Comput Syst Sci & Eng (2018) 1: 31–40
© 2018 CRL Publishing Ltd

International Journal of
Computer Systems
Science & Engineering

A scheduling extension scheme of the earliest deadline first policy for hard real-time uniprocessor systems integrated on POSIX threads based on linux

Vidblain Amaro-Ortega[1,*], **Arnoldo Díaz-Ramírez**[2], **Brenda Leticia Flores-Ríos**[1], **Félix Fernando González-Navarro**[1], **Frank Werner**[3] **and Larysa Burtseva**[1]

[1]*Autonomous University of Baja California, Mexico.*
E-mail: brenda. ores@uabc.edu.mx, fernando.gonzalez@uabc.edu.mx, burtseva@uabc.edu.mx
[2]*Technical Institute of Mexicali, Mexico. E-mail: adiaz@itmexicali.edu.mx*
[3]*Otto-von-Guericke University Magdeburg, Germany. E-mail: frank.werner@ovgu.de*

The Linux operating system has been employed to execute numerous real-time applications. However, it is limited to support soft real-time systems by two scheduling policies: First-In-First-Out and Round Robin. For real-time systems with critical constraints, the soft real-time support and these scheduling policies are still insufficient. In this work, the Earliest Deadline First scheduling policy, which has been shown in theory to be an optimal one in uniprocessor systems, is introduced as an extension of the Linux kernel. This policy is implemented into the real-time class, without the necessity of defining an additional class. The Linux kernel affords capabilities of a hard real-time operating system by an RT-Preempt patch, enabling the use of Linux to implement hard real-time systems. The integration is compliant with the POSIX real-time and thread standards, ensuring applications portability, employing the GLIBC library. In order to validate the proposed implementation, a set of experiments is conducted, showing that a real-time system that cannot be feasibly scheduled using existing policies, attains feasibility when it is scheduled using the integrated Earliest Deadline First policy

Keywords: Hard real-time task, scheduling, uniprocessor system, earliest deadline first policy, POSIX

1. INTRODUCTION

Nowadays Linux has positioned itself as one of the most interesting operating systems for the academic community and companies, for the development and execution of applications in several fields, e.g. control, computation, space, and military, to name a few [1, 2]. Like any operating system (OS), Linux is integrated by a set of programs that acts as an intermediary between the applications and the computer hardware, and that manages all the available resources. An OS belongs to one of two classes: a general purpose OS (GPOS) or a real-time OS (RTOS). The former aims to support the time-sharing applications and therefore allocates equitably the available resources to the system tasks. Instead, an RTOS gives support to real-time applications and incorporates policies that enable the compliance with the timing constraints of the tasks. In an RTOS, the tasks are commonly implemented as threads, which are tied to processes. An RTOS contains also a kernel that provides basic computer functions and takes control over the execution of the threads and the processes. It responds to the system calls, offers scheduling and timing services, as well as manages external interrupts. A system call allows a thread or a process to invoke the functions provided by the kernel.

*Correspondence Author. E-mail: vamaro@uabc.edu.mx

Linux has been increasingly employed as an RTOS due to its stability in the execution of the applications, allowing an efficient use of the computing resources. This is achieved largely by the scheduler, which is a component of the kernel that selects, on each scheduling instant, the next task to be executed from the list of active tasks. A standard interface, which is known as POSIX, an acronym for the Portable Operating System Interface, offers many advantages, such as portability and standardization of the application development [3]. It is aimed for software compatibility between UNIX-like OSs, defining an Application Programming Interface (API). Many operating systems are POSIX compliant, e.g., AIX, HP-UX, OS X, FreeBSD, Solaris, Android and Linux. POSIX has defined two extensions for the development of real-time systems. The first one is known as the IEEE Std 1003.1b-1993 extension, or simply POSIX.1b, which defines priority based scheduling, semaphores, real-time signals, among others. The second real-time extension is entitled IEEE Std 1003.1c-1995, or POSIX.1c, and defines thread creation, control, scheduling and synchronization.

The correctness of many systems and devices in our modern life depends not only on the effects or the results they produce, but also on the response time [4]. Such constraints set up an essential part of a real-time system (RTS), which requires a complete assignment of the resources and provides services in a timely manner. An RTS is considered as a system that interacts with an asynchronous environment to maintain an ongoing relationship, reacting opportunely to changes of the conditions and remaining synchronized with respect to the environment [5]. Such systems need to respond and to be executed within predefined time constraints. An RTS includes a set of tasks, which commonly are periodic due to a large number of control applications that require cyclical activities [6]. Every periodic task generates a sequence of activations called jobs.

The RTSs should be classified into hard and soft ones. An inherent characteristic of these systems is that the requirement specifications include timing information in the form of deadlines [7, 8]. In a hard RTS, a task must complete the execution before the associated deadline; otherwise a system is considered as failed. A soft RTS is less restrictive; the tasks may continue the execution beyond their respective deadlines up to a limit, while the system tries to minimize the consequences of missed deadlines. An example of this kind of systems is the people recognition for security issues [9].

