A Real-Time System Using the Preempt-RT patch and the RM, PIP and EDF scheduling policies

Andrés Buelna[1], Mayra L. Lizárraga[1], Vidblain Amaro[2], Arnoldo Díaz-Ramirez[1], Larysa Burtseva[2]
[1]Departamento de Sistemas y Computación, Tecnológico Nacional de México, Instituto Tecnológico de Mexicali, Mexicali, Baja California, México
{mayral.lizarraga, andres.buelna.b}@gmail.com, adiaz@itmexicali.edu.mx
[2]Instituto de Ingeniería, Universidad Autónoma de Baja California, Mexicali, Baja California, México
{vamaro, burtseva}@uabc.edu.mx

***Abstract*—This paper discusses how to implement a hard real-time system using the Preempt-RT Linux kernel patch. Two real-time controlling systems were developed to control the same physical system. One of them employs the Earliest Deadline First policy, thanks to a recent modification of the Linux kernel that implements it while complying with the POSIX real-time standard. The second one uses both the Rate Monotonic policy and the Priority Inheritance Protocol, due to the fact that two system tasks share the same critical section. A brief overview of the Preempt-RT patch is presented. Also, the paper discusses the implementation of the two real-time systems and the results of their execution.**

***Index Terms*—real-time systems, rate monotonic protocol, priority inheritance protocol, earliest deadline first policy, preempt-rt.**

I. Introduction

A real-time system (RTS) is a computing system that is expected to not only produce correct results; it is also expected to produce them satisfying specific timing constraints, in order to ensure the correct behavior and the predictability of the system [1]. These systems have been the subject of great interest due to the trend towards automation of everyday's systems and applications. Examples of them are self-driving cars, autonomous airplanes, or robots to care for the elderly. Some other applications include:

- Flight control
- Medical systems
- Industrial automation
- Robotics
- Military systems
- Cyber-physical systems

An RTS is comprised of a controlled or physical system (e.g., a robot) and a controlling system (e.g., a computational system), where the latter is comprised of *n* tasks, *m* processors, and *r* resources. Commonly, $n \gg m$. Also, an RTS includes the use of scheduling algorithms to assign processors and resources to the tasks, while guaranteeing the temporal restrictions of the system.

In order to design and implement RTSs, many models have been proposed. One of them is known as the periodic task model, in which every instance of the system tasks are released at a given constant rate, called the period. Typically, a single RTS encapsulates many periodic real-time tasks, each with their specific time constraints. The objective of an RTS designer is to guarantee the accomplishment of the timing restrictions of the tasks that comprise the system, through the selection and use of a real-time operating system (RTOS) and real-time scheduling algorithms.

The deadline is the most used parameter to define the timing constraint of each task. There are different types of RTSs, depending on the consequences that may arise if at least one of the tasks misses its deadline:

1) Soft RTS: missing a task deadline produces a performance degradation, but it still has some utility for the system;
2) Firm RTS: missing a task deadline does not cause any damage, but the result after its deadline is useless for the system;
3) Hard RTS: missing a task deadline may cause catastrophic consequences.

Liu and Layland's Rate Monotonic (RM) and Earliest Deadline First (EDF) [2] are two of the most known algorithms for scheduling hard real-time periodic tasks. RM is a simple algorithm that assigns the highest priority to the task with the shortest period. The RM policy is a static priority algorithm since the priority of each task is fixed during the whole execution of the system. In contrast, the EDF policy assigns the highest priority to the task with the earliest deadline. EDF is a dynamic priority algorithm since the priority of each task may vary on each activation. Both algorithms ensure the predictability and the schedulability of the system (i.e., the accomplishment of the task's deadlines), as long as the task set satisfies the algorithm schedulability test.

The utilization factor of a real-time task set is given by:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}, \tag{1}$$

where C_i represents the execution time of the task *i* and T_i represents its period.

Accordingly to Liu and Layland's sufficient schedulability test, a task set is schedulable on one processor under RM if the processor utilization factor U satisfies the inequality:

$$U \leq n(2^{1/n} - 1), \tag{2}$$

where *n* is the number of tasks in the system.

