HacIOSsim

Master's Degree in Cybersecurity - 2023/2024

Group 10

Full Name	Student ID
Alessio Mantineo	324267
Francesco Marrapodi	332062
Mikhael Russo	329165
Ming Su	330532

Teachers: Stefano Di Carlo, Tzamn Melendez, Vahid Eftekhari

Contents

1 | A brief theoretical overview

Before installation, understanding the tools and their implications is crucial. The main components discussed are Qemu, FreeRTOS, and ARM GNU Embedded

1.1 | Tools for Real-Time Operating Systems in Embedded Development

In this section, we'll delve into the details of three integral components: Qemu, its significance when used in conjunction with FreeRTOS, and the necessity of ARM GNU Embedded (*toolchain*):

- **Qemu (Quick EMUlator)**
  - A powerful open-source emulator for virtualizing different hardware and operating systems.
  - Used for software development, testing, and running multiple operating systems simultaneously.
  - Enables testing and debugging FreeRTOS-based applications without physical hardware.
  - Supports development, debugging, and portability testing.
- **FreeRTOS (Real-Time Operating System)**
  - It is an operative system designed for micro-controllers and small microprocessors.
  - It supports **non-preemptive**, **priority-based** and **Round-Robin** scheduling for efficient task execution.
  - Provides synchronization mechanisms like semaphores and mutexes for resource management.
- **ARM GNU Embedded tool-chain**
  - A collection of open-source tools for developing software for ARM-based processors.
  - Includes GCC for efficient code compilation and GDB for powerful debugging.
  - Supports both local and remote debugging on ARM processors.
  - Fosters a collaborative environment through its open-source nature

2 | Set up QUEMU, FreeRTOS and ARM GNU Embedded tool-chain to execute demos

2.1 | Installation Guide (Linux)

- Install QEMU dependencies.
- Download QEMU source code.
- Install ARM GNU Embedded tool-chain.
- Download FreeRTOS from his website.
- install Cmake.
- Start QEMU and run the FreeRTOS demo.
- Initiate GDB and connect to QEMU for comprehensive debugging.

2.2 | Installation Guide (MacOS)

- Utilize Homebrew, a package manager for simplified software installation on macOS.
- Download QEMU source code.
- Install ARM GNU tool-chain using Homebrew for efficient development.
- Download FreeRTOS from his website.
- Set up Visual Studio Code for FreeRTOS debugging using a launch configuration:
  - Customize configuration parameters, including specifying paths and server addresses.
  - Start debugging in Visual Studio Code, automating the process of launching QEMU and debugging the application.

2.3 | Installation Guide (Windows)

- install MYSYS2 following the instructions in this link.
- Download QEMU source code from MYSYS2.
- Install arm-none-eabi-gcc compiler from this link.
- Download FreeRTOS from his website.
- Add the path to your arm-none-eabi-gcc compiler, to your environment variables.
- Install Cmake.
- Now you are ready to execute your FreeRTOS demo on QEMU.

3 | Enhancing Learning Through Practical Scenarios: Applying Simulator and OS Concepts

In developing practical examples, we have chosen to create scenarios based on imaginary use case. This approach makes theoretical concepts tangible, allowing practical application and skill development.

3.1 | Resource Access Control in a Manufacturing Plant: FreeRTOS UseCase

- **Scenario Overview** In a manufacturing plant, multiple automated processes require controlled access to a critical resource like a robotic arm.
- **System Components**
  - Tasks: Represent automated processes
  - Resources: Represents the robotic arm

3.1.1 | Manage the access of various assembly task to the robotic arm through taskExample1:

- Tasks join a queue (accessQueue) for orderly resource (robotic arm) access, and release resources efficiently thanks to a mutex semaphore (resourceSemaphore).

Use Case Flow

- Task Initialization;
- Concurrent executions and Resource Access Management;
- Task Completion and System Efficiency;

3.2 | Data Management in Environmental Monitoring: FreeRTOS UseCase

Scenario: Database Access Management in an Environmental Monitoring System
Context: Imagine an environmental monitoring system with multiple sensors scattered throughout a large facility, such as an industrial plant. Each sensor collects data on various environmental parameters like temperature, humidity, pollutant levels, etc. These data are periodically sent to a central processing unit to be recorded in a database.

Use Case for taskExample2:

- Multiple sensors (tasks) send data to a central database, synchronized using a mutex for exclusive access.