Almost all existing RTOSs, including Linux, provide two priority-based scheduling policies: First-In First-Out (FIFO) and Round Robin (RR). Although these policies offer some degree of timing guarantees for the soft RTSs, they are not sufficient for systems with hard real-time constraints [10]. The two most known scheduling policies for hard RTSs are the Rate Monotonic (RM) and the Earliest Deadline First (EDF) [6, 11, 12, 13]. The RM algorithm, which assigns priorities inversely proportional to the task periods, is an optimal static priority assignment policy [14]. It can be emulated through FIFO. Instead, the EDF algorithm, which assigns the highest priority to the job with the earliest deadline, is an optimal dynamic priority assignment policy [7]. Unfortunately, EDF is not available in most of the existing RTOSs.

There are two approaches to allow Linux to provide real-time support: the hypervisor level [15, 16, 17, 18] and the OS scheduler level [19, 20, 21]. The first approach admits the coexistence of both, an RTOS and a generic OS. The first one has a higher priority over a non-RTOS. In contrast, the OS scheduler level provides real-time capabilities using multiple scheduling classes; each one has several scheduling policies, being the real-time scheduling class of the highest priority. Several commercial and open source projects use this approach. One of them is the RT-Preempt [19], which includes free open source patches. The correct choice of an RTOS is a fundamental aspect in the design of an RTS.

In this work, an integration of the EDF policy into the Linux kernel is achieved, employing the OS scheduler level approach and the RT-Preempt patch. In the same manner, we extend the API of the POSIX standard by enabling the use of the integrated policy via the library GLIBC [22]. This makes the development of hard RTSs possible, ensuring their portability.

The rest of the paper is organized as follows. In Section 2, the most remarkable works that complement Linux as an RTOS are surveyed. A theoretical background of RTSs and the notations used are given in detail in Section 3. In Section 4, the structure of the Linux scheduler is described internally in order to continue with the implementation of the scheduling policy and the experimental test in Section 5. Finally, in Section 6 some conclusions and subjects for future work are given.

2. RELATED WORK

Many modern GPOSs offer some kind of soft real-time support, mainly targeted for multimedia applications. Unfortunately, they lack a hard real-time support. One approach that has been used frequently to provide hard real-time capabilities is to modify the Linux kernel, avoiding any operating system operation whose duration is not predictable [9, 20, 21]. However, many of these proposals do not entirely comply with the POSIX standard since they define their own API for the development of RTSs. [10, 20, 21, 23]. In this section, we introduce some of the most known Linux variations that provide RTS support.

`SCHED_DEADLINE` [10, 23, 25] is a proposal that employs the scheduler level approach to implement the EDF algorithm as a policy; however, it defines proper functions, which differ in the POSIX standard. The RT-Preempt is a Linux kernel patch that offers low latency [19]. It provides real-time synchronization protocols, such as the priority inheritance protocol to limit the unpredictability caused by the management of shared resources. The RT-Preempt offers full support to the POSIX standard. Litmus-RT [20] is a patch that allows the fulfillment of non-critical real-time capabilities on the Linux OS. It offers a platform for the implementation of the scheduling policies and the synchronization of the tasks under multiprocessor architectures. This RTOS integrates an own system call interface and as a result, POSIX API is not supported. It offers various real-time scheduling algorithms implemented into the uniprocessor and multiprocessor architectures. ChronOS [21] is based on the RT-Peempt patch, providing hard RTOS capabilities. Furthermore, ChronOS incorporates scheduling policies across the interfaces that provide a set of APIs for the development of the RTSs.

Xenomai [16, 18], RTAI [15, 18] and RT-Linux [17] use a hypervisor level approach allowing Linux to behave like an RTOS.

They were developed with the purpose to execute an abstract micro-kernel in an OS like Linux. When a micro-kernel is used, an intermediate layer between the OS kernel and the hardware is created. The first one is notified if a system call or an interruption is generated. Linux is considered as a regular process within these RTOSs [15, 16, 17]. This solution turns out to be too invasive with respect to the original OS.

Commercial RTOSs, such as QNX Neutrino [26], VxWorks [27] and INTEGRITY [28], to mention a few, are also available on the market. They offer various services as remote support, POSIX-compliance, further available scheduling policies, APIs documentation, etc.; however the costs of these products or the necessity of custom requirements are out of reach in some cases.

The fact that the works [10, 20, 21, 23, 24] define own system call interfaces for the communication between an RTOS and the applications, is an obstacle to enable the RTSs portability. This problem is solved in this work by ensuring compliance with POSIX. Although some referred works [20, 21] allow a standard compliance, they do not support hard RTSs. In this paper, the running of such systems is also achieved.

