Real Time Systems – 2016/2017

Practical Assignment Number 1

Scheduling, Priorities Assignment, and Measurement of Computation Times

Luís Afonso 2013135191 — Class PL2 Group 3 — Cristiano Alves 2009109526

Contents

1 Introduction

The work to be performed consists on the implementation of an application that makes use of the POSIX routines, with the goal of studying the scheduling by priorities assignment and the measurement of the computation times.

2 Preparation

Task	Max. C. time	Activation freq.	Activation period
T1	30ms	5hz	200ms
T2	20ms	12Hz	83.(3)ms
T3	4ms	50hz	20ms

Table 1: Exercise data

2.1 Priorities in RMPO

What are the priorities that should be assigned to the tasks (1=high, 3=low). Consider rate monotonic priority ordering (RMPO).

Priorities RMPO (1=high, 3=low):
Task 1 - priority 3
Task 2 - priority 2
Task 3 - priority 1

2.2 Response times in RMPO

What is the response time of each task. Is it possible to meet all the temporal deadlines?

Using the iterative method:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \lceil \frac{w_i^n}{T_j} \rceil C_j \qquad (1)$$
$$Until: w_i^{n+1} = w_i^n$$

Task 3 response time:

$$R_3 = 4ms \tag{2}$$

Task 2 response time:

$$\begin{aligned} w_2^0 &= 20 \\ w_2^1 &= 20 + \lceil \frac{20}{20} \rceil * 4 = 24 \\ w_2^2 &= 20 + \lceil \frac{24}{20} \rceil * 4 = 28 \\ w_2^3 &= 20 + \lceil \frac{28}{20} \rceil * 4 = 28 \\ R_2 &= 28ms \end{aligned} \tag{3}$$

Task 1 response time:

$$\begin{aligned} w_1^0 &= 30 \\ w_1^1 &= 30 + \lceil \frac{30}{20} \rceil * 4 + \lceil \frac{30}{\frac{1}{12}} \rceil * 20 = 58 \\ w_1^2 &= 30 + \lceil \frac{58}{20} \rceil * 4 + \lceil \frac{58}{\frac{1}{12}} \rceil * 20 = 62 \\ w_1^3 &= 30 + \lceil \frac{62}{20} \rceil * 4 + \lceil \frac{62}{\frac{1}{12}} \rceil * 20 = 66 \\ w_1^4 &= 30 + \lceil \frac{66}{20} \rceil * 4 + \lceil \frac{66}{\frac{1}{12}} \rceil * 20 = 66 \\ R_1 &= 66ms \end{aligned} \tag{4}$$

Task	Activation period	Max. response time
T1	200ms	66ms
T2	83.(3)ms	28ms
T3	20ms	4ms

Table 2: Tasks activation period and maximum response time

From the calculations in equations 2, 3, 4 table 2 is filled and we can see that all tasks meet the deadlines

2.3 Response time in inverse RMPO

What are the response times of each of the tasks if it is used a prioritisation scheme inverse with respect to the one used in question 2a). The tasks can meet the temporal deadlines?

Priorities inverse RMPO (1=high, 3=low):
Task 1 - priority 1
Task 2 - priority 2
Task 3 - priority 3

Using again equation 1:

Task 1 response time:

$$R_1 = 30ms \tag{5}$$

Task 2 response time:

$$\begin{aligned} w_2^0 &= 20 \\ w_2^1 &= 20 + \lceil \frac{20}{200} \rceil * 30 = 50 \\ w_2^2 &= 20 + \lceil \frac{50}{200} \rceil * 30 = 50 \\ R_2 &= 50ms \end{aligned} \tag{6}$$

Task 3 response time:

$$\begin{aligned} w_3^0 &= 4 \\ w_3^1 &= 4 + \lceil \frac{4}{200} \rceil * 30 + \lceil \frac{4}{\frac{1}{12}} \rceil * 20 = 54 \\ w_3^2 &= 4 + \lceil \frac{54}{200} \rceil * 30 + \lceil \frac{54}{\frac{1}{12}} \rceil * 20 = 54 \\ R_3 &= 54ms \end{aligned} \tag{7}$$

Task	Activation period	Max. response time
T1	200ms	30ms
T2	83.(3)ms	50ms
T3	20ms	54ms

Table 3: Tasks activation period and maximum response time

From equations 5, 6, 7 and table 3 we calculated the response times and comparing to the activation periods we see that task 3 will no meet the deadline when using this prioritization scheme.

