read] = '\0';
        sscanf(buffer, "cpu %llu %llu %llu %llu %llu %llu %llu %llu",
               &stats->user, &stats->nice, &stats->system,
               &stats->idle, &stats->iowait, &stats->irq,
               &stats->softirq, &stats->steal);
    }
    close(fd);
}
```

The following snippet demonstrates how the tool interacts with the kernel's virtual filesystem to retrieve raw CPU ticks.

Signal Handling for Graceful Exit

```c
void handleSignal(int sig) {
    if (sig == SIGINT) {
        keep_running = 0;
        const char *msg = "\nCaught SIGINT. Exiting gracefully...\n";
        write(STDOUT_FILENO, msg, strlen(msg));
    }
}
```

This ensures that the "Continuous Monitoring" mode doesn't leave the terminal in a messy state when interrupted.

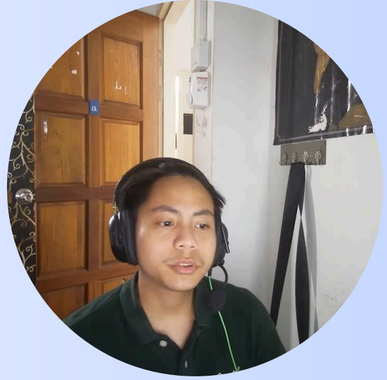


SOURCE CODE SNIPPETS

The snapshot show the function measures CPU usage by reading CPU statistics from /proc/stat at two different time points separated by a one-second interval. It calculates the difference between total and idle CPU time to determine the percentage of CPU utilization. The result is then displayed to the user and recorded in a log file for monitoring purposes.

CPU Usage function

```
void getCPUUsage() {
    CPUStats stat1, stat2;
    printf("\n--- CPU Usage ---\n");
    printf("Measuring CPU usage (sampling 1 second)...\n");

    //Take first CPU usgae snapshot
    read_cpu_stats(&stat1);
    sleep(1);
    read_cpu_stats(&stat2);  //Take second CPU Usage snapshot after 1 second

    unsigned long long total1 = stat1.user + stat1.nice + stat1.system + stat1.idle +
                                stat1.iowait + stat1.irq + stat1.softirq + stat1.steal;
    unsigned long long total2 = stat2.user + stat2.nice + stat2.system + stat2.idle +
                                stat2.iowait + stat2.irq + stat2.softirq + stat2.steal;

    unsigned long long idle1 = stat1.idle + stat1.iowait;
    unsigned long long idle2 = stat2.idle + stat2.iowait;

    unsigned long long total_delta = total2 - total1;
    unsigned long long idle_delta = idle2 - idle1;

    double cpu_usage = 0.0;
    if (total_delta > 0) {
        cpu_usage = (double)(total_delta - idle_delta) / total_delta * 100.0;
    }

    printf("CPU Usage: %.2f%%\n", cpu_usage);

    char log_buf[64];
    snprintf(log_buf, sizeof(log_buf), "CPU Usage checked: %.2f%%", cpu_usage);
    log_message(log_buf);
}
```



SOURCE CODE SNIPPETS

The snapshot demonstrate function retrieves system memory information by reading /proc/meminfo and extracting key values such as total and available memory. It calculates the used memory and usage percentage, then displays the results in a user-friendly format. This approach allows the program to monitor memory usage efficiently using the Linux kernel's virtual filesystem.

Memory Usage function

```
void getMemoryUsage() {
    int fd = open("/proc/meminfo", O_RDONLY);
    if (fd == -1) {
        perror("Error opening /proc/meminfo");
        return;
    }

    char buffer[BUFFER_SIZE];
    ssize_t bytes_read = read(fd, buffer, sizeof(buffer) - 1);
    close(fd);

    if (bytes_read <= 0) {
        perror("Error reading /proc/meminfo");
        return;
    }
    buffer[bytes_read] = '\0';

    unsigned long long total_mem = 0;
    unsigned long long available_mem = 0;

    // Parse manually or using sscanf on lines
    char *line = strtok(buffer, "\n");
    while (line != NULL) {
        if (strncmp(line, "MemTotal:", 9) == 0) {
            sscanf(line, "MemTotal: %llu kB", &total_mem);
        } else if (strncmp(line, "MemAvailable:", 13) == 0) {
            sscanf(line, "MemAvailable: %llu kB", &available_mem);
        }
        line = strtok(NULL, "\n");
    }
```

```
    printf("\n--- Memory Usage ---\n");
    if (total_mem > 0) {
        unsigned long long used_mem = total_mem - available_mem;
        double used_percent = (double)used_mem / total_mem * 100.0;
        printf("Total Memory: %llu MB\n", total_mem / 1024);
        printf("Used Memory:  %llu MB (%.2f%%)\n", used_mem / 1024, used_percent);
        printf("Free Memory:  %llu MB\n", available_mem / 1024);

        char log_buf[128];
        snprintf(log_buf, sizeof(log_buf), "Memory checked: Used %llu MB (%.2f%%)", used_mem / 1024, used_percent);
        log_message(log_buf);
    } else {
        printf("Could not read memory info.\n");
    }
}
```

SOURCE CODE SNIPPETS

The listTopProcesses() function scans the Linux /proc filesystem to gather information about currently running processes. It opens the /proc directory, where each running process is represented by a directory named after its process ID (PID). The function iterates through all entries in /proc and filters out non-numeric directory names to ensure only valid process directories are processed. For each valid PID, it constructs the file path to /proc/[pid]/stat, which contains important process statistics. The function then opens this file, reads its contents into a buffer, and safely null-terminates the data for further processing. This information is later used to store process details and identify the top processes, typically based on resource usage such as CPU time. This approach allows the program to monitor system processes efficiently using real-time kernel data without relying on external system commands.

Top 5 processess function

```
void listTopProcesses() {
    DIR *dir;
    struct dirent *entry;
    ProcessInfo processes[1024];
    int count = 0;

    dir = opendir("/proc");
    if (dir == NULL) {
        perror("opendir /proc");
        return;
    }

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

        int pid = atoi(entry->d_name);
        char path[256];
        snprintf(path, sizeof(path), "/proc/%d/stat", pid);

        int fd = open(path, O_RDONLY);
        if (fd != -1) {
            char buffer[512];
            ssize_t bytes = read(fd, buffer, sizeof(buffer) - 1);
            close(fd);

            if (bytes > 0) {
                buffer[bytes] = '\0';

                // Parse /proc/[pid]/stat
                // Format: pid (comm) state ppid ... utime stime ...
                // The comm field is in parentheses and can contain spaces.
```

```
char *close_paren = strrchr(buffer, ')');
if (close_paren) {
    char *name_start = strchr(buffer, '(');
    if (name_start) {
        // Extract name
        size_t name_len = close_paren - name_start - 1;
        if (name_len >= sizeof(processes[count].name)) name_len = sizeof(processes[count].name) - 1;
        strncpy(processes[count].name, name_start + 1, name_len);
        processes[count].name[name_len] = '\0';

        // Parse stats after ')'
        unsigned long long utime = 0, stime = 0;
        // The fields after ) are: state, ppid, pgrp, session, tty_nr, tpgid, flags, minflt, cminflt, majflt, cmajflt, utime, stime
        // That is 13 fields to skip to get to utime (14th) and stime (15th)
        // sscanf format string with skipped fields using *

        // Easier: use sscanf on the substring starting after close_paren
        // Field 3 (state) is first char after space
        // 3  4  5  6  7  8  9  10 11 12 13 14 15
        // %c %d %d %d %d %d %u %u %u %u %u %llu %llu

        char state;
        int ppid, pgrp, session, tty_nr, tpgid;
        unsigned int flags;
        unsigned long minflt, cminflt, majflt, cmajflt;

        sscanf(close_paren + 2, "%c %d %d %d %d %d %u %lu %lu %lu %lu %llu %llu",
               &state, &ppid, &pgrp, &session, &tty_nr, &tpgid, &flags,
               &minflt, &cminflt, &majflt, &cmajflt,
               &utime, &stime);
```

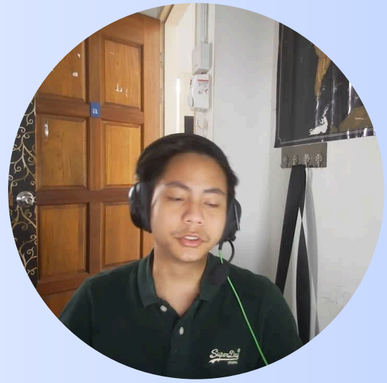
```
                    processes[count].pid = pid;
                    processes[count].cpu_time = utime + stime;
                    count++;
                    if (count >= 1024) break;
                }
            }
        }
    }
    closedir(dir);

    qsort(processes, count, sizeof(ProcessInfo), compare_processes);

    printf("\n--- Top 5 Processes (by Accumulated CPU Time) ---\n");
    printf("%-8s %-20s %-15s\n", "PID", "Name", "CPU Time (ticks)");
    for (int i = 0; i < 5 && i < count; i++) {
        printf("%-8d %-20s %llu\n", processes[i].pid, processes[i].name, processes[i].cpu_time);
    }
    log_message("Checked Top 5 Processes.");
}
```

SOURCE CODE SNIPPETS

The snapshot shows the function implements continuous system monitoring by repeatedly refreshing and displaying CPU usage, memory usage, and top processes at a user-defined time interval. It clears the terminal screen each cycle to provide a real-time dashboard view and continues running until the user presses Ctrl+C, which triggers a graceful exit via signal handling.

Continuous Monitoring Function

```
void continuousMonitor(int interval) {
    printf("\nStarting Continuous Monitoring... (Press Ctrl+C to stop)\n");
    log_message("Started Continuous Monitoring.");

    // Set up signal handler specifically for this loop if needed,
    // but global handler works too.