3. TASK MODEL

Several reference models have been proposed for RTSs. In this work, a periodic task model for RTSs is considered. In this model, an application is comprised by a set τ of n static tasks. Every task in τ gives rise to a potentially infinite sequence of activations or jobs. In addition, each task i, $1 \leq i \leq n$, is activated at a regular or semi-regular time interval T_i, which represents the minimum inter-arrival time between two consecutive jobs in τ_i. The least common multiple of T_i is called the hyperperiod of a periodic task set. The task execution time C_i, is the maximal time required for the execution of each job in τ_i; which denotes the worst-case execution time (WCET). It is assumed that the periods and execution times of all tasks are known at every instant, as well as the task period is equal to its deadline. A summary of the notations is given below.

3.1 Notations

τ	Task set.
i	Task index, $i = 1, 2, \ldots, n$.
j	Activation index, $j = 1, 2, \ldots, J$, where J is the number of the last activation.
r_{ij}	Release time of the task i in the activation j.
C_i	Execution time of the task i in the worst-case (WCET).
D_i	Relative deadline of the task i.
d_{ij}	Absolute deadline of the task i in the activation j.
T_i	Length of the activation period of the task i.
s_{ij}	Start time of the task i in the activation j.
f_{ij}	Finish time of the task i in the activation j.

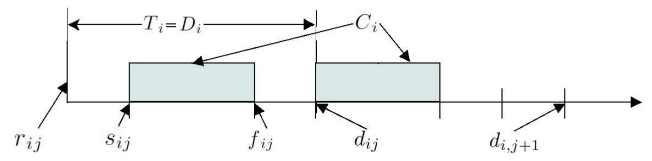

Figure 1 Main parameters of a task i.

The main parameters of a task are shown in Figure 1, where the execution of a task i, represented by a tuple (C_i, D_i, T_i) is illustrated. The task i is released at r_{ij} on activation j, and executed in the time interval $[s_{ij}, f_{ij}]$. The time C_i, which is used for the task execution, must be less than or equal to $D_i = T_i$.

3.2 The RTS

An RTS reacts dynamically on changes in the environment, whether they are caused by human behavior, either by a natural or an artificial phenomenon. A response must occur within a range of time according to the system constraints. The computer system and the environment are two elements that act differently on an RTS due to different time domains. An environment setting is governed by the exact duration on the clock time; while the computer system determines a sequence of the machine instructions, which must be performed in a chronological order [29].

In the context of the applications, a task in an RTS is a set of related jobs that provide some system function. Every job of a task may be released periodically, sporadically, or aperiodically. The majority of the constraints in an RTS are expressed by the task release (arrival) times, execution times, and deadlines.

An essential part of an RTOS is the scheduler. It defines the sequence of the jobs on the processor(s). A schedule is feasible if every job completes its execution by its deadline. Furthermore, a set of jobs is schedulable according to a policy if the scheduler always generates a feasible schedule when this policy is used. An optimal hard real-time scheduling policy always generates a feasible schedule, providing that a given set of jobs has feasible schedules [6, 11, 30].

The RM and the EDF rules are frequently used as scheduling policies. They have been extensively studied in [6, 11, 12, 13, 14, 31]. Both policies are based on assigning priorities to the system tasks according to respective deadlines, where a fresh review of priority assignment was presented by Davis et al. [32]. Assuming that the deadline of a task is numerically equal to its period length, the RM assigns static priorities inversely proportional to the period of a task; the task with the shortest period has the higher priority, and vice versa. The EDF assigns dynamic priorities to each job of a task according to the nearest expiring deadline [14].

For hard RTSs, it is essential to know if a task set is schedulable or not. This problem can be resolved with a mathematical condition, which is commonly called the feasibility test. It was proved by Baruah et al. [30] that the feasibility test in uniprocessor RTSs is a co-NP-complete problem in the strong sense for non-trivial computational models. In systems where the relative priorities of tasks (as in RM) or jobs (as in EDF) do not vary, the feasibility analysis may be less complex [7].

There are two kinds of feasibility tests: exact (sufficient and necessary) and inexact (sufficient but not necessary) ones [7]. If a task set is scheduled with a given algorithm and satisfies the exact test, then all the tasks will meet their deadlines. In contrast, if a task set does not satisfy the inexact feasibility test, it is not really known whether the tasks may complete their execution according to their deadlines. Liu and Layland [14] demonstrated that RM is an optimal policy for the static priority assignment. Their inexact test, shown in (1), states that a set of n periodic tasks is schedulable under RM if:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1). \tag{1}$$

