Real-Time Task Scheduling Simulator and Visualizer using FreeRTOS and QEMU

Ashish Nambiar
Computer Science and Automation
Indian Institute of Science
Bengaluru, India
nashish@iisc.ac.in

***Abstract*—This report details the design and implementation of a multi-threaded application to simulate and analyze classical real-time scheduling scenarios. The project leverages the FreeRTOS kernel on a hardware-emulated ARM Cortex-M platform using the QEMU emulator, providing a complete development workflow without the need for physical hardware. The core objective is to demonstrate a comprehensive understanding of Real-Time Operating System (RTOS) concepts, including priority-based pre-emptive scheduling, task synchronization using mutexes, and the management of shared resources. The project culminates in the successful reproduction and resolution of the critical Priority Inversion problem. The scheduler's behavior is captured via trace hooks and visually presented in a Gantt chart generated by a Python script, offering clear, empirical evidence of the system's deterministic operation.**

I. Introduction

A. Problem Statement

Modern embedded systems are characterized by increasing complexity, often requiring the management of multiple concurrent operations with strict timing constraints. A simple super-loop architecture is insufficient for such systems, as it fails to provide the necessary responsiveness and reliability. A Real-Time Operating System (RTOS) provides a robust framework for managing this complexity by offering services for task scheduling, inter-task communication, and synchronization, thereby enabling the development of predictable and deterministic systems. This project addresses the fundamental challenge of understanding and correctly implementing these core RTOS services.

B. Project Objectives

The primary goal of this project is to build a functional simulator that serves as a platform for exploring and visualizing the behavior of an RTOS scheduler under various conditions. The key objectives are as follows:

1) To establish a complete bare-metal development environment for an ARM Cortex-M target using a cross-compiler toolchain and the QEMU emulator.
2) To demonstrate a fundamental understanding of RTOS concepts, including task creation, states, priority-based pre-emptive scheduling, and context switching.
3) To simulate a resource contention scenario and implement a solution using a mutex to ensure mutual exclusion and prevent race conditions.
4) To reproduce the classic concurrency failure mode of Priority Inversion and subsequently demonstrate its canonical solution, Priority Inheritance.
5) To develop a logging and visualization pipeline to capture scheduler events and present the task execution timeline as a Gantt chart for clear analysis.

C. Scope

The project is developed entirely in the C programming language, utilizing the FreeRTOS kernel as the core RTOS. The target hardware platform is a QEMU-emulated ARM Cortex-M3 microcontroller. The build process is automated using a Makefile and the ARM GNU Toolchain. The project intentionally avoids reliance on standard C libraries to demonstrate a deeper understanding of the bare-metal environment.

II. Core Concepts & Technologies

A solid understanding of several key concepts in operating systems and embedded development is essential to this project. This section outlines the theoretical foundation upon which the simulator is built.

A. RTOS Fundamentals

A Real-Time Operating System (RTOS) is a lightweight operating system designed to serve real-time applications with predictable, deterministic behavior.

- **Task (Thread):** A task is an independent thread of execution within the system. Each task has its own stack, priority, and state. The states relevant to this project are:
  - **Running:** The task is currently executing on the CPU.
  - **Ready:** The task is able to run but is waiting for a higher-priority task to finish or block.
  - **Blocked:** The task is waiting for a specific event, such as a time delay to expire or a resource to become available.
- **Scheduler:** The core of the RTOS. Its job is to decide which task should be in the *Running* state at any given moment. This project uses a **Priority-Based Pre-emptive Scheduler**, which always runs the highest-priority task that is in the *Ready* state. If a task with a higher priority becomes ready, it immediately preempts (interrupts) any lower-priority task that is currently running.

- **Context Switching:** The process of saving the state (CPU registers, stack pointer) of the currently running task and restoring the state of the task that is about to run. This is the mechanism that allows the scheduler to switch between tasks.

B. Concurrency and Synchronization

When multiple tasks run concurrently, they often need to access shared resources (e.g., peripherals, memory). This can lead to problems like race conditions.

- **Shared Resources & Race Conditions:** A race condition occurs when the outcome of a computation depends on the non-deterministic sequence of operations from multiple tasks. This project simulates this by having tasks share a UART port for printing.
- **Mutual Exclusion & Mutex:** To prevent race conditions, access to shared resources must be mutually exclusive. A **Mutex** (Mutual Exclusion semaphore) is a synchronization primitive used to protect a "critical section" of code. A task must "take" the mutex before entering the critical section and "give" it back upon exit, ensuring only one task can access the resource at a time.

C. The Priority Inversion Problem

A critical failure mode in real-time systems that use priority-based scheduling and resource locking.