    while (keep_running) {
        clear_screen();
        printf("================================================\n");
        printf("   SysMonitor++ - Continuous Monitoring      \n");
        printf("   Refresh Interval: %d seconds              \n", interval);
        printf("   (Press Ctrl+C to stop)                    \n");
        printf("================================================\n");

        getCPUUsage();
        getMemoryUsage();
        listTopProcesses();

        printf("\nRefreshing in %d seconds...\n", interval);
        sleep(interval);
    }
    printf("\nContinuous monitoring stopped.\n");
}
```

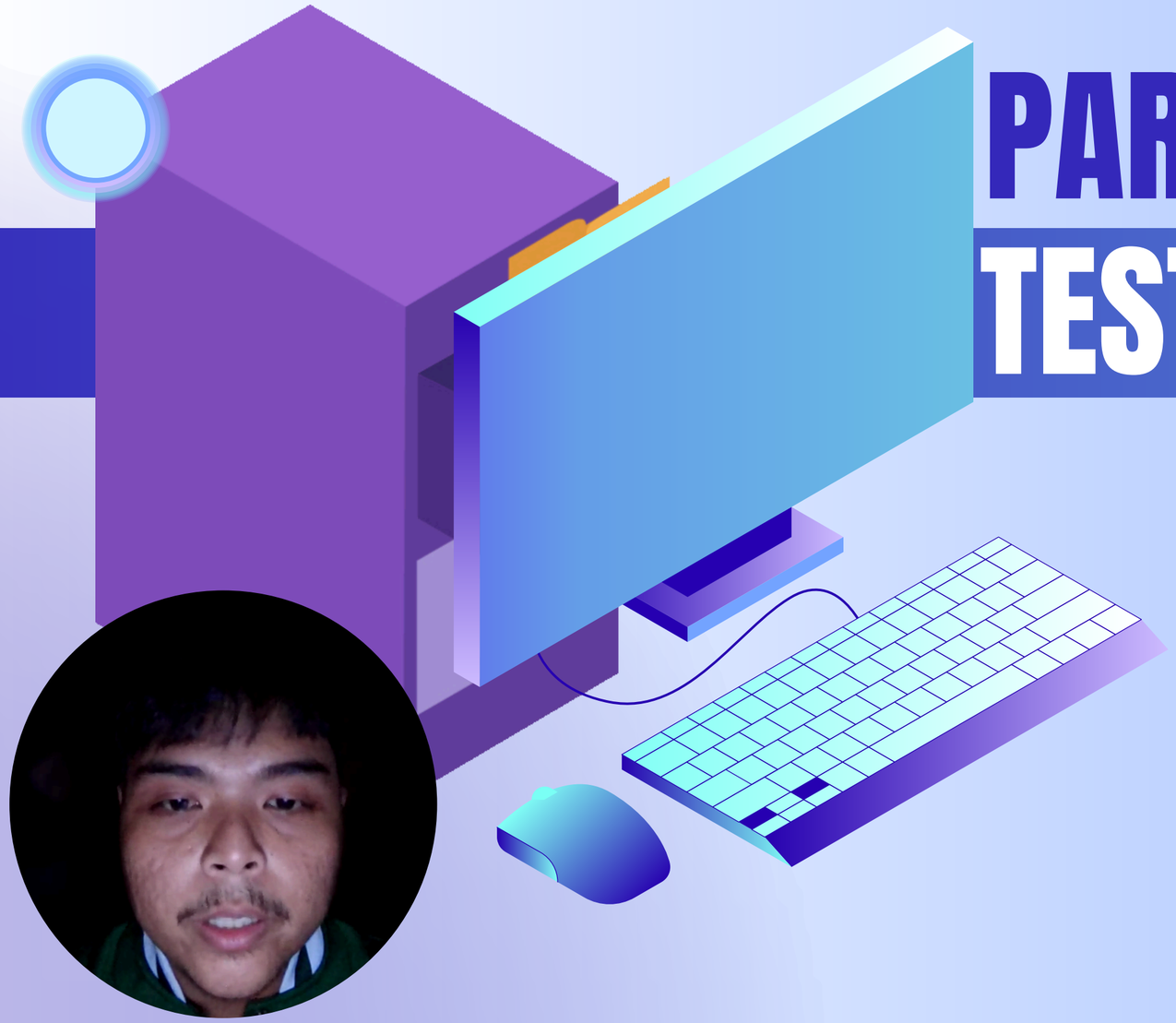


SOURCE CODE SNIPPETS

The snapshot shows that the function display the main menu of the system monitor in the terminal. It display a list of available options, allow users to select differrent monitoring features such as CPU Usage, Memory Usage, Continuous monitoring, Top 5 Processess and exit the program

Display Menu Function

