# Running FreeRTOS on QEMU: project report

Alessandro Genova, Alessandro Torrisi,
Emanuele Cornaggia, Giorgia Moscato, Simone Sambataro

Course: Computer architectures and operating systems

Academic year: 2023 - 2024

## Contents

## 1 Introduction

### 1.1 Operating System Selection: Reasons for Choosing FreeRTOS

We chose FreeRTOS for our project as we found the documentation and source code to be clear, well-commented, and straightforward. FreeRTOS is widely considered the most popular RTOS (real-time operating system), it provides a wide range of support for various processor architectures and it's a minimalistic but powerful RTOS for embedded systems.

### 1.2 Demo Selection and Emulated Board

We chose the demo emulating the MPS2-AN385 board, which has a Cortex M3 CPU. This is the recommended demo to start any FreeRTOS project with QEMU.

## 2 Creation of the tutorial

In line with the initial requirement of creating a tutorial we have developed a step-by-step guide. The guide is designed for various user type (Linux, Mac, Windows), providing detailed instructions on how to use FreeRTOS on QEMU and the essential tools needed to ensure the proper execution of our project.

## 3 Practical examples demonstrating the functionality of the operating system

We've implemented different examples and they are briefly described below:

### 3.1 UART Command Line

This example is designed to allow users to execute commands through the UART console efficiently. The implementation features a UART command line interface with interrupts, eliminating the need for resource-intensive polling methods. This approach ensures that when the user is not actively using the keyboard, the CPU can be used for other tasks or transition to an IDLE state to conserve resources and energy.

### 3.2 CPU Usage Statistics

This example provides detailed task information, including task state and CPU utilization percentage. A flag allows the activation or deactivation of CPU usage statistics, which are useful for debugging but can be resource-intensive and unnecessary during real-time operation. Setting this flag to 0 deactivates the CPU statistics code, optimizing performance by bypassing unnecessary statistics.

### 3.3 Memory WatchDog Example

In this example our goal was to to monitor and manage memory usage in a real-time operating system (RTOS) environment. With this example we are able to see statistics related to memory such as: Heap Used Space, Available Heap Space, Size Of Smallest Free Block In Bytes, and others.

### 3.4 Notifications Usage in FreeRTOS

In this example, we simulate a scenario where two sensors — one for temperature and one for humidity — read their respective values. The monitoring of their values is facilitated by FreeRTOS notifications. If the temperature tolerance threshold is exceeded, a notification is sent, and the temperature notification handler responds by illuminating the first half of the LEDs. Instead, if the humidity tolerance threshold is exceeded, the humidity notification handler lights up the second half of the LEDs.

### 3.5 LED Animations

In this example, interacting directly with the LED control registers on the board, we've implemented two LED animations: the Knight Rider Effect and the Constant Blink Effect.

### 3.6 Semaphore Usage in FreeRTOS

This example shows the use of mutexes in FreeRTOS in order to coordinate 2 tasks, a consumer and a producer: producer and consumer have a shared buffer, that the consumer initializes, based on a parameter passed at his creation, and only when the buffer is ready the consumer is able to read it.

### 3.7 Event Groups in FreeRTOS

In this branch a simple example is implemented to showcase the utilization of Event Groups functionality. The primary objective is to demonstrate how Event Groups can be employed for synchronization between tasks, allowing efficient communication and coordination in the FreeRTOS environment.

## 4 Implementation of new features

### 4.1 Best-fit and Worst-fit implementation

Our base demo use the heap_4.c, that implements a First-fit algorithm for memory management, so we tried other methods to allocate memory. We've implemented 2 algorithms:

- Worst Fit: allocates the block in the largest free block present in that moment.
- Best Fit: allocates the block in the smallest free block possible (big enough to contain the block).