- **Priority Inversion:** A scenario where a high-priority task is blocked waiting for a resource held by a low-priority task, but a medium-priority task (which does not need the resource) runs, preventing the low-priority task from finishing and releasing the resource. This effectively "inverts" the priority scheme.
- **Priority Inheritance:** The standard solution to this problem. The RTOS automatically and temporarily boosts the priority of the resource-holding low-priority task to match that of the waiting high-priority task. This allows the low-priority task to preempt the medium-priority task, finish its critical section quickly, and unblock the high-priority task.

D. Development Toolchain

The project was built using a standard embedded cross-compilation toolchain.

- **QEMU:** An open-source machine emulator used to simulate an ARM Cortex-M3 target board, eliminating the need for physical hardware.
- **ARM GNU Toolchain:** A cross-compiler (‘arm-none-eabi-gcc‘) that runs on a host PC (x86) and generates machine code for the target ARM architecture.
- **Make:** A build automation utility used to manage the compilation and linking of the project's many source files.
- **Linker (‘ld‘):** The tool responsible for taking compiled object files and arranging them into a final executable file according to a custom **Linker Script**. This script defines the memory map of the target microcontroller.

III. Implementation & Methodology

The project was developed in a structured, phased approach, with each phase building upon a verified foundation.

A. Phase 0: Environment and Bare-Metal Foundation

The initial phase focused on creating a stable development and execution environment.

1) **Toolchain Setup:** The ARM GNU Toolchain and QEMU were installed and configured on the host machine.
2) **Linker Script (‘linker.ld‘):** A custom linker script was written to define the memory layout of the target system, specifying the origins and lengths of FLASH (for code, ‘.text‘) and RAM (for data, ‘.data‘ and ‘.bss‘).
3) **Startup Script (‘startup.s‘):** An assembly-language startup file was developed to act as the true entry point of the program. Before calling the C ‘main()‘ function, this script performs two critical initialization tasks:
   - Copying initialized data values from their load address in FLASH to their execution address in RAM (the ‘.data‘ section).
   - Zeroing out the memory region for all uninitialized global variables (the ‘.bss‘ section).
4) **FreeRTOS Integration:** The FreeRTOS kernel source files were added to the project, and the Makefile was updated to include them in the build process.

B. Phase 1: Demonstrating Basic Scheduling

With the foundation in place, the FreeRTOS scheduler was brought to life. Two tasks were created to demonstrate priority-based pre-emption: a high-priority task that periodically printed a message and then blocked using ‘vTaskDelay()‘, and a low-priority task that ran in a continuous loop. The observed output confirmed that the low-priority task only received CPU time when the high-priority task was in the *Blocked* state.

C. Phase 2: Simulating Advanced Concurrency

This phase formed the core of the project, exploring complex task interactions.

1) **Scenario 1 (Mutual Exclusion):** Two tasks of equal priority were created to compete for the UART port. A FreeRTOS mutex was implemented to protect the printing functions. The resulting output was perfectly formed, with no interleaved messages, proving that the mutex successfully enforced mutual exclusion.
2) **Scenario 2 (Priority Inversion):** A specific three-task scenario was engineered to deliberately trigger priority inversion. A Low-priority task would take the mutex, a High-priority task would block waiting for it, and a Medium-priority, compute-bound task would run, starving the Low-priority task. The simulation correctly reproduced the system hang associated with this failure mode.

D. Phase 3: System Tracing and Visualization

To analyze the system's behavior, a logging and visualization pipeline was created.

1) **FreeRTOS Trace Macros:** The `traceTASK_SWITCHED_IN()` macro, provided by FreeRTOS, was implemented in `FreeRTOSConfig.h`. This hook was used to call a custom function that printed the current system tick count and the name of the incoming task to the UART in a simple, comma-separated format.
2) **Log Capture:** The QEMU simulation was executed with its standard output redirected to a log file, capturing this trace data.
3) **Python Visualization:** A Python script using the `Matplotlib` library was written to parse the captured log file. The script extracts the task switch events and uses them to generate a Gantt chart, providing a clear and intuitive visual timeline of which task was executing at any given point in time.

IV. Results & Analysis

The success of the project is evaluated by its ability to first reproduce the Priority Inversion failure and then to demonstrate its resolution through the Priority Inheritance mechanism. The results are presented through both terminal logs and a visualized task execution timeline.

