Politecnico di Torino

Computer Architectures and Operating Systems

HackOSsim

Vincenzo Longo, 328843 Giuseppe Famà, 333161
Rachid Youssef Grib, 331354

20 September 2024

Abstract

The goal of this project consists in analysing and using the real time OS **FreeRTOS** exploiting the QEMU[1] simulator. In particular the primary objective of the assignment includes create a detailed tutorial for the **installation** and usage procedures, develop some **practical examples** to demonstrate the functionality of the operating system, customize the OS to implement **new memory management solutions** and finally **evaluate the performance** achieved by the newly implemented solution.

[1]It allows to emulate several types of hardware architectures.

Contents

1 Installation and Usage Procedures' Tutorial

All the steps were performed on a 64 bits personal computer with Windows 11 operating system. For the installation and the configuration of QEMU and FreeRTOS we followed the online guide at this link `https://dev.to/iotbuilders/debugging-freertos-with-qemu-in-vscode-4j52`.

1.1 FreeRTOS Distribution Download

As the first thing, the FreeRTOS repository has been downloaded from GitHub at this link `https://github.com/FreeRTOS/FreeRTOS.git`. FreeRTOS is a market-leading embedded system RTOS supporting 40+ processor architectures. Moreover is a class of RTOS that is designed to be small enough to run on a microcontroller - although its use is not limited to microcontroller applications.

1.2 QEMU Machine Emulator Download

QEMU is a machine emulator that allows to virtualize hardware types, even across different architectures. This can be very helpful for embedded development because the applications can be run against hardware targets that you may not have immediate access to.
The QEMU installer for Windows has been installed and then executed. After it has been added to the PATH environment variable. Adding programs to the `PATH` environment variable simplifies executing those programs from any location in the terminal or command line without needing to specify the full path to the executable every time.

1.3 Editor and required tools Installation

[2] For running and debugging FreeRTOS on a specific hardware emulated by QEMU an editor is needed. For this purpose, VSCode has been installed. Before the configuration of the environment the following tools are required:

- **ARM GNU Toolchain**;
- **CMake**;
- **make**. [3]

At this point, ARM GNU Compiler, CMake, and "make" installation paths have been added to the PATH environment variable.

1.4 Environment Configuration

Now everything is ready to be properly configured. Launch VSCode, select 'File > Open Folder' in the menu. Then navigate to the FreeRTOS repository that has been downloaded before and select this subfolder: `.../FreeRTOS/FreeRTOS/Demo/CORTEX_MPS2_QEMU_IAR_GCC`.

After VScode loads the demo folder, open `.vscode/launch.json` in the editor. Find the `miDebuggerPath` parameter and change the value to the path where `arm-none-eabi-gdb` is located on your machine. Finally open 'main.c' and make sure that `mainCREATE_SIMPLE_BLINKY_DEMO_ONLY` is set to **1**. This will generate only the simple blinky demo. Next, press the **Run and Debug** button from the left side panel in VSCode. Select **Launch QEMU RTOSDemo** from the dropdown at the top and press the **play** button. This will build the code, run the program, and attach the debugger. From there, you can **Continue**, **Step Over**, **Step Into**, **Step Out**, and **Stop** from the button bar. You can also add breakpoints in the code by right clicking next to the line number.

1.5 Building the Demo

To build the demo run the `make` command in the `FreeRTOS/FreeRTOS/Demo /CORTEX_MPS2_QEMU_IAR_GCC/build/gcc` directory.
A successful build creates the `elf` file `FreeRTOS/FreeRTOS/Demo/CORTEX_MPS2_QEMU_IAR_GCC/build/gcc/output/RTOSDemo.out` [4].

[2]for more details follows the `installation.md` tutorial in the `docs/` directory
[3]it was installed from a WSL terminal
[4]the building process will be automatically performed if the project is tested in VSCode as described in 1.4

2 Demo Applications

This project provides some **demo applications** with practical examples which show the main **FreeRTOS** functionalities. **QEMU** was used for virtualizing the **ARM Cortex-M3 processor**, i.e., the target architecture, and for testing all the **demo applications**.