2.4 Pseudo-code to measure computation time of a task

Write a pseudo-code to measure the computation time of a task, using POSIX routines To measure the time of a task

a good way to do it is to put it in a function, example: f1(). Then just get the time when it started and the time when it ended. For that the most precise way in posix is to use the MONOTONIC clock like so:

Code 1: Basic code to measure function execution time

```
struct timespec temp1,temp2;
clock_gettime(CLOCK_MONOTONIC, &temp1);
f1(2,3);
clock_gettime(CLOCK_MONOTONIC, &temp2);

struct timespec execution_time =
    timespec_diference(temp2, temp1);
```

Then there is a need to get the delta time by subtracting the variable "temp2" by "temp1". For the function "timespec_diference" was created:

Code 2: timespec subtraction function

```
struct timespec timespec_diference(struct
    timespec second_time, struct timespec
    first_time){

   struct timespec time_delta;

   time_delta.tv_sec = second_time.tv_sec -
       first_time.tv_sec;

   if(first_time.tv_nsec > second_time.tv_nsec){
      time_delta.tv_nsec = 1000000000 +
          second_time.tv_nsec -
          first_time.tv_nsec;
      time_delta.tv_sec--;
   }
   else{
      time_delta.tv_nsec = second_time.tv_nsec -
          first_time.tv_nsec;
   }

   return time_delta;
}
```

3 Files and directories

For each exercise there is a folder with the source code (besides exercises 2 since it's just math). Since there are plenty of common code like the timespec math, there is a folder "common_files" containing:

- func.h
- func.o
- func2.c - replic functions of func.o
- rtai_mine.c - replic of rtai functions
- rtai_mine.h
- timespec_math.c - functions for required timespec math
- timespec_math.h

For each exercise there is a makefile. The make files in each exercise's folder will be configured to compile using those files. The source files for each exercise also include the files in that folder.

```
For any exercise:

To build simply use inside the folder of the
    exercise:
$make

To execute just do
$make play

To remove all object files, executables and
    files generated by the execution of the
    program:
$make clean
```

Due to the nature of the exercises, the pairs ex1 and ex5, ex3 and ex6_1, ex4 and ex6_2 main source file is an exact copy, with the only change being the makefile which in ex6_1 and ex6_2 uses func2.c instead of the func.o provided by the teacher.

4 Pratical Implementation

4.1 Exercise 1 - Functions computation time

To measure the execution time of each function a code like code 1was used. Not only that but:

- The whole process affinity is set CPU0 with posix function *sched_setaffinity*;
- The current and future memory is locked with *mlockall(MCL_CURRENT|MCL_FUTURE)*;

Some extra coding was added to get the average execution time as well the worst execution time. For this 2 more timespec math custom functions we're created, "timespec_sum" and "timespec_division_by_int"

```
struct timespec temp1,temp2, sum = {0,0},
    worst_time = {0, 0};
for(int i = 0; i < NUMBER_OF_AVERAGES; i++){
   clock_gettime(CLOCK_MONOTONIC, &temp1);
   f1(2,3);
   clock_gettime(CLOCK_MONOTONIC, &temp2);
   struct timespec delta_time =
       timespec_diference(temp2, temp1);

   struct timespec temp =
       timespec_diference(worst_time,
       delta_time);
   if(temp.tv_sec <0 || temp.tv_nsec < 0){
   worst_time = delta_time;
   }

   sum = timespec_sum(sum, delta_time);
}
```

```
struct timespec execution_average =
    timespec_division_by_int(sum,
    NUMBER_OF_AVERAGES);
```

Giving these average execution times:

function	time (ms)
f1	38
f2	58
f3	88

Table 4: Functions computation time in milliseconds.

The worst execution times we're always the same milliseconds except in some cases. In my tiny atom core, they sometimes gave diferences of up to 7ms if I had too much stuff running so I asume it was a problem with my small core and that the execution times are suposed to be very stable around the average.

