CA&OS Project

HaclOSsim

Rosucci Francesco s322675
Bedini Niccolò s327443
Perna Fabrizio s331806
Pinto Dario s331210
Valisheikhzahed Sajjad s329608

ACADEMIC YEAR 2023/2024

Contents

1 Point 1

In this section, we outline the steps required to install QEMU and the necessary utilities to compile and run FreeRTOS on QEMU, such as the GNU Make utility and the ARM-none-eabi-gcc compiler. The installation procedure depends on the operating system being used. We primarily worked on Arch Linux and Windows, following these steps:

Arch Linux:

1. Install QEMU by running:
   - **sudo pacman -S qemu-full**
2. Install the GNU Make utility by running:
   - **sudo pacman -S make**
3. Install the ARM-none-eabi-gcc compiler by running:
   - **sudo pacman -S arm-none-eabi-gcc**
   - **sudo pacman -S arm-none-eabi-newlib**

Windows:
There are several options depending on the development environment you want to use, such as IAR Embedded Workbench (which requires a subscription), Eclipse, or using WSL for both QEMU and the ARM compiler, following steps similar to those outlined for Arch Linux. The latter option is the one we adopted in our project by proceeding as follows:

1. Enable WSL by opening PowerShell as Administrator and running **wsl –install**, then set up a Linux distribution (Ubuntu is usually the default and was the one used for this project).
2. Once WSL with Ubuntu is set up, run the following commands:
   - **sudo apt install qemu-system-arm**
   - **sudo apt-get install libnewlib-arm-none-eabi**

Once the development environment is set up, download the latest version of FreeRTOS. FreeRTOS 202212.01 was used for this project.

2 Point 2

To verify the proper functioning of the previously downloaded operating system, we developed practical examples demonstrating the functionalities of FreeRTOS's logical flow and identified the necessary parameters to implement concepts discussed in class, such as different types of schedulers, timers, and mutexes.

2.1 Example 1: Non-Preemptive Scheduler

Initially, as the first example, we modified the default settings of the operating system to configure the non-preemptive scheduler. To achieve this, we modified the **FreeRTOSConfig.h** file by setting **configUSE_PREEMPTION** to 0. (Note: **INCLUDE_xTaskAbortDelay** also needs to be set to 0 since it can only be active in preemptive mode.) We created two tasks with different priorities. The task with the higher priority was scheduled first but then suspended itself. The second task, with lower priority, was then scheduled in its place. Despite having lower priority, it continued to hold the CPU even when Task 1 woke up and entered the ready queue, confirming the absence of preemption in the scheduling.
Code Reference: *main_noPreemption.c*

2.2 Example 2: Preemptive Scheduler

Next, we aimed to enable preemption and verify its correct execution. To achieve this, we restored the default settings in **FreeRTOSConfig.h**. In this example, we again created two tasks with different priorities. The difference this time is that Task 1 gradually increases the priority of Task 2 using the **vTaskPrioritySet()** function until both tasks have the same priority. At this point, preemption occurs. Task 2 executes when it reaches the same priority as Task 1, prints a message, and then suspends itself for a brief period. Since the preemption occured when Task 2's priority became equal to Task 1's priority, we investigated this unexpected behavior by analyzing the logical flow inside the **vTaskPrioritySet()** function:

```
if (uxNewPriority >= pxCurrentTCB->uxPriority) {
    xYieldRequired = pdTRUE;
}
```

This results in:

```
if(xYieldRequired != pdFALSE) {
    taskYIELD_IF_USING_PREEMPTION();
}
```

From this, we deduced that the initial preemption happens because this function sets the preemption flag when a task's priority becomes equal to or greater than the current task's priority. However, the default scheduler behavior only triggers preemption when a task with strictly higher priority enters the ready queue. To confirm this, we modified the condition >= to > inside **vTaskPrioritySet()**, and the result was as expected, with preemption occurring only when the second task's priority strictly surpassed that of the first. Finally, Task 1 further increases Task 2's priority, making Task 2 the highest priority task. Consequently, when Task 2 wakes up, preemption occurs, and Task 2 is executed.
Code Reference: *main_preemption.c*

2.3 Example 3: Round-Robin Scheduler

Eventually, we tested the Round-Robin (RR) scheduler for tasks of equal priority. To enable this, we added the following line in FreeRTOSConfig.h to activate the time slice used by RR:

```
#define configUSE_TIME_SLICING  1
```

The most intuitive way to experiment with this was by creating two tasks with equal priority, where each task prints a message every time it is executed. The correct functioning of RR scheduling was confirmed by observing the alternating messages from the two tasks. Since both tasks have equal priority, they share the CPU for a time slice defined by the parameter **configTICK_RATE_HZ** in **FreeRTOSConfig.h**.
Code Reference: *main_roundRobin.c*

