

# System Call Tracer

Operating Systems (IT253) Report

Submitted in partial fulfilment of the requirements for the degree of

BACHELOR OF TECHNOLOGY

In

INFORMATION TECHNOLOGY

by

Alapati Harshini (221IT006)

Bhoomika Deep Mahawar (221IT018)



DEPARTMENT OF INFORMATION TECHNOLOGY  
NATIONAL INSTITUTE OF TECHNOLOGY  
KARNATAKA SURATHKAL, MANGALORE -575025

March, 2024

## DECLARATION

I hereby *declare* that Operating Systems (IT253) Report entitled “**System Call Tracer**” which is being submitted to the National Institute of Technology Karnataka Surathkal, in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology in the Department of Information Technology, is a *bonafide report of the work carried out by me*. The material contained in this project report has not been submitted to any University or Institution for the award of any degree.

Alapati Harshini (221IT006)  
Bhoomika Deep Mahawar (221IT018)

Signature of the Students  
Department of Information Technology

Place : NITK, SURATHKAL  
Date : 25 March 2024

## **CERTIFICATE**

This is to certify that the Seminar entitled “**System Call Tracer**” has been presented by Alapati Harshini (221IT006) and Bhoomika Deep Mahawar (221IT018), students of IV semester B.Tech.(I.T), Department of Information Technology, National Institute of Technology Karnataka, Surathkal, on 25 March, 2024, during the even semester of the academic year 2022 - 2023, in partial fulfilment of the requirements for the award of the degree of Bachelor of Technology in Information Technology.

Examiner-1 Name

Signature of the Examiner-1 with Date

# **TABLE OF CONTENTS**

DECLARATION .....	
CERTIFICATE .....	
TABLE OF CONTENTS .....	1
LIST OF FIGURES .....	2
1.1 INTRODUCTION .....	3
1.2 PROBLEM STATEMENT .....	3
2 OBJECTIVES .....	4
3.METHODOLOGY .....	5
4 SYSTEM DESIGN FLOW CHART .....	11
5 IMPLEMENTATION .....	13
6 RESULTS .....	17
7 FUTURE WORK .....	24

## **LIST OF FIGURES**

1. Fig 4.1 System Design Flow Chart .....	11
2. Fig 5.1.1 System Call Interception .....	13
3. Fig 5.2.1 Logging Analysis .....	14
4. Fig 5.3.1 Multi process .....	15
5. Fig 5.4.1 Profiling and analysis.....	16
6. Fig 6.1 Target Code .....	17
7. Fig 6.2 Tracer .....	18
8. Fig 6.3 Logging .....	19
9. Fig 6.4 Filtering .....	19
10. Fig 6.5, 6.6 Multi Threads Support .....	20
11. Fig: 6.7 System Call Profiling .....	21
12 Fig 6.5.1: Total System Call Number .....	22
13. Fig 6.5.2: Avg System Call Number .....	23

# **Chapter 1. INTRODUCTION:**

## **1.1 Introduction**

System call tracing and analysis play a critical role in understanding the behavior and performance of software systems. By intercepting system calls made by processes and threads, developers and system administrators can gain insights into how programs interact with the underlying operating system. This report explores the implementation of a system call tracer with a focus on four key aspects: System Call Interception, Logging and Filtering, Multiple Processes/Threads Support, and Profiling and Analysis

Using techniques such as `ptrace` for interception, comprehensive logging with filtering capabilities, support for tracing multiple concurrent processes/threads, and profiling mechanisms to analyze system call frequency, duration, and other metrics, this report aims to provide a comprehensive overview of building an effective system call tracing tool. Through practical examples and discussions, we will delve into the technical details, challenges, and benefits of each aspect, ultimately showcasing the importance of system call tracing in understanding and optimizing software systems.

## **1.2 Problem Statement**

The project aims to develop a system call tracer that can intercept, log, and analyze system calls made by processes in a Linux environment. This tracer will provide insights into process behavior, aiding in debugging, performance analysis, and security auditing.

## **Chapter 2. OBJECTIVES:**

Understanding System Call Tracing and Analysis: Gain insight into the fundamentals of system call tracing and its significance in software development and system administration.

Exploring System Call Interception Techniques: Explore techniques such as ptrace for intercepting system calls and controlling the execution of programs.

Implementing Comprehensive Logging and Filtering: Develop proficiency in logging system call information and implementing sophisticated filtering mechanisms for targeted analysis.

Supporting Concurrent Processes and Threads: Investigate methods to support tracing of multiple concurrent processes and threads, ensuring accurate and coordinated tracing across the system.

Profiling System Call Activity: Master profiling techniques to analyze system call frequency, duration, and other metrics, facilitating performance optimization and troubleshooting.

Utilizing Visualization Tools: Utilize visualization tools and techniques to interpret and present complex system call tracing data effectively.

