Project Assignment

Computer Architectures and Operating Systems

Authors:
A. A. Antonini
F. Goffredo
G. Pizzenti
F. Rollo
Group 24

Professor:
Stefano Di Carlo

A.A. 2023-24

Contents

Part I
Setup Tutorial and first steps

1 Introduction

1.1 What are QEMU and Emulation?

This tutorial explains how to set up a *QEMU* to emulate a popular development board, the mps2 using an an385 FPGA (as described in the ARM AN385 Application Note). Then, we will run a Real Time OS (FreeRTOS) on the emulated environment.

In our case, an **emulator** is a **software** that makes an host system, our PC, behave like another, namely, a development board with an ARM Cortex-M3 processor on board. This is achieved in QEMU in (Full) System Emulation mode, which is used to build a virtualized version of the FPGA, on top of which we can run any OS developed for our chosen CPU.

As we can see from the FreeRTOS configuration file, the OS' internal clock is declared as 25MHz (we set it the same as the MPS2's SYSTICK), while the FreeRTOS Tick, generated from the internal clock, will produce interrupts at 0.02 MHz.

2 Installing QEMU and running the FreeRTOS demo

Now that we have laid the bases of Real Time operations, it's time to install the QEMU emulator on our system and say our Hello to the World (in our case it will be more of a friendly blink of a led, in perfect hardware-testing tradition).

2.1 Installing QEMU

Note

The following commands have been tested on Ubuntu Linux 22.10.

First of all, let's install QEMU on our system:

```
sudo apt install qemu-system
```

Then, let's download and unzip the FreeRTOS Kernel and the demo project bundled with it:

```
git clone https://github.com/FreeRTOS/FreeRTOS.git --recurse-submodules
```

Note

A lot of files in this folder are not needed for our purposes. Feel free to delete any subfolder dedicated to other platforms than the QEMU ARM Cortex-M3 (i.e. the `CORTEX_M3_MPS2_QEMU_GCC` folder).

Since we're supposedly working on a 64-bit Intel architecture, we are going to need a cross compiling toolchain to get ARM binaries from our C code. On a Debian-based Linux distribution, we can install the ARM toolchain (and a always useful debugger) with the following commands:

```
sudo apt install gcc-arm-none-eabi  // C compiler
sudo apt install gdb-arm-none-eabi  // GNU Debugger
```

Let's configure the GNU debugger to work with ARM architectures:

```
gdb-multiarch
(gdb) set architecture arm
(gdb) quit
```

2.2 QEMU and debugger use from terminal

Even if we'll default to using an IDE to give out our commands to the debugger, it's useful to know how to do it from a barebone terminal too:

```
cd <path_to_CORTEX_M3_MPS2_QEMU_GCC_folder>
make DEBUG=1
```

if the build has completed successfully, now we have a "build" folder, containing the RTOSDemo.axf file.
Now we run FreeRTOS with QEMU:

```
sudo qemu-system-arm -machine mps2-an385 -monitor null -semihosting \ --semihosting-config
    enable=on, target=native \ -kernel ./build/RTOSDemo.axf \ -serial stdio -nographic -s -S
```

Note

The `-s` option makes QEMU listen for an incoming connection on TCP port 1234; this is the port the GDB debugger will use.
The `-S` option starts the debugger in the paused state, because we need to wait for the incoming gdb connection.
The `-serial stdio -nographic` option redirects QEMU's output to the terminal we're using instead of a separate graphical window.

Once you're done, open another terminal and run the debugger on the RTOSDemo executable:

```
gdb-multiarch -q ./build/RTOSDemo.axf
```

Then set as target TCP port 1234:

```
(gdb) target remote localhost:1234
```

At this point, we can set breakpoints for whatever function we want:

```
(gdb) break main
```

Everything should be set up now. Run the FreeRTOS kernel with:

```
(gdb) c
```

To display the CPU registers' content:

```
(gdb) info reg
```

2.3 Setting up our IDE

Note

We will be using `CLion 2023.3` as our IDE.

Let's set up our IDE so that we can make use of our emulation tools while programming in a more sophisticated environment.
First, let's open CLion and open our `Demo` folder as a new project. Trust the source when you're asked and deny cleaning the project by clicking `cancel` when you're asked:

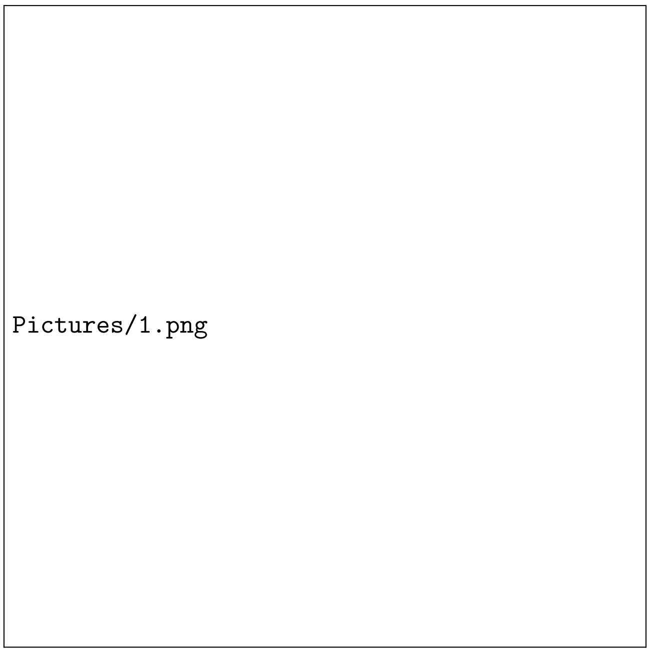

Figure 1

Now select the `main.c` file. At the top right corner of the window, click on `Add Configuration` to open the settings for running and debugging. Then, click `Add new...` and `Makefile target`.
Fill in the following parameters:

Name:	Makefile
Makefile:	the PATH of the Makefile
Arguments:	DEBUG=1

Note

You can find the Makefile in the `.../FreeRTOS/Demo/CORTEX_M3_MPS2_QEMU_GCC/Makefile` folder.

`Pictures/3.png`

Figure 2

Now click `Apply` and then `OK`.

It's time to compile: by pressing the `RUN` button (the small triangle) the IDE will automatically run the command

```
make DEBUG=1
```

which will use the commands contained in our specified Makefile to do its job. If everything went well, now we should see a folder named `Build` in the root folder of our project. Inside that, we'll find our RTOSDemo.axf file.

Note

".axf" is the extension for *ARM Executable Images*. It contains the **machine code**.

If you're reached this point, the basic setup to compile and run your code on QEMU is complete. Now, let's configure CLion to talk to the debugger.
We will now configure an **Embedded GDB Server** that will do exaclty the same things we did inside our Linux terminal before. First, start by opening the `Edit` menu and select `Edit Configurations...`. Now, click the 'plus' button (top right of the windows that has opened) and select `Embedded GDB Server`. Fill in the following fields:

Name:	FreeRTOSdemo
Executable binary:	Press on `...` and then select the `RTOSDemo.axf` file under the `Build` folder.
Debugger:	Bundled GDB multiarch.
Download executable:	*always*
'target remote' args:	tcp:localhost:1234
GDB server:	`/usr/bin/qemu-system-arm`

For the **Target** field, Press the `Settings` icon, then `Add target`, and `Make`. Fill the parameters with the configuration shown in Figure 3.

Pictures/2.png

Figure 3

For the **GDB server args** field, write: `-machine mps2-an385 -monitor null -semihosting`

`--semihosting-config enable=on,target=native -serial stdio -nographic -s -S`.

In the Before launch sub-pane, click on the `+` icon, then choose `Run another configuration`. Select the Makefile.

Very Important

Make sure that as 'Debugger' you have, next to 'Bundled GDB', **the word *multiarch* in grey** (See Figure 4). If not, select the path of the GDB we have installed previously. If you need help locating it, run the following command in a terminal:

```
which gdb-multiarch
```

(default location is `/usr/bin/gdb-multiarch`).

Pictures/1.5.png

Figure 4

Note

To get the actual path of the GDB Server, run in a terminal the command:

```
which qemu-system-arm
```

Default path for us will be `/usr/bin/qemu-system-arm`, because that's the program that will allow us to connect our Demo to the debugger.

We're done here. Click `Apply` and then `OK` to return to our code. From now on, every time we will start a debug session from CLion, the IDE will re-compile the executable code using the target Makefile, and run the latest version of the RTSDemofile.axf.

Now, let's enable full integration between the IDE and some key data structures in FreeRTOS; this will come particularly in handy because we will be able to navigate into the heap of our simulated OS. In CLion, click `File -> Settings... -> Embedded Development -> RTOS Integration` . Set the `Enable RTOS Integration` checkbox and choose `FreeRTOS` from the list of options.

Now we're really ready to start delving into any code using the debugger. Before doing so, make sure to close all other QEMU instances that you might have started before (the command `killall qemu-system-arm` in a shell should do the trick).

2.4 The FreeRTOS Demo

Let's find the file `main.c` in our demo project folder (`FreeRTOS/Demo/CORTEX_M3_MPS2_QEMU_GCC/`).
Since we want to fire up the simple "Blinky" demo, first we need to define `mainCREATE_SIMPLE_BLINKY_DEMO_ONLY` value to `1`.
Let's move on to `main_blinky.c` and take a look at what this piece of code does; basically, it demonstrates of to use a *queue* to pass a value between two tasks (`prvQueueReceiveTask()` function). To handle the succession of events of the exchange, it creates and uses a **software timer**.