ISBN: 978-607-97128-5-3

It can be seen from Eq. 2, that the least upper bound of the schedulablity test tends to 0.69, when $n \to \infty$. In other words, accordingly to the Liu and Layland's schedulability test for RM, a hard real-time task set is schedulable under RM as long as its total utilization is less than 70%.

Lehoczky *et al.* [3] developed a necessary and sufficient schedulability test for the RM policy. This exact test considers the processor utilization by the periodic task set as a function of time in a critical instant. Let τ be a set of *n* tasks of the periods $T_1 \leq T_2 \leq ... \leq T_n$, respectively, in an uniprocessor RTS. The cumulative demand on the processor by a set of tasks over the time interval $[0, t]$ at a critical instant is:

$$W_i(t) = \sum_{i=1}^{j} C_j \left\lceil \frac{t}{T_j} \right\rceil \quad (3)$$

where:

$$L_i(t) = \frac{W_i(t)}{t}$$

$$L_i = min_{\{0<t\leq T_i\}} L_i(t)$$

$$L = max_{\{1\leq i\leq n\}} L_i$$

$$S_i = \{kT_k | k = 1, \cdots, \lfloor {}^{T_i}/_{T_j} \rfloor ; j = 1, \cdots, i\}$$

Here L_i is the utilization factor required to fulfill the deadline of a task $\tau_i, 1 \leq i \leq n$, over the time range $[0, t]$; W_i is the cumulative demand on the processor by a set of tasks $\tau_1, \cdots, \tau_i$, over the time range $[0, t]$; S_i is the set of activation points of a task τ_i.

A task τ_i is schedulable under RM if and only if:

$$L_i = min_{\{t\in S_i\}} L_i(t).$$

Moreover, a set of n tasks is schedulable under the RM rule if and only if:

$$L_i = max_{\{1\leq i\leq n\}} L_i \leq 1.$$

Liu and Layland also defined a sufficient and necessary schedulability test for EDF, in which a real-time task set is schedulable on one processor under EDF if and only if its total utilization is less than or equal to one. That is, if the following condition is satisfied:

$$U \leq 1. \quad (4)$$

From Eq. 4 it can be observed that a hard real-time task set is schedulable under EDF as long as its total utilization is less than 100%.

When a task set requires the access to shared resources, problems such as priority inversion can be troublesome. Lui Sha *et al.* proposed the Priority Inheritance Protocol (PIP), that reduces the impact of the priority inversion problem [4]. Under the PIP protocol, a low priority task inherits the priority of a higher priority task if the former blocks the later while using a shared resource. By the use of the priority inheritance, the time a higher-priority task spends in a blocked state is reduced to the minimum, since a lower-priority task causing the blocking is allowed to complete its execution before other mid-priority tasks.

PIP can be used along with RM to allow resource sharing under a controlled environment. In order to test the schedulability of a task τ_i under PIP and RM, it is only needed to take into account the maximum blocking time of the task τ_i and the tasks with lower priority than τ_i . If the subset's processor utilization factor is less than or equal to $i(2^{1/i} - 1)$, as shown in Eq. 5, then the task's subset is schedulable under PIP and RM on one processor.

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \leq i(2^{1/i} - 1), \forall i, 1 \leq i \leq n \quad (5)$$

One of the main motivations of the use of an RTS is that it is possible to control a lot of physical systems using a low-resources computer system (e.g., an embedded system). For instance, consider the case of an autonomous vehicle. To reduce costs, wires can be removed from the vehicle and replaced by an RTS and a wireless network. The functions of the vehicle, such as the breaking system, are controlled by the RTS, guaranteeing that the temporal restrictions are met, such as stopping the vehicle on time.

However, to implement a hard RTS, a hard RTOS must be used. Preempt-RT is a patch that adds hard real-time capabilities to the Linux kernel. It includes the RM and PIP scheduling policies. Recently, a new patch that implements the EDF scheduling policy has been developed. Unfortunately, there is a lack of information on how to develop a hard RTS taking advantage of these scheduling policies. In this paper, the implementation of two RTSs is discussed. The main contribution of this document is a demonstration of the design and implementation of complete RTSs using the Preempt-RT patch, the RM, PIP and EDF scheduling policies, and the real-time POSIX standard extension [5].