Evaluating Real-World Applications: Evaluate the benefits and challenges of system call tracing and analysis in real-world scenarios, including software debugging and security auditing

## **Chapter 3. METHODOLOGY:**

### **3.1 Methodology Overview**

- The development of our System Call Tracer involved a systematic approach aimed at achieving the objectives outlined in the project scope. Our methodology focused on implementing key functionalities such as system call interception, logging, filtering, and analysis, while ensuring robustness, efficiency, and usability of the tracer. This section provides an overview of the methods and techniques employed throughout the development process.
- We began by researching existing techniques for system call interception and tracing, with a particular emphasis on understanding the `ptrace` mechanism available in the Linux operating system. Leveraging this knowledge, we devised a strategy to implement system call interception using `ptrace`, allowing our tracer to observe and control the execution of target processes at the system call level.
- Building upon the foundation of system call interception, we proceeded to implement logging and filtering functionalities to capture and process system call information effectively. We designed a flexible logging framework capable of recording system call numbers, arguments, return values, and errors, while also incorporating advanced filtering mechanisms to selectively log or process system calls based on various criteria such as process ID, thread ID, and system call type.
- Furthermore, we addressed the challenge of tracing multiple processes and threads simultaneously by implementing synchronization and communication mechanisms between the tracer and traced processes/threads. This ensured accurate tracing and minimized interference with the normal operation of the target processes/threads, enhancing the reliability and scalability of our tracer.
- To facilitate in-depth analysis of system call behavior, we integrated profiling capabilities



into our tracer, allowing us to track system call frequency, duration, and other metrics of interest. We developed tools for analyzing and visualizing profiling data, providing users with valuable insights into the performance and behavior of traced processes.

- Additionally, we implemented real-time monitoring features to enable immediate feedback on system behavior during tracing sessions. Users can interact with the tracer in real-time, pausing, resuming, or filtering system calls based on specific criteria, enhancing the flexibility and usability of the tool.
- Finally, we ensured cross-platform compatibility by modifying our tracer to support tracing on multiple operating systems, thereby expanding its potential user base and applicability.
- Overall, our methodology encompassed a comprehensive and iterative approach to developing a robust and feature-rich System Call Tracer, combining theoretical knowledge with practical implementation to achieve our project objectives effectively.

### 3.2 System Call Interception

To implement system call interception in our System Call Tracer, we followed a systematic approach:

- **Understanding ptrace:** We studied the ptrace system call and its usage in tracing system calls made by a target process. This involved understanding how ptrace can be used to attach to a process, intercept system calls, and retrieve information about them.
- **Forking and Executing the Target Process:** We used the fork system call to create a child process. In the child process, we called `ptrace(PTRACE_TRACEME, ...)` to enable tracing on itself. We then used `execl("./target", "target", NULL)` to replace the child process with the target program.
- **Parent Process (Tracer):** In the parent process, we waited for the child process to start using `waitpid(pid, &status, 0)`. Once the child process started, we entered a loop to trace its

system calls.

- **Tracing System Calls:** Within the tracing loop, we used `ptrace(PTRACE_SYSCALL, ...)` to trace each system call made by the child process. We waited for the child process to stop at a system call using `waitpid(pid, &status, 0)`.
- **Retrieving System Call Information:** When the child process stopped at a system call, we used `ptrace(PTRACE_GETREGS, ...)` to retrieve information about the system call, such as the system call number and arguments.
- **Handling Process Exit:** We checked for process exit using `WIFEXITED(status)` and exited the tracing loop if the child process exited.

This methodology provided a structured approach to implementing system call interception in our System Call Tracer. It helped us understand the key concepts and steps involved in tracing system calls using `ptrace`.

### 3.3 Logging and Filtering

In this section, we delineate the methodology employed for logging and filtering system calls in our System Call Tracer. The process encompasses the following steps:

- **Logging System Call Information:**
  - Upon intercepting a system call, the `log_syscall` function is invoked to record pertinent details about the system call.
  - The function logs information such as process/thread ID, timestamp, system call number, arguments, return value, and error code.
  - Timestamps are generated to denote when each system call occurred, facilitating chronological analysis of system call activity.
- **Opening and Writing to Log File:**
  - The `log_syscall` function opens the `syscall_log.txt` file in append mode to ensure that system call information is appended to the existing log.

- If the file opening operation fails, an appropriate error message is displayed, and the program terminates.
- Filtering System Calls:
  - To streamline the log output and focus on specific system calls of interest, a filtering mechanism is incorporated within the `log_syscall` function.
  - System calls are filtered based on their numbers using conditional statements. Only system calls of interest, such as `SYS_open`, `SYS_read`, and `SYS_write`, are logged, while others are skipped.
  - This filtering mechanism enhances the relevance of the logged information and facilitates targeted analysis of system call behavior.
- Timestamp Generation:
  - Before logging each system call, the `log_syscall` function retrieves the current timestamp using the `time` function and formats it for inclusion in the log entry.
  - Timestamps provide valuable temporal context, aiding in the analysis of system call patterns and sequences.
- Closing the Log File:
  - Upon completing the logging operation for a system call, the `log_syscall` function closes the `syscall_log.txt` file to ensure proper handling of file resources and data integrity.

This methodology outlines the systematic approach adopted to log and filter system calls in our System Call Tracer. It ensures the comprehensive recording of pertinent system call information while enabling targeted analysis through selective filtering.