Note

Queues are data structures mantained by the OS that work as "buffers"; two tasks that are supposed to exchange information will have access to the same buffer and write on there for the other to see, in a FIFO fashion (do note that the start of the queue is its tail, so if we're depicting the queue as a long "rectangle", we're reading from right to left).

When a process writes inside a queue, it will physically make a copy of the data and put it inside; this is particularly useful if we want a task to be able to get the value of one of our variables even after the variable has gone out of scope for the sender. On the other hand, choosing the correct size for a queue, neither too small nor so big that we waste too much space, can be a problem.

Queues can handle various multiprogramming setups: we can configure any number of tasks to read and/or write to a queue, although this also means we will need to take care of "traffic": what do we do if a task wants to read, but the queue is empty? And what if we want to write, but the queue is full? The programmer can specify the maximum time a task will be blocked while waiting for a change before giving up. If the situation solves itself before the timeout(e.g. another task sends to the queue the expected value), then the OS will take care of automatically moving the waiting task to the Ready state.

First, we set the frequency at which the "blinking" will happen:

```
const TickType_t xTimerPeriod = mainTIMER_SEND_FREQUENCY_MS;
```

Then, we create the queue using the `xQueueCreate` command:

```
xQueue = xQueueCreate( mainQUEUE_LENGTH, sizeof( uint32_t ) );
```

We create two tasks, one called *Rx* and the other called *TX*, to symbolise that they will be the sender and the receiver in the communication. They both have the same **priority**.

```
// creating task Rx:
xTaskCreate( prvQueueReceiveTask,
             "Rx",
             configMINIMAL_STACK_SIZE,
             NULL,
             mainQUEUE_RECEIVE_TASK_PRIORITY,
             NULL );
// we do pretty much the same for Tx.
```

We then create an auto-reloading software timer whose period is xTimerPeriod ticks; it will execute the prvQueueSendTimerCallback function everytime it expires:

```
xTimer = xTimerCreate( "Timer",
                        xTimerPeriod,
                        pdTRUE,
                        NULL,
                        prvQueueSendTimerCallback );
```

After starting the scheduler, the following events will trigger, each at its set frequency: - The TX task executes the prvQueueSendTask() function, which sends the value '100' to the xQueue queue every 200 ms. - Every two seconds, a timer goes off and the handler function that is called writes the value '200' in the xQueue. - The RX task is put in the ready state by the OS when something arrives in the queue. When it's its turn, it retrieves the message from the xQueue and prints a message.

Note

The scheduling algorithm adopted is **Round Robin** (as you can see from the `configUSE_PREEMPTION` and `configUSE_TIME_SLICING` configuration constants in the `FreeRTOSConfig.h` file). The periods are implemented with the vTaskDelayUntil() function, which puts the task in the blocked state until the specified time has expired.

Now, let's run the program. The final sequence of events is the following:

```
...
[Timer expired] Sent value to queue
[RX task] Message received from software timer
[RX task] Message received from task
[TX task] Sent value to queue
[RX task] Message received from task
[...]
[TX task] Sent value to queue
[RX task] Message received from task
[TX task] Sent value to queue
[Timer expired] Sent value to queue
[RX task] Message received from software timer
[RX task] Message received from task
...
```

Note

Do note that the milliseconds stated are only *simulated*, i.e. The code is actually expressed in timer ticks, but it also contains a formula to convert between the number of ticks that have passed and the corresponding ”real” time.

Part II
Demonstrating FreeRTOS' scheduling algorithms

3 The FreeRTOS Scheduler

The scheduling algorithm applied by FreeRTOS can be tweaked straighforwardly using the `configUSE_PREEMPTION` and `configUSE_TIME_SLICING` configuration constants defined in `FreeRTOSConfig.h`. Whatever the values chosen, the scheduling will be **priority-based**, and tasks with the same priority will follow the **Round Robin** schema.
As the names suggest, `PREEMPTION` enables the scheduler immediately replace the running task with a new one that has higher priority, while `TIME_SLICING` adds a **time quantum** to the Round Robin.

In the following pages we present a custom made program that we will run with different configurations as examples.

3.1 The program

The program we will use as example is composed of three kinds of tasks: "*server*", "*client setup*", and "*client ping*" (the latter has five different instances, each with a different IP); initially, the client setup task will contact the server to receive five IP configurations from it. After having assigned all the addresses in its pool, the server task quits and an infinite cycle of "pinging" (simulated by printing a message on screen) from the clients starts.
To implement this in its base form, each task is given a suitable priority (higher value = higher priority):

- the task that requests an IP has the highest priority (7)
- the task that receives the new IP has a low priority (5)
- the server task that sends out the configuration parameters will have a medium priority (6).
- the ping tasks have the same priority (1)

This way the requests from clients will come before the server does the first reading from the queue. The task that reads the server's reply, on the other hand, is secondary to the server's actions, so we will put it as last.
Inside every ping, we call the vTaskDelayUntil() API function; this function puts the task in the **Blocked state** until its period is over (specified as parameter). This is used to create periodic tasks with a predictable frequence of activation.

Let's focus on the pinging part. A first test, with `configUSE_PREEMPTION = 0` and `configUSE_TIME_SLICING = 0`, yields:

```
Ping sent from 10.0.1.1
Ping sent from 10.0.1.2
Ping sent from 10.0.1.3
Ping sent from 10.0.1.4
Ping sent from 10.0.1.5
[...]
Ping sent from 10.0.1.1
Ping sent from 10.0.1.2
Ping sent from 10.0.1.3
Ping sent from 10.0.1.4
Ping sent from 10.0.1.5
[...]
```

The following is the Gantt diagram (CPU clock at 20000000Hz, Tick interrupt at 20000Hz, 100 ticks of delay between two activations of the same task, for all tasks) from the experimental data:

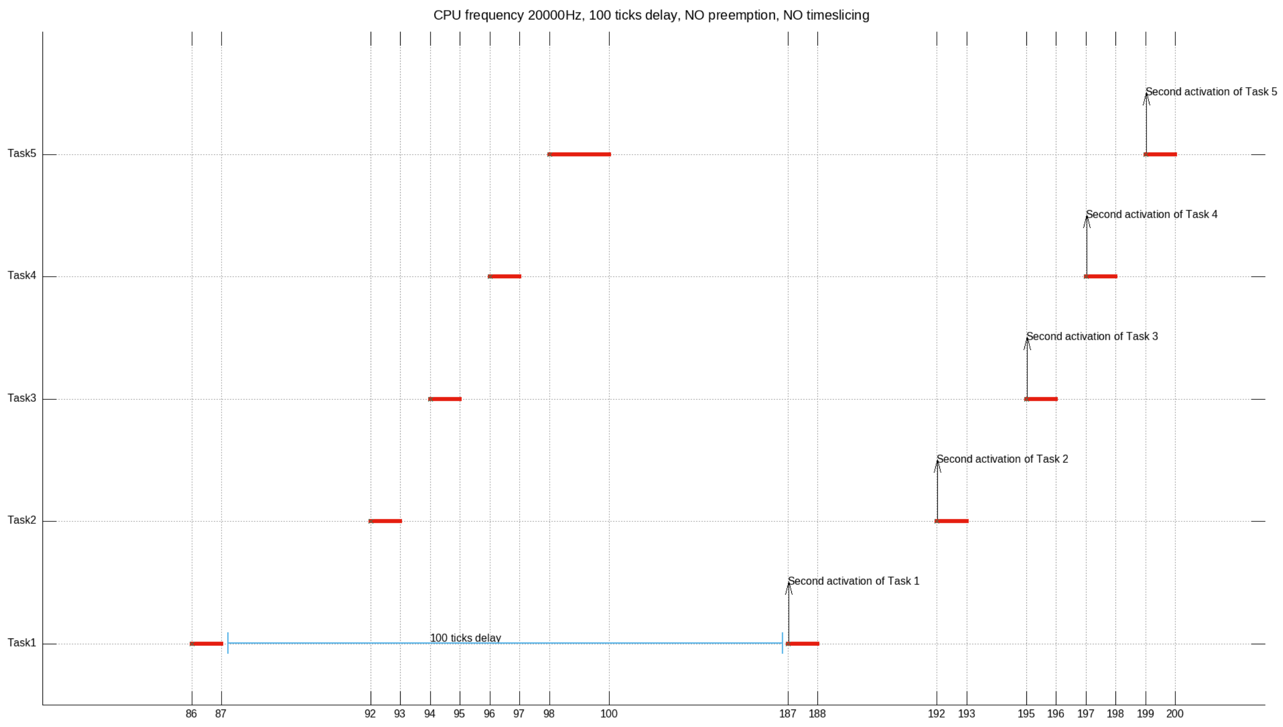

Figure 5

3.2 Adding preemption

In this other example, where we've **added preemption**, tasks 1,3,4 and 5 voluntary enter the blocked state using the vTaskDelayUntil() API function, but task (2) doesn't; instead, it begins an infinite loop with `NULL` instructions.
Since we have no set time quantum, and preemption is only applied when a task of higher priority becomes ready (we have none, since all pings have the same priority), the system remains stuck inside client 2's ping. The sequence of activations will be the following:

```
Ping sent from 10.0.1.1
Ping sent from 10.0.1.2
Ping task from 10.0.1.2 is running...
Ping task from 10.0.1.2 is running...
Ping task from 10.0.1.2 is running...
```

If we give task 5 a slighter higher priority than its peers, the following happens:

```
Ping sent from 10.0.1.5
Ping sent from 10.0.1.1
Ping sent from 10.0.1.2
Ping sent from 10.0.1.5
Ping sent from 10.0.1.3
[...]
```

During (2)'s infinite loop, (5) becomes active again. Thanks to its higher priority, the scheduler preempts in its favour, breaking 2's loop. After (5) puts itself in the blocked state again (by calling `vTaskDelayUntil()`), the CPU goes to (3) and (4), which have become ready in the meantime...