The authors proved that RM is able to schedule any periodic task set τ with implicit deadlines (periods are equal to the deadlines) if the total utilization of the processor satisfies $U_T \leq \ln 2 \approx 0.693$, where U_T is given as follows:

$$U_\tau = \sum_{i=1}^{n} \frac{C_i}{T_i}. \tag{2}$$

Lehoczky et al. [33] proposed a necessary and sufficient schedulability condition for the RM policy (exact test), which 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 $T_1 \leq T_2 \leq \cdots \leq T_n$, respectively, in a uniprocessor RTS. The cumulative demand on the processor by a set of tasks over the time interval $[0, t]$ at a critical instant is:

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

where:

$$\begin{aligned} L_i(t) &= \frac{W_i(t)}{t}, \\ L_i &= \min_{\{0<t\leq T_i\}} L_i(t), \\ L &= \max_{\{1\leq i\leq n\}} L_i, \\ S_i &= \{kT_k | k = 1, \cdots, \lfloor T_i/T_j \rfloor; \quad j = 1, \ldots, i\}. \end{aligned}$$

Here L_i is the utilization factor required to fulfill the deadline of a task 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 $\tau_1, \ldots, \tau_i$, over the time range $[0, t]$; S_i is the set of activation points of a task 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:

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

Liu and Layland [14] introduced an exact EDF schedulability test for any periodic task set (4). A periodic task set is schedulable under the EDF policy if and only if:

$$U_T \leq 1. \tag{4}$$

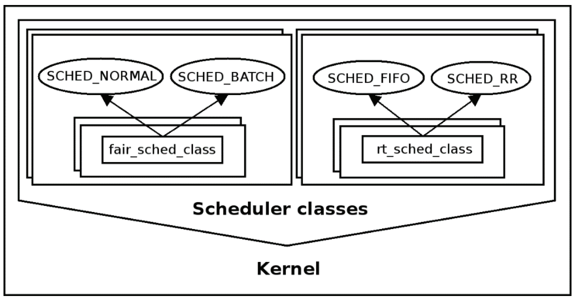

Figure 2 The policies and classes defined in the Linux kernel.

The authors proved that EDF is an optimal dynamic algorithm for uniprocessor architectures.

The availability of multiprocessor architectures, particularly multi-core ones, has motivated the interest in the scheduling theory for multi-processor RTSs [5], as well as the energy efficiency with shared resources [34]. However, there are still many applications that require uniprocessor architectures, due to reasons of predictability, application size, power consumption, to mention a few. In addition, new computing paradigms, such as cyber-physical systems, require the use of uniprocessor platforms [35, 36, 37, 38].

Among the existing uniprocessor real-time scheduling policies, a tendency resides in favor of the RM policy more than the EDF. Nevertheless, it has been shown that EDF allows a better exploitation of the available resources and significantly improves the system performance, as Buttazzo mentioned in [6]: "It is commonly believed that EDF introduces a larger runtime overhead than RM, however, EDF introduces less runtime overhead than RM, when context switches are taken into account." Unfortunately, EDF is barely available in existing RTOSs.

4. THE EDF POLICY INTEGRATION ON THE LINUX KERNEL

This section discusses the integration of the EDF policy named `SCHED_EDF` into the kernel. The integration was evaluated using the kernel version 3.4.61 and the RT-Preempt patch 3.4.61-RT77. The next subsection describes the architecture of the scheduler and the components, in order to continue with the implementation of the EDF policy.

4.1 The Linux Scheduler

On the Linux kernel, the policies are implemented through classes into the scheduler, referring to a class as a structure that groups the functions and variables used. Linux offers two scheduling classes, as it is shown in Figure 2. One of them is the `fair_sched_class`, which adopts two policies (`SCHED_NORMAL` and `SCHED_BATCH`). The other one is the `rt_sched_class`, which integrates two fixed priority policies (`SCHED_FIFO` and `SCHED_RR`) for the real-time tasks, as it is defined by the POSIX standard [3].

Linux provides 140 priority levels for the system tasks, which are managed by the two scheduler classes. The

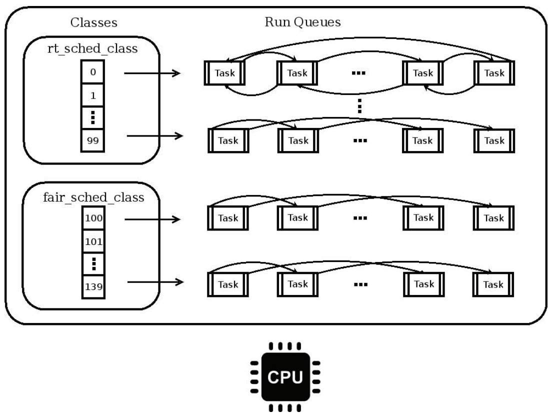

Figure 3 Run queues within the Linux scheduler.

`rt_sched_class` class defines priority levels ranging from 0 to 99, 0 being the highest priority level among all. This class is used to assign priorities to soft real-time tasks. The `fair_sched_class` class defines priority levels from 100 to 139, and it is used by general-purpose tasks [21].