### 3.4 Multiple Processes/Threads Support

Implementing support for tracing multiple processes/threads simultaneously in our System Call Tracer required careful design and coordination. Below is an overview of how we achieved this functionality:

- **Process Creation:**
  - We utilized the fork system call to create multiple child processes. Each child process represents a separate entity that executes its own logic and makes system calls.
- **Thread Creation for Tracing:**
  - After creating the child processes, we spawned a dedicated thread for each child process to perform system call tracing.
  - We employed the POSIX threads (pthread) library to create and manage threads efficiently.
- **Thread Functionality:**
  - Each tracing thread executes the trace\_child function, which simulates tracing activity for a specific child process.
  - The trace\_child function encapsulates the tracing logic, including the duration of the tracing period and the actions performed during tracing.
- **Synchronization and Communication:**
  - Synchronization mechanisms, such as signals, were employed to coordinate activities between the main process, tracing threads, and child processes.
  - We registered a signal handler for SIGINT (Ctrl+C) to gracefully handle termination requests and ensure proper cleanup.
  - Thread creation and joining operations were carefully synchronized to ensure that tracing threads are created before child processes execute and that all tracing threads complete their tasks before the main process exits.
- **Handling Child Processes:**
  - After creating the child processes and tracing threads, the main process waits for all tracing threads to finish using the pthread\_join function.
  - Additionally, the main process waits for each child process to exit using the waitpid function to ensure proper termination and collect exit status information.
- **Graceful Termination:**
  - To facilitate graceful termination of the tracer, we incorporated mechanisms to handle termination signals (SIGINT) and exit the program cleanly.

This approach enabled our System Call Tracer to effectively trace multiple processes/threads simultaneously, providing comprehensive insights into system call activity across multiple execution contexts. Through careful synchronization and communication, we ensured seamless coordination between the tracer and traced entities, enhancing the robustness and reliability of our tracing solution.

### 3.5 Profiling and Analysis Methodology

Profiling and analysis in our System Call Tracer are essential for understanding the behavior and performance of system calls. Here's how we implemented profiling capabilities to track system call frequency, duration, and other metrics, and how we analyzed and visualized the profiling data:

- **Profiling Data Collection:**
  - We utilized the `ptrace` system call to trace the execution of the target process and record information about each system call, including the system call number and the start and end times of the call.
  - For each system call, we incremented a counter in the `call_count` array and added the duration of the call to the `call_duration` array.
- **Data Storage:**
  - We stored the profiling data in arrays (`call_count` and `call_duration`) indexed by the system call number.
  - The `call_count` array stored the number of times each system call was made, while the `call_duration` array stored the total duration of each system call.
- **Profiling Data Analysis:**
  - After collecting the profiling data, we analyzed it to gain insights into system call behavior.
  - We calculated the average duration of each system call by dividing the total duration by the number of calls.
  - This analysis helped us identify which system calls were used most frequently and which ones took the longest time to execute.
- **Profiling Data Visualization:**
  - To visualize the profiling data, we used a Python script with the `matplotlib` library to create bar charts.
  - We created two bar charts: one showing the total duration of each system call and another showing the average duration of each system call.
  - These visualizations provided a clear overview of the profiling data and helped in identifying patterns and anomalies in system call behavior.
- **Insights and Optimization:**
  - By analyzing the profiling data, we gained insights into the performance of the target program and its interaction with the operating system.
  - This information can be used to optimize the target program by identifying bottlenecks in system call usage and improving overall efficiency.
  -

In summary, our approach to profiling and analysis involved collecting detailed information about system call usage, analyzing this data to gain insights into system call behavior, and visualizing the data to facilitate understanding and optimization of the target program.