The following diagrams illustrate the succession of tasks (the numbers in the parenthesis are the content of the ready queue at that moment. Every task is considered ready at the beginning). From this picture, we can deduce FreeRTOS' behaviour with respect to preemption at ready queues:

- As (2) is preempted because an higher priority task has become ready, it goes at the end of the queue.
- As (5) becomes active (highest priority), it goes at the top of the queue.

- All the tasks except for (5) go to the end of the queue when they return ready, because they all have the same priority.

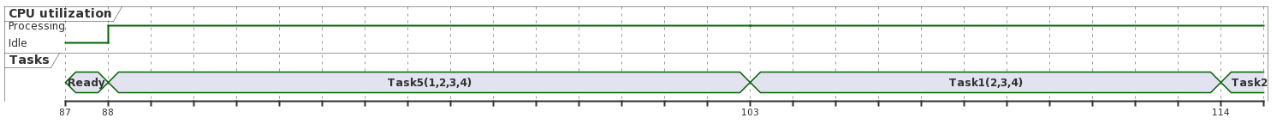

Figure 6

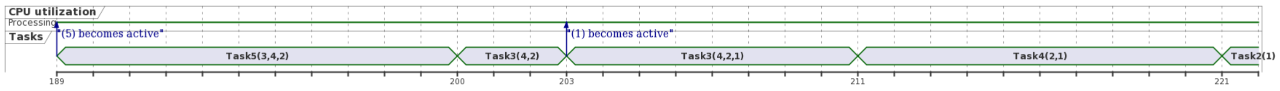

Figure 7

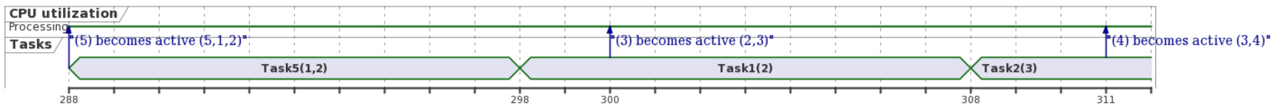

Figure 8

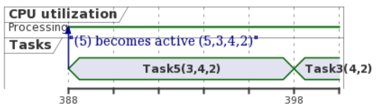

Figure 9

4 Adding variety to tasks

The example showed until now is just a taste of the many combinations of tasks that we could handle. For a more true-to-life experience, at last we add the two new types of tasks the system will handle, and a new revamped version of Ping:

- In Ping, we've added mathematical computation to the mix, to simulate the actual work the board should do to coordinate a network operation.
- The second type of task is a "ftp file transfer", where we can follow the passing of time using the CPU thanks to a "progress bar" printed to screen.
- The third task follows the example of the second one, but this time using the `wget` command.

This is the code we will run the next experiments on, but now let's see how it behaves when we add preemption and we adopt a Round Robin policy for the following set of tasks:

5 Running example

To test our new configuration, we used the following parameters (the deadline is supposed relative to the period):

Task	Priority	Deadline	Period	WCET
Ping Task	9	400	600	100
WGET	8	1200	1400	300
FTP	8	800	1200	400

A run with this parameters has given the following results:

Task	Turnaround	Waiting time
Ping	100	0
WGET	750	450
FTP	900	500

$$\left[TAT_{avg} = \frac{1760}{3} = 583ticks, \quad WT_{avg} = \frac{950}{3} = 316ticks\right]$$

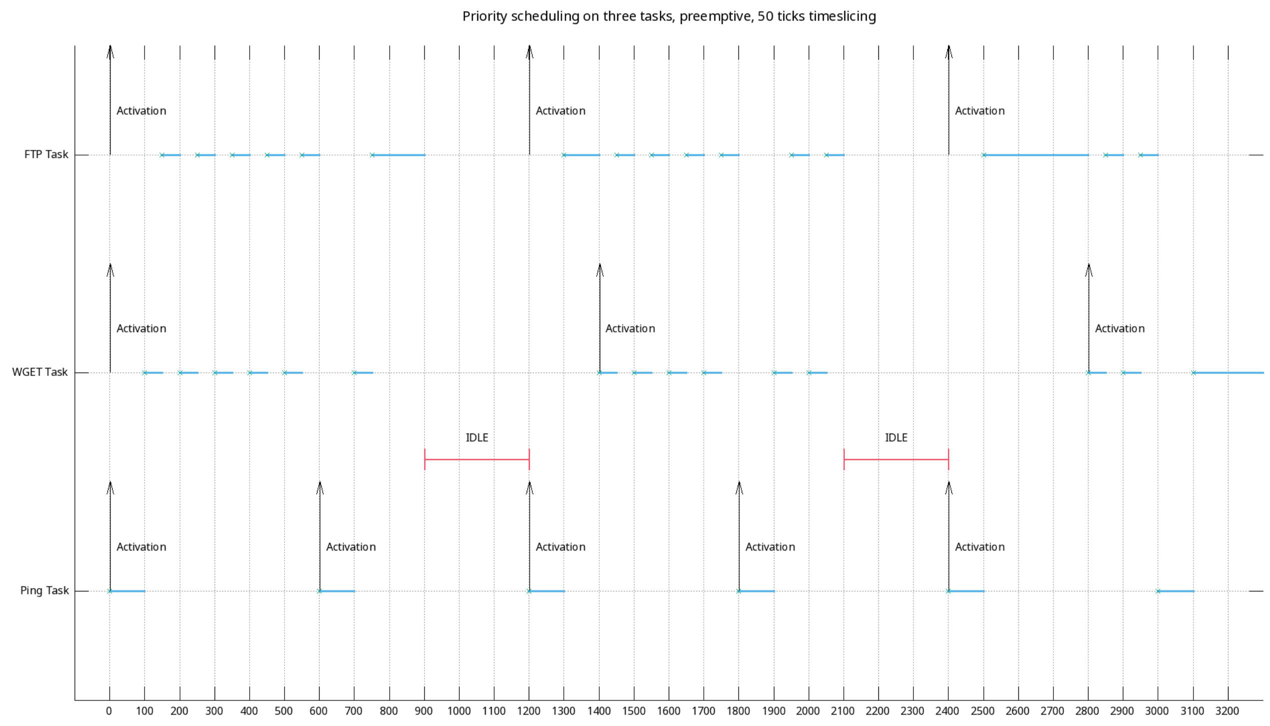

Figure 10

From the diagram we observe that FreeRTOS preempts every time the Ping task, the highest priority task, becomes active. The scheduling is feasible, so we never miss any deadline. Moreover, the WGET task and the FTP one have the same priority, and, in accordance to the Round Robin algorithm, they each get a fixed time slice before they are preempted to let the other run for the same time, until they are done with their work. Notice how the first activations of WGET and FTP are forced to resume their work after the second activation of the Ping task is over, and how the latter is able to work for more than its time quantum consecutively only because no other task is ready at the moment.

Part III
Implementing new Scheduling Algorithms into FreeRTOS

6 A first implementation of RMS

The following part shows the implementation of a new scheduling algorithm, the Rate Monotonic Scheduling, inside FreeRTOS, and the modifications that were made necessary to accommodate such change.

As we know, the Rate Monotonic scheduling algorithm schedules periodic tasks by giving them a static priority based on the **inverse of their period** (the shorter the period, the higher the priority, because it means that it will need to be ready to start again sooner than others). It's also preemptive, so if an higher-priority task becomes active while a lower one is running, the latter will get preempted.