Benefits of this Approach
The main benefits of this approach are:

- Efficient synchronization;

- controlled access order;
- scalability and flexibility;
- traceability;

3.3 | Security Alarm System Management in a Building: FreeRTOS Use Case

Scenario: Consider a security system in a large commercial building utilizing a network of sensors to detect suspicious activities, such as unauthorized movements or break-in attempts.

Use Case for taskExample3:

- High and Low Priority Sensors: High-priority sensors monitor critical areas, while low-priority sensors cover general areas.
- Incident Detection: When one or more sensors detect suspicious activity, they send a signal to the central system, triggering taskExample3 for each sensor..
- Event Waiting (xEventGroupWaitBits): Each taskExample3 waits for all sensors in taskExample2 to complete sending their signals, ensuring that all relevant information is collected before proceeding.
- Priority Handling: Once the signal is received that all sensors have sent their data:
  - *High-priority sensors* (e.g., those in critical areas) immediately activate advanced security procedures, such as alerting internal security or locking specific doors.
  - *Low-priority sensors* wait for all high-priority sensors to complete their operations. This may include sending additional alerts, activating alarm lights, or other less critical security measures.
- Notification and Task Conclusion: After performing the necessary operations, each taskExample3 concludes with vTaskDelete.

Benefits of this Approach:

- Effective Emergency Management: Provides a rapid and organized response to security incidents, with priority given to critical areas.
- Data Synchronization: Ensures that all relevant information is collected before activating response protocols.
- Resource Prioritization: Allows the system to allocate security resources more efficiently, focusing first on the most important areas.
- Flexibility and Scalability: Can be easily adapted to manage different security scenarios or sensor configurations.

4 | FreeRTOS customization

4.1 | implementation of Earliest Deadline First (EDF) scheduler

4.1.1 | EDF dynamic scheduling algorithm

The EDF algorithm for process scheduling is a dynamic priority scheduling algorithm that is used in real-time systems. In a dynamic priority scheduler, the tasks are executed in the order determined by their priority which is computed at run-time, also in this type of scheduler, as the name suggest, the priority assignment is dynamic, the same task may have different priorities at different time. In the EDF, the priorities are assigned to the tasks according to their absolute deadlines, and the task with the earliest deadline for execution have the highest priority.
A set of periodic tasks is schedulable with EDF if and only if the following *feasibility condition* is respected:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

Where C_i is the worst-case execution and T_i is the period of each task.

4.1.2 | EDF implementation

In order to ensure the correctness of the scheduler, the following assumptions are applied:

- Only periodic tasks are considered.
- Task deadline equals task period
- Only schedulable tasks are set.
- Independent tasks without shared resources or synchronization issues.

The implementation details of the proposed Earliest Deadline First (EDF) scheduler are as follows:

- A configuration variable, **configUSE_EDF_SCHEDULER**, is added to the FreeRTOS configuration file (FreeRTOS.h). Setting this variable to 1 enables the EDF scheduler.
- A new Ready List (**xReadyTasksListEDF**) is declared and the method *prvInitialiseTaskLists()* has been modified in order to inizialize properly this new list;
- The *prvAddTaskToReadyList()* method is modified to insert tasks into xReadyTasksListEDF based on their deadlines.
- Task structures are updated to include a **xTaskPeriod** variable to store task periods.
- A new task initialization method, *xTaskPeriodicCreate()*, is created.
- The management of the IDLE task is adjusted to ensure it remains at the lowest priority and due to this constraint, also the *vTaskStartScheduler()* method is modified.
- The context switch mechanism (*vTaskSwitchContext()*) is modified to ensure the running task is correctly updated based on the new Ready List.

4.2 | Float to Character Implementation

The provided implementation, addresses the issue of printing float values from the terminal in C. It introduces a function named *floatToChar*, which converts a floating-point number to a string.
Key points of the implementation include:

- **Handling Negative Numbers:** If the input number is negative, the function adds a minus sign ('-') to the output string and converts the number to its absolute value.
- **Extraction of Integer and Decimal Parts:** The function separates the floating-point number into its integer and decimal components. It converts the integer part to a string using *sprintf*, then proceeds to handle the decimal part separately.

- **Converting Decimal Part to String:** The decimal part is converted by multiplying it by 10 and extracting the integer part of the result successively to obtain each decimal digit. These digits are added to the string buffer, with the number of digits controlled by the precision parameter.
- **String Termination:** The resulting string is terminated with a null character to indicate the end of the string.