The Linux scheduler organizes the active tasks in run queues, implemented as red-black tree structures. Run queues are grouped into the scheduler classes used by the system. On each run, the tasks are sorted in a queue in order of their priority levels, as shown in Figure 3. In the case of multi-core architectures, each processor or core defines its own run queue structures. The Linux scheduler assigns tasks to each core where they are intended to execute during their lifetime, to avoid the overhead caused by cache misses and synchronization. However, the tasks could be moved to another core after running a load-balancing algorithm.

Since version 2.6.23, the Linux scheduler has been designed as an extensible module, allowing the incorporation of different policies across the classes. However, it is not capable to ensure that a hard RTS can be executed satisfying its timing restrictions, because it is designed to act as a GPOS. Namely, preemption is not permitted until the complete execution of the task. This generates latency in the response times of the active tasks.

The user programs are commonly restricted by the processor usage for long periods of time. The predictability in the OS response time and latency is required to achieve an accurate RTS execution. The accomplishment of this feature implicates that the kernel has to be preempted at any time, if a higher priority task requires to be executed.

4.2 RT-Preempt

The Linux kernel is designed in such a manner that it allows the shared resources to be accessed concurrently by different tasks. It implements some synchronization methods that are usually performed through semaphores or mutexes. Nevertheless, there are non-preemptive sections that make it unpredictable. A patch called RT-Preempt [19] has been developed and submitted into the kernel mainline. With some modifications in the kernel listed above, it provides the features that an RTOS should have to ensure the execution of the RTSs, modifying mainly the sections that cause Linux to be unpredictable:

Table 1 RTOS supports featured in projects.

Project	Hard RTSs	POSIX	GLIBC	Thread policy	References
SCHED_DEADLINE	+	−	−	−	[10, 23, 25]
Litmus-RT	−	−	+	+	[20]
ChronOS	−	+	+	+	[21]
Xenomai	+	+	+	−	[16, 18]
RTAI	+	+	−	−	[15, 18]
RT-Linux	+	+	−	−	[17]
SCHED_EDF	+	+	+	+	

- The primitives used to block the access to a resource are now preemptive by employing rt-mutexes instead of spinlocks. The last ones do not allow the kernel preemption once a resource is acquired.
- The priority inheritance protocol is implemented for in-kernel mutexes and not only for the user's processes.
- The interrupt handlers are now treated as preemptive kernel threads. In a GPOS, such processes are commonly performed immediately, even if there are other higher priority tasks in the system. These processes within Linux are designed as non-preemptive ones.
- RT-Preempt incorporates two high-resolution timers, turning the API Linux timer into separate infrastructures: one to provide kernel-level timers, and the other one for user programs, according to the POSIX standard. High-resolution timers enable that applications have a higher accuracy.

With these adjustments, the RT-Preempt patch allows Linux to be preempted in almost any instant of time. All these modifications are required for the immediate execution of tasks with a higher priority.

4.3 The SCHED_EDF policy

In previous works [10, 23, 25], the EDF policy was implemented creating a new scheduling class. In our case, considering the `rt_sched_class`, defined by POSIX, with the `SCHED_RR` and `SCHED_FIFO` policies, the EDF policy has been integrated into the real-time class to accomplish compliance with the POSIX standard; otherwise, the definition of new system calls would hinder the portability. Thus, integrating the EDF policy into the `rt_sched_class` allows the use of the system calls defined on the Linux API, ensuring the creation and implementation of POSIX-compliant systems. It is an important feature for RTS developers to get the portability of the applications. As the state of the art survey showed, no previous projects accomplish simultaneously with the features indicated in Table 1.

The EDF policy is implemented into the `rt_sched_class`, as it is shown in Figure 4. In addition, the incorporation does not interfere with the already defined code in the kernel, allowing the leverage of the capabilities offered by the `rt_sched_class`.

As Figure 4 shows, a single structure holds the active tasks within the real-time class. The tasks are scheduled under different policies: `SCHED_FIFO`, `SCHED_RR` or `SCHED_EDF`. This

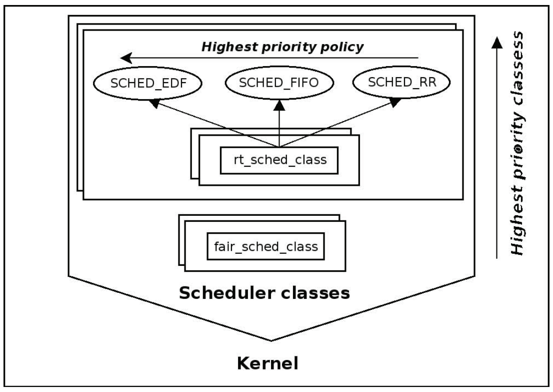

Figure 4 Incorporation of the EDF policy `rt_sched_class` into the kernel.

architecture guarantees that any task associated with the EDF policy is selected first even if other tasks with different policies exist. Therefore, the tasks associated with the EDF policy have the highest priority.