4.2 Exercise 2 - Calculate response times

For this point the objective is to determine the response time of each task when running in **RMPO** and in **Inverse RMPO** to later check in practical implementation these values.

4.2.1 RMPO ordering

Tasks	period (ms)	priority
f1	100	1
f2	200	2
f3	300	3

Table 5: Tasks activation periods and priority (lowest number = highest priority)

Task 1 response time:

$$R_1 = 38ms \tag{8}$$

Task 2 response time:

$$\begin{aligned} w_2^0 &= 58 \\ w_2^1 &= 58 + \lceil \frac{58}{100} \rceil * 38 = 96 \\ w_2^2 &= 58 + \lceil \frac{96}{100} \rceil * 38 = 96 \\ R_2 &= 96ms \end{aligned} \tag{9}$$

Task 3 response time:

$$\begin{aligned} w_3^0 &= 88 \\ w_3^1 &= 88 + \lceil \frac{88}{100} \rceil * 38 + \lceil \frac{88}{200} \rceil * 58 = 184 \\ w_3^2 &= 88 + \lceil \frac{184}{100} \rceil * 38 + \lceil \frac{184}{200} \rceil * 58 = 222 \\ w_3^3 &= 88 + \lceil \frac{222}{100} \rceil * 38 + \lceil \frac{222}{200} \rceil * 58 = 318 \\ w_3^4 &= 88 + \lceil \frac{318}{100} \rceil * 38 + \lceil \frac{318}{200} \rceil * 58 = 356 \\ w_3^5 &= 88 + \lceil \frac{356}{100} \rceil * 38 + \lceil \frac{356}{200} \rceil * 58 = 356 \\ R_3 &= 356ms \end{aligned} \tag{10}$$

With the calculations in equation 10, we see that task 3 response time is higher than the activation period. Since we want all tasks to meet the deadlines in RMPO, task 3 activation period was raised to 360ms.

Table 6 sumarizes the tasks activation period and priority

Tasks	period (ms)	priority
f1	100	1
f2	200	2
f3	360	3

Table 6: Tasks new activation periods and priority (lowest number = highest priority)

Using equation 1 again:

Response times in RMPO for each task will still be:

$$\boxed{R_1 = 38ms \quad R_2 = 96ms \quad R_3 = 356ms} \tag{11}$$

4.2.2 Inverse RMPO ordering

With the same process as before (with task 3's new activation period set at 360ms):

Tasks	period (ms)	priority
f1	100	3
f2	200	2
f3	360	1

Table 7: Tasks activation periods and priority (lowest number = highest priority)

Task 3 response time:

$$R_3 = 88ms \tag{12}$$

Task 2 response time:

$$\begin{aligned} w_2^0 &= 58 \\ w_2^1 &= 58 + \lceil \frac{58}{360} \rceil * 88 = 146 \\ w_2^2 &= 58 + \lceil \frac{146}{360} \rceil * 88 = 146 \\ R_2 &= 146ms \end{aligned} \tag{13}$$

Task 1 response time:

$$\begin{aligned} w_1^0 &= 38 \\ w_1^1 &= 38 + \lceil \frac{38}{360} \rceil * 88 + \lceil \frac{38}{200} \rceil * 58 = 184 \\ w_1^1 &= 38 + \lceil \frac{184}{360} \rceil * 88 + \lceil \frac{184}{200} \rceil * 58 = 184 \\ R_1 &= 184ms \end{aligned} \tag{14}$$

4.3 Exercise 3 - tasks running in RMPO

To make this test:

- Each function is placed in a thread;
- The 3 threads CPU afinity is set to CPU0. This is done using a Linux function because the closest posix function could only set the affinity for the entire process plus, in the MAN page, it was advised to use "*pthread_attr_setaffinity_np*";
- The current and future memory is locked with *mlockall(MCL_CURRENT|MCL_FUTURE)*;
- The scheduling policy is set to FIFO in the 3 threads. Task 1, 2 and 3 get respectively the priorities max, max-1 and max-2. The main thread just sleeps for 5 seconds until it cancels all other threads and reports the results;
- To make sure the 3 tasks start simultaneously, the first activation time of each thread is set to the same time (2 seconds after the main thread started), and all threads wait at the start. Example for task1:

```
//wait so that all tasks activate at the
    same time
clock_nanosleep(CLOCK_MONOTONIC,
    TIMER_ABSTIME, &activation_time[0],
    NULL);
```

The next tests using threads will always be with these configs unless, otherwise noted.