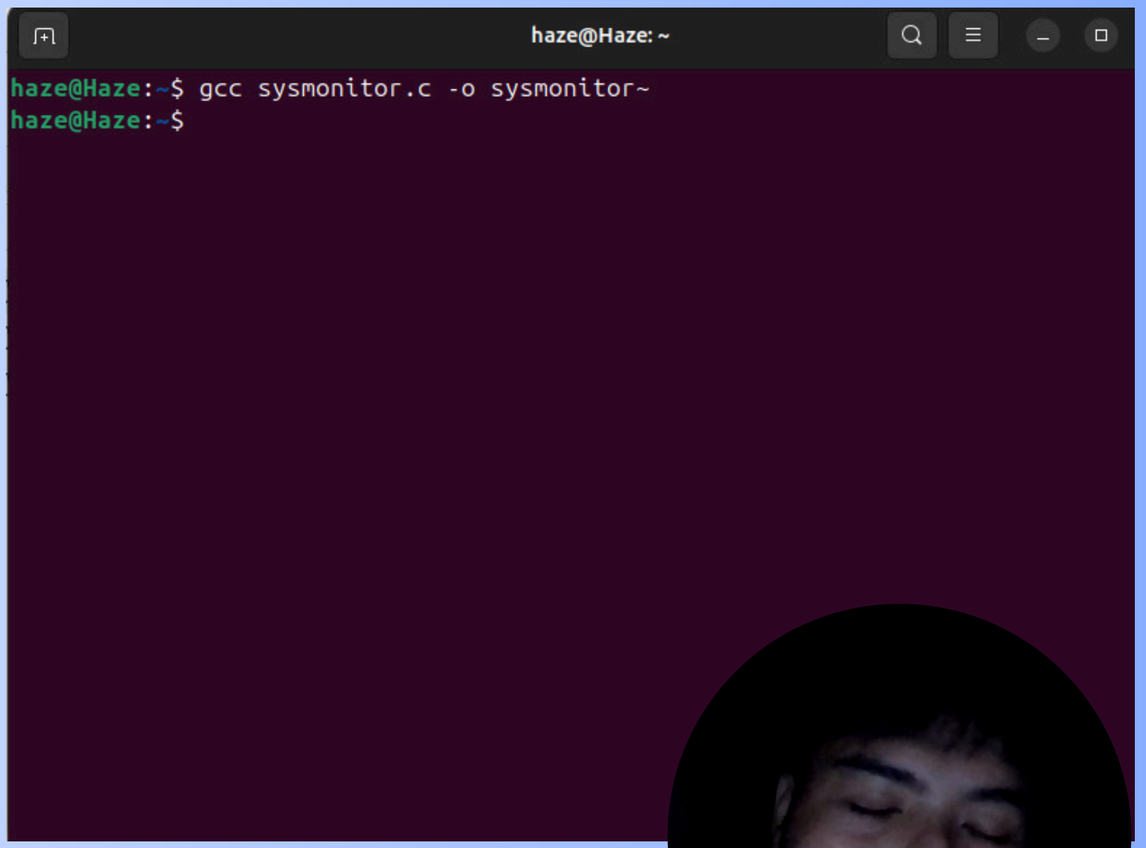
```
void displayMenu() {
    clear_screen();
    printf("========================================\n");
    printf("       SysMonitor++ - System Monitor    \n");
    printf("========================================\n");
    printf("1. CPU Usage\n");
    printf("2. Memory Usage\n");
    printf("3. Top 5 Processes (CPU)\n");
    printf("4. Continuous Monitoring\n");
    printf("5. Exit\n");
    printf("========================================\n");
}
```



PART B
TESTING AND EVALUATION

COMPILATION & BUILD TEST

- using command gcc sysmonitor -o sysmonitor~
- successfully compile without errors or warnings

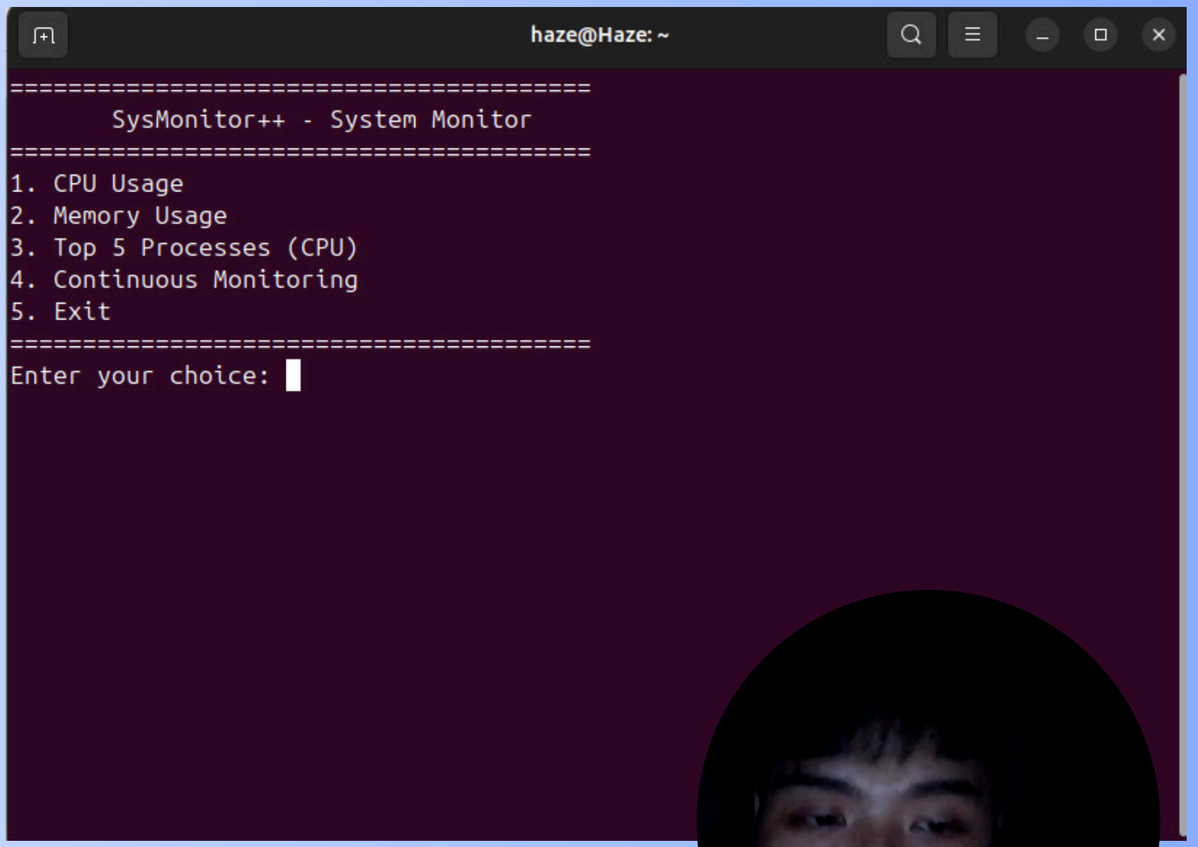


EXECUTION TEST

MENU MODE

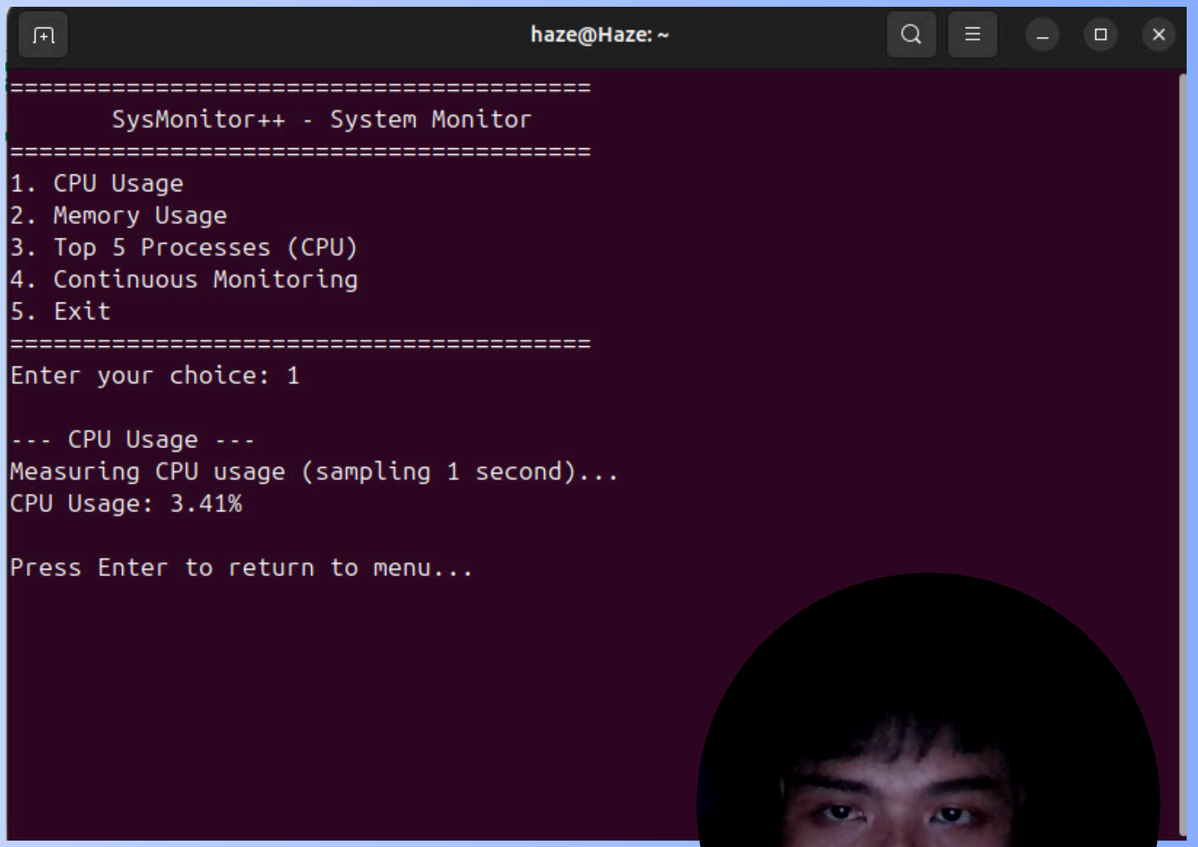
- using command ./sysmonitor to gave expected output
- Program display the main menu with numbered options

1. CPU Usage
2. Memory Usage
3. Top 5 Processes
4. Continuous Monitoring
5. Exit



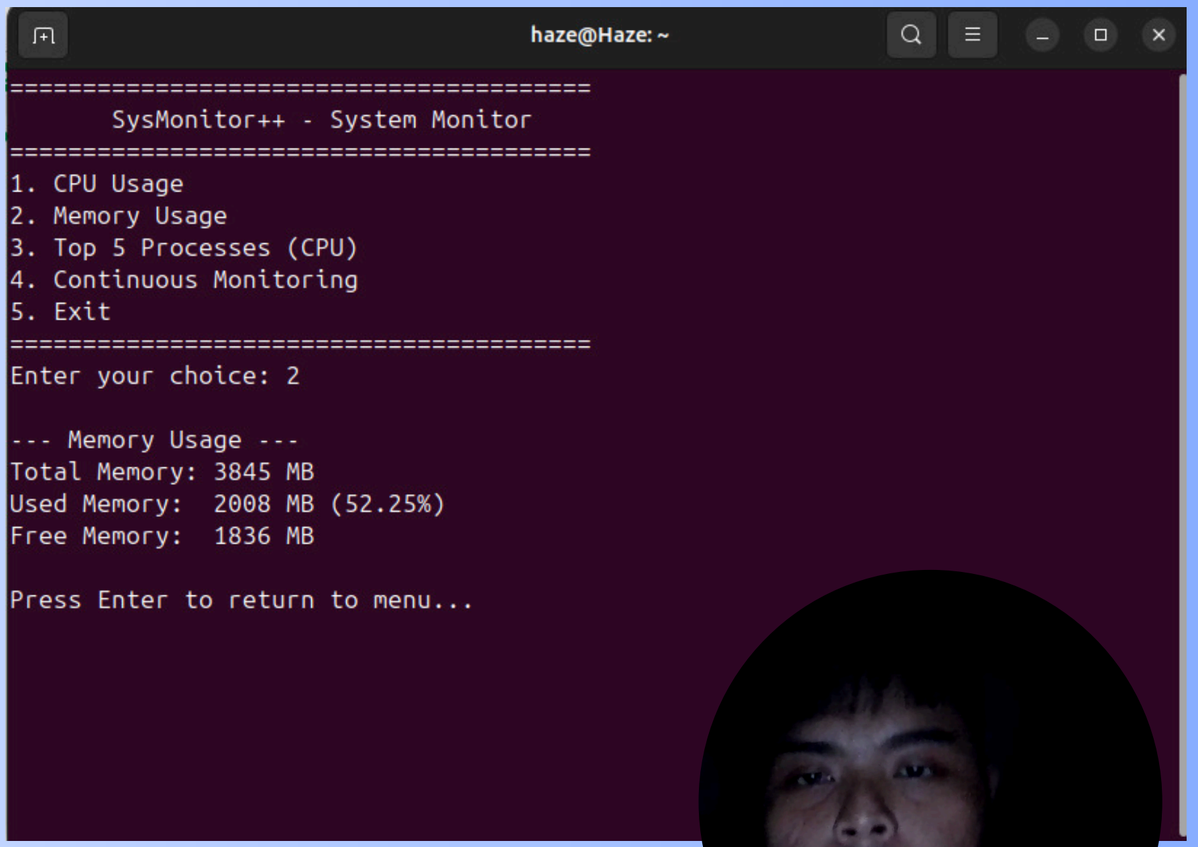
CPU USAGE