The integration of the EDF algorithm requires the incorporation of the relative and absolute deadline parameters. We added two attributes to the general structure `task_struct`, to allow the definition of the task deadlines:

```
struct task_struct {
    ...
    struct sched_rt_entity rt;
    #ifdef CONFIG_SCHED_EDF_POLICY
    struct timespec rel_deadline;
    struct timespec abs_deadline;
    #endif
    ...
};
```

The variables `rel_deadline` and `abs_deadline` store the deadlines, respectively. They are defined as `timespec` type, which accepts the time values with a resolution up to nanoseconds. In this way, real-time tasks can handle more realistic situations in which the time precision is important. Although the application developer specifies the relative deadline of a task, the absolute deadline is calculated and mapped internally, according to the relative deadline, each time a task is activated. This process is completely transparent for a user. The variable `sched_rel_deadline` is integrated into the `sched_param` structure, to allow the user to define the deadline of a task: relative and absolute

```
struct sched_param {
    int sched_priority;
    #ifdef CONFIG_SCHED_EDF_POLICY
    struct timespec sched_rel_deadline;
    #endif
};
```

A developer may schedule a process using the `SCHED_EDF` policy, and define the process parameters through the `sched_setscheduler` system call. This guarantees the compliance with the POSIX standard. A state diagram of a task is shown in Figure 5.

The Linux scheduler should be extended, particularly the real-time class, to consider first EDF among all policies, also to update the absolute deadlines of EDF-scheduled tasks, both at every scheduling time. For instance, when a task is added to the system, the `enqueue_task_rt()`[1] function is invoked. The insertion of a new task in the active task queue is done according to the policy and the priority level assigned, remembering that each priority level of the Linux system has a structure that keeps the tasks sorted according to the policy. Similarly, the `check_preempt_curr_rt()` function was modified. It is invoked when a task is activated after being on standby, either by an interrupt or by suspending for a certain time interval, the task must be rescheduled and the task queue reordered at the assigned priority level if it changed its priority. The `pick_next_rt_entity()` function selects the next task to be executed, according to the policies defined in the real-time class. In addition, this function verifies if there are tasks that are scheduled with the EDF policy, if that is the case, the first task of the structure is selected because they are sorted by their absolute deadline. If no other task of the EDF type exists, it continues to check the other existing real-time policies.

Usually the tasks are initialized with certain parameters, such as the scheduling policy and the task priority, among others. In addition, the task parameters are modified at runtime through the use of functions such as `switched_to_rt()` or `prio_changed_rt()`. In these cases, it is necessary to verify that the task scheduled by the RTOS is in the correct task queue, either in that class to which it now belongs, or at that priority level to which it changes. The `set_curr_task_rt()` function was also modified; it is invoked at any time instant once a task priority or a task parameter in the `task_struct` is assigned through the system call `sched_setscheduler()`. The major changes made in the kernel are presented in Figure 6.

It is important to note that despite the fact the new scheduling policy is intended for uniprocessor systems, it works correctly in multi-core platforms by selecting the use of a single core during the kernel configuration process, before its compilation.

4.4 Support threads through GLIBC

In Linux, the applications are executed using two ways: kernel and user modes [39]. In the first mode, the OS tasks, such as

[1]The new prototype `trace_sched_edf_abs_deadline` and modified functions `enqueue_task_rt()`, `check_preempt_curr_rt()`, `pick_next_rt_entity()`, `switched_to_rt()`, `prio_changed_rt()`, `set_curr_task_rt()` are described as separate files in the repository: https://drive.google.com/drive/folders/0ByXeZjrDUvExVUtnMWJyUVpvN1k?usp=sharing

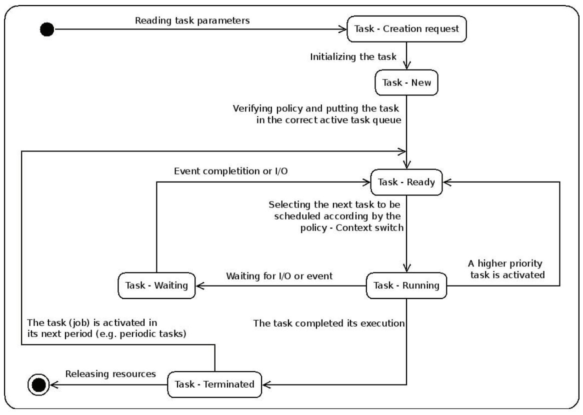
Reading task parameters
Task - Creation request
Initializing the task
Task - New
Verifying policy and putting the task in the correct active task queue
Event completition or I/O
Task - Ready
Selecting the next task to be scheduled according by the policy - Context switch
A higher priority task is activated
Task - Waiting
Waiting for I/O or event
Task - Running
The task (job) is activated in its next period (e.g. periodic tasks)
The task completed its execution
Releasing resources
Task - Terminated

Figure 5 State diagram of a task during its life cycle.