## Chapter 4. System Design Flow Chart

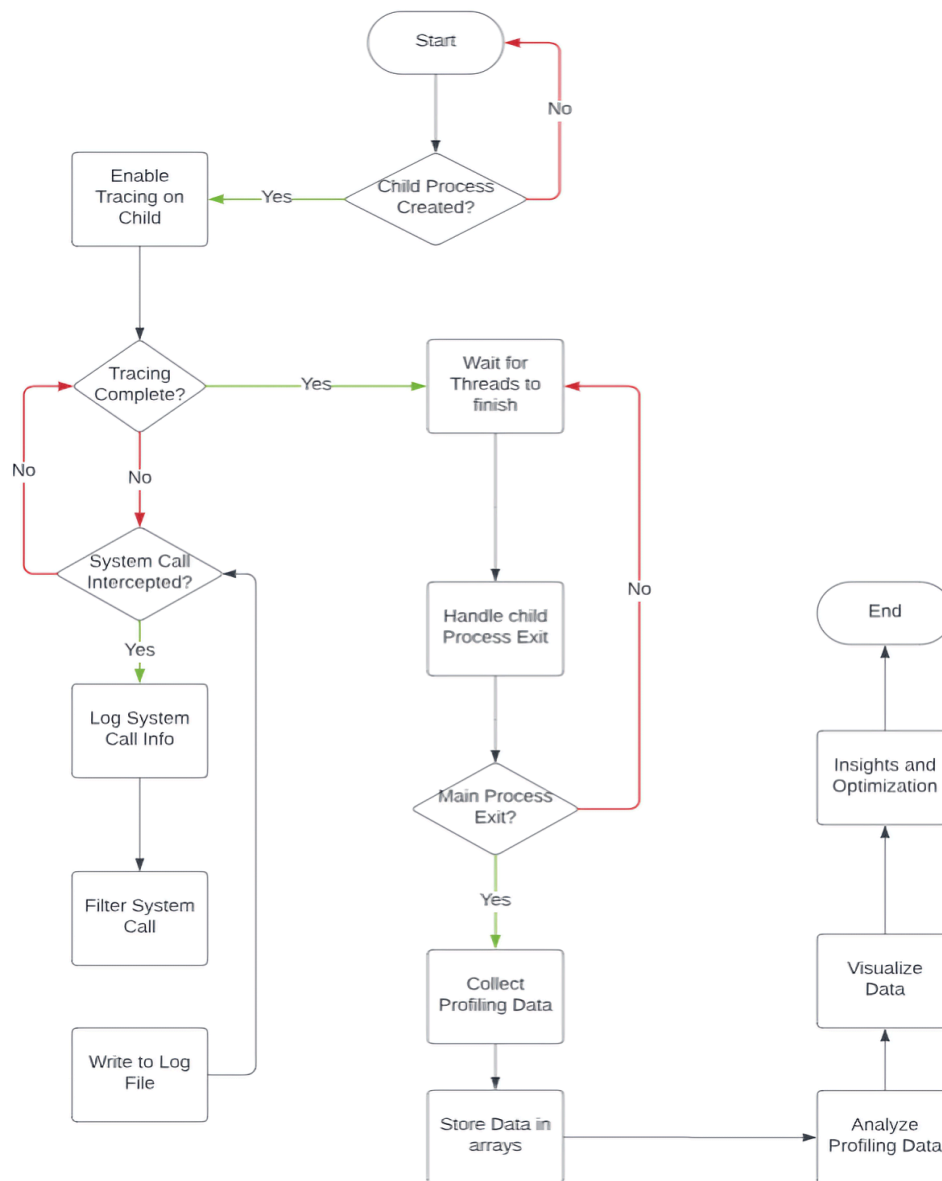


Fig 4.1: System Design Flow Chart

The flow chart illustrates the operation of our System Call Tracer:

- **System Call Interception:** Child processes are created to execute the target program, with tracing enabled using `ptrace`. The parent process traces system calls made by the child processes.
- **Logging and Filtering:** Intercepted system calls are logged, including relevant information such as process/thread ID, timestamp, and call details. Filtering ensures only specific system calls are logged.
- **Multiple Processes/Threads Support:** The tracer supports tracing multiple processes/threads simultaneously. Dedicated tracing threads are created for each child process, with synchronization mechanisms ensuring proper coordination.
- **Profiling and Analysis:** Profiling data is collected by tracing system call execution and stored for analysis. Insights gained help optimize the target program's performance.

This flow chart provides an overview of our System Call Tracer's design, highlighting its key functionalities and interactions.

## Chapter 5: Implementation

### 5.1 System Call Interception

```
// Launch the target process
pid = fork();

if (pid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
}

if (pid == 0) {
    // Child process (target code)
    ptrace(PTRACE_TRACEME, 0, NULL, NULL);
    execl("./target", "target", NULL);
    perror("execl");
    exit(EXIT_FAILURE);
}

// Parent process (tracer)
waitpid(pid, &status, 0);
```

This snippet demonstrates the creation of a child process using fork and the subsequent enablement of tracing on the child process using ptrace(PTRACE\_TRACEME, ...).

FIG 5.1.1 SYSTEM CALL INTERCEPTION

```
ptrace(PTRACE_SYSCALL, pid, NULL, NULL);
waitpid(pid, &status, 0);

if (WIFEXITED(status)) {
    printf("Target process exited.\n");
    break;
}

// Get system call number
ptrace(PTRACE_GETREGS, pid, NULL, &regs);
long syscall_num = regs.orig_rax;

printf("System call %ld\n", syscall_num);

// Continue execution
ptrace(PTRACE_SYSCALL, pid, NULL, NULL);
waitpid(pid, &status, 0);
```

Here, system calls made by the child process are traced using ptrace(PTRACE\_SYSCALL, ...).

Upon interception, the system call number is retrieved using PTRACE\_GETREGS.

if(WIFEXITED(status)) checks if the target process has exited, terminating the tracing loop if so.



## 5.2 Logging and Analysis

In the previous step of 'system call interception,' we traced the system calls. Here, we have added a new function to log and store information about these traced system calls.

```
// Function to log system call information
void log_syscall(pid_t pid, pid_t tid, struct user_regs_struct *regs, long ret) {
    // Open the log file in append mode
    FILE *logfile = fopen("syscall_log.txt", "a");
    if (!logfile) {
        perror("fopen");
        exit(1);
    }

    // Get current timestamp
    time_t now = time(NULL);
    struct tm *tm_info = localtime(&now);
    char timestamp[20];
    strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S", tm_info);

    // Log process/thread ID, timestamp, system call number, arguments, return value, and error code
    fprintf(logfile, "timestamp: %s, pid: %d, tid: %d, syscall: %llu, args: [%llu, %llu, %llu, %llu, %llu], ret: %lu, errno: %d\n",
        timestamp, pid, tid, regs->orig_rax, regs->rdi, regs->rsi, regs->rdx, regs->r10, regs->r8, regs->r9, ret, errno);

    // Close the log file
    fclose(logfile);
}
```

FIG 5.2.1 LOGGING ANALYSIS

This function logs system call information, including the process/thread ID, timestamp, system call number, arguments, return value, and error code, to a log file named `syscall_log.txt`.

For filtering, the `log_syscall()` function can be updated to include the following code snippet to log only those system calls that pass the filter.

```
// Filter system calls based on their number
if (regs->orig_rax != SYS_open && regs->orig_rax != SYS_read && regs->orig_rax != SYS_write) {
    return; // Skip logging this system call
}
```

This updated function filters system calls based on their number (e.g., `SYS_open`, `SYS_read`, `SYS_write`) before logging them to a log file named `syscall_log_filter.txt`.