- Select option ‘1’ from main menu executes CPU Usage monitoring function
- Display CPU utilization
- Showing both user and system time percentages



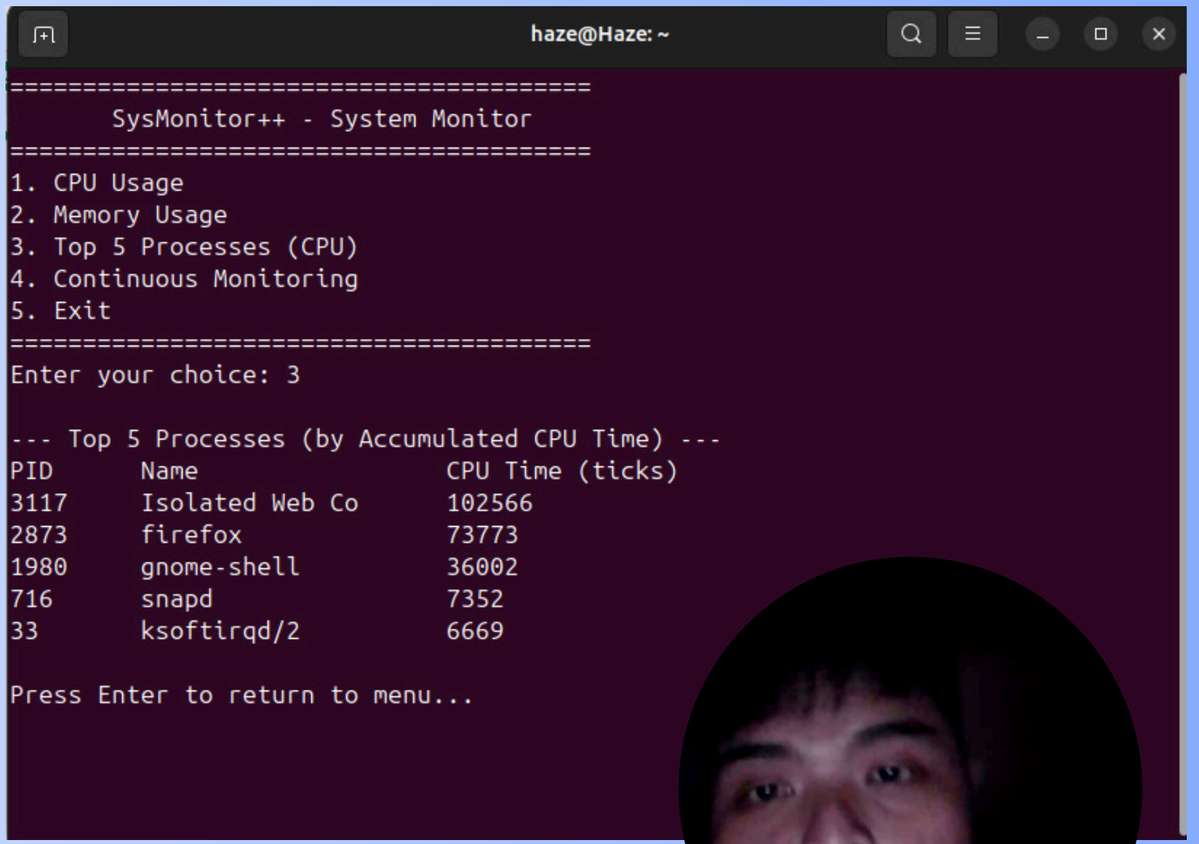
MEMORY USAGE

- Select option ‘2’ from main menu executes Memory Usage monitoring function
- Display memory statistics from system accurately
- Include total physical memory, amount of memory in use and remaining free memory



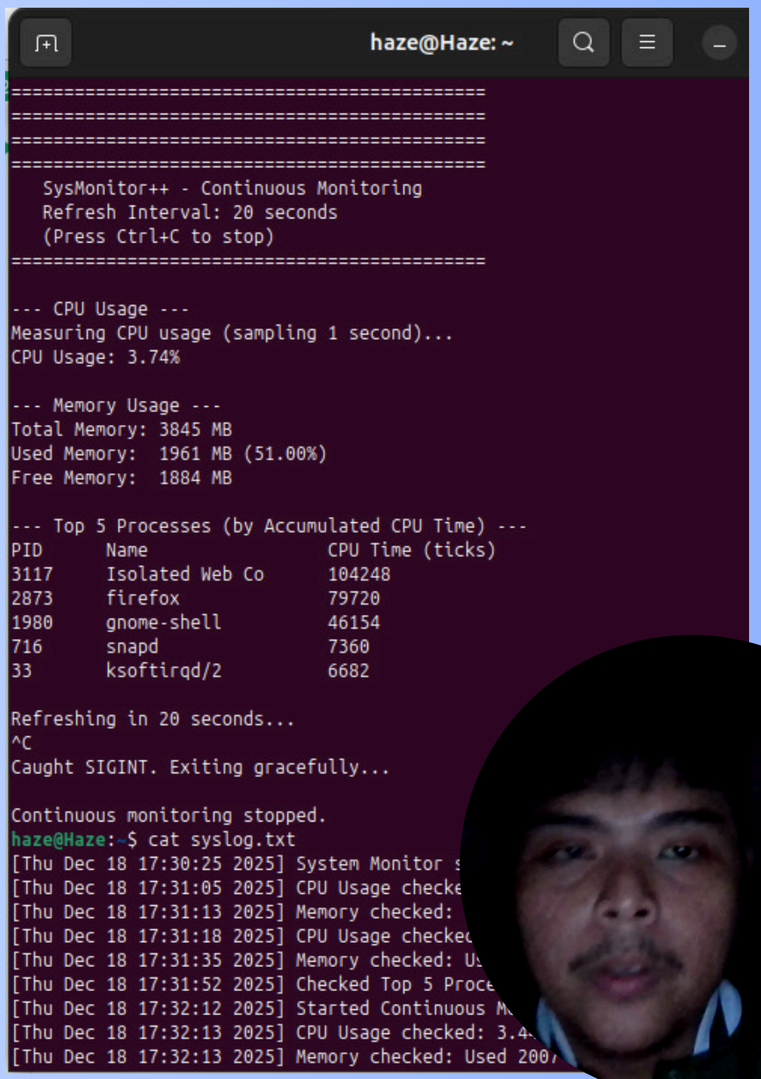
TOP 5 PROCESSES

- Select option ‘3’ from main menu executes Top 5 processes monitoring function
- Display a table or list contain PID, user running process, percentages of CPU and memory being consumed



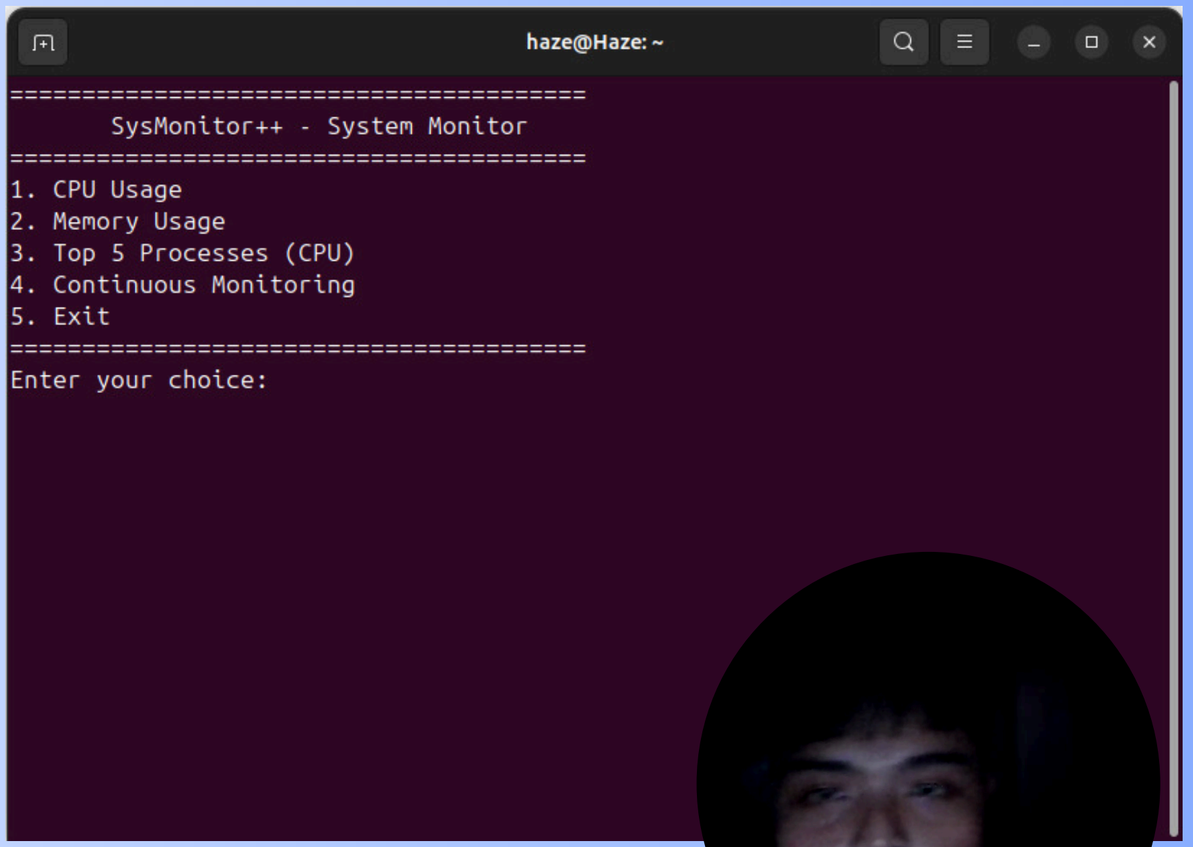
CONTINUOUS MONITORING

- Select option ‘4’ from main menu executes Continuoues monitoring features
- Repeatedly gather and display the system metric (CPU Usage, Memory Usage and top processes)
- Stop the monitoring by using CTRL + C



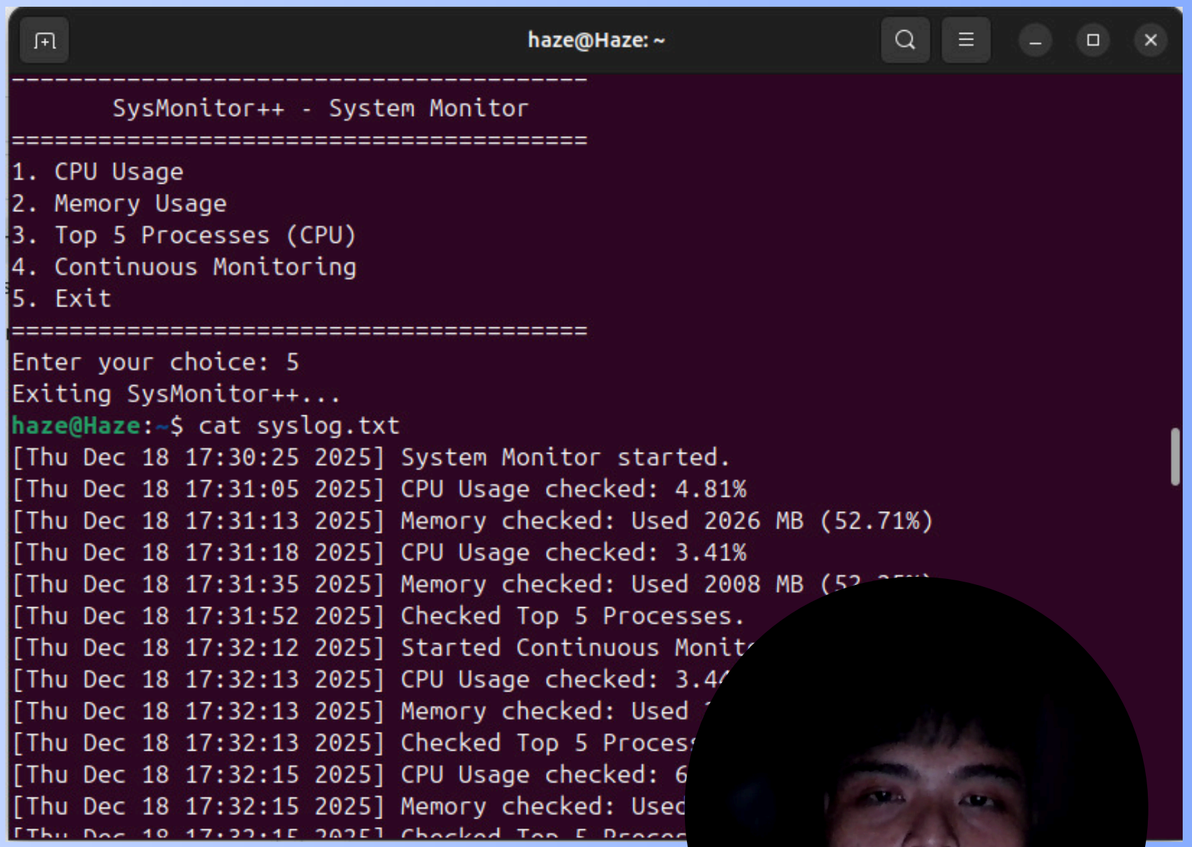
EXECUTION TEST

- Using command ./sysmonitor -m cpu