Results for exercise 3:

Tasks	Worst response time
T1	38ms
T2	96ms
T3	356ms

Table 8: Worst response time obtained when running in RMPO

Like calculated in equations 8, 9 and 10, the worst response times we're 38ms for task 1, 96ms for task 2 and 356ms for task 3.

4.4 Exercise 4 - switching priorities during execution time

When running in **IRMPO**, from the calculations made in equations 12, 13 and 14, we can see that Task 3 and 2, should meet the deadlines while task 1 response time should be higher than the activation period (100ms). When running in RMPO the behavior should be the same as in exercise 3.

Very important: the priority switch must always be done by the highest priority tasks and it should only lower it's priority after changing all the others. For this test the switch occurs periodically every 5 seconds.

The testing backs up those numbers with worst response time in IRMPO being:

Tasks	Worst response time
T1	184ms
T2	146ms
T3	88ms

Table 9: Worst response time obtained when running in Inverse RMPO

But the testing also showed that sometimes when running in RMPO, after switching from IRMPO to RMPO, the worst response times we're different from exercise 3.

The worst response time of task 1 and 2 were sometimes over their activation times (100ms and 200ms) during RMPO execution.

To see better what's happening, it was added some logging of various times into arrays, and then the values we're dumped into a file, in MatLab friendly format that allowed to analyze the data in MatLab. These we're the results:

In this graph the vertical lines mean:

- Orange -> switch from RMPO to IRMPO
- Black -> switch from IRMPO to RMPO
- Black -> simultaneous activation times of the 3 tasks
- The light blue dot -> when the task should be activated
- Red/Green/Blue dot -> when the task ended.
- The Y value of the dots has no meaning. The only purpose is to know to with which measure they are related too. Light blue dots with the same Y as a red/green/blue dot are pairs.

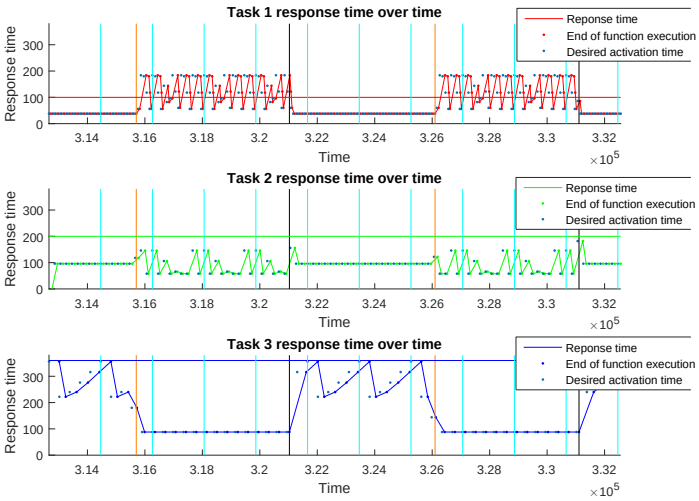

Figure 1: Graphs with ex4 exported data

From the graph we can see how the response time varied depending of the conditions.

The worst response times we're indeed when there was a simultaneous activation of the 3 tasks.

It's also possible to see when looking at the first finished execution of each task, when the switch from IRMPO to RMPO happened, that the execution time of task 1 and task 2 was particularly high. Task 1 actually took more than 100ms to finish. Although not shown in this data, Task 2 in some tests also went over the deadline, taking over 200ms to execute.

This can be explained by the activation time happening just before the priority switch. In IRMPO Task 3 caused big interference in Task 1 and 2. When the switch to RMPO occurred Task 1 and 2 ran with less interference but had already accumulated time. After that first execution, both tasks ran as expected in RMPO.