## 5.3 Multiple Processes/Threads Support

Creating Child Processes:

This code snippet creates multiple child processes (two in this case) using `fork` and executes a placeholder logic (e.g., running the `ls` command) in each child process.

```

// Create multiple child processes
int num_processes = 2;
pid_t child_pids[num_processes];
for (int i = 0; i < num_processes; i++) {
    child_pids[i] = fork();
    if (child_pids[i] == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (child_pids[i] == 0) { // Child process
        // Placeholder: Execute child process logic
        // For example, you can execute an external program using exec functions
        execl("/bin/ls", "ls", "-l", NULL);
        perror("execl");
        exit(EXIT_FAILURE);
    }
}

// Create a thread for each child process to trace system calls
pthread_t threads[num_processes];
for (int i = 0; i < num_processes; i++) {
    if (pthread_create(&threads[i], NULL, trace_child, (void *)&child_pids[i]) != 0) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
    }
}

```

Fig 5.3.1 Multi process  
Creating Threads for Tracing:

Here, a thread is created for each child process to perform system call tracing. The `trace_child` function is used to simulate tracing activity for each child process.

```

// Create a thread for each child process to trace system calls
pthread_t threads[num_processes];
for (int i = 0; i < num_processes; i++) {
    if (pthread_create(&threads[i], NULL, trace_child, (void *)&child_pids[i]) != 0) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
    }
}

```

Waiting for Threads and Child Processes:

```

// Wait for the threads to finish
for (int i = 0; i < num_processes; i++) {
    if (pthread_join(threads[i], NULL) != 0) {
        perror("pthread_join");
        exit(EXIT_FAILURE);
    }
}

// Wait for the child processes to exit
for (int i = 0; i < num_processes; i++) {
    int status;
    waitpid(child_pids[i], &status, 0);
    if (WIFEXITED(status)) {
        printf("Child process %d exited with status %d\n", child_pids[i], WEXITSTATUS(status));
    } else {
        printf("Child process %d exited abnormally\n", child_pids[i]);
    }
}

```

These snippets demonstrate waiting for the threads to finish and then waiting for the child processes to exit. The status of each child process is checked to determine if it exited normally or abnormally.

## 5.4 Profiling and Analysis

### Profiling System Calls

#### Profiling Data Recording:

```
void write_profiling_data() {
    FILE *file = fopen("profiling_data.txt", "w");
    if (!file) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < MAX_SYSCALLS; i++) {
        if (call_count[i] > 0) {
            fprintf(file, "%d %d %lld\n", i, call_count[i], call_duration[i]);
        }
    }

    fclose(file);
}
```

Fig 5.4.1 Profiling and analysis

This function writes the profiling data (syscall number, count, and total duration) to a file named `profiling_data.txt`.

#### Profiling Data Analysis:

```
void print_duration(int syscall_number, long long duration) {
    // Convert nanoseconds to milliseconds for better readability
    double duration_ms = (double)duration / 1000000;
    printf("Syscall %d: Count=%d, Total Duration=%.2f ms, Avg. Duration=%.2f ms\n",
        syscall_number, call_count[syscall_number], duration_ms, duration_ms / call_count[syscall_number]);
}
```

This function prints the total and average duration of each syscall for analysis.

#### Analysis and Visualization:

##### Python Script for Visualization:

```
# Plot bar chart for total duration
plt.figure(figsize=(10, 6))
plt.bar(syscall_numbers, total_durations, color='skyblue')
plt.xlabel('System Call Number')
plt.ylabel('Total Duration (nanoseconds)')
plt.title('Total Duration of System Calls')
plt.grid(axis='y')
plt.show()

# Plot bar chart for average duration
average_durations = total_durations / call_counts
plt.figure(figsize=(10, 6))
plt.bar(syscall_numbers, average_durations, color='salmon')
plt.xlabel('System Call Number')
plt.ylabel('Average Duration (nanoseconds)')
plt.title('Average Duration of System Calls')
plt.grid(axis='y')
plt.show()
```

This Python code reads the profiling data from `profiling_data.txt` and generates bar charts for the total and average duration of each system call, providing visual insights into system call behavior.

## Chapter 6: Results

First a target c code is compiled as shown in Figure 6.1:

```
bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ ls
filtering.c logging.c multi.c profiling_analysis.c profiling_analysis.py target.c tracer.c
bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ nano target.c
bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ gcc target.c -o target
bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ ls
filtering.c logging.c multi.c profiling_analysis.c profiling_analysis.py target target.c tracer.c
bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$
```

### 6.1 System Call Interception

- Implementation: System call interception was successfully implemented using ptrace, allowing the tracer to intercept and monitor system calls made by the target process.
- Insights: Analysis of traced system calls revealed patterns in system call usage, providing insights into the behavior of the target program.

```

bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ nano tracer.c
bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ gcc tracer.c -o tracer
bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ ./tracer ./target
System call 12
System call 158
System call 9
System call 21
System call 257
System call 262
System call 9
System call 3
System call 257
System call 0
System call 17
System call 17
System call 17
System call 262
System call 17
System call 9
System call 10
System call 9
System call 9
System call 9
System call 9
System call 3
System call 9
System call 158
System call 218
System call 273
System call 334
System call 10
System call 10
System call 10
System call 302
System call 11
System call 257
System call 1
System call 3
System call 231
Target process exited.
bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ |

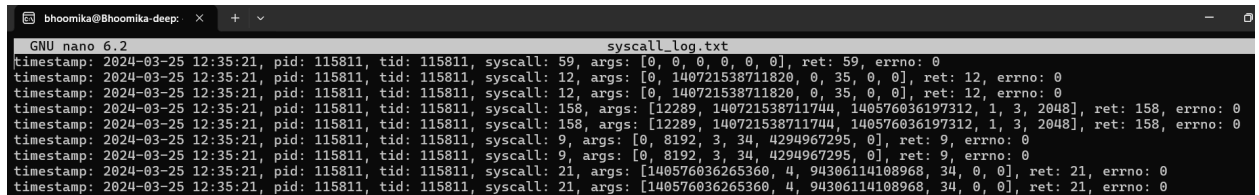
```