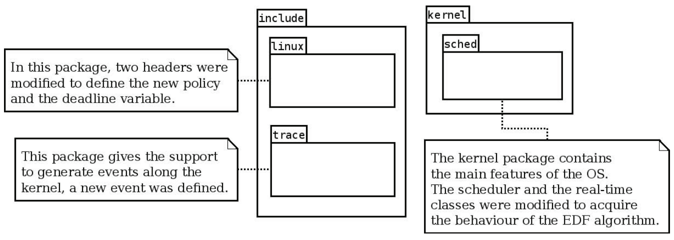
include
linux
trace
kernel
sched
In this package, two headers were modified to define the new policy and the deadline variable.
This package gives the support to generate events along the kernel, a new event was defined.
The kernel package contains the main features of the OS. The scheduler and the real-time classes were modified to acquire the behaviour of the EDF algorithm.

Figure 6 A package diagram of the major changes in the code and header files.

the scheduler, can be executed. In the second one, only user programs, as the text editor, are run. When a system call is invoked, it is executed in the kernel mode. Once completed, the program continues to run in the user mode. System calls are available by the use of an API, which carries out the communication between the user programs and the own OS functions, in a flexible and transparent manner. This lets developers to focus exclusively on the programming applications, without worrying about the internal OS architecture. A GLIBC library [22] is an API that enables the communication between the user programs and the kernel, providing a certain level of compatibility with the POSIX standard on the Linux environment (Figure 7). The GLIBC API was chosen for the integration of RTSs employing the new `SCHED_EDF` policy.

It was necessary to make certain adjustments in some code sections within the library for the development of RTSs employing threads and taking advantage of the `SCHED_EDF` policy. However, the GLIBC operation was not modified, only a few adaptations were done.

In Figure 8 it is shown how the new scheduling policy `SCHED_EDF` and the new parameter `sched_rel_deadline` were integrated into the `sched_param` structure of the GLIBC library. The `sched_param` structure is:

```
struct sched_param {
   int sched_priority;
   struct timespec sched_rel_deadline;
};
```

4.5 Tracer: FTRACE

FTRACE [40] is a tool that generates and stores the scheduling events to be analyzed thereafter. It is very helpful to verify the correctness of the system execution, to evaluate the performance, as well as for debugging. The event data are stored in a temporal ring buffer. In order to evaluate the performance of the implemented policy, a new event was integrated into the FTRACE, to store the data when an EDF-scheduled task is activated and its absolute deadline is updated. The last one was defined as a function called `trace_sched_edf_abs_deadline`. Its main purpose is to record the event in the ring buffer. Then the event is allowed to be exported into any user-defined format. It is important to mention that the events are implemented not as functions in the FTRACE, but as macros, which are called prototypes. Moreover, it should be mentioned that it is not the only way to obtain traces, but it is the most efficient way since it does not generate too much overhead.

After the definition, the prototype was integrated into the real-time class code, specifically in those sections, where the absolute deadlines of the tasks are updated. The use of the FTRACE tool is complex, due to the fact that it is scarcely documented, and perhaps it requires that the user has certain scripting skills. Another tool called TRACE-CMD eases the tracing process through the use of intuitive commands, to alleviate this problem [41].

TRACE-CMD executes the user programs whereby the event data are captured. The data produced is exported to a more understandable or friendly file format. In our integration, the tracing data is converted to the KIWI format. The KIWI application [42] allows the graphical inspection of the events generated by FTRACE. A procedure for the generation and conversion of the event data is given in Figure 9.

5. EXPERIMENTAL TEST

In this section, an experimental test of the EDF policy is discussed. The experiment was conducted on an Intel Pentium IV

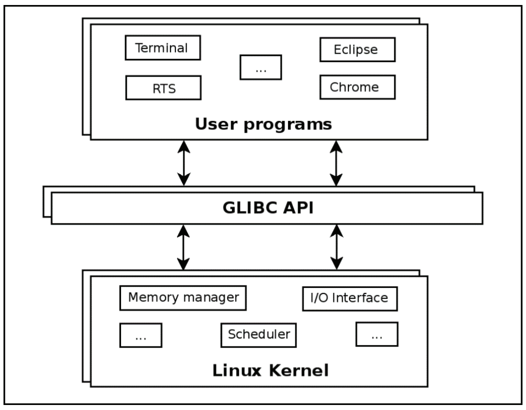

Figure 7 Communication between the user programs and the Linux kernel through GLIBC API.