- Expected Outome: Display CPU Usage only
- Actual Output: automatically went to main menu
- Pass/Fail: Fail



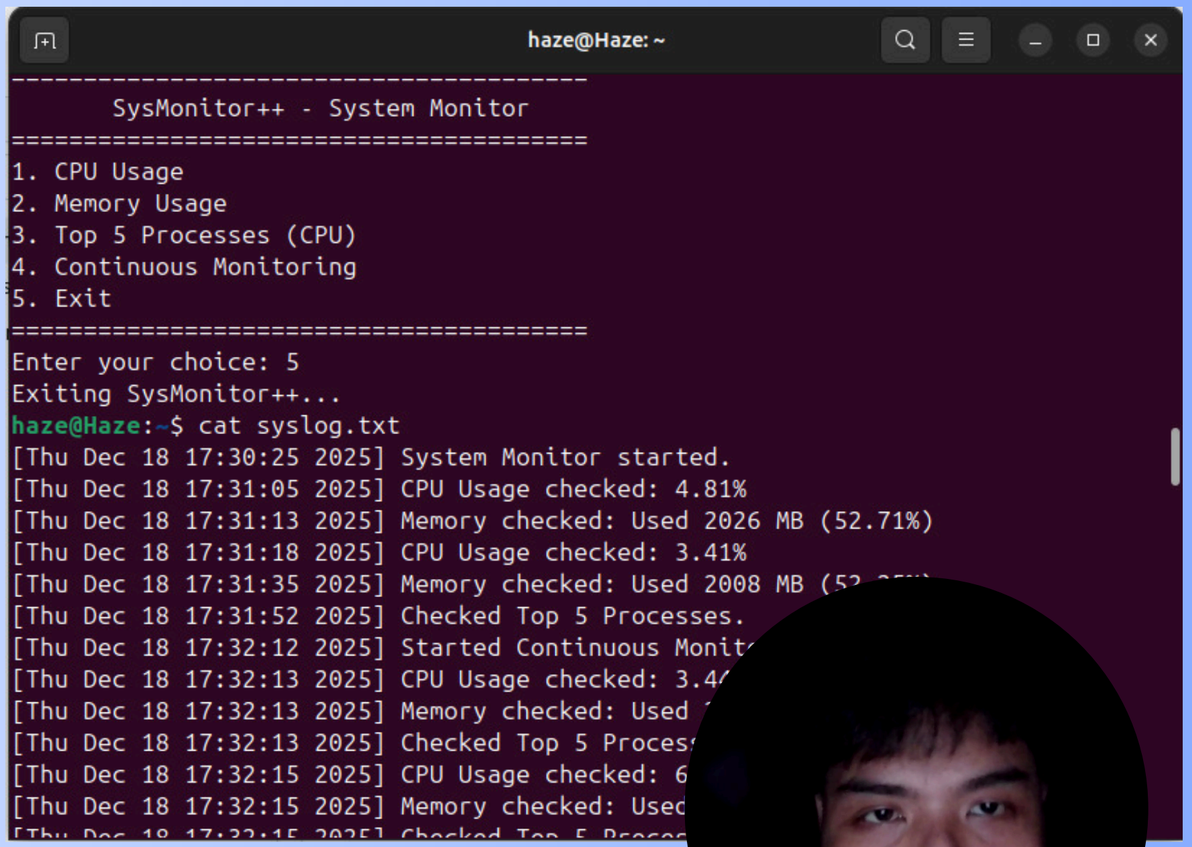
EXECUTION TEST

- Using command ./sysmonitor -m mem
- Expected Outome: Display Memory Usage only
- Actual Output: automatically went to main menu
- Pass/Fail: Fail



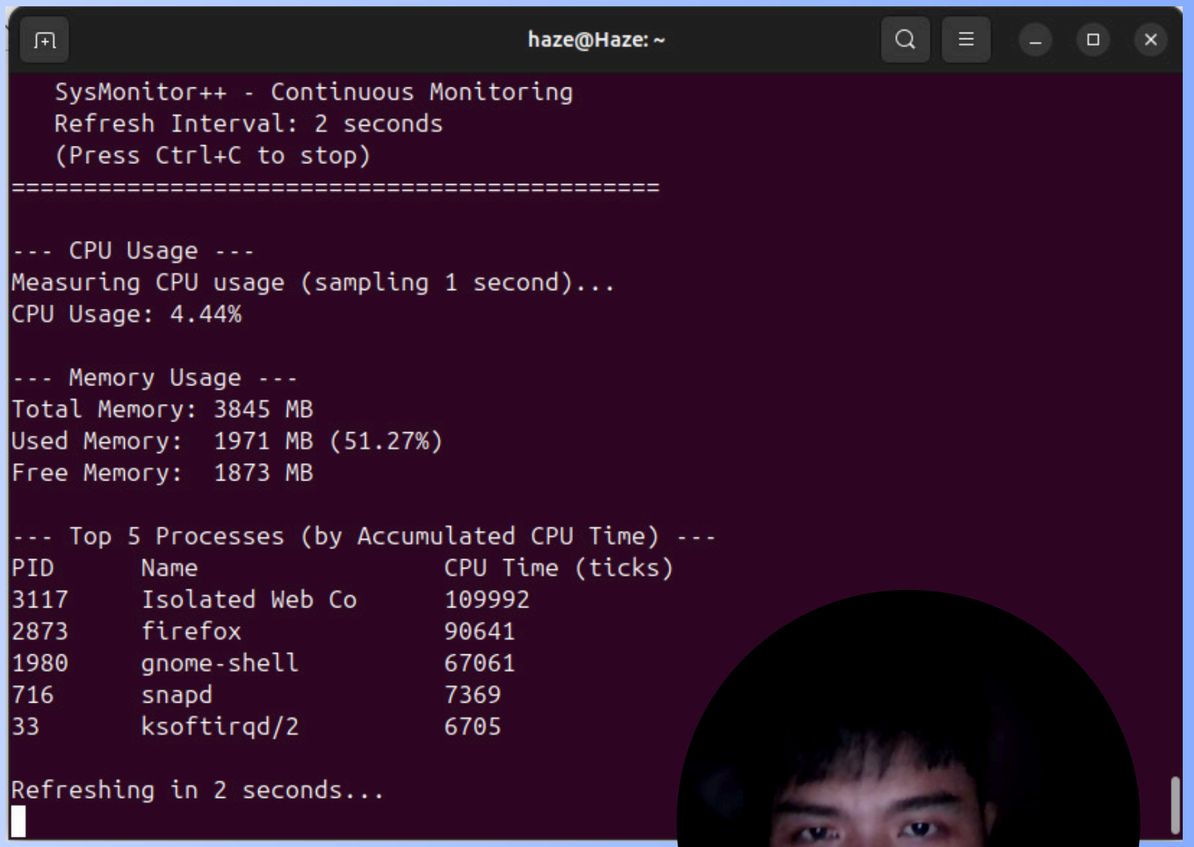
EXECUTION TEST

- Using command ./sysmonitor -m proc
- Expected Outome: List the top 5 most active processes
- Actual Output: automatically went to main menu
- Pass/Fail: Fail



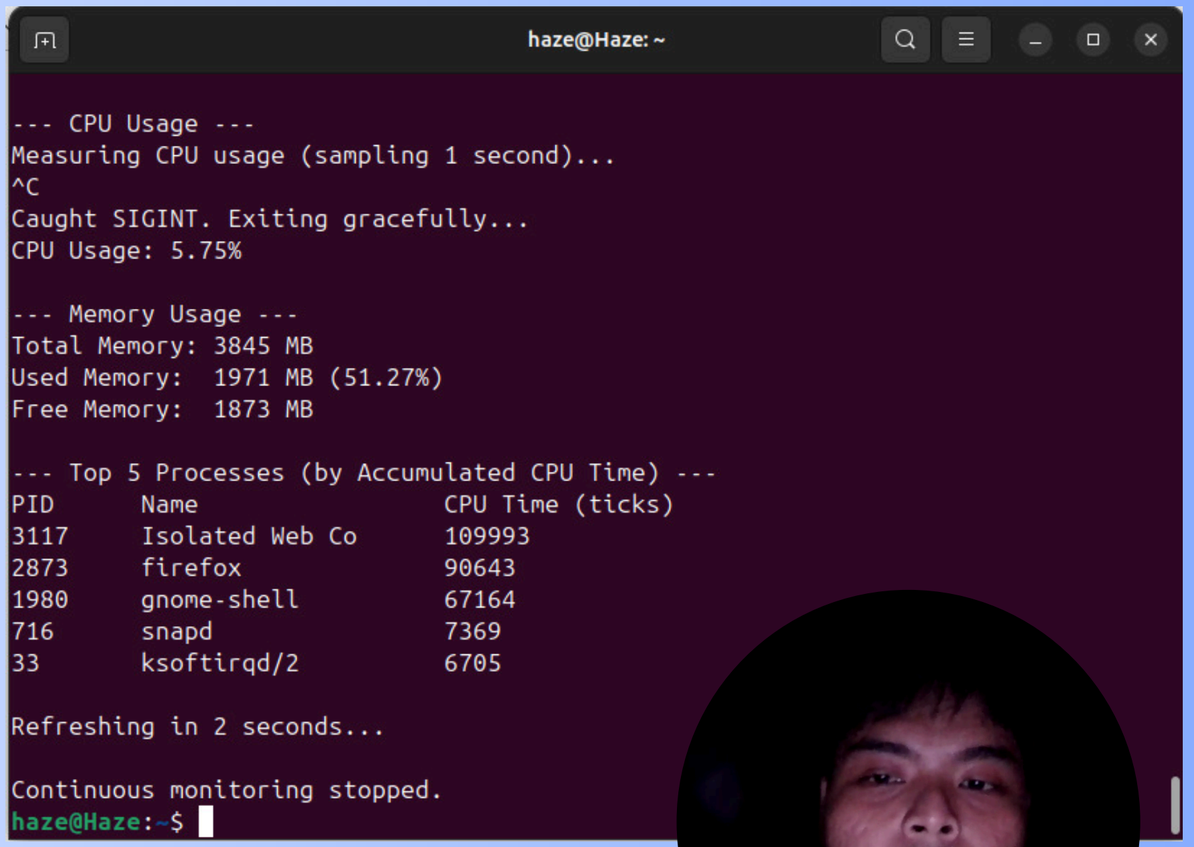
EXECUTION TEST

- Using command ./sysmonitor -c 2
- Expected Outome: Continuous monitoring at a 2 second interval
- Actual Output: Display continuous monitoring at 2 second interval
- Pass/Fail: Pass



EXECUTION TEST

- Using CTRL + C
- Expected Outome: Graceful exit the logs the action
- Actual Output: Exiting..... Saving log before terminating
- Pass/Fail: Pass



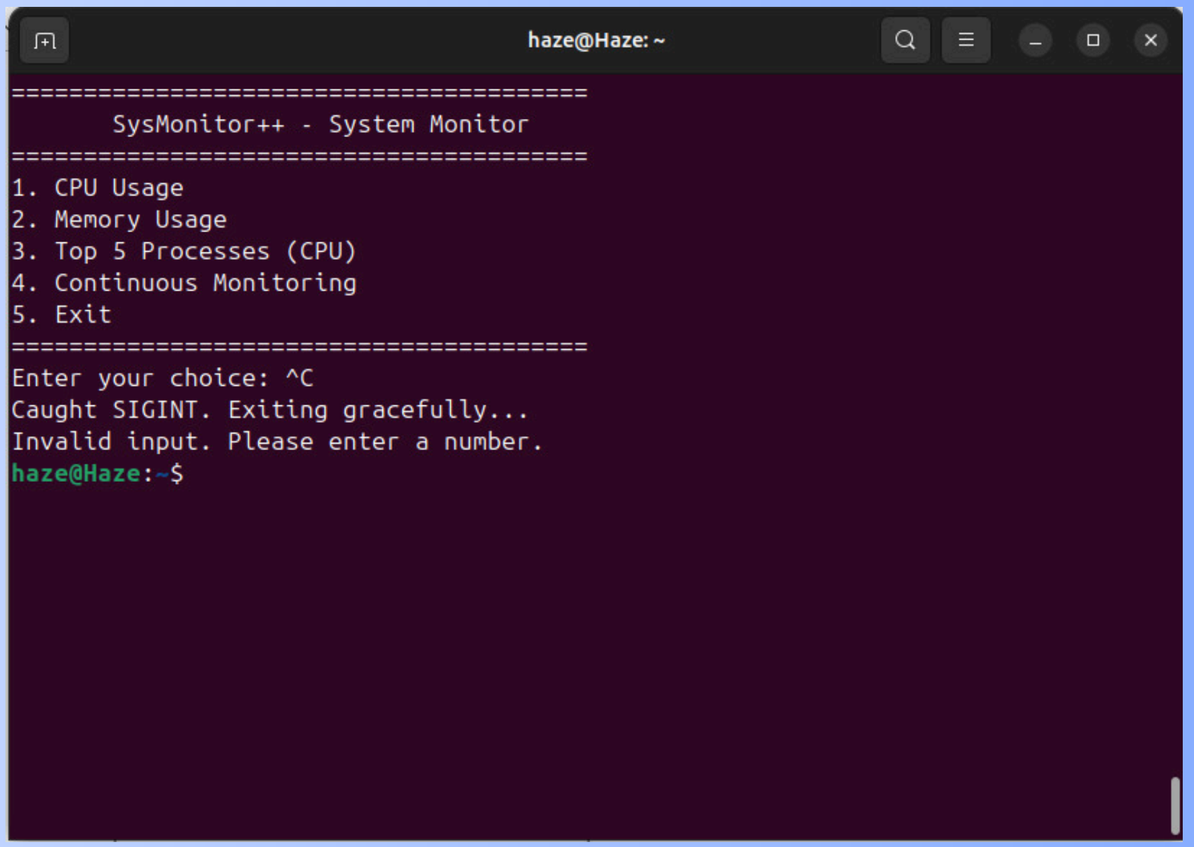