Fig 6.2 Tracer:

## 6.2 Logging and Filtering

- **Logging Mechanism:** The logging mechanism recorded detailed information about each intercepted system call, including process/thread ID, timestamp, system call number,

arguments, return value, and error code, and stored then in a file “syscall\_log.txt” whose snippet is given in figure 6.3:

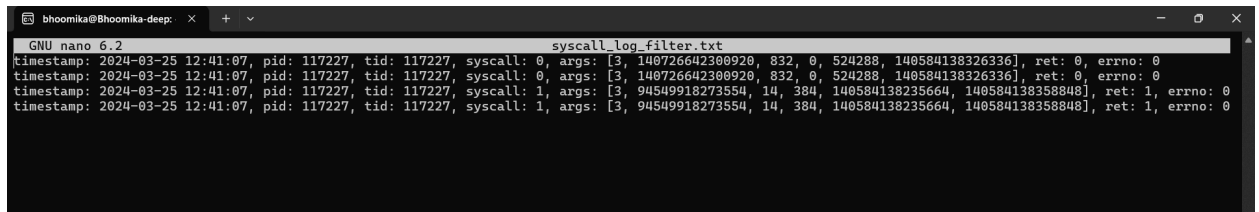


```

GNU nano 6.2 syscall_log.txt
timestamp: 2024-03-25 12:35:21, pid: 115811, tid: 115811, syscall: 59, args: [0, 0, 0, 0, 0, 0], ret: 59, errno: 0
timestamp: 2024-03-25 12:35:21, pid: 115811, tid: 115811, syscall: 12, args: [0, 140721538711820, 0, 35, 0, 0], ret: 12, errno: 0
timestamp: 2024-03-25 12:35:21, pid: 115811, tid: 115811, syscall: 12, args: [0, 140721538711820, 0, 35, 0, 0], ret: 12, errno: 0
timestamp: 2024-03-25 12:35:21, pid: 115811, tid: 115811, syscall: 158, args: [12289, 140721538711744, 140576036197312, 1, 3, 2048], ret: 158, errno: 0
timestamp: 2024-03-25 12:35:21, pid: 115811, tid: 115811, syscall: 158, args: [12289, 140721538711744, 140576036197312, 1, 3, 2048], ret: 158, errno: 0
timestamp: 2024-03-25 12:35:21, pid: 115811, tid: 115811, syscall: 9, args: [0, 8192, 3, 34, 4294967295, 0], ret: 9, errno: 0
timestamp: 2024-03-25 12:35:21, pid: 115811, tid: 115811, syscall: 9, args: [0, 8192, 3, 34, 4294967295, 0], ret: 9, errno: 0
timestamp: 2024-03-25 12:35:21, pid: 115811, tid: 115811, syscall: 21, args: [140576036265360, 4, 94306114108968, 34, 0, 0], ret: 21, errno: 0
timestamp: 2024-03-25 12:35:21, pid: 115811, tid: 115811, syscall: 21, args: [140576036265360, 4, 94306114108968, 34, 0, 0], ret: 21, errno: 0

```

- Filtering: A filtering mechanism was implemented to selectively log specific system calls, such as SYS\_open, SYS\_read, and SYS\_write, enhancing the relevance and focus of the logged information, and stored only those system calls that passes the filter in “syscall\_log\_filter.txt” as shown in figure 6.4



```

GNU nano 6.2 syscall_log_filter.txt
timestamp: 2024-03-25 12:41:07, pid: 117227, tid: 117227, syscall: 0, args: [3, 140726642300920, 832, 0, 524288, 140584138326336], ret: 0, errno: 0
timestamp: 2024-03-25 12:41:07, pid: 117227, tid: 117227, syscall: 0, args: [3, 140726642300920, 832, 0, 524288, 140584138326336], ret: 0, errno: 0
timestamp: 2024-03-25 12:41:07, pid: 117227, tid: 117227, syscall: 1, args: [3, 94549918273554, 14, 384, 140584138235664, 140584138358840], ret: 1, errno: 0
timestamp: 2024-03-25 12:41:07, pid: 117227, tid: 117227, syscall: 1, args: [3, 94549918273554, 14, 384, 140584138235664, 140584138358840], ret: 1, errno: 0