In FreeRTOS, there are five files for heap implementation (heap1, heap2, heap3, heap4, heap5). We've focused on the vanilla heap4.c which implements First-fit allocation, it also includes coalescing to mitigate fragmentation. To preserve this functionality, we maintained the order of the free list based on block addresses. Given that Best-fit allocates the smallest block, our approach involved iterating through the entire list to locate the smallest block capable of fulfilling the request. For Worst-fit allocation, we similarly traversed the list to find the largest block. In both cases, the free block list remains sorted by address to facilitate coalescing. Additionally, we introduced a Memory WatchDog feature that triggers whenever a malloc function is invoked. It verifies whether the requested allocation size exceeds a user-defined threshold of minimum available free memory.

### 4.2 IDLE low-power state

Referring to 3.1 example, at first, we tried to use the command line with polling method, but through this method we noticed that CPU usage was too high even when user is not interacting. A solution to this problem is to have a specific interrupt that gets activated when we press a button on the keyboard. This ensures that when the user is not actively using the keyboard, the CPU can be used for other tasks or it can transition to an IDLE state to conserve resources and energy. The low-power state is achieved through the Tickless IDLE feature, where the system remains inactive until awakened by an external event (such as a button press on the UART keyboard) or when it is necessary to resume a blocked task.

### 4.3 Non-Preemptive scheduling

For our project we've implemented different types of non-preemptive scheduler algorithms because the main goal was to compare the differences in performance. We can disable preemption by setting the macro configUSE_PREEMPTION to 0 in FreeRTOSConfig.h (the default algorithm in this case is first comes first served). We've implemented:

- Longest Job First: given a set of task we first compute the ones that have the biggest execution time.
- Shortest Job First: we schedule the task that has the smallest execution time.

The basic idea is to initially generate a set of random parameters for the 9 tasks that will be created. The primary task takes the given parameter as input and performs a for loop based on that number. We also record the start time and end time to compute statistics for average waiting time and average turnaround time. The greater the parameter, the longer the execution time of the task; this is a way to estimate the time required for task completion. Now, tasks can be rearranged based on their execution time. With LJF, tasks are rearranged in descending order (from slowest to fastest), while for SJF, tasks are ordered from the fastest to the slowest.

### 4.4 Earliest Deadline First scheduling

To implement EDF, we initially studied the FreeRTOS base code, examining task creation, scheduling, and context switch mechanisms. We also reviewed various online EDF implementations. Applying the knowledge gained from our research, we modified FreeRTOS to support EDF scheduling. Building upon the structure of the FreeRTOS codebase, we developed an implementation of xTaskCreate(), adjusted the TCB structure, and established an ordered list sorted by task deadlines and other relevant criteria aligned with the OS's task management principles. In order to implement correctly EDF we need a parameter that tell us when a task needs to end (deadline), so we added a new variable in the TCB called xTaskDeadline, that characterizes each task. From this parameter we decide which task to schedule (according to the earliest deadline), where new tasks must be placed in the ready list and so on. This helps us prioritize tasks according to their urgency achieving what EDF is supposed to do. To establish the deadline, we used a function that generates pseudo-random number, this happens before the creation of the tasks. When a task is currently running and another task with an earlier deadline becomes ready to execute, the scheduler is activated, and preemption takes place.

## 5 Evaluation and Performance improvement

### 5.1 Non-preemptive scheduling algorithm

To evaluate the performance for the different algorithms we run the code ten different time and look at the value of average waiting time and average turnaround time and compare them:

- First Come First Served: average waiting time it's around 324 unit and average turnaround time it's around 382 unit; like we have seen during the course the performance highly depends on which task arrives first.
- Longest Job First: average waiting time it's around 389 unit and average turnaround time it's around 468 unit, this is the worst of the three algorithms implemented.
- Shortest Job First: average waiting time is 173 unit while average waiting time it's around 218 unit, as we expected SJF is the best in terms of both parameters, but it's difficult to have an accurate estimation of the execution time for each task.

### 5.2 Preemptive scheduling algorithm: EDF

Since EDF is a scheduling algorithm that heavily relies on deadlines, and considering that we generate them (pseudo)randomly, the results depends on how these numbers are generated. In other words, if the generated deadlines are very close to each other, it will naturally be more challenging to respect them.