2.1 Background

The main FreeRTOS functions are contained in the following files:

- `task.c`
- `queue.c`
- `list.c`

Before running the selected demo application, there is the need to build the project (as already seen in 1.5) in order to compile the **FreeRTOS source code** and link it with our **demo application**.

Moreover, the processor requires some additional **RTOS code** for the target architecture. This part is located in the `FreeRTOS/FreeRTOS/Source/portable /[compiler]/[architecture]` directory. In our case the *compiler* is **GCC** (**GNU Compliler Collection**) and the *architecture* is **ARM_CM3** which refers to the **ARM Cortex-M3 processor**.

Furthermore, the files which handle the **memory management** are located in the `FreeRTOS/FreeRTOS/Source/portable/MemMang` and are called `heap_x.c`. Typically `heap_4.c` is used. In our project, a revised version of `heap_4.c` will be proposed and tested in the section 3.

Another important file is the `FreeRTOSConfig.h` header file which contains all the configurations options for the **Real Time OS**.

2.2 Intro to the demo applications

The demo applications are divided into two sections in order to provide a comprehensive overview about two of the main macro-themes of FreeRTOS distribution:

- **Task Management**
- **Queue and Tasks' Synchronization**

In this project, each single demo application can be selected by properly setting the `mainCREATE_SIMPLE_DEMO` value in the `main.c` file.

2.2.1 Task Management

This subsection provides three different demo applications:

1. `mainCREATE_SIMPLE_DEMO = 1` selects the `main_three_tasks_CRUDE.c` demo application.
2. `mainCREATE_SIMPLE_DEMO = 2` selects the `main_three_tasks.c` demo application.
3. `mainCREATE_SIMPLE_DEMO = 3` selects the `main_priority.c` demo application.

Each of them aims to let the reader understanding:

- how to implement a task, create one or more instances of a task.
- how FreeRTOS allocates processing time to each task.
- how FreeRTOS scheduler works.
- how the relative priority of each task affects its behaviour.

Simple Example with CRUDE DELAY. The `main_three_tasks_CRUDE.c` application is a simple example which shows the main **FreeRTOS API functions** for **creating**, **developing** and **managing tasks** in an embedded system. The application simply consists of **three tasks**. All of them have the same priority and each one implements the same function which **prints a message** and enters in a loop with the unique functionality to delay the task (a **CRUDE DELAY**, i.e. which does not move the task in the *waiting list*).

The general behaviour of the three tasks changes by modifying the scheduler configurations in the `FreeRTOSConfig.h` file. Totally there are three possibilities:

- by setting the `configUSE_PREEMPTION` and `configUSE_TIME_SLICING`'s values respectively to `1` and `0`, the scheduler will be **preemptive**, **priority-based** with **no time slicing** (i.e., **ROUND ROBIN DISABLED**). With this configuration, when a task with higher priority enters in the ready state preempts the running task (with lower priority). Tasks with the same priority do not share **CPU time** (i.e., no round robin behaviour).
  Considering our demo application, since all the **three tasks** have the **same priority**, the **first one** that enters in the running state, takes precedence and **continues to run**.

- by setting the `configUSE_PREEMPTION` and `configUSE_TIME_SLICING`'s values respectively to `1` and `1`, the scheduler will be **preemptive**, **priority-based** with **time slicing enabled** (i.e., **ROUND ROBIN-LIKE**). With this configuration, tasks with higher priority preempts tasks with lower one. When one or more tasks have the same priority of the running one, they share **CPU time** by alternating each other when a time slice occurs.
  Considering our demo application, the three tasks **alternate each other** in an unpredictable way.

- by setting the `configUSE_PREEMPTION` and `configUSE_TIME_SLICING`'s values respectively to `0` and `x`, the scheduler will be **NOT preemptive** and **priority-based**. This means that context switching happens only when the running task explicitly moves to a **NOT RUNNING STATE** (suspended, blocked, or ended).
  Considering our demo application, since all the **three tasks** have the **same priority**, the first one that enters in the **running state**, continues to run.