```

## 6.3 Multiple Processes/Threads Support

- Support for Simultaneous Tracing: The system successfully traced multiple processes/threads simultaneously, ensuring synchronization and communication between the tracer and traced entities. Output is given in figure 6.5 and figure 6.6
- Challenges and Solutions: Challenges related to synchronization and communication were addressed, ensuring seamless coordination and accurate tracing of multiple processes/threads.

```

bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ nano multi.c
bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ gcc multi.c -o multi
bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ ./multi
Tracing system calls for process 118359
Tracing system calls for process 118360
total 164
total 164
-rwxr-xr-x 1 bhoomika bhoomika 16552 Mar 25 12:45 a.out
-rwxr-xr-x 1 bhoomika bhoomika 16600 Mar 25 12:41 filtering
-rw-r--r-- 1 bhoomika bhoomika 2446 Mar 25 05:28 filtering.c
-rwxr-xr-x 1 bhoomika bhoomika 16600 Mar 25 12:35 logging
-rw-r--r-- 1 bhoomika bhoomika 2233 Mar 22 21:41 logging.c
-rwxr-xr-x 1 bhoomika bhoomika 16552 Mar 25 12:45 multi
-rw-r--r-- 1 bhoomika bhoomika 2783 Mar 23 02:23 multi.c
-rw-r--r-- 1 bhoomika bhoomika 2942 Mar 25 06:11 profiling_analysis.c
-rw-r--r-- 1 bhoomika bhoomika 813 Mar 25 06:11 profiling_analysis.py
-rw-r--r-- 1 bhoomika bhoomika 10679 Mar 25 12:35 syscall_log.txt
-rw-r--r-- 1 bhoomika bhoomika 610 Mar 25 12:42 syscall_log_filter.txt
-rwxr-xr-x 1 bhoomika bhoomika 16128 Mar 25 12:04 target
-rw-r--r-- 1 bhoomika bhoomika 633 Mar 17 15:37 target.c
-rw----- 1 bhoomika bhoomika 14 Mar 25 12:41 test.txt
-rwxr-xr-x 1 bhoomika bhoomika 16504 Mar 25 12:05 tracer
-rw-r--r-- 1 bhoomika bhoomika 1570 Mar 25 08:05 tracer.c

```

```

Tracing system calls for process 118359
Tracing system calls for process 118360
Tracing system calls for process 118359
Tracing system calls for process 118360
Tracing system calls for process 118359
Tracing system calls for process 118360
Tracing system calls for process 118359
Tracing system calls for process 118360
Tracing system calls for process 118360
Tracing system calls for process 118359
Tracing system calls for process 118360
Tracing system calls for process 118359
Tracing system calls for process 118359
Tracing system calls for process 118360
Tracing system calls for process 118359
Tracing system calls for process 118360
Tracing system calls for process 118360
Tracing system calls for process 118360
Tracing system calls for process 118359
Trace for process 118360 complete.
Trace for process 118359 complete.
Child process 118359 exited with status 0
Child process 118360 exited with status 0
bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ |

```

## 6.4 Profiling and Analysis

- Profiling Data: Profiling capabilities were implemented to track system call frequency, duration, and other metrics.
- After profiling, the details like system call number, frequency, duration and the average duration found after analysis are printed as shown in figure 6.7

- Analysis: Analysis of profiling data revealed the total and average duration of each system call, providing valuable insights into system call behavior and performance, and it stored system call number, frequency and duration in a “profiling\_data.txt” file for further analysis and visualization as shown in figure 6.8

```

bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ nano profiling_analysis.c
bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ gcc profiling_analysis.c -o profiling_analysis
bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ ./profiling_analysis ./target
System call profiling:
Syscall 0: Count=1, Total Duration=0.61 ms, Avg. Duration=0.61 ms
Syscall 1: Count=1, Total Duration=0.45 ms, Avg. Duration=0.45 ms
Syscall 3: Count=3, Total Duration=0.77 ms, Avg. Duration=0.26 ms
Syscall 9: Count=8, Total Duration=2.93 ms, Avg. Duration=0.37 ms
Syscall 10: Count=4, Total Duration=1.42 ms, Avg. Duration=0.36 ms
Syscall 11: Count=1, Total Duration=1.12 ms, Avg. Duration=1.12 ms
Syscall 12: Count=1, Total Duration=0.93 ms, Avg. Duration=0.93 ms
Syscall 17: Count=4, Total Duration=1.61 ms, Avg. Duration=0.40 ms
Syscall 21: Count=1, Total Duration=0.49 ms, Avg. Duration=0.49 ms
Syscall 158: Count=2, Total Duration=1.06 ms, Avg. Duration=0.53 ms
Syscall 218: Count=1, Total Duration=0.30 ms, Avg. Duration=0.30 ms
Syscall 231: Count=1, Total Duration=0.41 ms, Avg. Duration=0.41 ms
Syscall 257: Count=3, Total Duration=1.36 ms, Avg. Duration=0.45 ms
Syscall 262: Count=2, Total Duration=0.52 ms, Avg. Duration=0.26 ms
Syscall 273: Count=1, Total Duration=0.30 ms, Avg. Duration=0.30 ms
Syscall 302: Count=1, Total Duration=0.32 ms, Avg. Duration=0.32 ms
Syscall 334: Count=1, Total Duration=0.30 ms, Avg. Duration=0.30 ms
bhoomika@Bhoomika-deep:~/IT253/SystemCallTracer$ |

```

Fig:6.4.1 System Call Profiling:



```
GNU nano 6.2
0 1 605615
1 1 454909
3 3 774217
9 8 2925649
10 4 1420700
11 1 1117580
12 1 934501
17 4 1610349
21 1 492978
158 2 1063912
218 1 304695
231 1 411784
257 3 1355229
262 2 522570
273 1 302037
302 1 317656
334 1 300183
```

## 6.5 Visualization

- Bar Charts: Visualizations, in the form of bar charts, were generated from the profiling data to illustrate the total and average duration of system calls as shown in figure 6.9 and figure 6.5.1, respectively.
- Insightful Visualizations: These visualizations facilitated a clear understanding of system call behavior and performance, aiding in optimization and debugging efforts.