Since FreeRTOS only deals with **integer** priorities, the code contains a function that "translates" over the $[min\ priority \rightarrow max\ priority]$ interval the actual priorities computed according to the definition, which belong to the $]0, 1[$ interval. The following pseudocode explains how the related C code works:

```
tasks[n];

// we order the tasks[] array using its members' periods (i.e. period(tasks[x]) < period(tasks
    [x+1]))
orderTasksByPeriod;

//we start assigning a priority that is one step higher than the idle task's priority, since
    that is supposed to be the lowest one.
nextPriorityToAssign = idle_priority + 1;


for(all tasks)
{
  changePriorityOfTask(nextPriorityToAssign);

  if (the period of the next task is the same as the period of the current one)
  {
    skip increasing the nextPriorityToAssign;
  }
  else
  {
    nextPriorityToAssign++;
  }
```

The assigned priorities will be something like (using example data):

period of a task (in ticks)	priority
399	1
640	2
640	2
950	3

Albeit simplicistic, this first version of RMS is working as intended; for example, if we create three Ping tasks with the following characteristics

Task	Period (ticks)	WCET (ticks)
T1	37	10
T2	43	10
T3	47	10

And we compile the Gantt diagram of the actual execution, using experimental data:

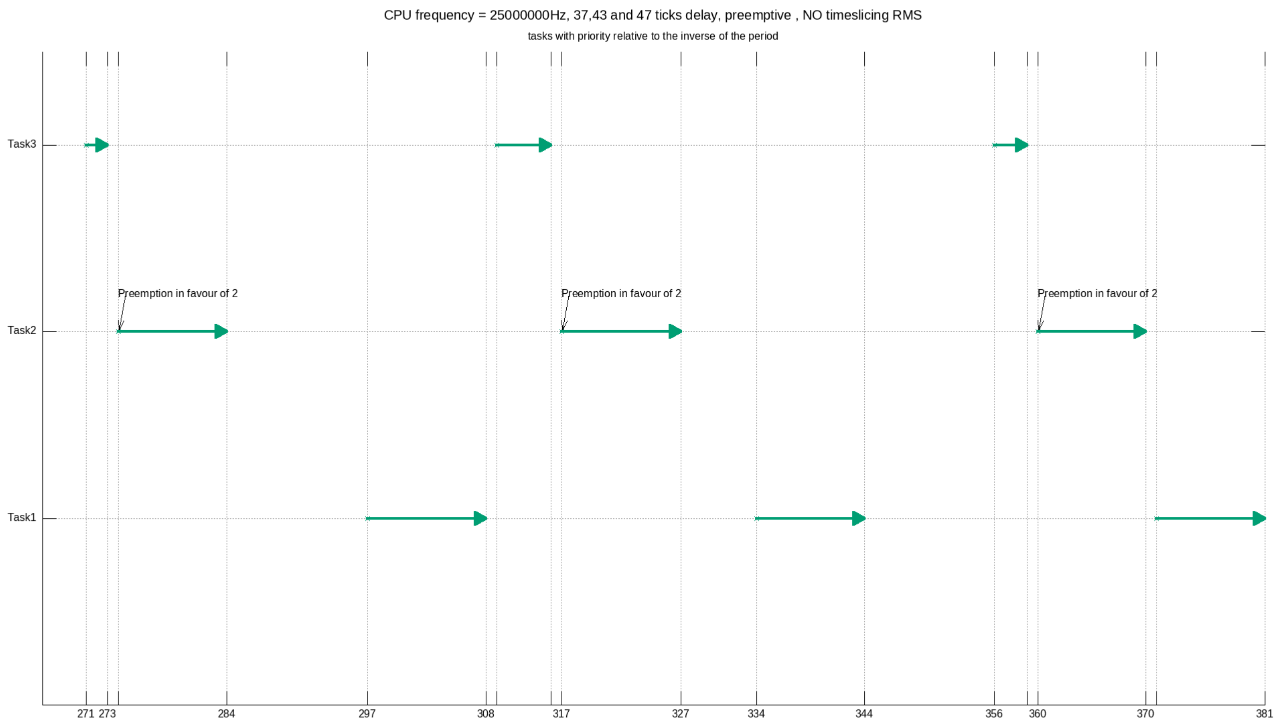

Figure 11

We can confirm that the tasks are **activating respectively every 37, 43 and 47 ticks**, and using the CPU each for 10 ticks (again, when not preempted first) before putting themselves in the blocked state. We also showed that the tasks have been **ordered according to their priority**: Task2 takes precedence on all the others because it has the shortest period, while T1 runs with the least.

Note

Task3's periods are less consistent than the other two tasks' because it's often preempted before it has the chance to set its next wake-up time using `vTaskDelayUntil()`.

7 A more elaborate RMS implementation

The last implementation tried to make do with the structures and methods of FreeRTOS by, essentially, just hijacking the function to create tasks and delaying it until we have a full view of all the tasks that are supposed to run in the system, and then call it again "under the hood" with modified parameters. After carefully studying the original `task.c` code, we've delved into creating custom OS structures such a custom Task Control Block for out RMS-controlled tasks.

Note

In our effort, we tried to remain truthful to the original developers' code style and data types, using the official Style Guide.

```
typedef struct tskCustomTaskControlBlock{

    // [...] original TCB members

    TickType_t xLastWakeTime;        // Parameter to track the last wake time of the task
```

```c
    TickType_t xArrivalTime;         // Arrival time of the task
    TickType_t xPeriod;              // Period of the task
    TickType_t xWCET;                // Worst Case Execution Time
    TickType_t xTimeSpent;           // Time elapsed since task start
    TickType_t xDeadline;            // Deadline of the task

}cTCB_t;
```

With this we will have a way, built-in into each task's core, to always keep track of critical parameters for our scheduling decisions.

```c
void vPeriodicTaskCreate( // [...] original TaskCreate parameters
                          TaskHandle_t *pxTaskHandle,
                          TickType_t xArrivalTime,
                          TickType_t xPeriod,
                          TickType_t xDeadline,
                          TickType_t xWCET
                        )
{
  // we start as normal:
    cTCB_t *pxTCB;
    pxTCB = pvPortMalloc(sizeof( cTCB_t ));
    // [...]
    // ... then we assign the new data to their field inside our custom TCB:
    pxTCB->xArrivalTime = xArrivalTime;
    pxTCB->xPeriod = xPeriod;
    pxTCB->xWCET = xWCET;
    // [...]

}
```

Now that we've created a new control structure, and we've succesfully initialized it, we need to modify how we call the scheduler to go along with the new information; to do it, we're adding a new function, `vTaskStartRealTimeScheduler()`, which will take care of the initialization of RMS-related fields and then, before exiting, it will call the canonical `vTaskStartScheduler()` to tie all out changes together with what is already a part of FreeRTOS:

```c
void vTaskStartRealTimeScheduler(){
    #if( configENABLE_RM == 1 )     //custom config parameter definition in freeRTOSconfig.h
      prvAssignPriorityRMS();
    #endif

    prvCallTaskCreate();      // use xTaskCreate to submit a task to FreeRTOS' scheduler
    xStartTime = xTaskGetTickCount();
    vTaskStartScheduler();      //start FreeRTOS' scheduler
}
```

Note

We've declared some static global variables at the top of the file (`RealTimeScheduler.c`):

```c
static TickType_t xStartTime = 0;       // Time elapsed since the scheduler started
static BaseType_t xIdleFlag = 0;        // Flag to know if Idle Task is active.

#if( configENABLE_RM == 1 )
    static List_t xTASK_List;
    static List_t *pxTASK_List = NULL;
#endif
```

Where `xTASK_List` is a list of all the custom TCBs we've created, and `*pxTASK_List` points to the beginning of said list.

At last, let's dive in depth in the most important part of the implementation, the actual priority assigning behaviour defined in `prvAssignPriorityRMS()`:

```
    static void prvAssignPriorityRMS() {
        [...]
    current priority = highest priority -1;

while( head of TCB list != tail of TCB list )
{
            priority of current TCB = current priority;
            current priority--;

            current TCB = next TCB in list;
        }
        }
```

Note

In this code we're just assigning priorities from higher to lower at each step, because the list is already sorted by ascending periods as a result of calling `prvInitialiseTCBItemRMS(cTCB_t *pxTCB)` (more specifically, the `listSET_LIST_ITEM_VALUE( &pxTCB->pxTCBItem, pxTCB->xPeriod);` line sets the period field of a TCB as the field that the sorting is done on).

FreeRTOS' lists are not "normal" C lists: they have been enhanced to handle TCB members specifically. For example, each member of a list must be given a reference to the list it is part of (its *container*).
An example of how we used such lists is inside `vInitScheduler()`: first we initialize the Task List, and then we assign a global variable with the pointer to its beginning.

```
void vInitScheduler(){
    #if( configENABLE_RM == 1)
        vListInitialise( &xTASK_List );     // Initialization of Task List
        pxTASK_List = &xTASK_List;          // Assigning the global variable
    #endif
}
```

For each custom TCB we need to initialize it as a list item and add it to it using `prvInitialiseTCBItemRMS( cTCB_t *pxTCB )` :

```
    static void prvInitialiseTCBItemRMS( cTCB_t *pxTCB ) {

        /* vListInitialiseItem initializes the container to null so the item
         * does not think that it is already contained in a list*/
        vListInitialiseItem( &pxTCB->pxTCBItem );

        /* The owner of a list item is a pointer to the object (usually a TCB) that
    holds the list item's contents */
        listSET_LIST_ITEM_OWNER( &pxTCB->pxTCBItem, pxTCB );

        listSET_LIST_ITEM_VALUE( &pxTCB->pxTCBItem, pxTCB->xPeriod);

        /* Insert the Item in the Task List */
        vListInsert( pxTASK_List, &pxTCB->pxTCBItem );
    }
```

Now, we present the content of `prvCallTaskCreate`: this function calls FreeRTOS' xTaskCreate to submit our tasks to FreeRTOS' scheduler, one by one, reading from our TCB list:

```
static void prvCallTaskCreate(){
    [...]
    while( pxTCB_Pointer != pxTCB_Tail ){
        pxTCB = listGET_LIST_ITEM_OWNER( pxTCB_Pointer );
        xTaskCreate( prvPeriodicTaskMaster, //pointer to the code to execute during the task
                     pxTCB->pcName,
                     pxTCB->ulStackDepth,
                     pxTCB->pvParameters,
                     pxTCB->xPriority,
                     pxTCB->pxTaskHandle);
        pxTCB_Pointer = listGET_NEXT( pxTCB_Pointer );
    }
}
```