The rest of the document is organized as follows: a brief description of Preempt-RT patch as well as the EDF extension are presented in Section II. Section III discusses the implementation of the controlled system, whereas the design of the controlling system using the EDF scheduling policy is presented in Section IV. In Section V the controlling system is described, using the RM scheduling policy and the PIP protocol to control the access to the shared resources. Finally, the conclusions are presented in Section VI.

II. PREEMPT-RT

An operating system (OS) is a software that acts as intermediary between the computer hardware and the user applications. The OS kernel is a software module that is loaded into memory after the computer is booted, and controls all the operations of the computer. Among its main activities, the kernel controls the process management, the interrupt handling and the process synchronization. A general purpose OS (GPOS) aims to share equally all the computer resources among the user's applications, in order to provide multi-tasking services and improve user's interactive responses. In

ISBN: 978-607-97128-5-3

contrast, an RTOS is designed to support specialized real-time applications [6]. Unlike a GPOS, an RTOS does not care about sharing resources equally among the system tasks since it always preempts a lower-priority task to execute a higher-priority one [7]. Additionally, the operations of an RTOS are time-bounded, since predictability is crucial to guarantee the timing restrictions of the RTS.

The activities of an RTS are commonly implemented as tasks or threads. An RTOS provides three important functions to serve tasks, which are:

- Scheduling
- Dispatch
- Intercommunication and synchronization.

The scheduler is an OS module that selects, from the ready tasks list, the next task to be executed, whereas the dispatcher is a module that performs the necessary bookkeeping actions to start the execution of the chosen task. Intercommunication and synchronization services assures that the tasks cooperate correctly, avoiding racing conditions and related anomalies.

To guarantee the temporal restrictions of the RTS, the scheduler implements real-time scheduling algorithms, such as RM, EDF, and synchronization protocols, such as PIP. The RTS designer chooses, among the available scheduling algorithms, those that better satisfy the RTS characteristics. There are many commercial and free-software RTOS's. Some examples are:

- VXworks [8]
- OSE [9]
- MaRTE OS [10].

Another approach to provide hard-real time capabilities is to modify an existing GPOS. For instance, the Linux kernel only provides support for soft RTSs. This makes Linux distributions not suitable for hard real-time applications. However, due to the fact that Linux is distributed using the GPL license and its source code is freely available, it is an attractive choice. Preempt-RT [11] is a kernel patch which makes a Linux system predictable and deterministic. The patch makes almost all kernel code preemptable. This is possible due to the following changes:

- Converting interrupt handlers into preemptable threads
- Implementing PIP for kernel mutexes, spin-locks and semaphores.
- Making critical sections protected preemptable using the functions spin-lock_t and rwlock_t.

Preempt-RT, as most RTOSs, provides two priority-based scheduling policies: First-In First-Out (FIFO) and Round Robin (RR), as well as the PIP synchronization protocol. The RM algorithm can be easily used through the FIFO policy. However, as was mentioned previously, the EDF algorithm allows a higher processor utilization than RM. Unfortunately, EDF is not implemented into the Linux kernel. Recently, a patch that implements the EDF scheduling algorithm has been developed [12]. The EDF scheduling policy is integrated complying with the POSIX Thread standard. The GLIBC library was also modified to accomplish it. Additionally, the FTRACE tool [13] is employed to allow the tracing of the kernel scheduling events. When the FTRACE option is activated, the scheduling events are traced and stored in a trace file, usually with the .dat file extension, which is parsed into the KIWI format. KIWI is an application that receives as input a *.ktr file, and produces graphs that show the scheduling events through time [14].

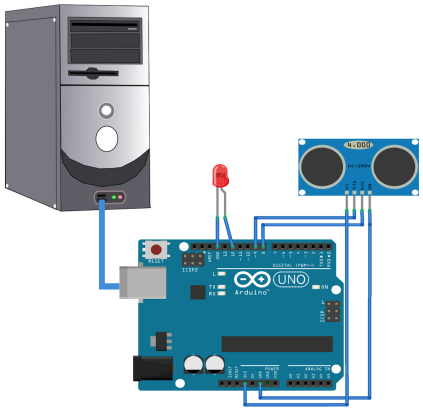

Figure 1. Architecture of the controlled system

In the next sections, the implementation of the two RTSs using the Preempt-RT patch and the POSIX standard will be discussed. One of the systems uses the EDF scheduling policy. The other one uses the RM and PIP algorithms. Both systems control the same physical system, which is described next.

