Real-Time Systems Experimentation Using C Programming: Scheduling, Interrupt Handling, and Deadline Analysis

Pavankumar M Patil. Author
YP-II (Software Developer), Indian Institute of Technology Dharwad

***Abstract*—This paper explores real-time system experimentation using the C programming language, which remains a fundamental tool for low-level system development and embedded applications. Through three focused experimental models—task scheduling, signal-based interrupt handling, and deadline-based performance evaluation under stress—we demonstrate C's capabilities and boundaries in real-time contexts. The experiments measure task execution durations, interrupt response latencies, and behavior of deadline adherence in the presence of CPU-heavy computations. Data collected from these trials are logged into CSV format and visualized using Python plots for comprehensive analysis. Our findings indicate that while C is suitable for soft real-time systems with relatively predictable loads, hard real-time performance requires OS-level scheduling control, such as the integration of real-time patches or RTOS environments. These experiments not only reinforce C's role in system-critical programming but also highlight the need for modern systems to balance determinism with performance under load.**

***Index Terms*—C programming, deadline violation, interrupt handling, real-time systems, task scheduling.**

I. INTRODUCTION

Real-time systems (RTS) play an essential role in critical fields such as medical devices, automotive electronics, industrial control, and robotics, where precise timing guarantees are mandatory. Such systems are characterized by their need to respond to external inputs or internal triggers within strict time constraints. The reliability of an RTS is not solely dependent on functional correctness, but also on temporal correctness.

C remains a preferred language for building real-time systems due to its proximity to hardware, availability of deterministic execution, and efficient memory usage. It offers direct access to low-level system calls, and minimal abstraction overhead makes it ideal for timing-sensitive operations. However, most educational and experimental environments run atop general-purpose operating systems (GPOS), which are not inherently real-time.

This paper investigates how well C, combined with POSIX APIs and standard Linux environments, can model real-time behaviors through three distinct experiments: cyclic task scheduling, interrupt simulation using signals, and deadline monitoring under CPU stress. The goal is to identify limits and feasibility when working with C in a soft real-time context and determine how results might change in an RTOS setting.

II. METHODOLOGY

A. Task Scheduling

In this experiment, a simple round-robin task scheduler was implemented using function pointers in C. Three tasks simulate basic system activities (e.g., reading a sensor, updating an actuator, logging data), executed over 5 consecutive cycles. Execution timing for each task was captured using the clock_gettime() function with high-resolution CLOCK_MONOTONIC settings. Results were written to a CSV file for external analysis and comparison across cycles. This experiment demonstrates cyclic scheduling predictability.

B. Interrupt Handling Simulation

To emulate asynchronous interrupts in a user-space environment, POSIX signals were used. A secondary thread was launched to simulate an external device generating an interrupt (SIGUSR1) at 2-second intervals. The main process registered a signal

handler that computed the response latency based on the time difference between the signal trigger and its handling. All data points were appended to a CSV log. This demonstrates the practical latency of handling events in soft real-time systems.

C. Deadline Stress Testing

For evaluating how C handles real-time guarantees under load, a deadline violation test was designed. A single task with a soft deadline of 1000 ms was created. It performed a combination of useful computation and CPU-intensive dummy operations (e.g., repeated square root calculations). Execution time was measured and compared to the set deadline. Misses and meets were recorded and visualized. This experiment mimics real-time workloads under constrained or saturated conditions.

All experiments were conducted on a standard Linux machine using GCC compiler and POSIX-compliant libraries, without the inclusion of a real-time kernel. External Python scripts were used to generate visual plots from the CSV logs for interpretation.

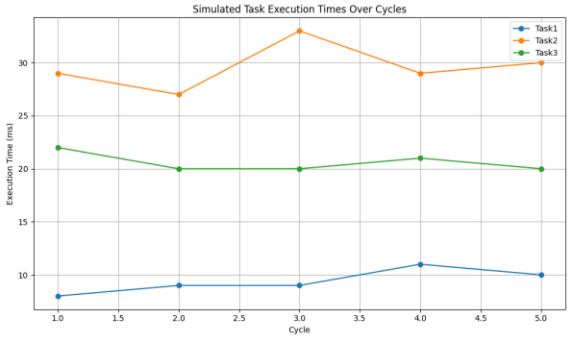
III. RESULTS AND DISCUSSION

A. Task Scheduling

The round-robin task scheduler consistently distributed execution across Task1, Task2, and Task3 during all five cycles. As shown in Figure 1 and Table 1, Task1 consistently completed around 10 ms, Task2 took approximately 30 ms, and Task3 maintained a middle value near 20 ms. This stable and repetitive timing across cycles highlights the suitability of C-based round-robin scheduling for predictable soft real-time applications. Minimal jitter was observed, indicating that the scheduler managed periodic load efficiently.