Very Important

Notice that every task we're creating is technically instructed to execute **one function in all cases**: `prvPeriodicTaskMaster`. The `prvPeriodicTaskMaster` will be the core of our most complex decision taking, because it's the *wrapper function* that will take care of selecting which Task function code to run (the ping, the WGET or the FTP) every time.

```
static void prvPeriodicTaskMaster( void *pvParameters )
{
    // We retrieve the TCB of the task we want to run:
    cTCB_t *pxTask = prvGetTCBFromListByHandleRMS(xCurrentTaskHandle);

    for ( ; ; )
    {
        if (IdleFlag == 1)
        {
         printf "IDLE";
         IdleFlag = 0.
        }

        else
        {
        // Now we call the task function code specified in the TCB (WGET, Ping or FTP)
        pxTask->pxTaskCode( pvParameters );

        printf(info on the running task);
        [...]
        printf( name, length of this CPU burst  and WCET);

        putTaskInBlockState(TCB->period);
        }
    }
}
```

7.1 Running example

This is the final result for our three tasks:

Task	Period	Arrival time	WCET
Ping Task	600	0	100
WGET	1400	0	300
FTP	1200	0	400

The schedule is theoretically feasible:

$$\left(\frac{1}{6}+1\right)\cdot\left(\frac{3}{14}+1\right)\cdot\left(\frac{1}{3}+1\right)\leq 2$$

The turnaround and waiting time for each task (in ticks) is:

Task	Turnaround	Waiting time
Ping	100	0
WGET	800	600
FTP	500	100

$$\left[TAT_{avg} = \frac{1500}{3} = 500ticks, \quad WT_{avg} = \frac{700}{3} = 233ticks\right]$$

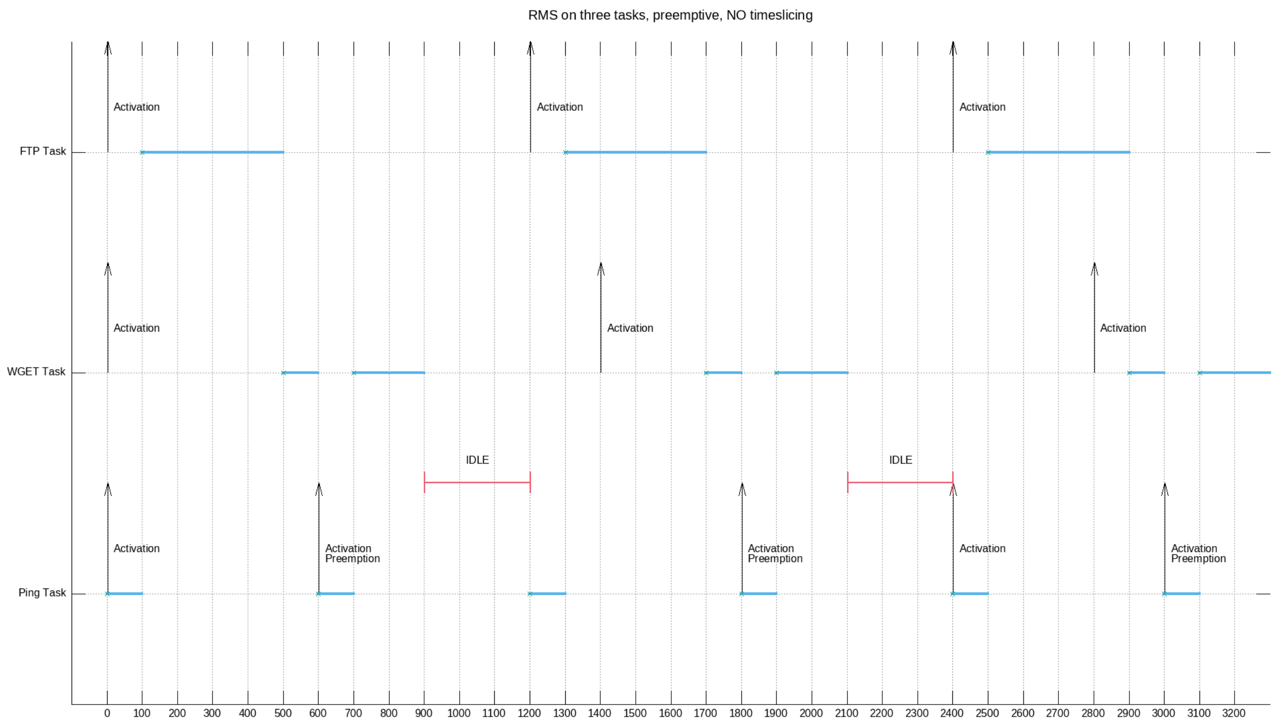

Figure 12

8 Implementing Earliest Deadline First

Earliest Deadline First is another scheduling algorithm that is particularly suited for periodic tasks. We built upon what we learned while working with RMS to implement EDF too.

The most challenging aspect of this implementation was going from computing priorities once to making **decisions every time a scheduling event occurs**, such as when a task is put in the blocked state or when a task becomes active again after some waiting. The second major problem was adding the logic behind scanning the ready queue and **finding the process closest to its deadline at any moment**.

- The Custom TCB was improved with the `TickType_t xAbsDeadline;` field, to keep the result of the kernel's computation of the current deadline for a process.
- We implemented new functions to adapt to the TCB changes, such as`prvGetTCBFromListByHandleEDF` and `prvInitialiseTCBItemEDF`.
- The number and nature of the tasks list had to be modified too: we added a temporary task list (`xTASK_List_tmp`) that will work with the original one to deal with changes to the scheduling order.
- the creation of a new periodic task has the added complication of initializing the current deadline, i.e. the system time the task will need to be completed by. We do it by adding together the task's arrival time and the set time limit (`xDeadline`):

```
void vPeriodicTaskCreate(){

    [...]
    #if( configENABLE_EDF == 1 )
    pxTCB->xAbsDeadline = pxTCB->xDeadline + pxTCB->xArrivalTime + xStartTime;
    prvInitialiseTCBItemEDF( pxTCB );
    #endif
    [...]

}
```

- The `vApplicationTickHook` function is updated to adapt to the TCB changes and the presence of the custom EDF scheduler.
- The `prvPeriodicTaskMaster` was, predictably, one of the most worked on functions. We analyze its changes in the following subsection.
- The scheduler is now a *EDF Scheduler*, which is a task that executes a function called `prvSchedulerEDFCode`:

```
static void prvSchedulerEDFCode(){
        for( ; ; ){
          // Update priorities
            prvCheckPrioritiesEDF();
            // Give the signal to unblock the next task in line
            ulTaskNotifyTake( pdTRUE, portMAX_DELAY );
        }
    }
```

8.1 Changes related to PeriodicTaskMaster

`prvPeriodicTaskMaster()` has been though major changes, such as a new `printf` that accounts for new information (the task's deadline), and the `prvNotifySchedulerEDF()` function, which we need to **notify the scheduler** that the current task is done running, so it's the right time to **update its priorities**.

```
static void prvPeriodicTaskMaster( void *pvParameters ){
    TaskHandle_t xCurrentTaskHandle = xTaskGetCurrentTaskHandle();
   [...]
    #elif( configENABLE_EDF == 1 )
        cTCB_t *pxTask = prvGetTCBFromListByHandleEDF(xCurrentTaskHandle);
    #endif

    [...]

    for ( ; ; )
    {
        [...]
        #if( configENABLE_EDF == 1)
            prvNotifySchedulerEDF();
            printf(Tick Count, Task name, last Wake Time, Abs deadline, Priority);
        #endif

        [...]

       //run the current task's code

       #if( configENABLE_EDF == 1 )
           pxTask->xAbsDeadline = pxTask->xDeadline + pxTask->xLastWakeTime + pxTask->xPeriod;
           // Notify scheduler that the task has been executed and update priorities:
            prvNotifySchedulerEDF();
       #endif

     vTaskDelayUntil( &pxTask->xLastWakeTime, pxTask->xPeriod );


    }
}
```

The `prvNotifySchedulerEDF` function is another new addition to our code. The function is used to first notify the EDF Scheduler to update the priorities, and then request a context switch to the highest priority task.

```
  static void prvNotifySchedulerEDF(){
        BaseType_t xHigherPriorityTaskWoken;
        // Notify the scheduler task to run its code again.
        vTaskNotifyGiveFromISR( xSchedulerEDFHandle, &xHigherPriorityTaskWoken );
        // Request a context switch to the higher priority task returned by the scheduler
        portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
    }
```

`prvNotifySchedulerEDF` is called inside the `PeriodTaskMaster` code after the current Ping/WGET/FTP function (`pxTaskCode`) returns; this will in turn alert the scheduler task (whose handle is `xSchedulerEDFHandle`) that we should temporarily block `PeriodTaskMaster` while we update our priorities; this is done by calling `vTaskNotifyGiveFromISR` with a flag parameter, *xHigherPriorityTaskWoken*, that is set to 'false'. After the update, the scheduler will then set the flag to 'true' again, acting like a "green semaphore" for `PeriodTaskMaster`.