III. A REAL-TIME CONTROLLED SYSTEM

In order to illustrate the implementation of an RTS, a prototype of a controlled system was designed. It implements a periodic task (thread) that monitors if an object is detected within a predefined range. For instance, the system could be seen as a robot that is moving and checks periodically for objects that interfere with its trajectory. If an object is detected, it turns an LED on immediately. Conversely, when the object is no longer detected, the LED is turned off. These actions are equivalent to the re-adjusting of the robot's trajectory, within a specific timing window, to avoid a collision.

An ultrasonic HC-SR04 was used to detect obstacles. It is capable to detect objects and calculate the distance from the sensor to an object in a range of 2 to 450 cm. The sensor operates sending a start pulse and measuring the width of the return pulse. Additionally, an Arduino UNO micro-controller was used to communicate the ultrasonic sensor with the controlling system (e.g., a computer). Fig. 1 depicts the architecture of the RTS prototype. The LED was connected to the GND pin and the pin 12 of the Arduino micro-controller in order to represent the turning ON/OFF of the lights. Also, the pins 9 and 8 are used for the trigger and echo of the sensor, respectively. Finally, the VCC and GND pins are connected to the 5v and GND pins of the Arduino UNO.

If an object is detected within the detection range, the system checks if the LED light is turned off. In that case, the LED is turned on. In contrast, if the LED light is already on, no action is performed. When the task is activated on its next period and the object is no longer within the detection range, the system checks if the light is ON to turn it OFF.

ISBN: 978-607-97128-5-3

These actions are performed by the controlling system, which communicates with the Arduino micro-controller through the USB serial port, using the Arduino-Serial library [15].

Two different controlling systems were designed for demonstration purposes. In both systems, there are three periodic tasks. Nevertheless, the tasks behave different on each system. One of them schedules the hard real-time tasks using the EDF policy. Only one task interacts with the controlled system, turning the LED ON and OFF. The second system schedules the hard real-time tasks using the RM policy. Since two system tasks share a critical section, the PIP protocol is used. Two tasks interact with the controlled system. The task with the highest priority turns the LED ON if an object is detected. The second task turns the LED OFF if no object is detected and the LED is already turned ON.

The controlling systems were executed on a Dell Vostro 260 with 6 Gb of RAM and an Intel® Core™ i3-2100 CPU @ 3.10GHz. However, since a mono-processor kernel was configured, only one core and 1 Gb of RAM were used. The system ran on a Debian GNU/Linux 7.4.0 Wheezy i386 OS, using the PREEMPT RT 3.4.61edfV2-rt77+ and the SSELINUX-EDF patches. The system used the EDF policy through the SSELINUX-EDF patch developed by Amaro *et al.* [12]. In the following sections each of the two controlling systems are explained briefly.

IV. Example: EDF-based controlling system

To test the performance of the EDF policy, an experiment was carried out using a controlling system comprised by three hard real-time tasks, that were implemented as POSIX threads. One task interacted with the controlled system, whereas the other two tasks were *independent*; i.e., they are supposed to interact and control some other physical systems.

Table I shows the worst execution time and period for every task.

Table I
Task set for EDF example

τ_i	τ_1	τ_2	τ_3
T_i	1000000000ns	800000000ns	700000000ns
C_i	481099035ns	234889412ns	664758422ns

According to the EDF schedulability test, a set of tasks is schedulable if and only if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{6}$$

Using the previous schedulability test, it can be concluded that the task set fulfills the schedulability condition since

$$\frac{481099035}{1000000000} + \frac{234889412}{800000000} + \frac{664758422}{700000000} \leq 1,$$

$$0.949654888 \leq 1.$$

Fig. 2 shows the execution of the tasks for a given period of time. It can be observed that all the tasks complete execution by their respective deadlines.

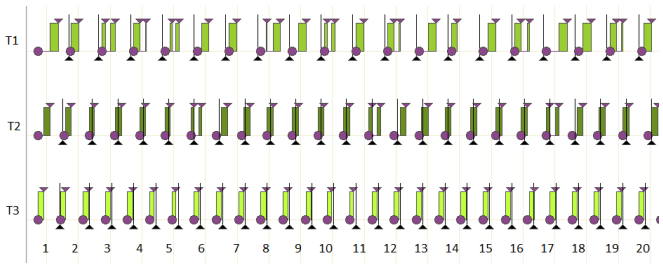

Figure 2. Task set scheduled under EDF

The following code shows the function `main()` where every thread, with respective deadline and initiation, is declared using the EDF policy.