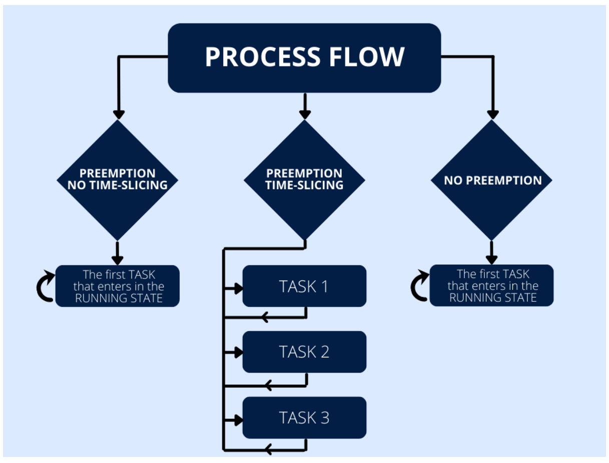
PROCESS FLOW
PREEMPTION NO TIME-SLICING
PREEMPTION TIME-SLICING
NO PREEMPTION
The first TASK that enters in the RUNNING STATE
TASK 1
TASK 2
TASK 3
The first TASK that enters in the RUNNING STATE

Figure 1: Process Flow - main_three_tasks_CRUDE.c

Example with DELAY/SLEEP FUNCTION. The `main_three_tasks.c` application is a revised version of the previous one. The unique difference consists in the **implementation of the delay**. In this case the **API function** `vTaskDelayUntil()` offered by **FreeRTOS** is used to delay the task. This function just moves the task in the **blocked state**, making room for tasks in the **ready state**

Considering our demo application, the three tasks **alternate each other** independently from the **scheduler configuration**.

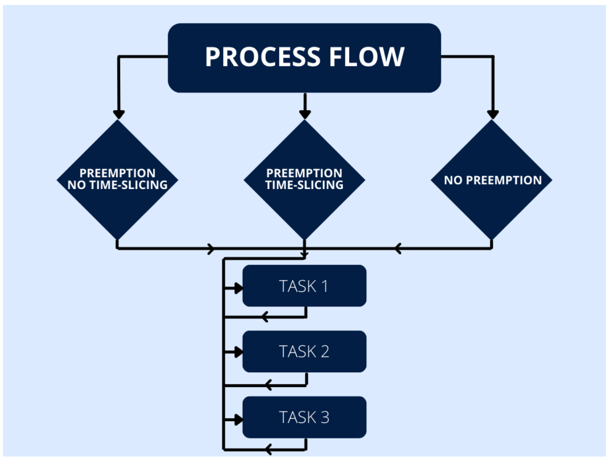

Figure 2: Process Flow - main_three_tasks.c

Example with Dynamic Priority. The `main_priority.c` application is a more advanced example which aims to show how **FreeRTOS scheduler** works by analysing the behaviour of two tasks where one of them has dynamic priority. The demo application consists of two tasks:

- the **TASK 2** with lower priority (PRIORITY = 1), which creates the second task **TASK 1** (at the beginning with PRIORITY = 2), then enters in an infinite loop wherein it prints a message.
- the **TAKS 1** which continously changes its priority, first making it equal to **TASK 2**'s one, then increasing it again and so on...

BOTH the tasks implement a **CRUDE DELAY** with a for which consumes **CPU time** (i.e., the tasks are not moved into the waiting list).

The general behaviour of the three tasks changes by modifying the scheduler configurations:

- `configUSE_PREEMPTION = 1` and `configUSE_TIME_SLICING = 0` → the scheduler will be **preemptive**, **priority-based** with **no time slicing** (i.e., ROUND ROBIN DISABLED). In this case, **TASK 2** creates **TASK 1** which preempts **TASK 2** and continues to run.
- `configUSE_PREEMPTION = 1` and `configUSE_TIME_SLICING = 1` → the scheduler will be **preemptive**, **priority-based** with **time slicing enabled** (i.e., ROUND ROBIN-LIKE). In this case, **TASK 2** creates **TASK 1** which preempts **TASK 2**. When **TASK 1** decreases its priority, the two tasks alternate each other. At a certain point **TASK 1** increases again its priority continuing to run, and the same cycle will be repeated indefinitely.
- `configUSE_PREEMPTION = 0` and `configUSE_TIME_SLICING = x` → the scheduler will be **NOT preemptive** and **priority-based**. In this case, **TASK 2** creates **TASK 1** which **cannot** preempt **TASK 2** which continues to run indefinitely.