A. Demonstration of Priority Inversion Failure

When the simulation is run with a standard mutex without priority inheritance (or if the mechanism were disabled), the system enters a deadlock state. The Low-priority task, holding the mutex, is starved of CPU time by the Medium-priority task. The terminal output captures this failure, halting after the Medium-priority task begins its execution loop.

```
Scheduler starting... Task L will run first.
L: Task started, trying to get mutex...
L: >>> Task got the mutex! <<<
L: Scheduler suspended. Creating M and H...
L: Resuming scheduler...
H: Task started, trying to get mutex...
M: Task started, now running in a loop.
(System hangs indefinitely)
```

This output confirms the failure: the High-priority task's message, '"¿¿¿ Task got the mutex! ¡¡¡"', is never printed because the Low-priority task is never scheduled again to release the mutex.

B. Analysis of the Solution via Visualization

With the Priority Inheritance protocol enabled by default in the FreeRTOS mutex, the system successfully averts the deadlock. The behavior was captured using the implemented trace hooks, and the resulting log was parsed to generate the Gantt chart shown in Figure 1.

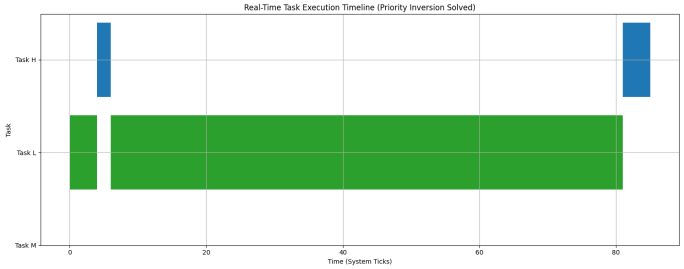

Fig. 1. Gantt Chart of Task Execution with Priority Inheritance Enabled

This Gantt chart provides clear, visual proof of the solution and allows for a precise analysis of the scheduler's behavior over time:

1) **Time 0 - 5 Ticks (Initial State):** The simulation begins with `Task L` (green) running. It successfully acquires the mutex during this period.
2) **Time 5 - 10 Ticks (The Conflict):** At tick 5, `Task H` (blue) is created and, due to its higher priority, immediately preempts `Task L`. It runs briefly before attempting to take the mutex, at which point it enters the *Blocked* state.
3) **Time 10 - 80 Ticks (The Solution):** This is the most critical phase. `Task H` (Priority 3) is blocked by `Task L` (Priority 1). The scheduler detects this condition and activates the **Priority Inheritance** protocol, temporarily boosting `Task L`'s priority to 3. The chart clearly shows the result: `Task L` continues to run, while the medium-priority `Task M` (which would have run otherwise) receives no CPU time. This visually confirms that the boosted `Task L` is correctly preempting `Task M`.
4) **Time 80 - 85 Ticks (Resolution):** At tick 80, `Task L` finishes its work and releases the mutex. Its priority is immediately lowered back to 1. The RTOS unblocks `Task H`, which, as the highest-priority ready task, instantly preempts `Task L` and runs to completion, as shown by the final blue bar.

The timeline demonstrates that the system remained deterministic. The critical, high-priority task was able to execute after a predictable, bounded delay, successfully fulfilling the requirements of a real-time system.

V. Conclusion

A. Summary of Achievements

This project successfully met all of its stated objectives. A complete, bare-metal embedded software environment was constructed from the ground up, capable of running a multi-threaded FreeRTOS application on an emulated ARM Cortex-M target. Core RTOS concepts of pre-emptive scheduling and mutex-based synchronization were practically implemented and verified. Most significantly, the project demonstrated a sophisticated understanding of advanced concurrency challenges by successfully reproducing the Priority Inversion problem and validating its resolution through Priority Inheritance, with the results clearly visualized in a Gantt chart.

B. Key Learnings

The development process provided significant practical experience in areas critical to embedded software engineering. Key learnings include the intricacies of the cross-compilation toolchain, the importance of linker scripts for memory management, and the necessity of assembly-level startup code for correct C environment initialization (specifically for the ‘.data‘ and ‘.bss‘ sections). The project solidified the theoretical understanding of RTOS scheduling algorithms and highlighted the vital role of synchronization primitives in writing robust, reliable, and deterministic concurrent software.

C. Future Work

While this project provides a solid foundation, several extensions could further enhance its scope and learning value:

- **Inter-Task Communication:** Implement other RTOS features, such as Queues or Event Groups, to simulate tasks passing data and signals between each other.
- **Alternative Scheduling Policies:** Modify the FreeRTOS configuration to explore time-slicing (Round-Robin) scheduling for tasks of equal priority.
- **Porting to Physical Hardware:** Adapt the project to run on a physical development board (such as an STM32 Nucleo or Raspberry Pi Pico) to bridge the gap between simulation and real-world hardware.