2.4 Example 4: Mutex

In addition to the various types of schedulers, we tested the functionality of mutexes. In this example, four tasks are created, all running with the same priority. Each task utilizes a mutex to synchronize access to a shared variable. The shared variable and the mutex are encapsulated in a structure (**xMutexParameters**), which is dynamically allocated and passed to each task. The mutex (**xMutex**) ensures that only one task can access the shared variable at a time. When a task needs to access the variable, it attempts to acquire the mutex using **xSemaphoreTake**. If successful, the task can safely read and modify the shared variable. To demonstrate that the critical section is enforced, each task suspends its execution for a short period of time before releasing the mutex. During this time, the other tasks are blocked from modifying the shared variable, ensuring exclusive access. After completing its operations, the task releases the mutex with **xSemaphoreGive**, allowing other tasks to gain access. The printed output from each task shows the value of the shared variable before and after modification. This helps to track changes in the variable as tasks interact with it. If a task fails to acquire the mutex within a specified timeout period, it prints a message

indicating that it was unable to access the mutex.
Code Reference: *mutex_example.c*

3 Point 3

In this section, we describe our customization of FreeRTOS to implement two new features aimed at modifying task scheduling and ensuring fair CPU allocation: an aging-like mechanism to mitigate starvation of lower-priority tasks and a critical task flag to prioritize critical tasks among equal priority ones. This approach modifies both the scheduling logic and process management within the FreeRTOS kernel.

3.1 TCB Modifications and Tasks creation

The first modification implemented to realize our custom solutions was to modify the task control block (TCB) of the processes. We introduced two new fields: **uxAging**, a counter that tracks the aging by keeping count of the number of CPU bursts a process has executed, and **xCritical**, a flag to indicate whether the process should be considered critical or not. Subsequently, we modified the operating system primitives involved in task creation and the initialization of the TCB. FreeRTOS provides the **xTaskCreate** primitive for task creation, which we renamed **xTaskCreateModified**, adding the ability to pass an additional parameter required to initialize the **xCritical** flag to a value chosen during task creation. Inside **xTaskCreate**, the function **prvInitialiseNewTask** is invoked. This function also underwent slight modifications, with the **xCritical** flag added to its parameters and code added within the function to initialize the new TCB parameters.

To avoid altering the behavior of tasks created by the operating system itself, and thus minimizing changes to the existing codebase, we made use of macros. Specifically, the following macros

```
#define xTaskCreate( pxTaskCode, pcName, usStackDepth, pvParameters, uxPriority,
   pxCreatedTask) xTaskCreateModified( pxTaskCode, pcName, usStackDepth,
   pvParameters, uxPriority, pxCreatedTask, pdTRUE )
#define cxTaskCreate( pxTaskCode, pcName, usStackDepth, pvParameters, uxPriority,
    pxCreatedTask, xCritical ) \
    xTaskCreateModified( pxTaskCode, pcName, usStackDepth, pvParameters,
   uxPriority, pxCreatedTask, xCritical )
```

allowed us to associate the normal **xTaskCreate** call made by the operating system with **xTaskCreateModified**, with the critical flag set to true. This allowed us to distinguish between tasks created by users, using the **cxTaskCreate** primitive, and those created internally by FreeRTOS, using the **xTaskCreate** primitive, ensuring that system tasks are always classified as critical. This distinction is important because, since tasks marked as critical are not affected by the aging mechanism, it prevents FreeRTOS's internal processes from being penalized by the aging mechanism or preempted by user tasks, which could lead to undesirable behavior or even system instability. In this way, tasks generated by FreeRTOS are automatically marked as critical, ensuring they are always given the appropriate priority. In conclusion, the use of macros for task creation serves two purposes. First, it avoids the need to modify every task creation call across the codebase, seamlessly integrating the new logic without disrupting the original task scheduling mechanisms. Second, it ensures that system tasks are always treated with higher priority than user tasks when necessary, maintaining overall system stability.

3.2 Aging Mechanism

The first customization involves introducing an aging mechanism to dynamically adjust task priorities based on their CPU usage. This is particularly aimed to mitigate the issue of starvation, where lower-priority tasks might never get executed because higher-priority tasks dominate the CPU. The core idea behind this mechanism is that if a **non-critical** task continuously occupies the CPU, its aging counter increments at the end of each CPU burst, when I/O waits, interrupts, or other events occur. When this counter exceeds a predefined threshold, defined in **FreeRTOSConfig.h** with the parameter **configMAX_AGING_COUNTER**, the task's priority is decremented, effectively allowing lower-priority tasks to eventually be scheduled. Additionally, this mechanism can be enabled or disabled by modifying the **configUSE_AGING** parameter within the same configuration file where the threshold is defined. The implementation of this aging mechanism primarily involves modifying the context-switching logic in the **vTaskSwitchContext()** function. If the current task is non-critical, its aging counter is incremented each time the task is switched out. If the counter exceeds the threshold, the counter resets and the task's priority is decremented, but only if it is greater than 1, ensuring that tasks' priority never reaches IDLE task's priority.

```
if(pxCurrentTCB->xCritical == pdFALSE)
{
    pxCurrentTCB->uxAging++;
    if(pxCurrentTCB->uxAging == configMAX_AGING_COUNTER)
    {
        pxCurrentTCB->uxAging = 0;
        if(pxCurrentTCB->uxPriority > 1)
            vTaskPrioritySetCustom(pxCurrentTCB,pxCurrentTCB->uxPriority-1);
    }
}
```