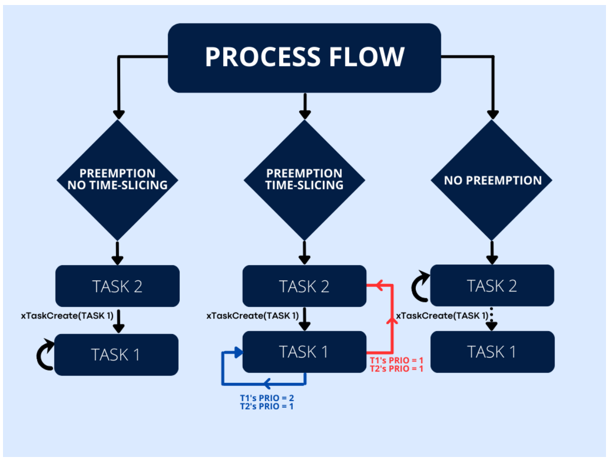

Figure 3: Process Flow - main_priority.c

2.2.2 Queue and Tasks' Synchronization

This subsection provides three different demo applications:

1. `mainCREATE_SIMPLE_DEMO = 4` selects the `main_queue.c` demo application.
2. `mainCREATE_SIMPLE_DEMO = 5` selects the `main_semaphore.c` demo application.
3. `mainCREATE_SIMPLE_DEMO = 6` selects the `main_semaphore2.c` demo application.

Each of them aims to let the reader understanding:

- how to use queues for task-to-task communication.
- how use queues and semaphores for tasks'synchronization.

Simple Queue Example . This demo application shows the main **FreeRTOS API** functions for **Queue Management**. **Queues** provide **Task-to-Task** communication mechanism. The `main_queue` creates *four tasks*:

- three **PRODUCERS** which send items in the queue;
- one **CONSUMER** which reads from the queue.

The **CONSUMER** has the highest priority, the three **PRODUCERS** have the same priority.

When the scheduler starts the **CONSUMER** task tries to read from the queue which is empty. The **CONSUMER** task will block for `100 ms` waiting for data to be available. At this point one of the **PRODUCERS** will send data to the queue, unblocking the **CONSUMER** which will read the data from the queue. The **CONSUMER** task will then block again waiting for data and the cycle will continue. In this way the **PRODUCERS** will synchronize with the **CONSUMER** by just leveraging the **FreeRTOS queue implementation**.

Sometimes if the `100 ms` elapses while the queue is still empty, the **CONSUMER** task will print an **error message**. Moreover, each task contains a **CRUDE DELAY** implementation for **demonstration purposes**. The general behaviour is depicted in the following diagram:

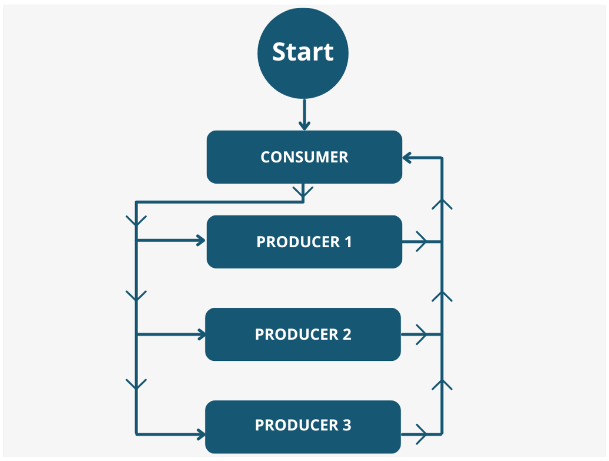

Figure 4: Precedence Diagram - Queue

Queue and Semaphores: Example 1 . This demo application shows the main **FreeRTOS** API functions for **Tasks' Synchronization** using **semaphores**. **Semaphores** are objects used to send interrupts for unblocking tasks. This results in tasks synchronization with interrupts. The `main_semaphore` creates the **queue**, the **semaphores** and **four tasks** with the **SAME priority**:

- three **PRODUCERS** which send items in the queue.
- one **CONSUMER** which reads from the queue.