SIGNAL HANDLING TEST

haze@Haze: ~

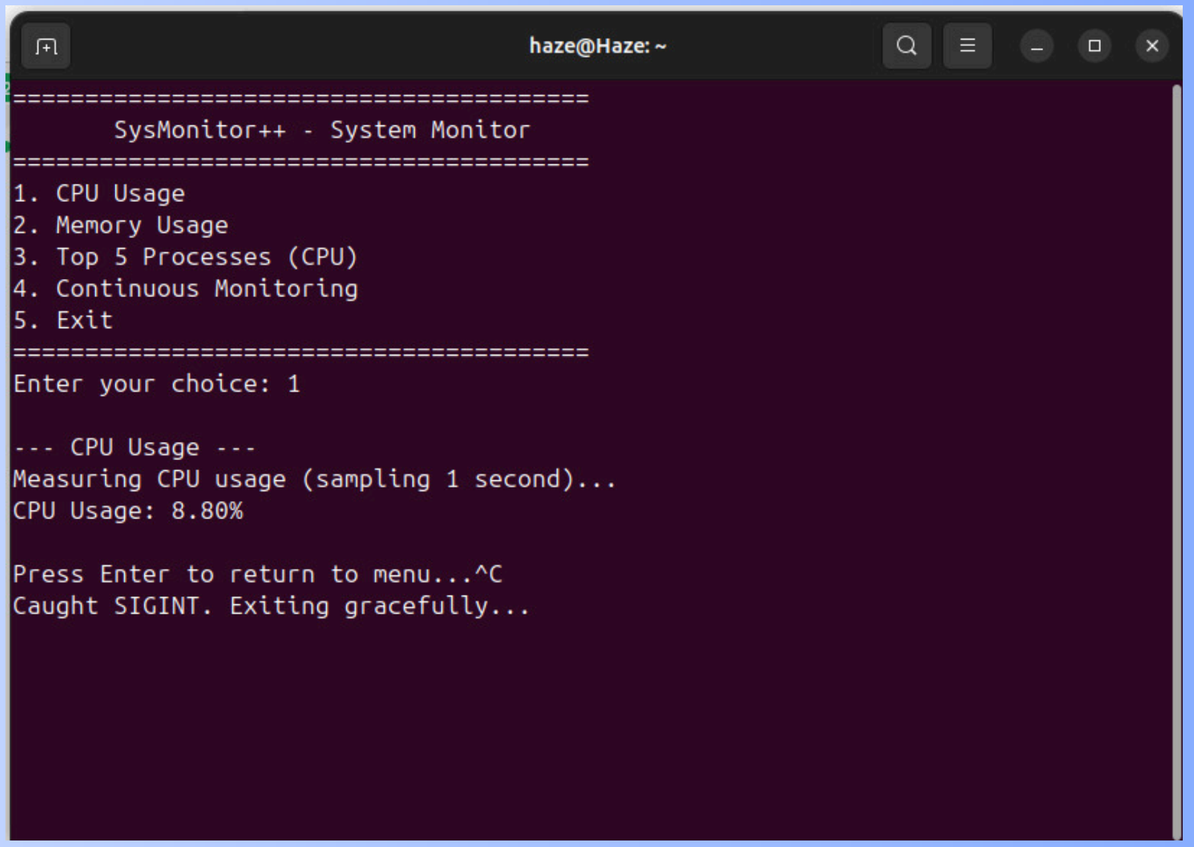
```
========================================
       SysMonitor++ - System Monitor
========================================
1. CPU Usage
2. Memory Usage
3. Top 5 Processes (CPU)
4. Continuous Monitoring
5. Exit
========================================
Enter your choice: ^C
Caught SIGINT. Exiting gracefully...
Invalid input. Please enter a number.
haze@Haze:~$
```

SIGNAL HANDLING TEST

- Using CTRL + C
- Expected Outcome (Purpose): Ctrl + C initiates a graceful exit and logs the action.
- Actual Output Observed: Pressing Ctrl + C displayed: Exiting... Saving log before terminating.
- Pass/Fail: Pass



SIGNAL HANDLING TEST



- Expected Outcome (Purpose): Ctrl + C initiates a graceful exit and logs the action, returning to the command prompt
- Actual Output Observed: Pressing Ctrl + C displayed: Exiting... Saving log before terminating.
- Pass/Fail: Pass

SIGNAL HANDLING TEST

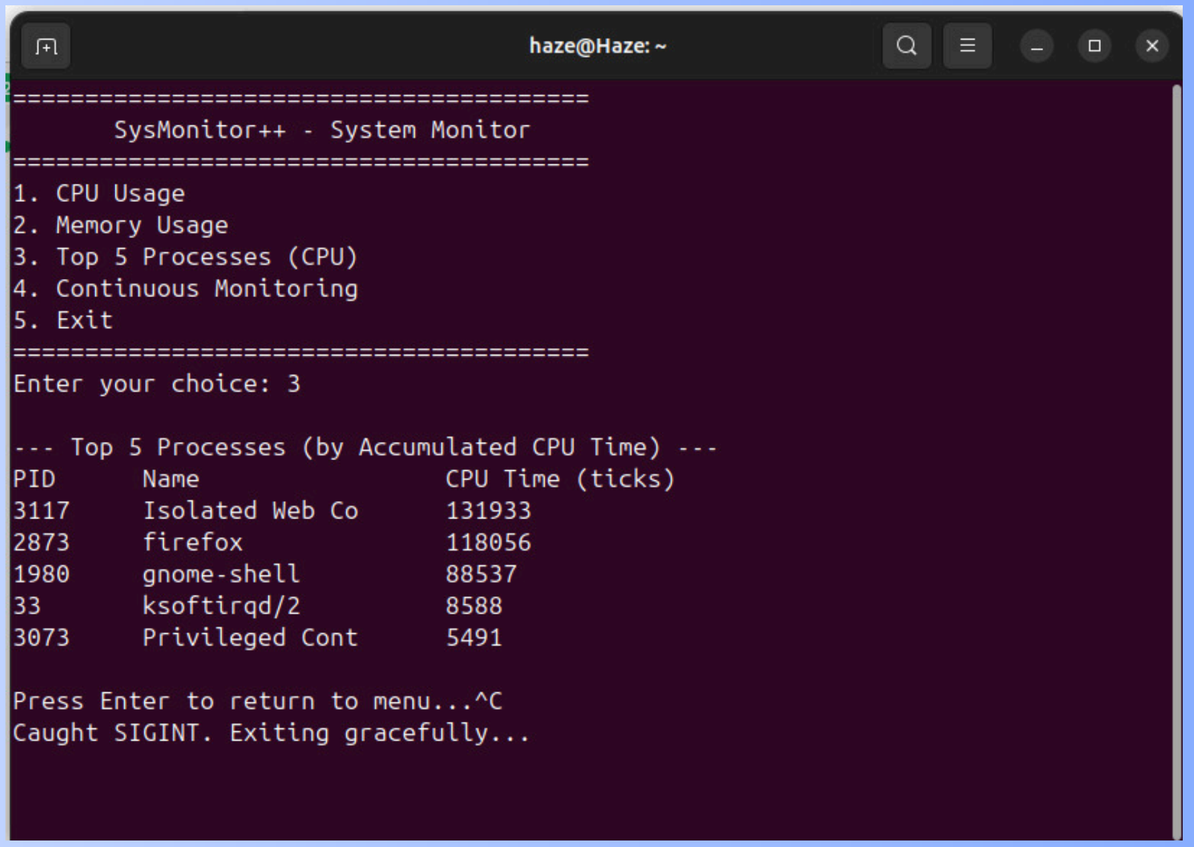
haze@Haze: ~