This graph can also be used to conclude that:

- **in RMPO:**
  - Task 1 ran without interference, making the response time constant at 38ms, the same as the computation time.
  - Task 2 ran with interference and the response time was always calculated 96ms which means there was actually no jitter (at least bigger than 1ms)
  - Task 3 ran with interference from both task 1 and 2 and although the response time never went over the calculated 356ms, there was lots of jitter and the lowest response time was just above 200ms

- **in inverse RMPO:**
  - Task 3 in this case was the highest priority so it ran without any interference, making the response time constant at 88ms, the same as the computation time
  - Task 2 response time was never over 146ms as the calculations predicted. But in this prioritisation scheme, there was lots of jitter in the response time, oscillating from low values like 58ms up until the worst response of 146ms. A repeating pattern can be seen in the response times.
  - Task 1 was as expected the most affected, with the worst response time being over the activation period of 100ms, reaching up to 184ms. But there we're actually lots of times where the response time was over 100ms but lower than the worst response. Even in a soft realtime application that's a pretty bad performance, with more than 50% of the times not meating the deadline.

4.5 Exercise 5 - Creating a source code that imitates f1, f2 and f3

To make this, Posix time keep functions we're used.

The idea was to start with something kinda like this (pseudo-code for f1)

```
first_time = clock_gettime();
do{

second_time = clock_gettime();
}while(second_time - first_time < 38ms);
```

Of course this would only work if there we're no tasks with higher priority. If there we're, the function would run a fraction of the 38ms, be interrupted, and when it came back the time to run would be less than it should - it could even have already passed the rest of the 38ms and exit the while right away.

To solve this instead the function would wait "n" times a fraction the 38ms (again, example for f1).

Code 3: f1 replic function

```
struct timespec temp1,temp2, dt;
for(int i= 0; i < 38; i++){
   clock_gettime(CLOCK_MONOTONIC, &temp1);

   do{

      clock_gettime(CLOCK_MONOTONIC, &temp2);
      dt = Calculate_dt(temp1,temp2);

   }while(dt.tv_nsec < 1000000);
}
```

In this case it was opted for fractions of 1ms. Anything smaller, the actual execution time of the "while" would be significant to the total execution time. Since all the math execution time is very unstable, it should be much smaller than the while time (in this case 1ms) to keep a very steady total execution time of the function.

For f2 an f3 a similar code 3 is used, but the "for" instead execute 58 and 88 times respectively.

4.6 Exercise 6 - Testing func2.c

In this exercise the objective was to replicate exercise 3 and 4 but using func2.c as source for f1(), f2() and f3(). The main code for ex6_1 and ex6_2 is the exact same as ex3 and ex4.

The results:

The testing produced similar results with ex6_1 and ex6_2 giving about the same response times and ex6_2 showed the same behavior when exporting the data into a graph like figure 1

4.7 Exercise 7 - Executing in Round Robin

Assigning equal priorities to the tasks, test the application with the Round Robin scheduling method

Round-robin tries to divide in equal slices, the processor time of each task with the same priority when they try to run at the same time. The worst case is when the 3 tasks activate at the same time - there the worst possible response time will be the sum of all the computation times. Meaning that the absolute worst response time for any of the tasks should be:

$$38ms + 58ms + 88ms = 184ms \tag{15}$$

When executing the 3 tasks in Round Robin, having all the same priority, the worst execution times we're as expected, very close together.

$$\begin{aligned} R1 &= 184ms \\ R2 &= 184ms \\ R3 &= 184ms \end{aligned} \tag{16}$$

But, sometimes task 3 would get execution times around 200ms. It was not analyzed if this was due to the small processor being used - meaning other processes could increase the testing tasks response time, or simply that the round-robin method did not perfectly divide each task's processor time very well.

4.8 Exercise 8 - RTAI replic functions

For this exercise a test code was created. It simply has 3 periodic tasks that say "hi" every period.

To replicate the functions **rt_task_make_periodic()**, **rt_task_make_periodic_relative_ns()** and **rt_task_wait_period()**,[1] and[2] were consulted to know the behaviour or each function.