Totally there are two **binary semaphores**:

- one for **PRODUCERS**.
- one for the **CONSUMER**.

At the beginning the main unblocks the **PRODUCERS**, calling `xSemaphoreGive(xSemaphoreProducer)` which start to fill the queue.
When the queue is full, the **CONSUMER** can start to read (the `xSemaphoreGive(xSemaphoreConsumer)` is called). When the **CONSUMER** reads out all the values in the **queue**, it unblocks the **PRODUCERS** and the process will be repeated again. It's clear that in yhis case the **priority** fo the tasks is not important, because the sempahores are used to syncronize the tasks. **Any priority can be used**. The general behaviour is depicted in the following diagram:

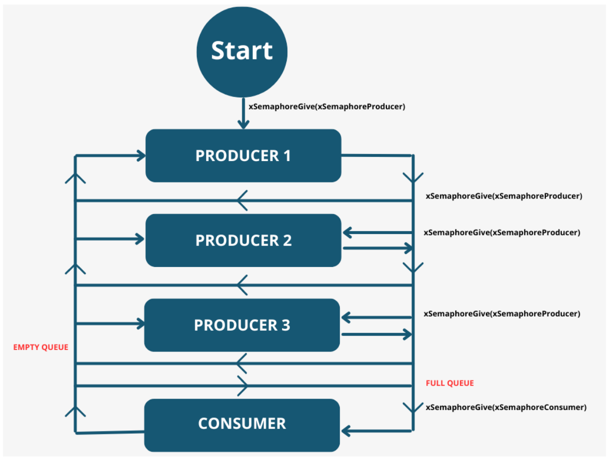
Start
xSemaphoreGive(xSemaphoreProducer)
PRODUCER 1
xSemaphoreGive(xSemaphoreProducer)
PRODUCER 2
xSemaphoreGive(xSemaphoreProducer)
PRODUCER 3
xSemaphoreGive(xSemaphoreProducer)
EMPTY QUEUE
FULL QUEUE
xSemaphoreGive(xSemaphoreConsumer)
CONSUMER

Figure 5: Precedence Diagram - Semaphores (Example 1)

Queue and Semaphores: Example 2 . This demo application is a more advanced example which shows the usage of semaphores for tasks' synchronization. The `main_semaphore2` creates the **queue**, the **semaphores** and **four tasks** with the **SAME priority**:

- three **PRODUCERS** which send items in the queue;
- one **CONSUMER** which reads from the queue.

Totally there are four **semaphores**:

- three **binary semaphores** for the three **PRODUCERS**;
- one **counting semaphore** for the **CONSUMER**.

Each **PRODUCER** inserts its value in the queue increasing the value of the counting semaphore of the **CONSUMER** (i.e, the value of `xSemaphoreConsumerCounting` represents the number of items currently available in the queue). Then it unblocks the next task as depicted in the following diagram:

