Politecnico di Torino

Politecnico di Torino
Department of Control and Computer Engineering

Group Project – Computer architectures and operating systems

HaclOSsim (Nr. 2)

Group Nr. 09

Mat. 323386 – Antonio Capece
Mat. 328833 – Federico Cagnazzo
Mat. 331031 – Giovanni de Maria
Mat. 331618 – Andrea Carcagnì

Accademic Year 2023-2024

Contents

Abstract

The HackOSsim project involves the exploration and enhancement of the FreeRTOS embedded operating system within the QEMU environment, an ISA simulator. One of the main goals of this project is to achieve proficiency in using QEMU. To facilitate this, a detailed tutorial is provided, which explains the installation and usage processes of the ISA simulator for emulating an ARM system. Additionally, practical examples are created to demonstrate the features of FreeRTOS. A significant achievement of this project is the implementation of a scheduler, which is crucial for managing task execution within FreeRTOS.

1 Description of the software used

1.1 QEMU

QEMU (Quick EMUlator) is an open-source emulator that allows users to run virtual machines and emulate various hardware architectures. Initially created for x86, QEMU has expanded its capabilities to include a diverse array of architectures such as ARM, MIPS, and PowerPC. It provides a flexible and powerful platform for developing, testing, and debugging software in a virtual environment.

1.2 FreeRTOS

FreeRTOS is a leading real-time operating system (RTOS) designed for embedded devices. It offers a versatile kernel that supports the development of software applications across various hardware platforms. The primary reason for using an RTOS is the complexity involved in managing multiple tasks in real-time environments. Embedded systems frequently need to respond to events promptly and predictably. FreeRTOS addresses these challenges by providing essential features like task scheduling, inter-task communication, and synchronization, ensuring efficient and reliable operation.

2 Usage Procedures

2.1 Usage of FreeRTOS

In order to use FreeRTOS, the FreeRTOS source code must be used in the project. It is designed to be simple and easy to use, requiring only three source files common to all ports and one microcontroller-specific file.

In the official FreeRTOS project, the files to be included are located in:

- `FreeRTOS/Source/tasks.c`
- `FreeRTOS/Source/queue.c`
- `FreeRTOS/Source/list.c`
- `FreeRTOS/Source/portable/[compiler]/[architecture]/port.c`
- `FreeRTOS/Source/portable/MemMang/heap_x.c`

The following header file paths must be included in the project:

- `FreeRTOS/Source/include`
- `FreeRTOS/Source/portable/[compiler]/[architecture]`
- `[path to FreeRTOSConfig.h]`

Each project must include a `FreeRTOSConfig.h` file to configure the RTOS according to the specific needs of the application.

2.2 Usage of QEMU

The following commands are used to compile FreeRTOS and then run QEMU:

```
make --directory=./build/gcc
qemu-system-arm -machine mps2-an385 -cpu cortex-m3 -kernel ./build/gcc/output/RTOSDemo.out -
    monitor none -nographic -serial stdio
```

Ensure you are in the demo folder before executing the commands. In our project, the source code of the demos are located in Examples folder.

The command qemu-system-arm launches a QEMU virtual machine emulating an ARM Cortex-M3 system using the mps2-an385 machine type. It specifies RTOSDemo.out as the kernel image, disables the QEMU monitor and graphical output, and redirects the serial console to the terminal.

We would like to highlight that in every folder of each project demo, there is a .vscode/tasks.jso file. This file was instrumental during the development phase, as it automated the compilation process by including the make and qemu-system-arm commands.

3 Demos - Practical examples

Hands-on projects have been created to illustrate the process of building applications with QEMU and FreeRTOS on a Cortex-M3 microcontroller. In each project, the simulated **mps2 - an385** development board is utilized due to its comprehensive support for the Cortex-M3 architecture and its robust simulation capabilities, which provide a realistic development environment without the need for physical hardware.

3.1 demoTimer.c – Timer-Based Message Delay

This demo uses a timer to create increasing delays between output messages. Starting with a 1-second delay, each subsequent message delay increases by 1 second. The demo is limited to 7 repetitions but can be adjusted for more. A message is printed on the console after an increasing timeout. The vTimerCallback function checks and increments repetitions, updates and prints the current delay, stops the timer after 7 repetitions.

3.2 demoSemaphores.c – Semaphore-Based task coordination

This demo implements three mutex semaphores and three tasks to manage access to a critical section. Each task prints a message and uses a timer with respective delays of 1, 2, and 3 seconds. An array of semaphore handles (xSemaphores) synchronizes tasks. Each task waits for its semaphore, prints a message, delays, prints a wake-up message, and releases the semaphore to the next task.

The core function initializes semaphores, starts the first semaphore, creates tasks, and starts the FreeRTOS scheduler. Tasks execute cyclically, with each task blocking the others while accessing the critical section.

3.3 demoStats.c – Runtime statistics and task coordination

This program is an example to show how much space remains unallocated in the heap. It creates two tasks, run concurrently. The former is a simple counter, the latter call the xPortGetFreeHeapSize() function, which returns the total amount of head space that remains unallocated.