8.2 Assigning priorities

The `prvSetEDF` method takes care of initializing the priorities for the first run of the EDF scheduler. The logic behind it is, obviously, giving higher priority of tasks that have a deadline closer in time. We do this by just traversing the TCB list and setting the priorities from highest to lowet value, since the list is already sorted by ascending deadlines.

```
    /* Assign priorities while traversing the Sorted List */
    static void prvSetEDF(){
        cTCB_t *pxTCB;
        [...]

        while( currentTCB_Pointer != pxTCB_Tail ){
            currentTCB = listGET_LIST_ITEM_OWNER( currentTCB_Pointer );

            currentTCB->xPriority = xHighestPriority;
            xHighestPriority--;

            currentTCB_Pointer = listGET_NEXT( pxTCB_Pointer );
        }
    }
```

Now, let's see how these priorities are **changed on the fly** based on their newly computed absolute deadlines:

```
/* Assign new priorities to tasks based on their new absolute deadlines */
    static void prvCheckPrioritiesEDF(){
        [...]

        currentTCB_Pointer = listGET_HEAD_ENTRY( pxTASK_List );
        const ListItem_t currentTCB_Tail = listGET_END_MARKER( pxTASK_List );

        while( traversing TCB list ){
           [...]

            // Sort again the list by the updated absolute deadlines
            listSET_LIST_ITEM_VALUE( currentTCB_Pointer , currentTCB->xAbsDeadline );

            // We keep a temporary list while updating the pxPrevious fields of each list
    member to make it point to the correct previous TCB in the list
            currentTCB_Pointer_tmp = currentTCB_Pointer;
            currentTCB_Pointer = listGET_NEXT( currentTCB_Pointer );
            uxListRemove( currentTCB_Pointer->pxPrevious );

      // Adding the current TCB to the list
            vListInsert( pxTASK_List_tmp, currentTCB_Pointer_tmp );
        }

        swap(currentTCBlist,tempTCBlist);

        // Update priorities based on new task order, from highest to lowest, while traversing
     the ordered TCB list

           currentPriorityToAssign = SCHEDULER_PRIORITY - 1;
        [...]
        while( traversing TCB list ){
    [...]
            currentTCB->xPriority = currentPriorityToAssign;

            // Set the new updated priority to the target task
            vTaskPrioritySet( currentTCB->pxTaskHandle, currentTCB->xPriority );

            currentPriorityToAssign--;
            [...]
        }
    }
```

9 Running example

To test our new configuration, we used the following parameters (the deadline is supposed relative to the period):

Task	Start Time	Deadline	Period	WCET
Ping Task	0	400	600	100
WGET	0	800	1200	400
FTP	0	1200	1400	300

The turnaround and waiting time for each task (in ticks) is:

Task	Turnaround	Waiting time
Ping	100	0
WGET	900	600
FTP	500	100

$$\left[TAT_{avg} = \frac{1500}{3} = 500 ticks, \quad WT_{avg} = \frac{700}{3} = 233 ticks \right]$$

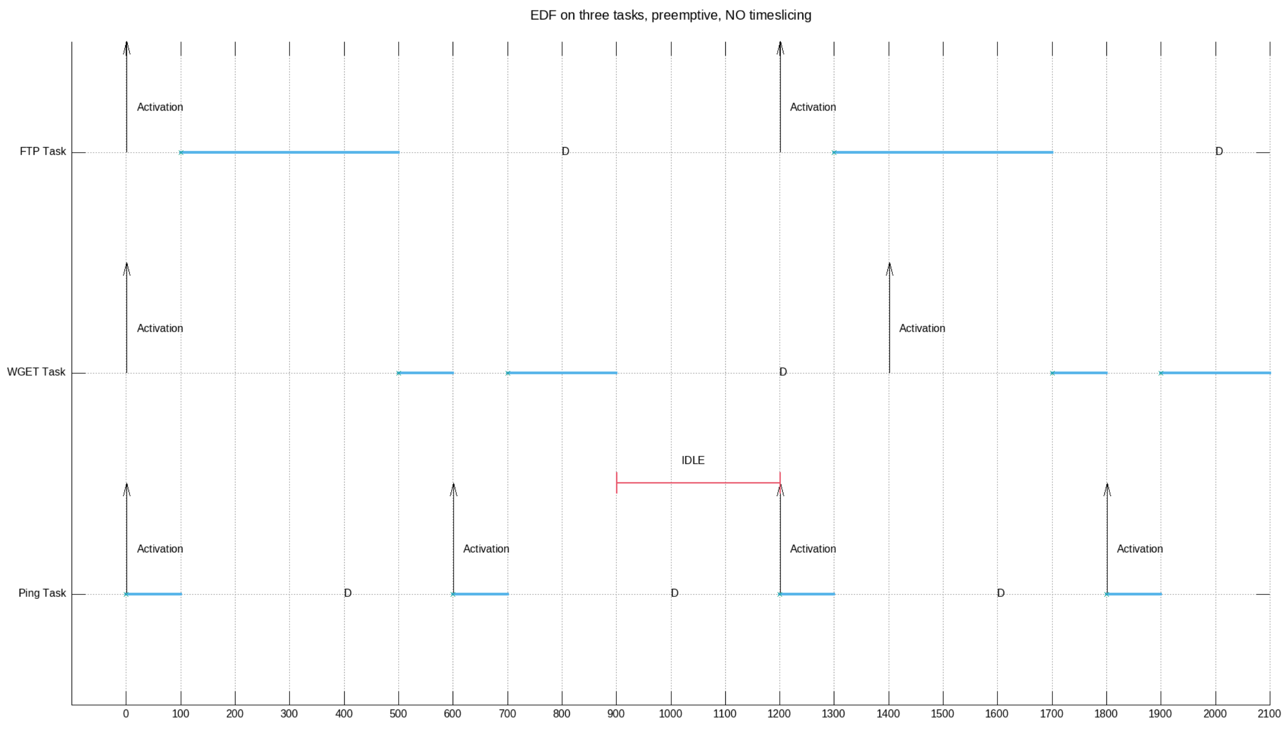
EDF on three tasks, preemptive, NO timeslicing
FTP Task
WGET Task
Ping Task
Activation
D
IDLE
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100

Figure 13

10 Adding aperiodic tasks: Polling Servers implementation

In this last section we present a solution to deal with aperiodic tasks, namely the Polling Servers algorithm. As before, we need to make changes throughout the code:

- Our custom TCB has two more fields: `xPS`, **a flag that signals that a task is a Polling Server**, and `xBudgetPS`, **the budget assigned to said Server**.

```
typedef struct tskCustomTaskControlBlock{
 [...]

    #if( configENABLE_APERIODIC == 1)
        BaseType_t xPS;              // Flag for Polling Server
        TickType_t xBudgetPS;        // Budget of Polling Server
    #endif

    [...]

}cTCB_t;
```

- We define yet another type of TCB: `tskAperiodicTaskControlBlock`, to contain all information needed for its scheduling:

```
 typedef struct tskAperiodicTaskControlBlock{
       [...]
        TickType_t xWCET;                // Worst-Case-Time-Execution
        ListItem_t pxTCBAItem;           // Item object for task list (owner)
        BaseType_t xArrival;             // Arrival Time used as key for sorting the queue
    } cTCBA_t;
```

- The custom TCB's fields are first populated by `vAperiodicTaskCreate`. Notice how, each time a new aperiodic task comes, we assign the next value of an integer counter to keep track of the relative order of arrival of this kind of tasks:

```
    void vAperiodicTaskCreate( ... ){
        xArrival++;
        [...]
        pxTCBA->xWCET = xWCET;
        pxTCBA->xArrival = xArrival;
        prvInitialiseTCBAItem( pxTCBA );
}
```

- The function that takes care of inserting the new TCB in the list, `prvInitialiseTCBAItem`, gives us an insight on **how aperiodic tasks will organize among themselves**:

```
 /* Initialize TCB Item and insert it in Aperiodic Task List */
    static void prvInitialiseTCBAItem( cTCBA_t *pxTCBA ){
        vListInitialiseItem( &pxTCBA->pxTCBAItem );
        [...]
        // The list is sorted in ascending order by arrival time (FIFO):
        listSET_LIST_ITEM_VALUE( &pxTCBA->pxTCBAItem, pxTCBA->xArrival);
        // Insert the new TCB in the Aperiodic FIFO queue:
        vListInsert( pxAperiodicTASK_List, &pxTCBA->pxTCBAItem );
    }
```

- The Real Time scheduler needs to be aware of aperiodic tasks and initialize their lists. It will also take care of creating the Polling Server task (calling the `prvPollingServerInit()` function). `prvPollingServerInit()`'s inner workings are pretty simple: it just creates a task, called Polling Server, with its relevant parameters (period and deadline, set at 800ms, and maximum budget). The Polling Server function is called `prvPollingServerCode`. We'll analyze it in the following section.