```
========================================
       SysMonitor++ - System Monitor
========================================
1. CPU Usage
2. Memory Usage
3. Top 5 Processes (CPU)
4. Continuous Monitoring
5. Exit
========================================
Enter your choice: 2

--- Memory Usage ---
Total Memory: 3845 MB
Used Memory:  1985 MB (51.64%)
Free Memory:  1859 MB

Press Enter to return to menu...^C
Caught SIGINT. Exiting gracefully...
```

- Expected Outcome (Purpose): Ctrl + C initiates a graceful exit and logs the action, returning to the command prompt.
- Actual Output Observed: Pressing Ctrl + C displayed: Exiting... Saving log before terminating.
- Pass/Fail: Pass

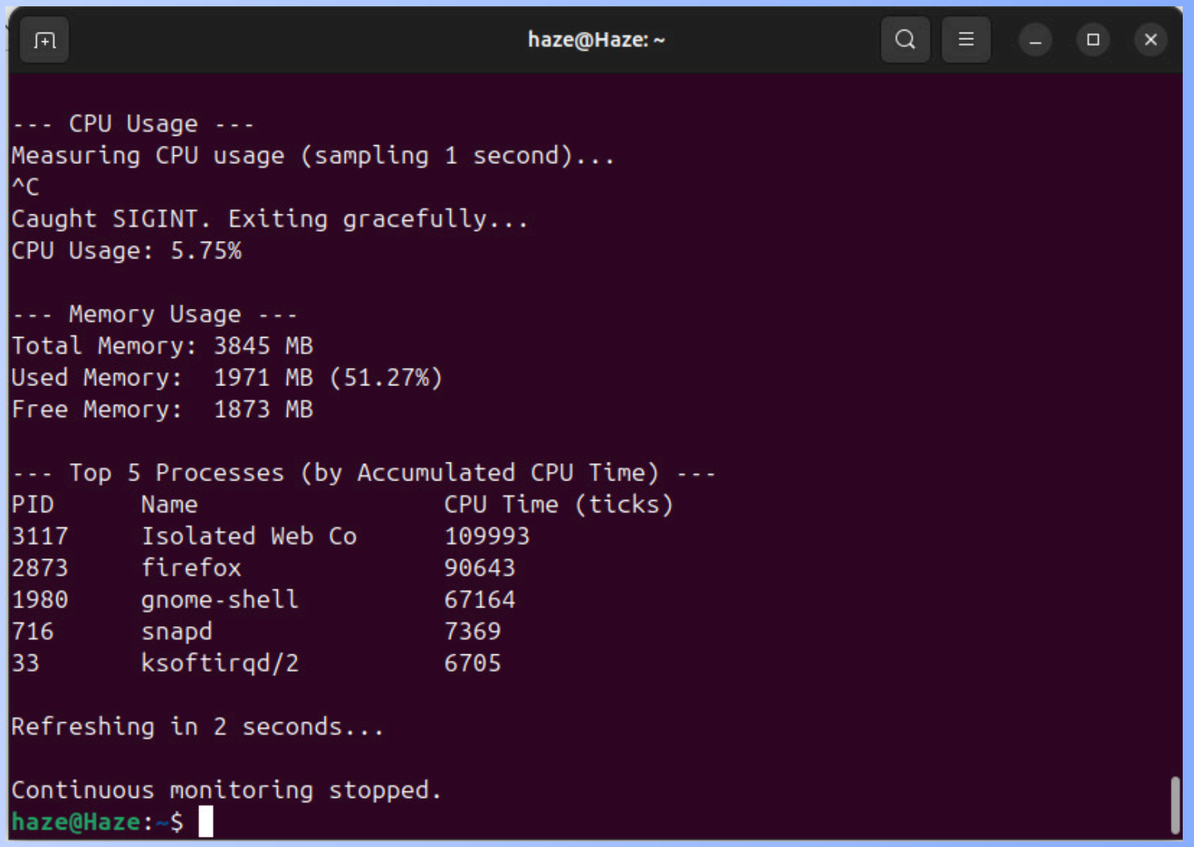
SIGNAL HANDLING TEST

- Expected Outcome (Purpose): Ctrl + C initiates a graceful exit and logs the action, returning to the command prompt.
- Actual Output Observed: Pressing Ctrl + C displayed: Exiting... Saving log before terminating.
- Pass/Fail: Pass



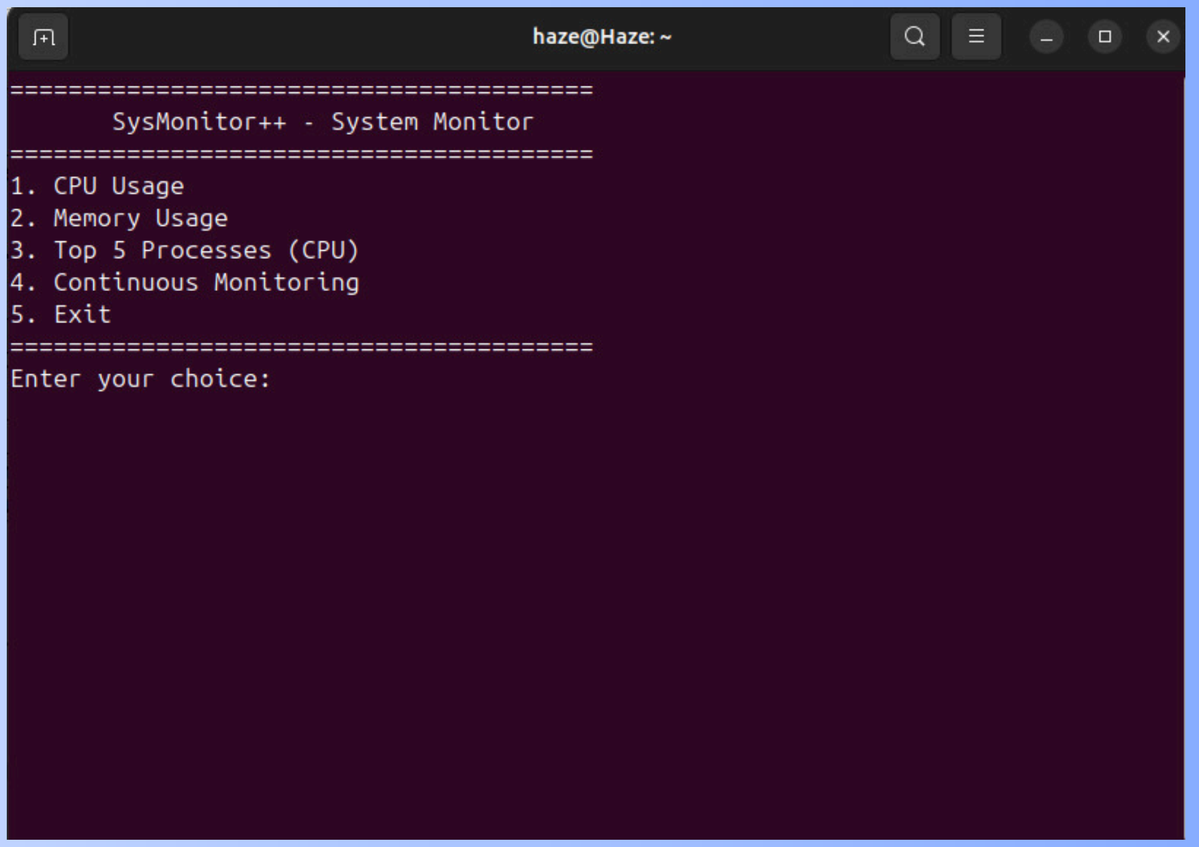
SIGNAL HANDLING TEST

- Expected Outcome (Purpose): Ctrl + C initiates a graceful exit and logs the action, returning to the command prompt.
- Actual Output Observed: Pressing Ctrl + C displayed: Exiting... Saving log before terminating. The continuous loop stopped immediately.
- Pass/Fail: Pass



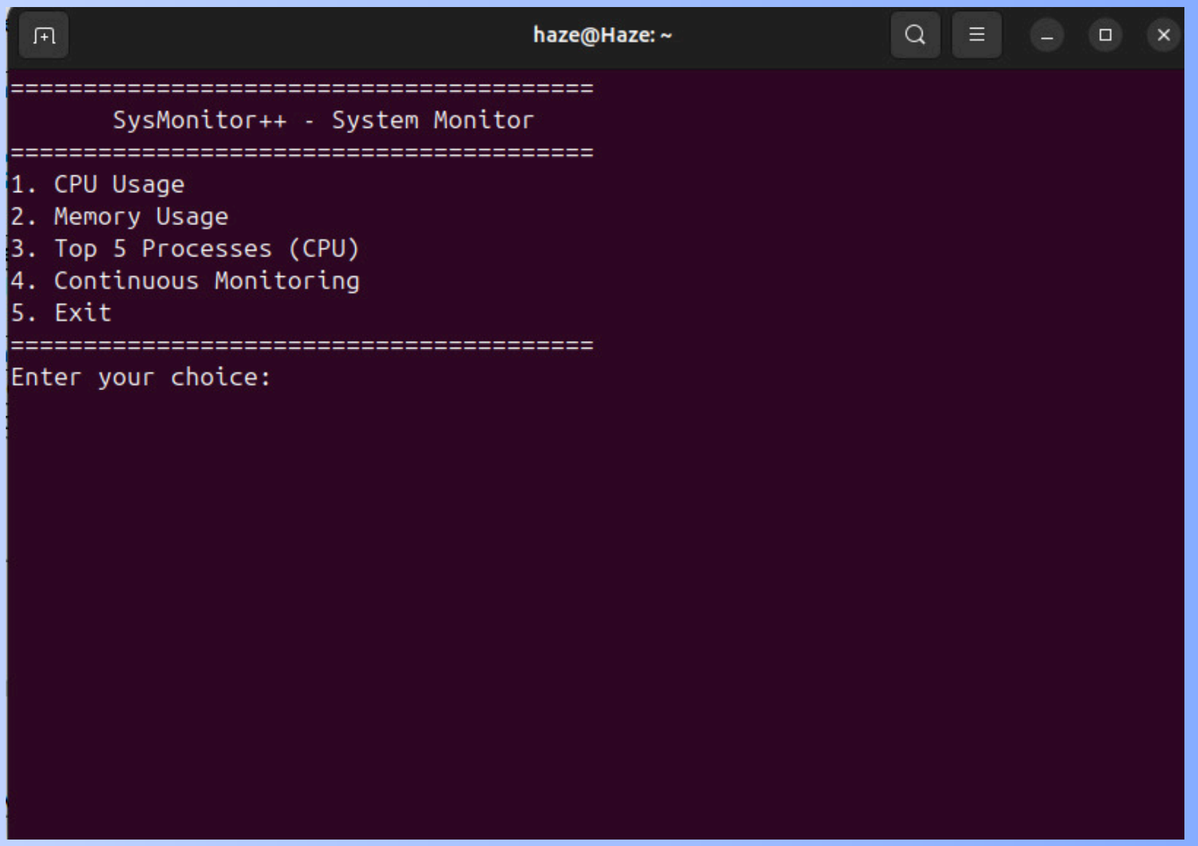
ERROR-HANDLING & EDGE-CASE TEST

- Missing Argument (./sysmonitor -m)
- Test: Running without a mode argument.
- Result: Program directly displayed main menu. (Fail)



ERROR-HANDLING & EDGE-CASE TEST

- Invalid Option (./sysmonitor -x)
- Test: Running with an invalid command-line flag.
- Result: Program directly displayed main menu. (Fail)

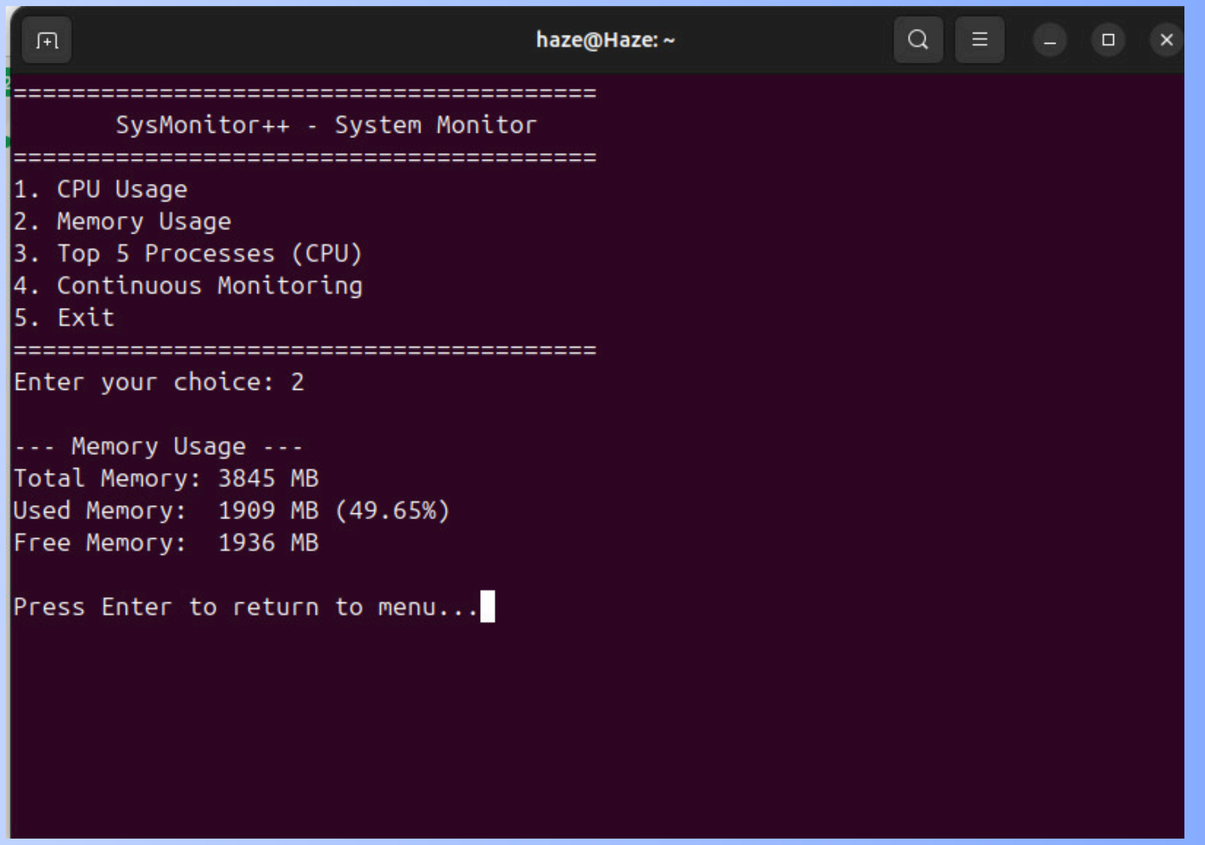


LOG VERIFICATION

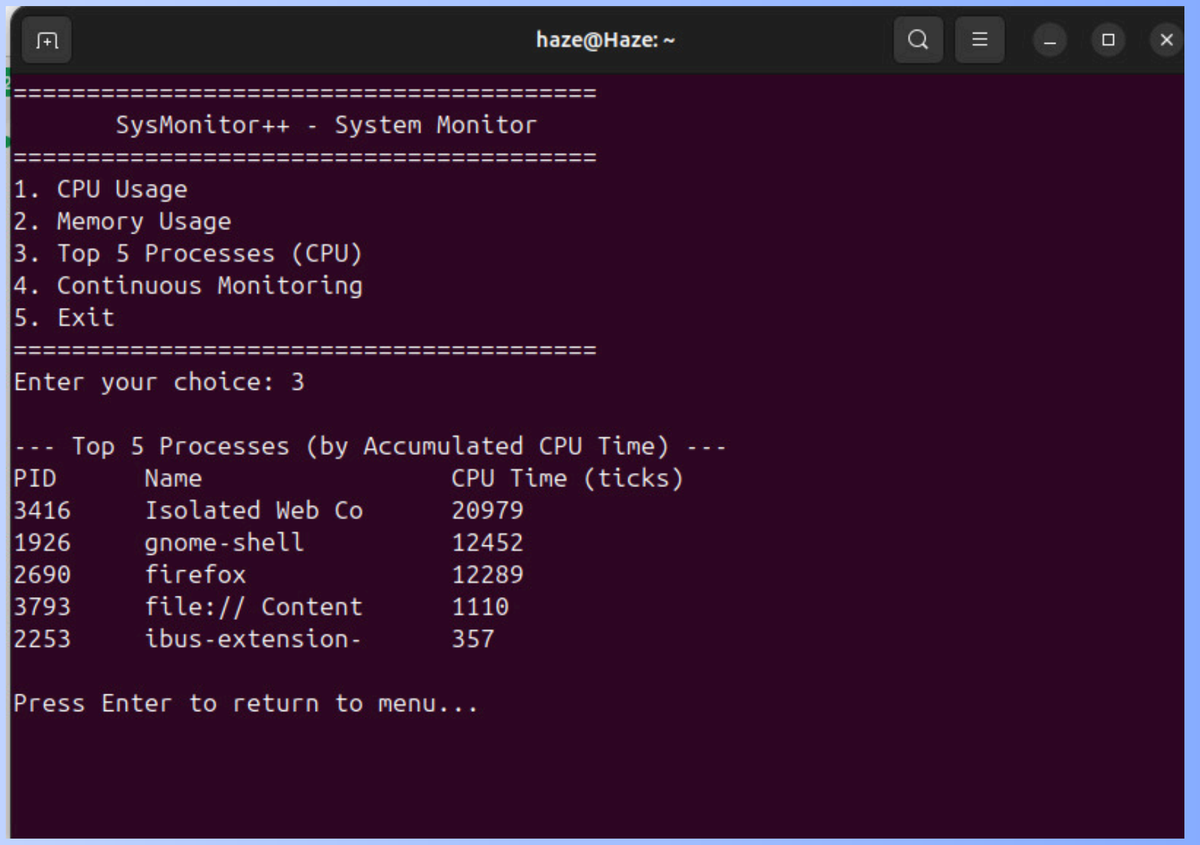
haze@Haze: ~

