Computer Architectures & Operating System
(a.a. 2023/2024)

Group 39 - HaclOSsim

Luigi Bartolomeo, 324471
Riccardo Bruno, 323563
Federico Civitareale, 323440

February 2025

Contents

1 Methodology & Tools

1.1 QEMU

QEMU (Quick Emulator) is a free and open-source machine emulator and virtualizer that can run various guest operating systems and architectures on a single host system. Unlike traditional virtualization solutions, QEMU is not just a hypervisor but an emulator that can run guest operating systems on a wide range of host platforms. It is a powerful tool for software development, testing, and deployment.

1.2 FreeRTOS

An RTOS (Real-Time Operating System) is an operating system designed to manage tasks in a predictable and timely manner, ensuring that systems respond within strict deadlines. While general-purpose operating systems like Unix or Windows offer multitasking by rapidly switching between different tasks, an RTOS ensures deterministic behavior by meeting real-time requirements. This is crucial for embedded systems, which often have time-sensitive tasks. In an RTOS, tasks are assigned priority and the scheduler uses those priorities to determine which task to execute next.

FreeRTOS is a lightweight RTOS designed primarily for microcontrollers, which are resource-constrained processors and often used in embedded systems. Unlike more extensive systems, microcontrollers typically have limited memory and are designed for specific, dedicated tasks. FreeRTOS provides essential real-time scheduling, inter-task communication, and synchronization, making it suitable for these applications. It is often referred to as a real-time kernel or executive, as it focuses on core functionality, with additional features such as networking stacks or command interfaces available through add-ons.

2 Installation & Configuration

2.1 FreeRTOS installation

Firstly, clone FreeRTOS or install from freertos.org.
The folder of interest is **CORTEX_MPS2_QEMU_IAR_GCC**; all others can be eliminated. To use FreeRTOS, the user must include the FreeRTOS source code in their project, usually contained in the folder /FreeRTOS. Every project must consist of a FreeRTOSConfig.h file, which is used to configure the RTOS to the application's specific needs.

2.2 QEMU, ARM Toolchain and GDB

2.2.1 QEMU

Launch these commands in a Ubuntu/Debian system to install these tools:

```
apt-get install qemu
apt-get install gcc-arm-none-eabi
apt-get install gdb-arm-none-eabi
```

For other OSs (Arch, MacOS), check the Tools Installation section file in the associated repo's README. The package *qemu-system-arm* is used to emulate a 32-bit ARM system. Mandatory arguments are machine, CPU, and kernel image. The machine argument specifies the machine type to emulate, while the CPU argument specifies the CPU model, the kernel image is the kernel to be loaded. In this project the following command is used to run QEMU:

```
qemu-system-arm -s -S -M mps2-an385 -cpu cortex-m3 -monitor none
-nographic -serial stdio -kernel ./output/RTOSDemo.out
```

2.2.2 GDB

In order to debug the application running on qemu, the GDB debugger can be used, and the command to run is:

```
arm-none-eabi-gdb RTOSDemo.out -ex "target remote localhost:1234"
```

3 Demo Application

3.1 demoSmokers

The Smokers Problem is a classical synchronization problem in operating systems, similar to the Dining Philosophers Problem. It involves three smokers and one agent, highlighting resource-sharing issues and process synchronization. There are 3 kinds of subjects: the ingredients are paper, tobacco, matches, the smokers(who roll the cigarette and smoke), and an agent (who provides the ingredients). The problem is described as follows:

- Anyone can smoke as long as it has papers, tobacco and matches
- The smokers have one random ingredient
- The agent puts 2 random ingredients on the table
- who does not have the ingredients on the table, smokes

In this demo it is used:

- **SemaphoreHandle_t**: to manage the taking of ingredients

And the following task is created:

- taskSmoker: the smokers wait for all ingredients, then smoke

3.2 demoDinner

The Dining Philosophers Problem is a classic synchronization problem in operating systems that illustrates the challenges of resource sharing and deadlock prevention. In brief, there are 5 philosophers seated around a circular table, each philosopher has a plate of food, between each pair of philosophers, there is one fork (in total 5 forks). A philosopher needs two forks (left and right) to eat. There are 4 routines in which the philosophers can be: Thinking (don't use any fork), Hungry (try to pick up forks), Eating(have both forks and consume), and Releasing (release the forks).