4.8.1 rt_task_make_periodic

Citing[2]

```
int rt_task_make_periodic ( RT_TASK * task,
                    RTIME start_time,
                    RTIME period
                    )

Make a task run periodically.

rt_task_make_periodic mark the task task,
    previously created with rt_task_init(), as
    suitable for a periodic execution, with
    period period, when rt_task_wait_period() is
    called.

The time of first execution is defined through
    start_time or start_delay. start_time is an
    absolute value measured in clock ticks.
    start_delay is relative to the current time
    and measured in nanoseconds.

Parameters:
   task is a pointer to the task you want to
       make periodic.
   start_time is the absolute time to wait
       before the task start running, in clock
       ticks.
   period corresponds to the period of the task,
       in clock ticks.

Return values:
   0 on success. A negative value on failure as
       described below:
     - EINVAL: task does not refer to a valid
         task.
```

To emulate this in Posix, RT_TASK * task, will be the thread ID of the type "pthread_t" and RTIME will be a "struct timespec". Since it is not asked to emulate "rt_task_init()", all tasks are "valid" so returning "EINVAL" has no point.

Each time you call this function, the task will be added to the periodic tasks list. **rt_task_make_periodic()** should not be used more than once for the same thread, it was not made to be prepared for that.

4.8.2 rt_task_make_periodic_relative_ns

```
int rt_task_make_periodic_relative_ns ( RT_TASK
    * task,
                          RTIME start_delay,
                          RTIME period
                          )

Make a task run periodically.

rt_task_make_periodic_relative_ns mark the task
    task, previously created with
    rt_task_init(), as suitable for a periodic
    execution, with period period, when
    rt_task_wait_period() is called.

The time of first execution is defined through
    start_time or start_delay. start_time is an
    absolute value measured in clock ticks.
    start_delay is relative to the current time
    and measured in nanoseconds.

 Parameters:
   task is a pointer to the task you want to
       make periodic.
   start_delay is the time, to wait before the
       task start running, in nanoseconds.
   period corresponds to the period of the
       task, in nanoseconds.

 Return values:
   0  on success. A negative value on failure
       as described below:
      -EINVAL: task does not refer to a valid
          task.
```

This function will work like **rt_task_make_periodic()** but instead of directly adding the parameters to the list, it will first get the current time and add **start_delay** to that.

There is a bit of discrepancy in the description vs the parameters description. The first says that **start_delay** is in clock ticks while the parameters says it's in nanoseconds. For the posix implementation **start_delay** is a struct timespec.

4.8.3 rt_task_wait_period

```
void rt_task_wait_period ( void  )

Wait till next period.
   rt_task_wait_period suspends the execution of
       the currently running real time task
       until the next period is reached. The
       task must have been previously marked for
       a periodic execution by calling
       rt_task_make_periodic() or
       rt_task_make_periodic_relative_ns().

Note:
   The task is suspended only temporarily, i.e.
       it simply gives up control until the next
       time period.
```

By this description, it seems that this function will just wait until the next period is reached, ex:

```
start time = 20ms
period = 100ms
current time = 440ms

So like this the time sleeping would be
    (440-20) / 100 = 4.2 -> 4 integer.
4 + 1 = 5; 5*100 + 20 = 520ms. So the task
    should sleep until 520ms time is reached.
```

This can be a problem because this means, if the task response time is higher than the activation period, then the task will wait until the next period instead of activating right away. Because of this more search was done and acording to[1]:

```
 Returns:

 0 if the period expires as expected. An
     abnormal termination returns as described
     below:
   - RTE_UNBLKD: the task was unblocked while
       sleeping;
   - RTE_TMROVRN: an immediate return was taken
       because the next period has already
       expired.
```

The fact that it can return **RTE_TMROVRN** means that the behaviour of **rt_task_wait_period()** is not simply "wait until next multiple of the period", but instead it will actually count the periods and exit right away if the next period has passed.

4.8.4 Periodic task list

To assure the behaviour of **rt_task_wait_period()**, the list will be composed by this struct for each task:

```
typedef struct{
   pthread_t thread;
   struct timespec start_time;
   struct timespec period;
   unsigned long n;
}rtai_node;
```

The **unsigned long n** is just to know when to not wait when the task response time is higher than the period.