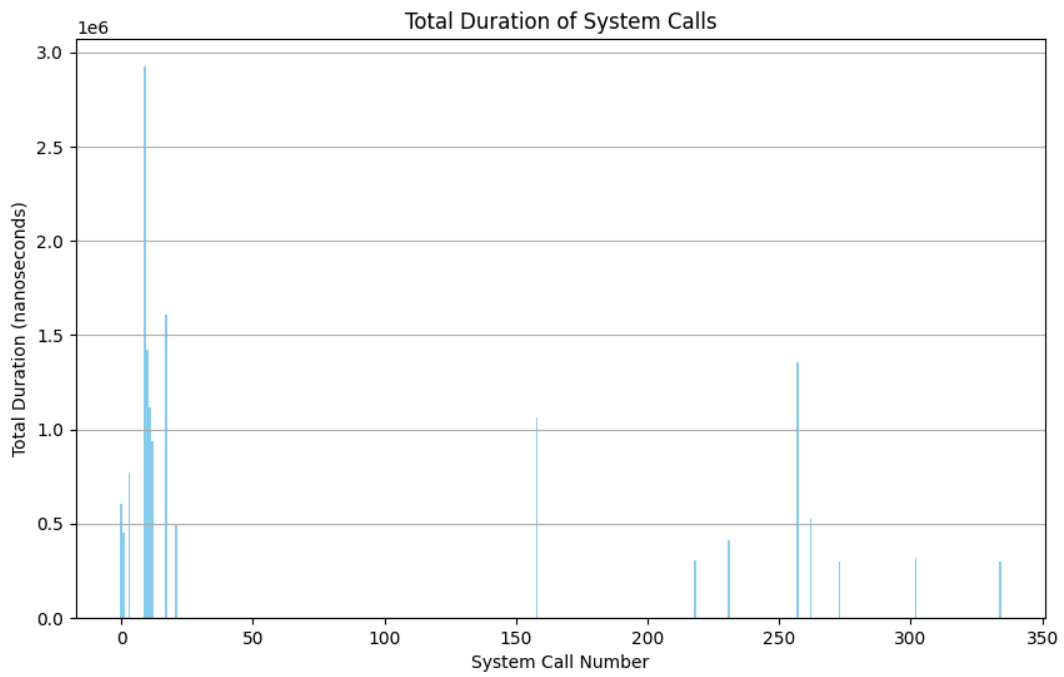


Fig 6.5.1: System Call Number

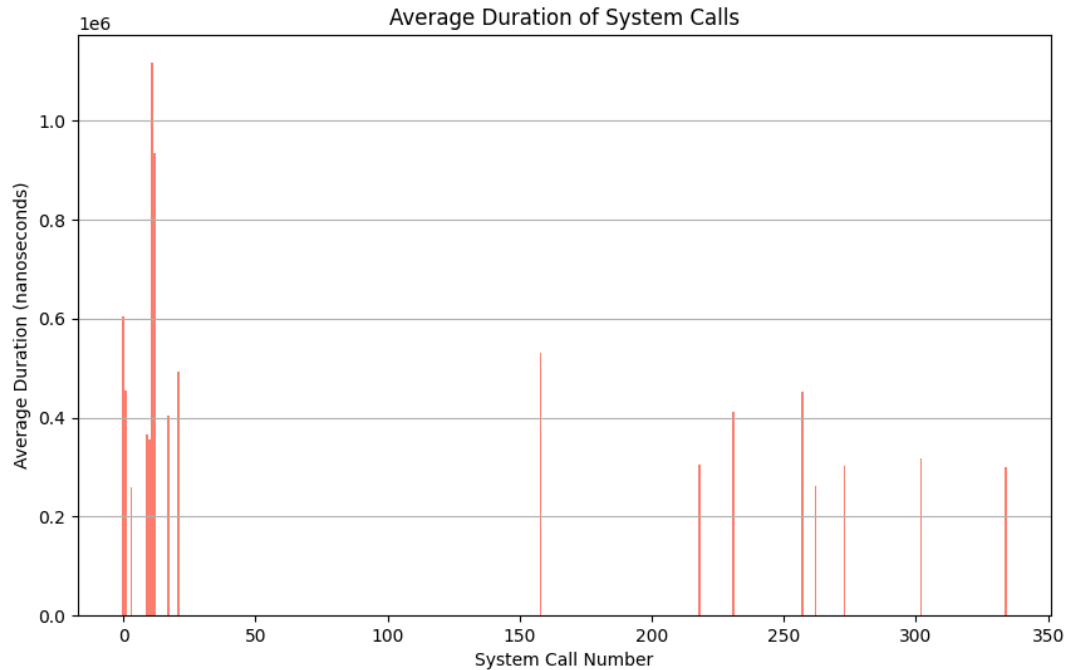


Fig 6.5.2: Avg System Call Number:

## 6.6 Overall Impact and Insights

- **Significance:** The System Call Tracer has provided valuable insights into system call behavior, helping in understanding the performance and usage patterns of system calls.
- **Future Optimization:** The insights gained from the traced data can be used to optimize system call usage and improve the overall efficiency of the target program.

## Chapter 7. Future Work

- **Enhancements:** Future enhancements could include adding support for more system calls, improving filtering capabilities, and enhancing visualization techniques for better analysis.
- **Further Integration:** Integration with debugging tools and real-time monitoring capabilities could further enhance the utility and functionality of the System Call Tracer.