In order to create and configure the system threads, the first step is to declare the threads identifiers and their attributes identifiers, as it is shown in line 6 and line 7, respectively. It is important to note that, to communicate the application with the Arduino micro-controller, the Arduino-Serial library was used. For instance, in line 9 the `fd` and `bitrate` identifiers are declared, where `fd` represents the connection to the serial port and `bitrate` is the transmission data rate. Lines 10 and 11 initialize the connection to the Arduino Card.

The RTS designer selects and sets the thread's scheduling policy through the function `pthread_attr_setschedpolicy`, as shown in line 17. The Preemp-RT patch implements the SCHED_FIFO and SCHED_RR scheduling policies. Additionally, the SCHED_EDF policy can be used if the kernel is properly patched. When the SCHED_EDF policy is used, it is necessary to specify the deadline of the thread, through the use of the `sched_param` data structure, as shown in line 20. Afterwards, the thread attributes are set (line 21).

Through the use of the function `clock_gettime` and the clock CLOCK_MONOTONIC, the amount of time elapsed since an unspecified point in the past, such as the system start-up, it is obtained and stored in `initialTime` (line 14). Since all the system threads start the execution at almost the same time, once the current time is obtained, the initial time is set to one second later (line 15).

```
1 #include "../arduino-serial/arduino-serial-lib.h"
2 #include <pthread.h>
3 struct timespec initialTime;
4 int main(int argc, char **argv) {
5   /** Declare variables **/
6   pthread_t in, pr, out;
7   pthread_attr_t inAttr, prAttr, ouAttr;
8   struct sched_param param1, param2, param3;
9   int fd = -1,bitrate=9600;
10  fd = serialport_init("/dev/ttyACM0", bitrate);
11  serialport_flush(fd);
12  ...
13  /** Do some operations **/
14  clock_gettime(CLOCK_MONOTONIC,&initialTime);
15  initialTime.tv_sec+=1;
16  //Set schedule policy
17  status = pthread_attr_setschedpolicy(&inAttr,
                    SCHED_EDF);
18  ...
19  //Set deadline
20  param1.sched_rel_deadline = t;
21  status = pthread_attr_setschedparam(&inAttr,
                    &param1);
```

```
22   ...
23 }
```

The code below describes the basic structure of the functions that the threads execute on each activation. Each thread has different parameters, such as the period, and performs different operations. Since the controlling system uses the periodic task model, each instance of the threads is activated periodically. For this reason, when a thread is created, the initialization parameters are set, as can be observed in lines 5-7. Afterwards, the thread executes an infinite loop (lines 9-13). The first operation inside the loop is the invocation to the `clock_nanosleep()` function (line 10), which suspends the thread until its period is elapsed, or a signal is delivered to the calling thread. At the end of the loop, once the thread operations are executed, the value of the thread period is added to the `nextActivation` variable (line 12), which is used by the `clock_nanosleep()` function to set the new activation instant for the thread.

```
1  void * inThread(int* fd) {
2   struct timespec periodActivation;
3   struct nextActivation,nextdeadline;
4   int period_time;
    ...
5   period_time=1000000000;
6   periodActivation.tv_sec = 0;
7   periodActivation.tv_nsec = period_time;
8   nextActivation = initialTime;
    /** Do some operations **/
    ...
9   do{
10    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
11    &nextActivation, NULL);
    /** Do some operations **/
    ...
12    timespec_add(&nextActivation, &periodActivation);
    ...
13  } while (1);
14  pthread_exit(NULL);
15  }
```