- The hook function called inside the System Tick Handler now updates the Polling Server's budget every time there's an interrupt while the Polling Server is executing:

```
void vApplicationTickHook( void )
{
    TaskHandle_t xCurrentTaskHandle = xTaskGetCurrentTaskHandle();
    #if( configENABLE_RM == 1 )
        cTCB_t *pxTask = prvGetTCBFromListByHandleRMS(xCurrentTaskHandle);
        if( pxTask != NULL && xCurrentTaskHandle != xTaskGetIdleTaskHandle())
        {
            pxTask->xTimeSpent++;
            #if(configENABLE_APERIODIC == 1)
                if(pxTask->xPS == pdTRUE)
                    pxTask->xBudgetPS--;      // Updating budget
            #endif
        }
    [...]
}
```

10.1 The Polling Server's logic: prvPollingServerCode

The following is the code that is executed every time the Polling Server takes the CPU. First, we start every execution with checking if there are aperiodic tasks that are waiting to be served. If there aren't, then there's no need for the Polling Server. We will go on with the other periodic tasks.

```
    static void prvPollingServerCode(){
        for( ; ; ){
            if( pxAperiodicTASK_List->uxNumberOfItems == 0 ){
                printf("[PS] No Aperiodic Tasks to Serve\n");
                return; //The Polling Server is done for now.
            }
            [...]
    }
```

If instead we find some aperiodic task that is waiting for execution, now it's time to do it:

```
    static void prvPollingServerCode(){
     [...]
            else
            {
                [...]

                // Get pointer to the TCB of the Polling Server
                *pxTCB = prvGetTCBFromListByHandleRMS(handle of current task);
                [...]

                // Get the first TCB from the Aperiodic FIFO queue
                pxTCBA = listGET_LIST_ITEM_OWNER(aperiodic task list head);

                // Check if the budget of the PS is enough for the WCET of that task...
                if(pxTCBA->xWCET < pxTCB->xBudgetPS){
                // ... if it is, execute the task
                    pxTCBA->pxTaskCode( pxTCBA->pvParameters );
                    printf("Task tame, current budget");
                }
                else    //... if it is not ...
                {
                 // ... reset the PS budget for next time ...
                    pxTCB->xBudgetPS = configMAX_BUDGET_PS;
                    // ... print an error message with the details ...
                    printf("Task name, new PS budget");
                // ... exit the cycle ...
                    return;
                }
                // ... and remove the offending task from the list.
                uxListRemove(pxTCBA_Pointer);
            }
        }
    }
```

10.2 Execution example

The following is an example of what the QEMU system prints to terminal when executing our program. The aperiodic tasks are represented by two back-to-back updates to the DNS' database or to the firmware of the board:

Task	Period	Deadline	WCET
Ping Task	600	400	600
100			
WGET	0	1400	1200
300			
FTP	0	1200	800
400			
Polling Server	1600	1200	100
300			
DNS	–	–	48
Firmware update	–	–	60

In this example we have prioritized serving the periodic tasks over the aperiodic ones by assigning to the Polling Server a period longer than the others.

The canonic test for RMS tells us that the scheduling is feasible:

$$\left(\frac{1}{6}+1\right)\cdot\left(\frac{3}{14}+1\right)\cdot\left(\frac{1}{3}+1\right)\cdot\left(\frac{1}{18}+1\right)\leq 2$$

Notice how at time 2200 we can only do one firmware update instead of two; this is because the polling server's budget is 100, but running two firmware updates would take 120 ticks. The second update will manage to run at the next Polling Server run, i.e. at *tick*=3300.

The Gantt diagram is:

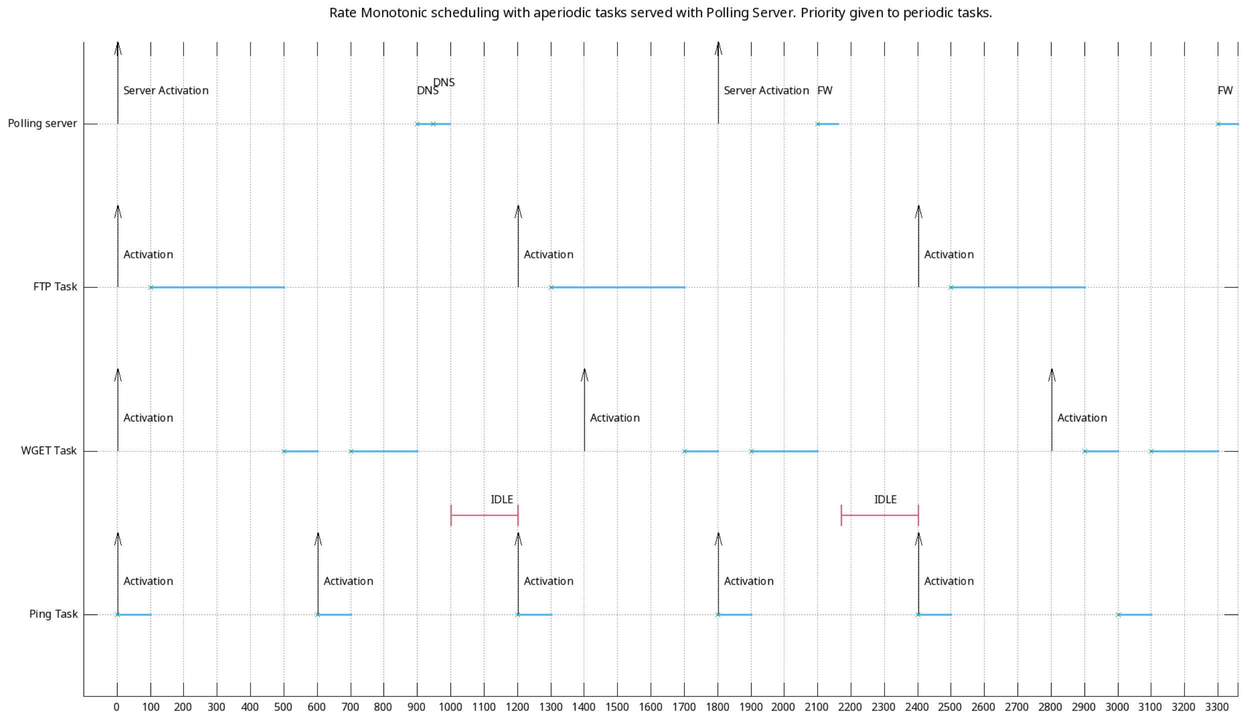

Figure 14

The output on screen will resemble this:

```
------------------------------------------------------------------------------
[START] Tick count 1 - Task Client1 - LastWakeTime 0 - Priority 9
[CLIENT] Client 1 pings: 64 Bytes
[END] Execution time 50 (WCET: 100) - Task Client1
------------------------------------------------------------------------------
[START] Tick count 51 - Task Client3 - LastWakeTime 0 - Priority 8
[CLIENT] File Transfer Status: 10%
[...]
[CLIENT] File transfer completed!
[END] Execution time 274 (WCET: 400) - Task Client3
------------------------------------------------------------------------------
[START] Tick count 325 - Task Client2 - LastWakeTime 0 - Priority 7
[CLIENT] Page status: 10%
[...]
[CLIENT] Request completed!
[END] Execution time 181 (WCET: 300) - Task Client2
------------------------------------------------------------------------------
[START] Tick count 506 - Task PS - LastWakeTime 0 - Priority 6

[SERVER] DNS Update
[PS] Aperiodic Task Server executed - Polling Server Budget = 83

[SERVER] DNS Update
[PS] Aperiodic Task Server executed - Polling Server Budget = 66
[PS] Next Aperiodic Task Server not schedulable - Reset Polling Server Budget to 100

[END] Execution time 34 (WCET: 100) - Task PS
--------*[IDLE]*--------
------------------------------------------------------------------------------
[START] Tick count 600 - Task Client1 - LastWakeTime 600 - Priority 9
[CLIENT] Client 1 pings: 64 Bytes
[END] Execution time 46 (WCET: 100) - Task Client1
--------*[IDLE]*--------
------------------------------------------------------------------------------
[START] Tick count 1200 - Task Client1 - LastWakeTime 1200 - Priority 9
[CLIENT] Client 1 pings: 64 Bytes
[END] Execution time 45 (WCET: 100) - Task Client1
------------------------------------------------------------------------------
[START] Tick count 1245 - Task Client3 - LastWakeTime 1200 - Priority 8
[CLIENT] File Transfer Status: 10%
[...]
[CLIENT] File transfer completed!
[END] Execution time 275 (WCET: 400) - Task Client3
------------------------------------------------------------------------------
[START] Tick count 1519 - Task Client2 - LastWakeTime 1400 - Priority 7
[CLIENT] Page status: 10%
[...]
[CLIENT] Request completed!
[END] Execution time 185 (WCET: 300) - Task Client2
------------------------------------------------------------------------------
[...]
```

11 Adding Feasibility Tests to our algorithms

We have taken advantage of the extensive literature on scheduling algorithms to write some tests that could help diagnose problems or just understand better what is going on in our system.
In general, static scheduling algorithms are suitable for periodic computations with hard deadlines, while dynamic algorithms are better suited for sporadic or aperiodic tasks. The scheduler will always work to determine the optimal sequence for executing computations. A feasible scheduling ensures that the timing constraints of all computations are met. Schedulability analysis hinges upon prior knowledge of the WCET for each computation.

We know that the **utilization factor U** represents the fraction of CPU time used by the computation and it is defined by the following equation:

$$U_i = \frac{C_i}{T_i}$$

Where C_i is the WCET of the computation i and T_i is the period.