5 | Implementing Task Synchronization and Periodic Scheduling in FreeRTOS

5.1 | Overview

The code defines two periodic tasks, TSK_A and TSK_B, each with its own execution time (task1_C and task2_C) and period (task1_T and task2_T).
The main part of the code checks the schedulability of these tasks using isSchedulable function and, if schedulable, creates the tasks with xTaskPeriodicCreate and starts the FreeRTOS scheduler with vTaskStartScheduler.

5.2 | Main Concepts:

- Tasks: In FreeRTOS, tasks are independent threads of execution. Each task is defined by a function, such as TSK_A and TSK_B, that includes the task's code.
- Periodicity: Each task is designed to execute periodically, with its period specified by task1_T and task2_T. The period is the time interval between successive starts of the task.
- Execution Time: The execution time (task1_C and task2_C) represents how long each task takes to complete its duties before it yields control back to the scheduler.
- Schedulability Check: Before starting the tasks, the code checks if the tasks are schedulable (isSchedulable function), ensuring that the system can meet all task deadlines under the specified constraints.
- Task Creation and Scheduling: Using xTaskPeriodicCreate, each task is created with a specified priority, stack size, and period. The scheduler is then started with vTaskStartScheduler, at which point the tasks begin executing according to their periods and priorities.
- xTaskGetTickCount: This function returns the current tick count, providing a time reference for scheduling periodic activities.

5.3 | Key FreeRTOS Concepts:

- traceTASK_SWITCHED_OUT: This macro logs a message every time a task is switched out of the CPU. It uses pcTaskGetName(NULL) to get the name of the currently running task, which is being switched out, and prints a message indicating that this task has been switched out.
- traceTASK_SWITCHED_IN: Similar to the previous macro, but in this case, it logs a message every time a task is switched into the CPU. It also uses pcTaskGetName(NULL) to get the name of the task that is now running and prints a message indicating that this task has been switched in.
- traceTASK_DELAY_UNTIL: This macro logs a message when a task is put into a delay until a specified time (xTimeToWake). It's used to trace when tasks are being delayed, which is common in RTOS for managing task execution timing and ensuring tasks do not run longer than necessary. The macro also uses pcTaskGetName(NULL) to identify the task being delayed.

5.4 | Tests and Results

Each task function obtains the current tick count upon entry (xLastWakeTime = xTaskGetTickCount()) to establish a reference time for its periodic execution.

They simulate the task's execution time by delaying for task1_C or task2_C ticks, respectively.

Then, they use vTaskDelayUntil to wait until the next period, effectively pacing the task to run at its specified periodic rate.

When the EDF scheduler is enabled, the snippet defines three macros (traceTASK_SWITCHED_OUT, traceTASK_SWITCHED_IN, and traceTASK_DELAY_UNTIL) that are used for tracing task behavior. These macros make use of the printf function to log messages related to task switching and delays, as example below:

```
FEASIBLE CPU UTILIZATION: 100.0
TASK A SWITCHED IN
TASK A DELAYING UNTIL 2
TASK A SWITCHED OUT
TASK B SWITCHED IN
TASK B DELAYING UNTIL 2
TASK B SWITCHED OUT
TASK IDLE SWITCHED IN
TICK : 1
TICK : 2
TASK IDLE SWITCHED OUT
TASK A SWITCHED IN
TASK A DELAYING UNTIL 4
TASK A SWITCHED OUT
TASK B SWITCHED IN
TASK B DELAYING UNTIL 4
TASK B SWITCHED OUT
TASK IDLE SWITCHED IN
TICK : 3
TICK : 4
TASK IDLE SWITCHED OUT
```

- Task Alternation and CPU Utilization: The output shows that tasks A and B alternate in execution, respecting their WCETs of 2 time units and waiting for their next period to be executed again. This is in accordance with the CPU utilization calculation that indicates 100
- Delays Until Specific Ticks: The output shows that each task executes for a period, then puts itself on hold (”DELAYING”) until the specified tick. This pattern repeats, with tasks being scheduled to execute at their appropriate periods. This behavior is consistent with a system that uses EDF to handle periodic tasks with fixed deadlines.
- Task Idle: There are periods when the ”IDLE” task is scheduled between the execution of tasks A and B. This indicates that, after executing each task and waiting until the next deadline, the processor has no tasks to execute, correctly reflecting the periods when tasks A and B are inactive due to their scheduled waits.