Since this should work for a variable number of tasks and dynamic sized lists is something we learn in the first year, a connected list was created:

```
typedef struct _list_rtai{
   rtai_node data;
   struct _list_rtai *next_rtai_node;
}list_rtai;
```

And all the required functions to add and find elements:

```
list_rtai *list1=NULL;
void _rt_add(pthread_t *_thread, struct
    timespec _start_time, struct timespec
    _period){

   list_rtai *temp_list = list1;
   if(list1 == NULL){
      sem_init(&(writting_sem), 0, 1);
      sem_wait(&writting_sem);
      list1 =
          (list_rtai*)malloc(sizeof(list_rtai));
      list1->next_rtai_node = NULL;
      temp_list = list1;
   }
   else{
      while(temp_list->next_rtai_node != NULL){
         temp_list = temp_list->next_rtai_node;
      }
      sem_wait(&writting_sem);
      temp_list->next_rtai_node =
          (list_rtai*)malloc(sizeof(list_rtai));
      temp_list = temp_list->next_rtai_node;
   }


   temp_list->data.n = 0;
   temp_list->data.thread = *_thread;
   temp_list->data.start_time = _start_time;
   temp_list->data.period = _period;

   printf("thread: %lu; start_time_ns: %lu,
       _period_ns: %lu\n",
   temp_list->data.thread,
       temp_list->data.start_time.tv_nsec,
       temp_list->data.period.tv_nsec);
   sem_post(&writting_sem);
}

int _rt_find_element(pthread_t *_thread,
    list_rtai **_return){

   list_rtai *temp_list = list1;
   while(temp_list != NULL){

      if(temp_list->data.thread == *_thread){
         *_return = temp_list;
         return 0;
      }

      temp_list = temp_list->next_rtai_node;
   }

   return -1;
}
```

With all this, this will allow any number of tasks to be used with the RTAI replic functions. Altough remember, you can't call **rt_task_make_periodic()** and **rt_task_make_periodic_relative_ns()** for the same thread twice.

4.9 Exercise 9 - Testing RTAI replics

In this exercise the objective was to replicate ex3 and ex4 but now using the RTAI functions created and taking the most advantage out of them.

To take even more advantage of the RTAI, a new function is created to get the last activation time to allow calculating

the response time. Otherwise a solution very similar to exercise 3, 4, 6.1 and 6.2 would have to be used. This is the function:

```
int rt_give_last_activation(struct timespec
    *wake_time){
   pthread_t _thread = pthread_self();
   list_rtai* element = NULL;
   if(_rt_find_element(&_thread, &element) == 0){
      if(element->data.n == 0){
         wake_time->tv_nsec = 0;
         wake_time->tv_sec = 0;
         return 0;
      }

      struct timespec dt = {0, 0};
      if(element->data.n > 1)
      dt =
          timespec_mult_by_int(element->data.period,
          element->data.n-1);

      *wake_time =
          timespec_sum(element->data.start_time,
          dt);

      return 0;
   }
   else{
      printf("rt_task_wait_period. Error\n");
      return -1;
   }

}
```

The results obtained we're the same as exercise 3 and 4. A graph was also produced in ex9_2 but since it was also equal to figure 1 there was no point on including it.

5 Conclusion

After this assignment our understanding about real time systems and how scheduling policies and choice of priorities can impact the response times has greatly improved. The procedure on how to implement priorities, scheduling and periodic threads in Posix was something that this assignment made great to explore.

We believe the extra work done in analyzing the data, from response times to times of activation, in MatLab, using graphs, was key to understand the behavior of the system and make evident some key aspects.

One thing we think could be improved is the execution of exercise 5, in developing func2.c. We believe there could be better ways to implement a function with deterministic execution time.

Another point we would have liked to improve on this assignment would be trying to use Gantt charts on exercise 7 and maybe exercise 4 (and similar ones like 6_2 and 9_2)to analyze the data and determine why sometimes the response times in 7 we're over 184ms. Unfortunately MatLab did not support Gantt charts of the shelf and we did not have time to implement a custom method.

References

[1] rtai sourcearchive. rt_task_wait_period extra info,. http://rtai.sourcearchive.com/documentation/3.7.1-1/api_8c_ed4176714390da3fc9e23f9091dc353f.html.

[2] http://rtai.sourcearchive.com/. rtai.org api documentation,. https://www.rtai.org/userfiles/documentation/magma/html/api/.