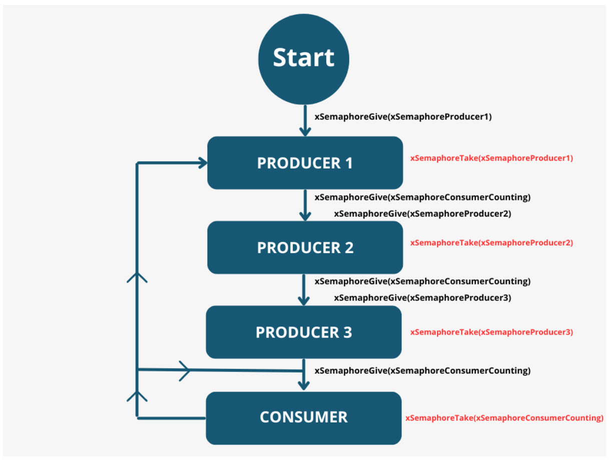
Start
xSemaphoreGive(xSemaphoreProducer1)
PRODUCER 1
xSemaphoreTake(xSemaphoreProducer1)
xSemaphoreGive(xSemaphoreConsumerCounting)
xSemaphoreGive(xSemaphoreProducer2)
PRODUCER 2
xSemaphoreTake(xSemaphoreProducer2)
xSemaphoreGive(xSemaphoreConsumerCounting)
xSemaphoreGive(xSemaphoreProducer3)
PRODUCER 3
xSemaphoreTake(xSemaphoreProducer3)
xSemaphoreGive(xSemaphoreConsumerCounting)
CONSUMER
xSemaphoreTake(xSemaphoreConsumerCounting)

Figure 6: Precedence Diagram - Semaphores (Example 2)

Overall, each **PRODUCER** inserts its value and unblock the next task; then the **CONSUMER** reads out all the values previously inserted by the **PRODUCERS** before unblocking the **first PRODUCER** and the cycle continues indefinitely. Again here the **priority** does not change the behaviour of the task.

3 Memory Management

This section introduces the implementation of some heap allocation algorithms. **FreeRTOS** already provides some implementation for dynamic allocation (`heap_x.c`). This project provides a revised version of `heap_4.c` (`heap_4_revised.c`). The standard implementation of `heap_4.c` uses by default the **first-fit** algorithm. The `heap_4_revised.c` version offers the possibility to choose an allocation algorithm among **first-fit**, **best-fit** and **worst-fit** by properly setting the `configHEAP_ALLOCATION_TYPE`'s value in the `FreeRTOSConfig.h` file.

3.1 Best-fit

Best-fit. The best fit memory allocation strategy allocates the smallest available block that is still large enough to satisfy the requested size.

```
if (configHEAP_ALLOCATION_TYPE == 1)
    configASSERT( heapSUBTRACT_WILL_UNDERFLOW( pxBlock->xBlockSize, xWantedSize ) == 0 )
        ;
    /* traverse the whole free block list */
    while( pxBlock->pxNextFreeBlock != heapPROTECT_BLOCK_POINTER( NULL ) )
    {
        /* Check if the current block is a valid option and if another valid block
           was found before and check whether is a best fit */
        if  (   ( pxBlock->xBlockSize >= xWantedSize )
                &&
                (   pxBlockTmp == NULL
                    ||
                    ( ( pxBlock->xBlockSize - xWantedSize ) < ( pxBlockTmp->xBlockSize -
                        xWantedSize ) )
                )
            )
        {
            pxPreviousBlockTmp = pxPreviousBlock;
            pxBlockTmp = pxBlock;
        }
        pxPreviousBlock = pxBlock;
        pxBlock = heapPROTECT_BLOCK_POINTER( pxBlock->pxNextFreeBlock );
        heapVALIDATE_BLOCK_POINTER( pxBlock );
    }
    pxPreviousBlock = pxPreviousBlockTmp;
    pxBlock = pxBlockTmp;
```

Listing 1: Custom implementation of Best-Fit algorithm added to heap_4.c

3.2 Worst-fit

Worst-fit. The worst fit memory allocation strategy allocates the largest available block of memory that can accommodate the requested size.

```
#elif (configHEAP_ALLOCATION_TYPE == 2)
    configASSERT( heapSUBTRACT_WILL_UNDERFLOW( pxBlock->xBlockSize, xWantedSize ) == 0 )
        ;
    /* traverse the whole free block list */
    while( pxBlock->pxNextFreeBlock != heapPROTECT_BLOCK_POINTER( NULL ) )
    {
        /* Check if the current block is a valid option and if another valid block
           was found before and check wheter is a worst fit */
        if  (   ( pxBlock->xBlockSize >= xWantedSize )
                &&
                (   pxBlockTmp == NULL
                    ||
                    ( ( pxBlock->xBlockSize - xWantedSize ) > ( pxBlockTmp->xBlockSize -
                         xWantedSize ) )
                )
            )
        {
            pxPreviousBlockTmp = pxPreviousBlock;
            pxBlockTmp = pxBlock;
        }
        pxPreviousBlock = pxBlock;
        pxBlock = heapPROTECT_BLOCK_POINTER( pxBlock->pxNextFreeBlock );
        heapVALIDATE_BLOCK_POINTER( pxBlock );
    }
    pxPreviousBlock = pxPreviousBlockTmp;
    pxBlock = pxBlockTmp;
```