This ensures that tasks with excessive CPU usage are gradually deprioritized, making space for other tasks to execute. Since FreeRTOS system tasks are always marked as critical, they are excluded from the aging mechanism to avoid unwanted behavior in core kernel functions. Another addition made to implement this mechanism is the creation of the **vTaskPrioritySetCustom** primitive. This function is derived from FreeRTOS's native **vTaskPrioritySet**, which performs all necessary operations for changing a task's priority, including modifying the TCB and moving the task to the appropriate queue. However, we removed all sections where the scheduler might be invoked if the task whose priority is being set meets certain conditions. This was done because this function is already invoked within the scheduler and is intended solely to change the task's priority without altering the scheduling

3.3 Critical tasks

The second customization we introduced is the addition of the critical task flag to the *Task Control Block (TCB)*. This flag allows us to mark certain tasks as 'critical,' meaning they should be given preference over non-critical tasks of the same priority level during scheduling. This feature is particularly useful in scenarios where critical tasks need to be prioritized to meet real-time constraints, even when competing with other tasks at the same priority. To efficiently implement this feature without adding significant overhead to the scheduling logic, we modified the FreeRTOS process queue structure by adding a critical task counter to the **xLIST** struct in **list.h** file, which represents the head of a queue of tasks, then initialized modifying the function **vListInitialise** in **list.c** file. This counter keeps track of how many critical tasks are present in a given priority queue. This allows the system to bypass unnecessary searches when there are no critical tasks in the list. Specifically, if the counter indicates that there are critical tasks, the scheduler checks for the first critical task by traversing the list until it finds one. If no critical tasks are present, the system reverts to its original FIFO logic for scheduling tasks of equal priority, simply choosing the first task in the ready list. The function **listGET_OWNER_OF_NEXT_ENTRY()**, which is invoked to choose the next task to be scheduled, was modified to check for the presence of critical tasks.:

```
if(pxConstList->xCriticalCounter>0)
{
    do{
        ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;
        if( ( void * ) ( pxConstList )->pxIndex == ( void * ) &( ( pxConstList )
    ->xListEnd ))
        {
            ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;
        }
        ( pxTCB ) = ( pxConstList )->pxIndex->pvOwner;
        }while (pxTCB->xCritical == pdFALSE);
```

The critical task counter is updated in two parts of the operating system. It is incremented when a task enters a ready queue and decremented when it is removed from the ready queue. This was achieved by modifying two functions: the first is **prvAddTaskToReadyList** defined in **tasks.c**, where a check was added to increment the **xCriticalCounter** of the list if the added task is critical; the second is **uxListRemove** defined in **list.c**, where, through functions defined by us such as **isListItemCritical** and **vDecreaseCriticalCounter**, it is checked whether the task being removed from a list is critical, and the critical counter is decremented if it is greater than 0.

4 Point 4

To evaluate the performance and correctness of the new mechanisms implemented in Point 3, we designed a test case that highlights the key features of our modifications.
The example involves the creation of a timer and four tasks: one with a higher priority and three with lower and equal priority, one of which is marked as critical. The scheduler is configured to use round-robin scheduling.
Initially, the higher-priority task (Task 1) runs but is periodically interrupted by the timer. Each time Task 1 is swapped out, its aging counter is incremented, and as a result, its priority gradually decreases, following the aging mechanism described earlier. Once Task 1's priority reaches the same level as the other tasks, the round-robin scheduling kicks in. However, due to our critical task implementation, only Task 5, the critical task, will execute, demonstrating the effectiveness of this enhancement. Task 5 takes over, disables the timer, and prints a message five times. It continues to run uninterrupted because critical tasks are unaffected by the aging mechanism. After Task 5 terminates its execution, the remaining three tasks, all with equal priority, begin alternating execution due to round-robin scheduling. Each of these tasks, upon completion of their time slice, increments their aging counter, leading to a gradual decrease in priority. Eventually, two of the three tasks will drop to a lower priority, leaving only one task to run. Since it now has the highest priority, it will continue executing without being replaced by the others, thus not triggering the aging mechanism any further.
This test clearly demonstrates the functionality of both the aging mechanism, which prevents high-priority tasks from monopolizing the CPU, and the critical task flag, which ensures that essential tasks always maintain precedence when needed.

$Task$	$Time_{original}$	$Time_{aging}$
Task1	3	3
Task3	∞	14
Task4	∞	15
Task5	∞	12

Table 1: Response time comparison

The comparison of response times in the table further supports the effectiveness of the aging mechanism in ensuring fair scheduling. While Task 1 maintains its original response time of 3 ticks, the aging mechanism allows previously neglected tasks to run, with Task 3, Task 4, and Task 5 receiving response times of 14, 15, and 12 ticks respectively, as opposed to the ∞ value, which indicates they were not executed in the original configuration.