The problem is that the forks are a shared resource, which can lead to deadlock(nobody consumes) or starvation(someone will ever consume). The solution to this problem is to implement an asymmetric solution: The odd-numbered philosopher picks up their right fork first and then the left fork, and the even ones pick up their left fork at first and then the right fork.
In this demo will be used:

- **SemaphoreHandle_t**: for managing the forks state(picking up)

The following tasks were created:

- **philosopherTask**: The philosopher thinking is simulated by *pdMS_ TO_ TICKS(1000)* and the fork taking is simulated by the use of semaphore (*xSemaphoreTake*), if the philosopher has 2 fork eat and release the fork with *xSemaphoreGive*

3.3 demoBarber

For this demo application, Edsger Dijkstra's Sleeping Barber Problem was chosen, which is a classic problem in operating systems to illustrate issues related to process synchronization using semaphores and message passing. Briefly, there is a barber who always wants to sleep when no customer is in the shop, C customers ($NUM\_CUSTOMERS$) and N seats (NUM_SEATS) in the waiting room. Each customer arriving at the shop either:

- has its **hair cut**, after waking up the barber
- **sits in the waiting room**, if the barber is not free
- **leaves**, if there is no chair available

In this demo will be used:

- **QueueHandle_t**: for storing customers sitting in the waiting room
- **SemaphoreHandle_t**: binary semaphore to signal if the barber is available or not

For this purpose, the following tasks were created:

1. **vBarberTask**: wakes up if the queue is not empty, blocks the semaphore for an indefinite timeslot ($portMAX\_DELAY$), cuts hair and frees the semaphore
2. **vCustomerTask**: if the queue is empty, it deletes itself; else sits or leaves
3. **vCustomerGeneratorTask**: generates all the vCustomerTasks, introducing a delay space-out between each one

4 demoScheduler

4.1 Description

The scheduler demo modifies the barber's one to improve its performance. In this new version have been added some customers feature, namely the arrival time, the service time (how long it takes for the haircut) and the expiration time (after how long the customer leaves angrily because he's been waiting too long for the barber to cut his hair).

Like in the previous scenario, there is just one barber, and the scheduler has been tested with 5 customers arriving and a waiting room of a maximum of 3 seats. The customers are scheduled based on their expiration time using the Earliest Deadline First (EDF) scheduling algorithm.

A timer has been set to simulate customer arrival, and upon its expiration, a new task is created for the relative customer. The RTOS then schedules them according to 2 modes of operation (which can be set through a variable in the FreeRTOSConfig.h file):

- Non-preemptive Mode

  The barber fully completes a customer's haircut before moving on to the next. Even if a customer with a shorter deadline arrives, the barber does not switch tasks. This simulates a real-world barber shop, where once a customer is seated, they receive a full haircut before the next customer is served. The EDF policy is applied only if 2 customers arrive simultaneously.

- Preemptive Mode

  If a customer with an earlier deadline arrives, the current haircut is interrupted, and the new customer is served first. The unfinished customer is placed back in the queue and will be served later. This ensures that the most urgent customers are always prioritized, but can result in multiple customers receiving partial haircuts before being fully served.

4.2 Performance evaluation

The results show the difference between the preemptive and non-preemptive modes. In the former, all the customers receive their haircut within their deadline. In the latter, the second customer doesn't start his haircut as soon as he arrives, resulting in a delay and, thus, a deadline not met.

Scheduling without preemption

Customer	**Arrival Time**	**Deadline**	**Service Time**	**Start Time**	**End Time**	**Finished**
1	1	10	2	2	4	Yes
2	3	8	5	4	8	No
3	5	12	1	10	11	Yes
4	1	7	1	1	2	Yes
5	4	15	2	8	10	Yes

Table 1: Scheduling without preemption

Scheduling with preemption

Customer	**Arrival Time**	**Deadline**	**Service Time**	**Start Time**	**End Time**	**Finished**
1	1	10	2	2	9	Yes
2	3	8	5	3	8	Yes
3	5	12	1	9	10	Yes
4	1	7	1	1	2	Yes
5	4	15	2	10	11	Yes

Table 2: Scheduling with preemption

5 Statement of Work

Luigi Bartolomeo	demoDinner & Scheduler DEV, Tutorial, Report, Slides writing
Riccardo Bruno	demoBarber & Scheduler DEV, Tutorial, Report, Slides writing
Federico Civitareale	demoSmokers & Scheduler DEV, Tutorial, Report, Slides writing