Listing 2: Custom implementation of Worst-Fit algorithm added to heap_4.c

3.3 First-Fit

First-Fit. The first fit memory allocation strategy allocates the first block of memory that is large enough to satisfy the requested size.

```
#else
    while( ( pxBlock->xBlockSize < xWantedSize ) && ( pxBlock->pxNextFreeBlock !=
        heapPROTECT_BLOCK_POINTER( NULL ) ) )
    {
        pxPreviousBlock = pxBlock;
        pxBlock = heapPROTECT_BLOCK_POINTER( pxBlock->pxNextFreeBlock );
        heapVALIDATE_BLOCK_POINTER( pxBlock );
    }
#endif
```

Listing 3: Default implementation of First-fit algorithm

4 Performance Evaluation

This project provides a **simple demo application** for evaluating the behaviour of the **allocation algorithms** previously introduced. The demo application can be selected by setting the `mainCREATE_SIMPLE_DEMO` value to `7` in the `main.c` file.

4.1 Demo Application

The **main** function performs the following steps:

1. **Allocates 1000 bytes**
2. **Allocates 1000 bytes**
3. **Allocates 1500 bytes**
4. **Allocates 100 bytes**
5. **Allocates 100 bytes**
6. **Allocates 100 bytes**

7. **Allocates 100 bytes**
8. **De-allocates 1000 bytes (second block)**
9. **De-allocates 100 bytes (fourth block)**
10. **De-allocates 100 bytes (fifth block)**
11. **De-allocates 100 bytes (sixth block)**
12. **Allocates 300 bytes**
13. **Allocates 1000 bytes**
14. **Creates a task (TASK 1)**

After each operation, **the free heap space** and **the minimum ever free heap space** are recorded. If an allocation fails, the system prints an error message using the `vApplicationMallocFailedHook` function in 'main.c'.

4.2 Evaluation

For this test, the heap size (`configTOTAL_HEAP_SIZE`) in the `FreeRTOSConfig.h` file is set to **4 kB**.

4.2.1 First-Fit Algorithm

When the default implementation is selected (i.e., **First-Fit**), the allocation fails before the last block of **1000 bytes** is allocated. The output obtained is shown in Table 1.

Message	**Free Heap (B)**	**Minimum Ever Free Heap (B)**
1. Before allocating memory blocks	0	0
2. After allocated 1000 bytes	3072	3072
3. After allocated 1000 bytes	2064	2064
4. After allocated 1500 bytes	552	552
5. After allocated 100 bytes	440	440
6. After allocated 100 bytes	328	328
7. After allocated 100 bytes	216	216
8. After allocated 100 bytes	104	104
9. After deallocated the second block (1000 bytes)	1112	104
10. After deallocated the fourth block (100 bytes)	1224	104
11. After deallocated the fifth block (100 bytes)	1336	104
12. After deallocated the sixth block (100 bytes)	1448	104
13. After allocated 300 bytes	1136	104
Malloc failed..		