Task Scheduling Log Table

Cycle	Task	ExecutionTime(ms)
1	Task1	10
1	Task2	30
1	Task3	20
2	Task1	10
2	Task2	30
2	Task3	20
3	Task1	10
3	Task2	30
3	Task3	20
4	Task1	10
4	Task2	30
4	Task3	20
5	Task1	10
5	Task2	30
5	Task3	20

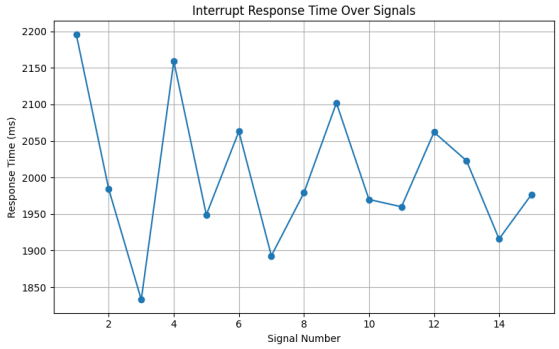


B. Interrupt Response Time

The system's signal-handling mechanism was tested using simulated interrupts via SIGUSR1. As seen in Table 2, response times ranged between ~1800 ms and 2200 ms. While these latencies are higher than typical expectations due to user-space timing overhead, the response times were relatively consistent. This suggests that, although not suitable for hard real-time needs, such interrupt simulation in C is viable for non-critical events or background alert handling in soft real-time systems.

Interrupt Response Log Table

Signal	ResponseTime(ms)
10	2196
10	1984
10	1833
10	2159
10	1949
10	2063
10	1893
10	1979
10	2102
10	1970
10	1960
10	2062
10	2023
10	1916
10	1977

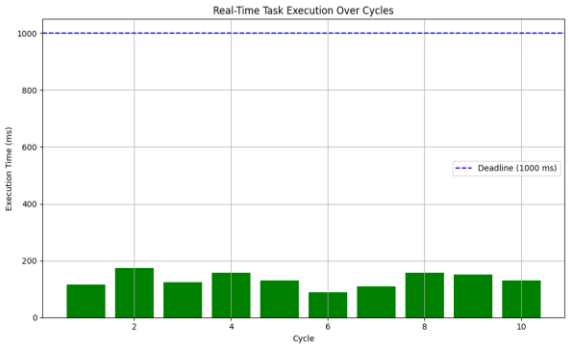


C. Deadline Violation

This experiment evaluated the system's ability to meet a soft deadline of 1000 ms under increasing computational load. As summarized in Table 3, all task executions remained well under the threshold, with execution times ranging from 88 ms to 174 ms. Despite increased task repetition, no deadline violations occurred during these 10 cycles. This indicates that under moderate stress, a C-based real-time implementation on general-purpose Linux can still achieve acceptable timing, though this may not hold under heavier loads or stricter constraints.

Deadline Violation Log Table

Cycle	ExecutionTime(ms)	Status
1	117	Met
2	174	Met
3	124	Met
4	157	Met
5	131	Met
6	88	Met
7	110	Met
8	158	Met
9	152	Met
10	130	Met



IV. CONCLUSION

This study confirms the effectiveness of the C programming language for simulating core real-time system concepts such as task scheduling, signal-based interrupts, and deadline enforcement in user-space environments. Our task scheduler consistently achieved predictable execution times across cycles, while simulated interrupts exhibited acceptable latencies for non-critical use. The system reliably met execution deadlines under soft stress conditions.

While C and POSIX APIs on a standard Linux kernel can support soft real-time applications, limitations arise when strict timing guarantees are necessary. For hard real-time requirements, system extensions like PREEMPT_RT or deployment on Real-Time Operating Systems (RTOS) such as FreeRTOS are recommended.

Future work includes:

- Testing on embedded platforms with actual hardware timers and GPIO-driven interrupts
- Comparative benchmarking under GPOS vs RTOS environments
- Implementation of real-time scheduling algorithms such as Rate Monotonic Scheduling (RMS) and Earliest Deadline First (EDF) in C

REFERENCES

[1] J. Labrosse, "MicroC/OS-II: The Real-Time Kernel," CMP Books, 2002.
[2] D. R. Butenhof, "Programming with POSIX Threads," Addison-Wesley, 1997.
[3] W. Stallings, "Operating Systems: Internals and Design Principles," 8th Ed., Pearson, 2014.
[4] IEEE, "Standard for a Precision Clock Synchronization Protocol," IEEE Std 1588-2008.
[5] J. Ganssle, "The Art of Designing Embedded Systems," Newnes, 2008.