```
========================================
       SysMonitor++ - System Monitor
========================================
1. CPU Usage
2. Memory Usage
3. Top 5 Processes (CPU)
4. Continuous Monitoring
5. Exit
========================================
Enter your choice: 1

--- CPU Usage ---
Measuring CPU usage (sampling 1 second)...
CPU Usage: 5.65%

Press Enter to return to menu...
```

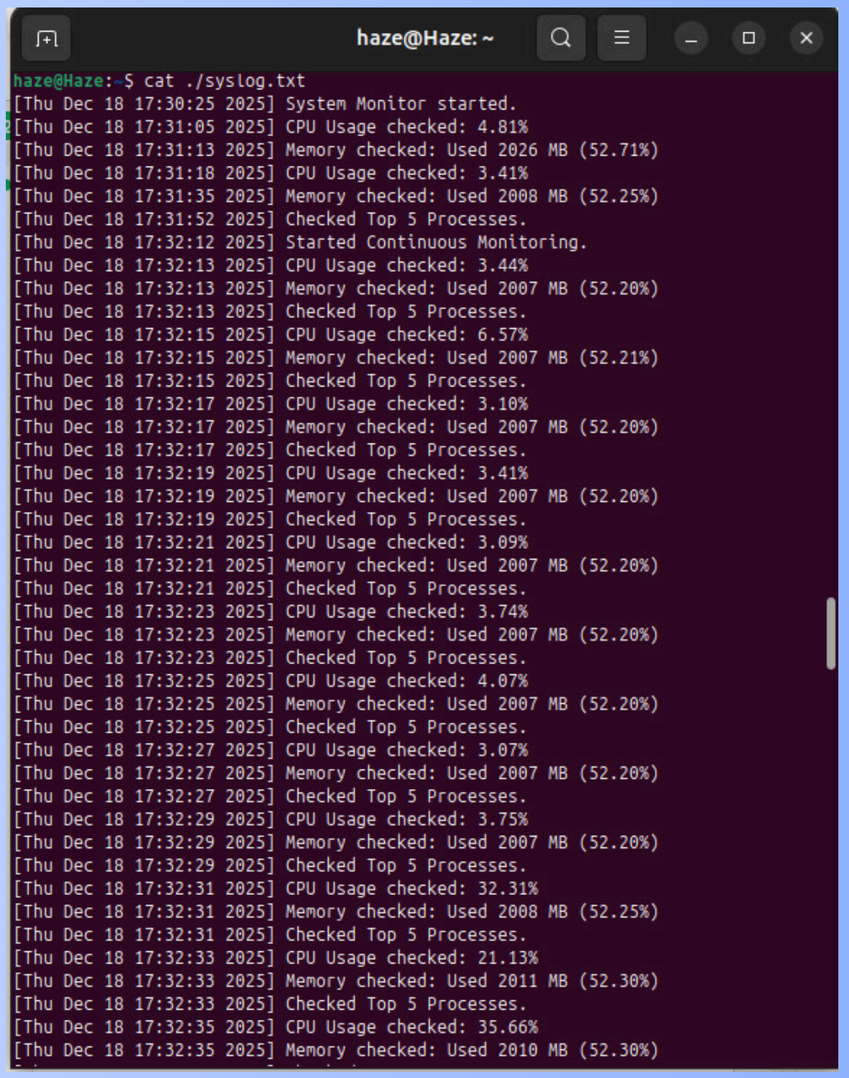
LOG VERIFICATION



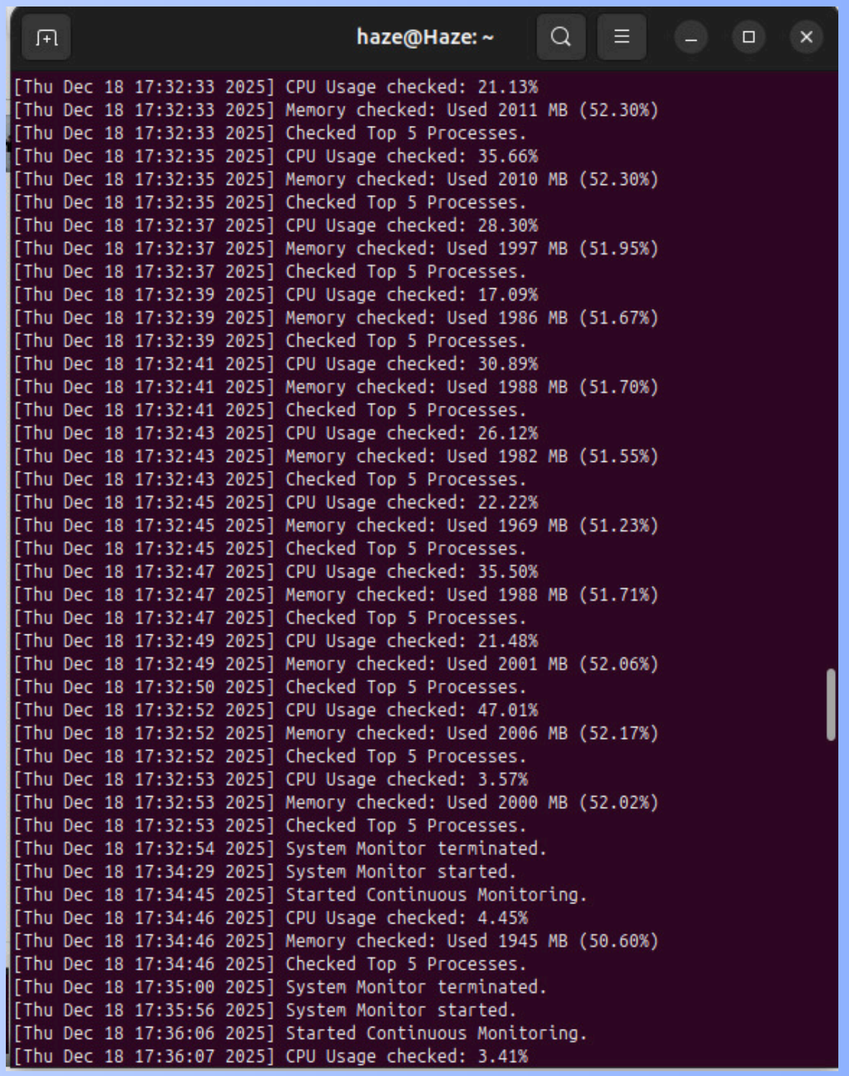
LOG VERIFICATION



LOG VERIFICATION



LOG VERIFICATION



SUMMARY FOR TESTING

The system is fully functional in Menu Mode, successfully monitoring CPU, memory, and processes with robust Signal Handling (Ctrl+C).

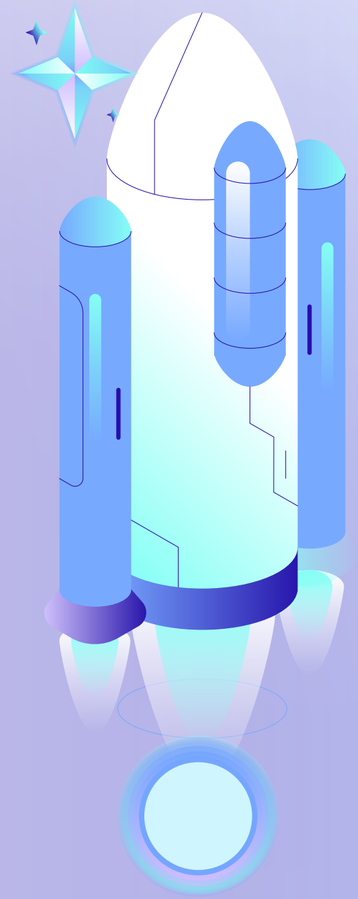
However, Command-Line Mode and Error-Handling failed; the program does not correctly parse flags (like -m), defaulting to the main menu instead. Refinement of the argument-parsing logic is required to support non-interactive execution.

GITHUB COLLABORATION

Link for repository:
https://github.com/Misopawa/TMN4133-Group12-SysMonitor-

Link for video:
https://youtu.be/BMgjvj79bZ8

CONCLUSION



In conclusion, SysMonitor++ successfully meets the management requirement for a lightweight, native monitoring utility. By leveraging direct kernel interactions via the /proc filesystem, the tool delivers accurate real-time telemetry with minimal resource overhead. While the interactive menu functions flawlessly, future iterations will focus on refining command-line argument parsing to better support automated scripting.

THANK YOU!