CAOS report

Fabio Lorenzato, Alessandro Milani, Matteo Scursatone

April 2024

Contents

Choice of the project

Our group choose the hackOSsim assignment, because we wanted to try out real-time OS using our prior knowledge of the C programming language. Moreover, playing around with the scheduler was on our checklist of things we wanted to try out, therefore we started to work on this project. Firstly, we had to choose the Operating System for our virtual board, and after some brainstorming, we agreed to try out the FreeRTOS os. Our choice was based on different factors:

- The extensive documentation available
- A really active forum
- That one being open source, of course
- Prior knowledge of it could be useful in the future, because it's backed by Amazon

As specified by the guidelines, we had to use the QEMU emulator, so we chose a compatible board. We opted to emulate the Luminary Micro Stellaris LM3S6965EVB, which includes the following devices:

- A Cortex-M3 CPU
- 256k Flash memory and 64k SRAM
- Lots of featured and interfaces available (Timers, UARTs, ADC, I2C, SSI interfaces, . . .)
- Some graphical drivers already available

All this allowed us to work with a good amount of RAM and ROM compared to other board. Moreover, the possibility to control a display and having a lot of guided examples lead us to choose this board.

1 Setup

Firstly, it is necessary to install qemu and set up the correct embedded operating system to be emulated.
You can find the installation commands for most of the package managers here Qemu. For the FreeRTOS os version, we opted for version 202212.01, which is the latest tag on the master branch available at the time of writing this. You can download the correct build from here, FreeRTOS.
We still need to compile the binaries correctly, meaning that it is necessary to install the ARM gnu toolbox. We wrote the following snippet of code to help with that, we hope that it still works.

```
ARM_TOOLCHAIN_VERSION=$(curl -s
↪  https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads |grep -Po '<h4>Version
↪  \K.+(?=</h4>)')
curl -Lo gcc-arm-none-eabi.tar.xz
↪  "https://developer.arm.com/-/media/Files/downloads/gnu/${ARM_TOOLCHAIN_VERSION}/binrel/arm-gnu-tool
sudo mkdir /opt/gcc-arm-none-eabi
sudo tar xf gcc-arm-none-eabi.tar.xz --strip-components=1 -C /opt/gcc-arm-none-eabi
echo 'export PATH=$PATH:/opt/gcc-arm-none-eabi/bin' | sudo tee -a
↪  /etc/profile.d/gcc-arm-none-eabi.sh
source /etc/profile
rm -rf gcc-arm-none-eabi.tar.xz
```

You can verify a correct installation with the command `arm-none-eabi-gcc --version`

2 Demos

We decided to create 4 different demos to show the potentiality of the board:

1. Display Demo
2. Input Demo
3. Chaos Demo
4. Pong Demo

Each demos use the display connected to the virtual board and has different interaction with it. The first one is the simplest, and it shows some task that sequentially take control of the board, every task does not do anything, but attend with a timer and then leave the CPU and rejoin the queue.

The second one, called Input Demo, takes the input from the keyboard and then show it on the display. There are two tasks, one for the displays and one to get the input from the keyboard. The second one can send to a queue the letter to display, meanwhile the first task attends the queue to be occupied to take the letter and then show on the display.

The Chaos Demo shows an initial image that, after some seconds, will be destroyed from "worms". N tasks will be created to destroy the image, the starting position of every "worm" is pseudo-random and will eat the image translating to the right side, moreover with the 'r' or 'R' button the image will be reset.

The Pong Demo is the classic pong game, where the player and the computer play against each other to throw the ball behind the adversary bar. The program uses three tasks to do basically everything, and to separate the various logic aspects:

- one that handles all the logic(the status of the game, if the ball collides with everything, ...) and check that the state of the game is consistent, that being `game_engine_task`
- another one to handle how the state of the game is shown to the player, `graphical_engine_task`
- the last one to get the input from the player, that one being `p_input_task`

Each component handles a different aspect of the game, which allowed us to make changes without having to modify everything each time, because each component has a low coupling degree. That being said, the task communicate using two distinct message queue, one between `graphical_engine_task` and `game_engine_task`, while the other one between `p_input_task` and `game_engine_task`. The communication is unidirectional from the `game_engine_task` to the other two, with no acknowledgement mechanism implemented. The state of the game is drawn each time either a player input is received, with a small timeout afterward, or enough time is passed to assume that no input is required(70 ticks should be enough, but it depends on their frequency). You can move the player paddle with "w" and "s".