Table 1: Worst-Fit Allocation Algorithm Results

The **heap configuration** at **step 13** (i.e., before the failure) is the following:

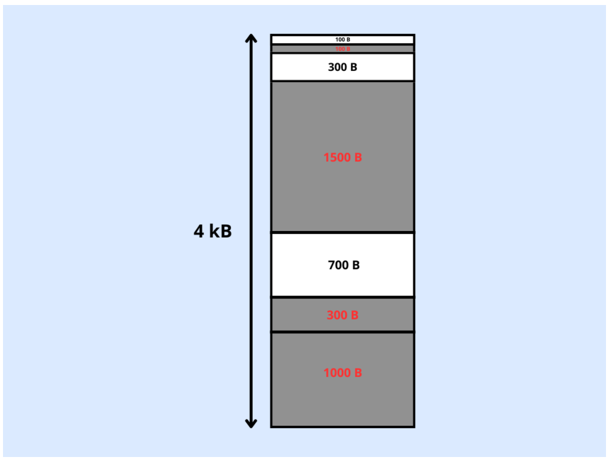

Figure 7: Heap Configuration - First-Fit

4.2.2 Best-Fit Algorithm

When the Best-Fit algorithm is used, the system fails to allocate memory for creating the task at step 14 because there is insufficient space. The output obtained is shown in Table 2.

Message	Free Heap (B)	Minimum Ever Free Heap (B)
1. Before allocating memory blocks	0	0
2. After allocated 1000 bytes	3072	3072
3. After allocated 1000 bytes	2064	2064
4. After allocated 1500 bytes	552	552
5. After allocated 100 bytes	440	440
6. After allocated 100 bytes	328	328
7. After allocated 100 bytes	216	216
8. After allocated 100 bytes	104	104
9. After deallocated the second block (1000 bytes)	1112	104
10. After deallocated the fourth block (100 bytes)	1224	104
11. After deallocated the fifth block (100 bytes)	1336	104
12. After deallocated the sixth block (100 bytes)	1448	104
13. After allocated 300 bytes	1136	104
14. After allocated 1000 bytes	128	104
Malloc failed...		

Table 2: Best-Fit Allocation Algorithm Results

At step 14, the allocation fails because there is not enough space to create the **TCB** (**Task Control Block**). The **heap configuration** at **step 14** (i.e., before the failure) is the following:

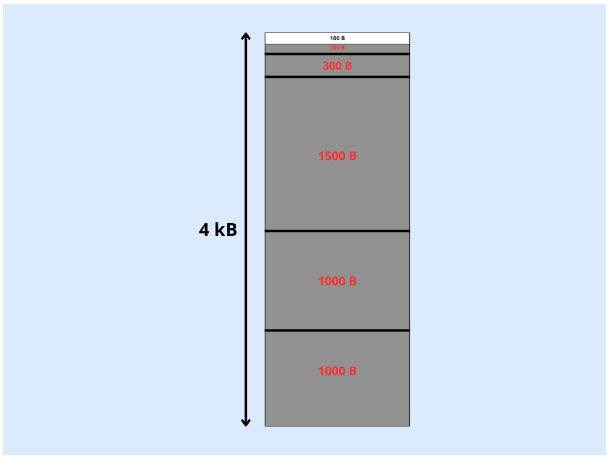

Figure 8: Heap Configuration - Best-Fit

4.2.3 Worst-Fit Algorithm

The output is the same obtained with **First-Fit** algorithm in Table 1 (i.e., in this specific example the two algorithms behave in the same way). Obviously also the heap configuration before the failure is the same 7.

5 Conclusions

Our project aims to show the use of **FreeRTOS** on **Qemu** through tutorials, exercises and modifying some solutions of the operating system. Some demo applications were developed to provide a comprehensive overview on FreeRTOS' **task management**, **queue and semaphore's usage** for **task-to-task communication and tasks' synchronization**. Next, new implementations of heap allocation algorithms were implemented by revising the `heap_4.c` file provided with the FreeRTOS distribution. Particularly, the **Best-Fit** and **Worst-Fit** algorithms are added to the **First-Fit** one, which is the default choice. Finally, a demo application was developed to evaluate the newly implemented solution. Overall, developing this project helps us to gain the main skills in **FreeRTOS programming**, to better understand **embedded systems** and **real time operating systems**.

6 Statement Of Work

The work of each member of the group is summarised below.

Member	**Work Done**
Rachid Youssef Grib	Memory Management, Queue and Tasks Synchronization
Giuseppe Famà	Installation, Queue and Tasks Synchronization
Vincenzo Longo	Installation, Tasks Management

The job was done in **presence**, **all the participants actively participated** to this project.