The following code explains the most important actions performed by the controller tasks. In order to allow the communication between the physical system and the controller thread, the next parameters were declared as shown in lines 11-13 of the code below. The functions `serialport_read_until` and `serialport_write` shown in lines 19 and 22 are used to communicate with the Arduino micro-controller. Next, it can be seen in lines 20 to 24 that the threads, on each activation, check if an object is detected. In such a case, the LED light is turned ON. Otherwise, it is turned OFF.

```
1 void * inThread(int* fd) {
2   struct timespec periodActivation;
3   struct nextActivation,nextdeadline;
4   ...
5   period_time=1000000000;
6   periodActivation.tv_sec = 0;
7   periodActivation.tv_nsec = period_time;
8   nextActivation = initialTime;
9   /** Do some operations **/
10  ...
11  const int buf_max = 256;
12  char serialport[buf_max],eolchar='\n',buf[buf_max];
13  int timeout = 5000,dist;
14  do{
15    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
            &nextActivation, NULL);
16    /** Do some operations **/
17    memset(buf,0,buf_max);
18    serialport_flush(*fd);
19    serialport_read_until(*fd, buf, eolchar,buf_max,
                  timeout);
20    dist=atoi(buf);
21    if(dist<=30)
22      rc = serialport_write(*fd, "1");
23    else
24      rc = serialport_write(*fd, "0");
25    ...
26    timespec_add(&nextActivation, &periodActivation);
27    ...
28  }while (1);
29  pthread_exit(NULL);
30 }
```

V. Example: RM and PIP based controlling system

The second controlling system works with the RM policy and the PIP protocol and implements three threads. One of them is supposed to control a different physical system. The other two of the threads cooperate to control the physical system, sharing a critical section, which is the connection port with the physical system.

Since the task set shown in Table II does not satisfy the Liu and Layland's sufficient schedulability test, it is not known wether or not the task set is schedulable using RM. However, using the necessary and sufficient schedulability test proposed by J. Lehoczky *et al.* in [3] and adding the necessary maximum blocking times B_i, it can be concluded that the task set is schedulable using the RM and PIP protocols.

Table II
Task set for the RM and PIP example

τ_i	τ_1	τ_2	τ_3
T_i	700000000ns	800000000ns	1000000000ns
C_i	360164690ns	222114638ns	334387711ns
B_i	246928780ns	—	—
U_i	0.86727638	0.27764329	0.33438771

Proof: Using Lehoczky's sufficient and necessary schedulability tests, and including the blocking times due to the use of critical sections, we obtain:

1) Task τ_1: Check $C_1 + B_1 \leq 700000000$. Since $360164690+246928780 \leq 700000000$, task τ_1 is schedulable.
2) Task τ_2: Check whether either $C_1 + C_2 + B_2 \leq 700000000$; $360164690 + 222114638 + 0 \leq 700000000$ Or
   $C_1+C_2+B_2 \leq 800000000$ $360164690+222114638+0 \leq 800000000$
   Since either of the two conditions needs to hold true we conclude that task τ_2 is schedulable
3) Task τ_3: Check whether either $C_1 + C_2 + C_3 \leq 700000000$ $360164690 + 222114638 + 334387711 \leq 700000000$ Or
   $C_1+C_2+C_3 \leq 800000000$ $360164690+222114638+334387711 \leq 800000000$ Or
   $C_1+C_2+C_3 \leq 1000000000$ $360164690+222114638+334387711 \leq 1000000000$
   Since the third condition holds true, we conclude that the task set is schedulable.

ISBN: 978-607-97128-5-3

Fig. 3 shows the execution chart of the task set under RM with shared resources. The circles represent the activation of a job and the triangles represent the end of the execution of the job. We can observe that there are cases where Thread 3 is blocked by Thread 1 due to the utilization of the shared resources. It also can be observed that all the tasks meet their respective deadlines in the period of time depicted in Fig. 3 .