3.4 demoMatrix.c – Matrix Multiplication

This demo performs matrix multiplication between two predefined matrices. Each task handles the multiplication of a specific row from matrix A with a column from matrix B, inserting the result into the corresponding cell of the result matrix C.

The demo includes standard I/O and FreeRTOS headers for task management. It defines matrix dimensions and initializes matrices A and B with predefined values. Tasks are created to handle individual element multiplications and to print the result matrix.

3.5 demoHospital.c – Patient Management System

This demo simulates the management of a hospital system. The hospital has a limited number of operating rooms, with only one patient being operated on at a time in each room. Patients are operated on based on a priority system determined by color codes, which define both the priority and the maximum waiting time before the condition worsens.

To manage concurrent access to the operating rooms, semaphores are used to count the number of available rooms. Semaphores are taken when a room is available and released upon the completion of an operation. Queues are utilized to maintain the list of patients waiting for operations, with separate queues for patients with red and green codes. Tasks populate these queues with the arrival times of patients.

Timers manage the duration of operations in each operating room. Tasks run in parallel, allowing for a realistic simulation of the hospital environment.

3.6 demoHospital2.c – Emergency Room Management

This demo represents an improvement over the previous hospital management system, with new functionalities and optimizations to better manage patient operations based on priorities. The new system continues to manage patients according to a priority system based on color codes but introduces a more detailed data structure for patients and a more advanced scheduling logic. The additional functionalities include managing patient priorities, removing patients from the list in case of death, and enhancing operation management using task notifications.

Specifically, the ‘taskPazient‘ function handles the operation of a single patient. It uses a waiting loop to simulate the operation time and notifies the task scheduler upon operation completion using ‘xTaskNotifyGive‘. This notification mechanism allows effective communication between the task performing the operation and the task scheduler, improving synchronization and flow control.

The task scheduler calculates priorities based on critical time, operation duration, and arrival time, allowing for a more efficient and realistic management of operations.

4 Scheduler implementation

This newly implemented feature aims to simplify the implementation of the previous demos, previously discussed in 3.5 and 3.6, which shows an actual implementation of an hospital management system.

Each patient has a unique ID, an arrival time, a critical time after which the condition worsen, an operation time and a priority code.

The system presume the presence of only an operation room, and schedules the patients based on their criticals time.

To simulate the arrival of a patient, a periodic task executed every second is used which creates a task for each patient, and the Operating System is used to schedule them.

In this way, there is no need to implement a big task that occupies its time to schedule each patient, but only one to create them.

To schedule them, the OS implements a polling server scheduler. Its purpose is to execute each periodic task with the highest possible priority. The scheduler implements a way to use preemption on each aperiodic task, which is by default disabled for the purpose of this demo, because we need to simulate a single operating room, and it would not be appropriate to have each patient use the operating room only for a small portion of time, and then give it up to other patients until they can use it again.

For this case, if the patient is not being operated before his critical time, it will die. This case also happens when an operation was not completed before the critical time. When a patient is being operated for the amount of time requested, which is fixed for each patient, it will free the operating room and be saved.

4.1 Scheduler performance evaluation

To compare the demo with the original FreeRTOS scheduler, the default preemption is used. This is needed because it is the only way to schedule the periodic task correctly, but imply that the hypothesis make prieviously is not respected.

The comparation of the performance of the FreeRTOS scheduler with preemption and our custom preemptive scheduler, is shown in the tables below. In the FreeRTOS scheduler, all patients (P1 to P4) are operated on within their deadlines, with none of them experiencing fatal delays. In contrast, our custom preemptive scheduler employs a different task prioritization mechanism, resulting in delayed start times for some operations. Specifically, Patient P3 misses the deadline and dies due to the delayed start of their operation.

FreeRTOS Scheduler with preemption

Patient	Arrival time	Starting Operation	Duration	Deadline	Ending Operation	Died (yes/no)
P1	2	2	3	9	5	no
P2	2	2	4	8	6	no
P3	6	6	4	12	10	no
P4	8	8	1	10	9	no

Our Scheduler with preemption

Patient	Arrival time	Starting Operation	Duration	Deadline	Ending Operation	Died (yes/no)
P1	2	6	3	9	5	no
P2	2	2	4	8	6	no
P3	6	10	4	12	-	yes
P4	8	9	1	10	10	no

5 Statement of work and final declaration

All members participated actively and collaboratively. The work of each member is explained in this table:

Antonio Capece	Project management, Scheduler implementation and showcasing
Federico Cagnazzo	Demos development, Slides writing, Tutorial writing
Giovanni de Maria	Demos development, Report writing, Tutorial writing
Andrea Carcagnì	Co-project management, Demos development

We, the undersigned students, members of Group Nr. 09, hereby declare that the contents presented in this document and in the Git repository are the result of our original work.

Antonio Capece (323386), Federico Cagnazzo (328833),
Giovanni de Maria (331031), Andrea Carcagnì (331618)
Turin, June 27, 2024