3 Commands

Starting from the folder group 29, we have to move into build fold

```
cd build
```

then we can decide which demos we would compile.

```
make all DEMO = x
```

where x is the number of the demo we choose. After that the command whichever command we choose the command to run the demo is:

```
make run
```

To clean the produced file or before choose another demo to run it is important to run

```
make clean
```

4 Scheduler Implementation

As per what concerns the required customization of the system, we decided to play around with the scheduler, as previously explained. The default scheduler implemented by FreeRTOS is designed to have a tiny footprint, as it's basically a fixed priority preemptive algorithm, which means that it will always run the highest priority task that is able to. It is also possible to make it use a round-robin scheme or turn off preemption.
Our proposed feature is a earliest deadline first algorithm, which should be optimal to minimize the lateness of a schedulable set of tasks. It is possible to enable it via the `configUSE_EDF_SCHEDULER` flag, set in the `FREERTOS.h` file. Most of the changes were done in the `tasks.c` file in the Freertos source code.
The algorithm leverages a new task list(`xReadyTasksListEDF`), in which the ready tasks will be pushed. The provided function to add a task to the queue(`vListInsert`) was also modified, which by default in the task based on the priority(at `pxReadyTaskList[task_priority]`). Now it will add the tasks TCB in the new list based on their `xStateListItem.xItemValue` value in such a way that the list is sorted in ascending order according to this value, which has been modified to contain the deadline.

The task control block(*TCB*) has been modified to hold the period, in ticks. We also had to modify the context switch mechanism, handled by the `vTaskSwitchContext` function, which changes the current task running by updating a pointer to a TCB, to always choose the head of the ready queue instead of the process with the highest priority. This works because the tasks are pushed in the ready list based on their deadline, as just explained.

In the original implementation, the need for a context switch is checked at every tick increment, which means that we also had to change this part of the logic (implemented in the `xTaskIncrementTick`) to check if the awakened task has a lower deadline than the current one.

By default, handling periodic task is not implemented, so we introduced a new task creation primitive(xTaskPeriodicCreate). Enabling the EDF scheduler flag will also disable all the task creation function, as we were unable to add compatibility with the codebase, because no function overloading is possible in C.

The idle task was also changed to be periodic. The FreeRTOS specification requires a task to be running at any instant, which means in the default scheduler it is initialized as an infinite task with the lowest priority. We kept the logic kind of the same, but to ensure that it will never run unless it's the only ready one, its default period is increased by the period of the task with the longest one each time it runs.

4.1 Scheduler Benchmarks

To evaluate the benefits of the new scheduler, a benchmark test was created. This test involves three periodic tasks executed under both scheduling algorithms: the original scheduler and the EDF (Earliest Deadline First) scheduler. For the default scheduler, all tasks were executed with the same priority. The configuration for the EDF scheduler is as follows:

X	Task Periods	Execution Time
Task 1	40	11
Task 2	8	3
Task 3	6	2

Table 1: Benchmark's Tasks

During the execution of the tasks, we went to use the functions provided by the Trace module of FreeRTOS, to obtain information regarding the execution status, specifically (in the source code are the files benchmark_edf_scheduler and benchmark_default_scheduler. The benchmark use two files for the needs to apply some change needed to make the code compatible with the schedulers):

- `tracePERIODIC_TASK_CREATE`: For get a log message at the creation of the task
- `traceTASK_SWITCHED_IN`: For get a log message at the switch in execution state of the task
- `traceTASK_SWITCHED_OUT`: For get a log message at the switch in ready state the task

Other these three the `vApplicationTickHook` was utilized to keep track of the passage of ticks.

4.1.1 Benchmark Results

Given the use of these functions, the benchmark tasks were run for a total of 1100 ticks for both schedulers, and the resulting logs were saved and analyzed via a python script to extract comparable information. In particular, from the logs where get information about:

- The idle time for each scheduler
- The number of switch in
- The lateness of each task

X	Default	EDF
Idle ticks	103	58
Switch in	1289	546
Lateness Task 1	-13.93	-6.31
Lateness Task 2	-1.47	-4.32
Lateness Task 3	-1.57	-4.0

Table 2: Results of the Log's analysis

From these results we can highlight how the use of the EDF scheduler leads to wasting much less time in idle (approximately the 56%) and less than half lost in switch in and out of the various tasks between the running state and ready (a reduction from 1289 switch in to about 546). The main cause of this difference is given by the general behavior of the default scheduler that implement a round-robin time-slicing of equal priority tasks. To finish we can also underline a reduction of the maximum lateness that went from a max of -1.47 in the default scheduler for the task 2 to a max of -4.0 for the task 3 with the use of EDF scheduler.

In conclusion, we can say that in our analysis, with the settings we used, the EDF scheduler has proven to be more effective. If one wanted to delve further into the difference between the two schedulers, a more fine-tuned use of priorities in the default scheduler or a test on multicore systems might lead to more comprehensive results, but are out of our scope and possibility.