11.1 Feasibility of RMS

Let $\Gamma = \tau_1,..., \tau_n$ be a set of n periodic tasks, where each task τ_i is characterized by a processor utilization U_i.

Γ is schedulable with RM if

$$\prod_{i=1}^{n}(U_i + 1) \leq 2$$

If the condition is not valid it does not mean that the algorithm does not work but we can prove that it may still. This feasibility test is also referred as a **sufficient** but not necessary test under RM schedulability.

11.2 Feasibility of EDF

Schedulability of periodic task set handled by EDF can be verified through the processor utilization factor.

A set of periodic tasks is schedulable with EDF if and only if:

$$\sum_{i=1}^{n} U_i \leq 1$$

This feasibility test is also referred as a **sufficient and necessary** test under EDF schedulability.

11.3 Adding basic feasibility tests inside our code

The following code is called inside the `vTaskStartRealTimeScheduler` function, to immediately inform the user of the theoretical feasibility of the tasks.

```
static BaseType_t prvCheckFeasibilitySTD( void ){
    #if( we use RMS )
        float xU = 1.0;        // U is CPU Utilization Factor
        float adder = 1.0;      // The constant we need inside the RMS formula
    #elif( we use EDF scheduling )
        float xU = 0.0;
    #endif
    [...]
while( traversing TCB list ){

#if( we use RMS )
  xU *= (float) currentTCB->xWCET / currentTCB->xPeriod + adder ;   // U = U*(WCETi/PERIODi)+1
#elif( we use EDF scheduling )
  xU += (float) currentTCB->xWCET / currentTCB->xPeriod;      // U = U+(WCETi/PERIODi)
#endif

        [...]
    }
// test the result against our thresholds:

    #if( we use RMS )
        if( xU > 2 ){
            return false;
        }

    #elif( we use EDF scheduling )
        if( xU > 1 ){
            return false;
        }
    #endif

    return true;
}
```

11.4 The Worst-Case Response Time approach

The utilization factor is just a non-exact test to verify if a set Γ of periodic tasks is schedulable with EDF, because if the response is non-negative, it still does not guarantee that all $\tau_i \in \Gamma$ are schedulable.

As we can see experimentally, this is the case for RMS tests too.
All of this has also been demonstrated by numerous studies, using the **Worst-Case Response Time** (**WCRT**) analysis approach [1].

The worst-case response time of a task is the length of the longest interval from a task's release until its completion. To determine the worst-case response time, we consider the so-called *critical instants*[2].

The worst case response time R_i of a task τ_i is given by the smallest R that satisfies:

$$R^0 = C_i$$

$$R^{m+i} = C_i + \sum_{j<i} \left\lceil \frac{R^m}{T_j} \right\rceil C_j$$

Assuming a critical instant at time zero, the factor $\left\lceil \frac{R^m}{T_j} \right\rceil$ gives the worst-case number of preemptions that an execution of task τ_i suffers from task τ_j in an interval $[0, n)$ with $(m < n)$. The R is the interval between the release time and the end of the execution of the computation and it defines the WCRT. The computation is schedulable when the maximum response time R_i is less than or equal to its deadline $(R_i \leq D_i)$.

Note

This approach is not only suitable for Earliest Deadline First (EDF) scheduling but also provides a necessary condition for Rate-Monotonic (RM) schedulability, where in this scenario the deadline is equal to the period.

11.4.1 Code implementation

To implement the WCRT inside our code, we first added a new variable, `TickType_t xWCRT`, to our custom TCB; then, we implemented a new function, `prvCheckFeasibilityWCRT`.
In the following excerpt we present **the iterative procedure we devised to calculate worst-case response times**, calling I the R^{m+i} quantity:

```
static BaseType_t prvCheckFeasibilityWCRT(void) {

  long int I;
  currentTCB = first TCB of list;

  [...]

  long int R = currentTCB -> xWCET;

  [...]

  if (currentTCB -> WCRT > currentTCB -> Period)
    return FALSE;

  currentTCB = next TCB in list;

  while (traversing TCB list) {
    R = R + currentTCB -> xWCET;

    while (R <= currentTCB -> xDeadline) {
      I = 0;
      [...]
      hpTCB = highest priority TCB in list;
      do {
        [...]
        I = I + (CEIL(R, hpTCB -> xPeriod) * hpTCB -> xWCET);
        hpTCB = next item in list; //get the next highest priority task
      } while (hpTCB != currentTCB);

      I = I + currentTCB -> xWCET;

      if (R == I)
        break;
      else
        R = I;
    }

    if (R > currentTCB -> xDeadline)
      return FALSE;

    currentTCB -> xWCRT = R;
    printf("Task, WCET, WCRT"));

  currentTCB = next TCB in list;
}

return TRUE;
}
```

11.4.2 Pratical example

Here we present a practical example, using three ”Client” Tasks with the following characteristics:

Task	Period	WCET	WCRT
Ping1	300	100	100
Ping2	800	300	500
Ping3	800	200	800

The standard RM feasibility test **fails**:

$$\left(\frac{1}{1}+1\right)\cdot\left(\frac{3}{8}+1\right)\cdot\left(\frac{2}{9}+1\right)\leq 2$$

is **not true**, but we still might be able to get a feasible scheduling with WCRT ... which is exactly what the tests done by our program tell us:

```
U_max > 2

STANDARD FEASIBILITY TEST NOT PASSED
```

```
Task Client1 - WCET 100 - WCRT 100
Task Client2 - WCET 300 - WCRT 500
Task Client3 - WCET 200 - WCRT 800

FEASIBILITY WCRT TEST PASSED
```

The resulting scheduling is the following:

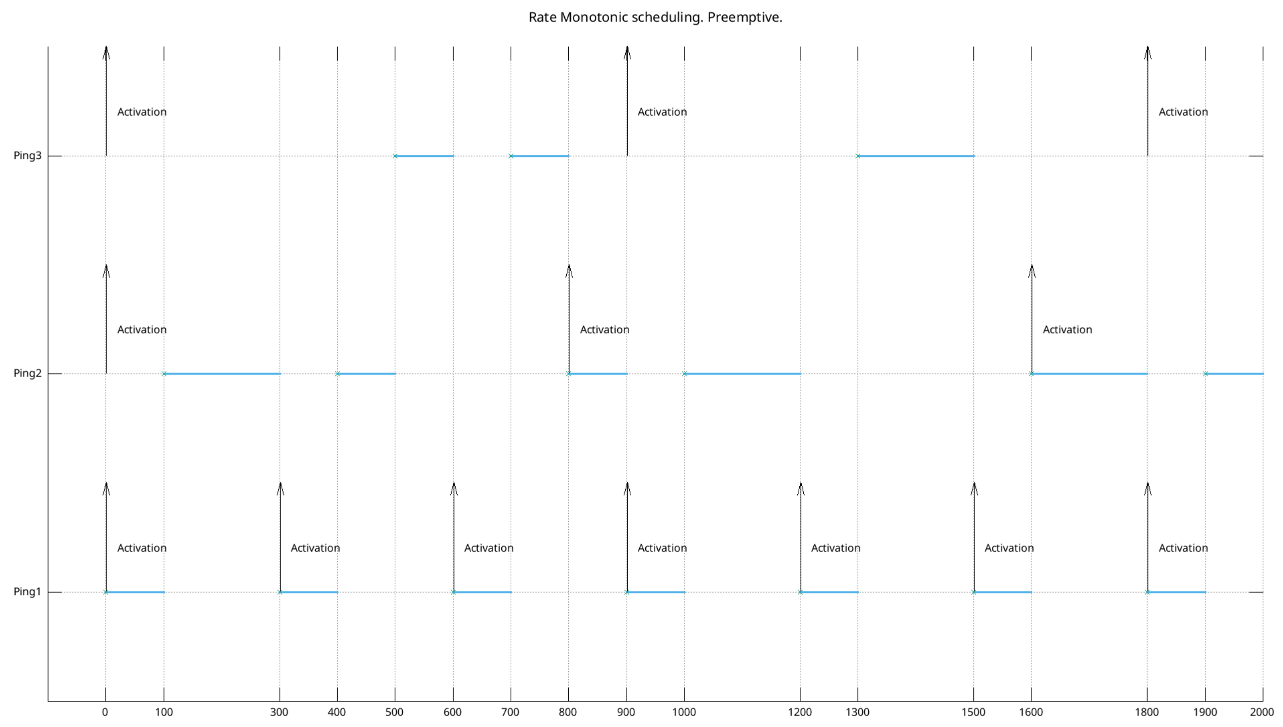

Figure 15

References

[1] M. Joseph and P. Pandya. "Finding Response Times in a Real-Time System". In: The Computer Journal 29.5 (Jan. 1986), pp. 390–395. ISSN: 0010-4620. DOI: 10.1093/comjnl/29.5.390. eprint: https://academic.oup.com/comjnl/article-pdf/29/5/390/1314410/290390.pdf. URL: https://doi.org/10.1093/comjnl/29.5.390.

[2] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: J. ACM 20.1 (Jan. 1973), pp. 46–61. ISSN: 0004-5411. DOI: 10.1145/321738.321743. URL: https://doi.org/10.1145/321738.321743.