### 5.2.1 Evaluation on respected deadlines

In this first scenario, we conducted a test on 10 tasks that all arrive simultaneously and execute only once. The test is based on a seed provided as input by the user, and by using the same seed, the generated deadlines are, of course, identical. We compared the performance in terms of respected deadline between EDF and Round Robin (that is the base preemptive algorithm implemented by FreeRTOS). We did that in order to show that Round Robin generally doesn't respect deadlines. We enforce the fact that the results highly depend on the machines used because the rand_r function used to generate the random parameters changes implementation based on the standard library present on the OS. We run the code multiple times and those are the average of missed deadlines and respected deadlines for EDF and RR.

- Percentage of respected deadlines for RR: 30%
- Percentage of respected deadlines for EDF: 88%

To obtain these results, we conducted several tests with different seeds on a MacOS, so therefore tests conducted on different machines could lead to different results.

### 5.2.2 Checking correctness of EDF

At first we were not sure of the correctness of our scheduler, so in order to verify that the task scheduled was the one with the earliest deadline, we made a practical example to show when context switch happens and what task actually gets scheduled. With this test we can observe that when a task with an earlier deadline get scheduled, the context switch happens, moving the current task in a suspended state and executing the just arrived task with the earlier deadline. Had this not happened, we would have missed the deadline of the newcome task. We have also implemented another task just to show that the deadlines that get compared are not the starting deadline declared in the xTakeCreateDeadline, but the correct ones which their values in ticks that get decremented with the execution of the os.

### 5.2.3 Periodic task evaluation

We wanted to test our system on real time situation, in order to do that we have created an example showing the full correctness of our algorithm testing it on a real time situation. We have create 2 task with 2 different deadline, that respect the formula of the feasibility of EDF and the we showed when context switch happens and how each task respects its deadline.

## 5.3 Comparison between Worst-fit and Best-fit

We compared our implementation of Worst-fit and Best-fit in terms of fragmentation, using this formula found on StackOverflow :

$$\frac{\text{free memory} - \text{largest free block}}{\text{free memory}}$$

The test consists in 2 parts:

- The first test consists in allocating a fixed number of blocks each time to see how the fragmentation behaves for both algorithms
- In the second test (using a pseudo random function) we allocate a random number of bytes and then disallocates them:

Running the code those are the major difference between Worst-fit and Best-fit:

- for the random test: for the Worst-fit we have an average fragmentation of 8.52% while for the Best-fit we have an average fragmentation of 3,71%. Even if this test does not stress too much our system we have a small conclusion about the the fact that best fit behaves better in terms of fragmentation compared to worst fit. (see table 1)
- for the fixed allocation test: for the Worst-fit we have an average fragmentation of 4,85% and for the Best-fit we have a average fragmentation of 1.89% (see table 2). The consideration for this test are the same as the previous

Table 1: Feasibility Percentage for Best-fit and Worst-fit algorithms: random size test

| **Iteration** | **Worst Fit** | **Best Fit** |
|---|---|---|
| 1 | 7.0% | 1.7% |
| 2 | 8.0% | 2.7% |
| 3 | 20.0% | 13.3% |
| 4 | 0.4% | 0.4% |
| 5 | 10.0% | 7.9% |
| 6 | 4.0% | 6.1% |
| 7 | 15.0% | 0.7% |
| 8 | 12.8% | 1.2% |
| 9 | 0% | 0% |
| 10 | 8.0% | 3.1% |

Table 2: Fragmentation Percentage for Best-fit and Worst-fit: fixed size test

| **Iteration** | **Worst Fit** | **Best Fit** |
|---|---|---|
| 1 | 0% | 0% |
| 2 | 0% | 0% |
| 3 | 0% | 0% |
| 4 | 17.0% | 17.0% |
| 5 | 17.0% | 1.2% |
| 6 | 2.6% | 0% |
| 7 | 2.6% | 0% |
| 8 | 3.0% | 0.4% |
| 9 | 3.0% | 0.3% |
| 10 | 3.3% | 0% |