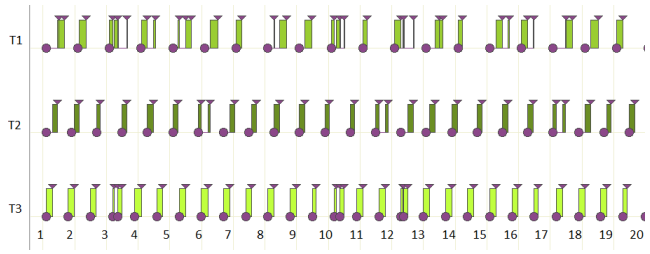

Figure 3. Task set scheduled under RM with shared resources

As it was explained in the previous case, the header files, as well as the main's thread parameters related to the connection, need to be included in the program's code. However, due to the fact that two tasks use the same port to communicate to the Arduino micro-controller, they request access to the critical section using the `pthread_mutex_lock` function and release it using the `pthread_mutex_unlock` function, after completing execution of the critical section. Both functions are shown in line 14 and 21 of the code below. Similarly, the communication is established making use of the `serialport_read_until` and `serialport_write` functions in lines 17 and 20, respectively.

```
1  void * Thread3(...) {
2    struct timespec periodActivation,nextActivation;
3    ...
4    nextActivation = initialTime;
5    const int buf_max = 256;
6    char serialport[buf_max],eolchar='\n',
                                    buf[buf_max];
7    int timeout = 5000,dist;
8    do{
9      /*** The thread is set to sleep until the next
10     activation time ***/
11     clock_nanosleep(CLOCK_MONOTONIC,
                          TIMER_ABSTIME,
              &nextActivation, NULL);
11     /*** Do some operations ***/
12     pthread_mutex_lock(&mutex);
13     memset(buf,0,buf_max);
14     serialport_flush(*fd);
15     serialport_read_until(*fd,buf,eolchar,buf_max,
                        timeout);
16     dist=atoi(buf);
17     if(dist<=30)
18       rc=serialport_write(*fd, "1");
19     pthread_mutex_unlock(&mutex);
20     /*** Do some operations ***/
21     ...
22     timespec_add(&nextActivation,
                           &periodActivation);
23   }while (1);
24   pthread_exit(NULL);
25 }
```

VI. Conclusions and Future Work

To implement a hard RTS, a hard RTOS must be used. Preempt-RT is a Linux kernel patch aimed to bound the latency of the kernel operations, thus providing hard real-time capabilities to Linux. Preempt-RT offers the FIFO scheduling policy and the Priority Inheritance Protocol as a synchronization protocol. Although the RM algorithm can be easily mapped onto FIFO, there are real-time task sets with high processor utilization that cannot be scheduled using RM but are scheduled using the EDF policy. Recently, a patch to the Linux kernel was developed, implementing the EDF policy offering compliance with the POSIX standard.

Unfortunately, there is no information regarding how to implement a hard RTS using the Preempt-RT kernel and the RM, PIP and EDF policies. This paper aims to fill this gap by discussing how to implement a hard RTS using these tools.

A prototype of a physical system was built, along with two hard RTS controlling systems. The schedulability analysis, the system code and the results of their executions were discussed.

As a future work, we plan to evaluate the RTSs introduced in this paper using a minimal hard real-time kernel.

References

[1] G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, ser. Real-Time Systems Series. Springer US, 2011. [Online]. Available: https://books.google.com.mx/books?id=h6q-e4Q_rzgC

[2] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973. [Online]. Available: http://doi.acm.org/10.1145/321738.321743

[3] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *[1989] Proceedings. Real-Time Systems Symposium*, Dec 1989, pp. 166–171.

[4] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep 1990.

[5] T. O. G. 1998, *POSIX .13 IEEE Std. 1003.13-1998. Information Technology—Standardized Appli- cation Environment Profile—POSIX Realtime Application Support (AEP)*, 1998.

[6] S. R. Reddy, "Selection of rtos for an efficient design of embedded systems," *IJCSNS International Journal of Computer Science and Network Security*, vol. 6, no. 6, pp. 29–37, 2006.

[7] B. Furht, *Real-time UNIX systems: design and application guide*, ser. Kluwer international series in engineering and computer science. Kluwer Academic Publishers, 1991.

[8] Vxworks website https://www.windriver.com/products/vxworks/.

[9] Ose website http://www.enea.com/ose.

[10] Marte os website http://marte.unican.es/.

[11] Rt preempt home page https://rt.wiki.kernel.org/.

[12] *V. Amaro-Ortega, L. Burtseva, A. Díaz-Ramírez. SSELINUX-EDF. Registro Público de Derecho de Autor, No. 03-2016-042911515700-01, 3 Mayo, 2016. INDAUTOR, México.*

[13] Ftrace home page. https://www.kernel.org/doc/documentation/trace/ftrace.txt.

[14] Kiwi home page http://www.gti-ia.upv.es/sma/tools/kiwi/index.php.

[15] Arduino-serial source page https://github.com/todbot/arduino-serial.