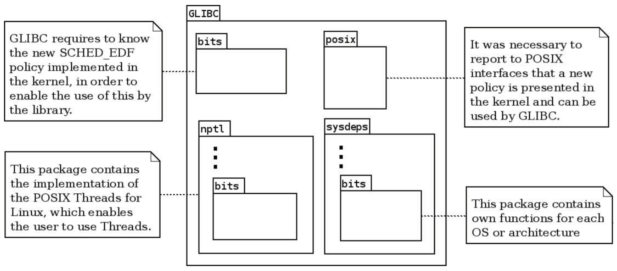
GLIBC
bits
posix
nptl
sysdeps
bits
bits
GLIBC requires to know the new SCHED_EDF policy implemented in the kernel, in order to enable the use of this by the library.
It was necessary to report to POSIX interfaces that a new policy is presented in the kernel and can be used by GLIBC.
This package contains the implementation of the POSIX Threads for Linux, which enables the user to use Threads.
This package contains own functions for each OS or architecture

Figure 8 A package diagram of the files modified in GLIBC.

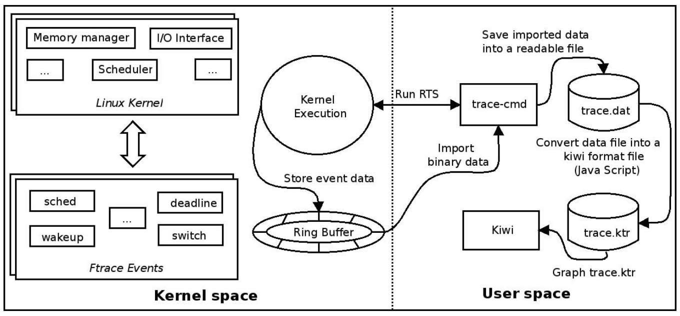

Figure 9 Data collection for an end user.

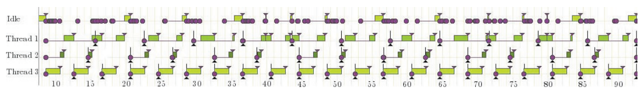

Figure 10 Task set scheduled using RM.

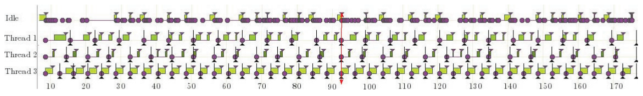

Figure 11 Task set scheduled using EDF (two hyperperiods).

2.8 GHz computer with 1.5 GB RAM at 400 MHz, 20 GB HD at 150Mpbs. It has only one core to provide a uniprocessor behavior. The test of the EDF and RM policies was effectuated using two corresponding applications developed with the aim to simulate realistic scenarios. Three independent threads were created, with different deadlines and WCETs. The parameters of the tasks used are shown in Table 2.

Calculating the total processor utilization U_T we get:

Table 2 Parameters of the task set (time is given in ms).

Thread	C_i	D_i	T_i
1	290	700	700
2	50	600	600
3	190	400	400

$$U_\tau = \frac{290}{700} + \frac{50}{600} + \frac{190}{400},$$
$$U_\tau = 0.972.$$

The exact schedulability test introduced by Lehoczky [33] was used to verify whether the task set is schedulable using the RM scheduling policy:

$$
\begin{aligned}
S_3 &= \{400, 600, 700\},\\
W_1 &= C_1 + C_2 + C_3 = 530 \le 400,\\
W_2 &= C_1 + C_2 + C_3 = 720 \le 600,\\
W_3 &= C_1 + 2C_2 + 2C_3 = 770 \le 700.
\end{aligned}
$$

The test showed that Thread 1 is not capable to accomplish the execution time before the corresponding deadline even in the first activation. Threads 2 and 3 show the same behavior.

Figure 10 shows that the tasks are not schedulable under the RM policy. This means that some deadlines are not accomplished, because the first job of Thread 1, which has the lowest priority, misses the deadline. If the parameters cannot be modified, the tasks would not be executed correctly under the existing scheduling policies in RT-Preempt. However, since the total processor utilization satisfies $U_t \le 1$, the task set can be correctly scheduled using the EDF policy.

Figure 11 shows the execution chart of the task set using EDF. The arrows in upward direction represent the activations and deadlines of the threads. It is also observed that with the EDF policy:

- All task jobs meet the respective deadlines.
- Non-real-time tasks (grouped in Idle) are processed only when real-time tasks do not require to be executed.
- The schedule is similar on both hyperperiods, which are separated by a vertical red line.

In the program, which performs the tests[2], one can observe how the tasks were created using both `SCHED_EDF` and `SCHED_FIFO` policies. It is noteworthy that the `SCHED_FIFO` policy is used to resemble the behavior of RM.

The operations performed by the tasks are described below. The `clock_nanosleep()` function was used to define the periodic activation of a task:

```
void * inThread(...) {
  struct timespec periodActivation,
  nextActivation, now;
  ...
  /*Start the execution of each thread
  at the same time*/
  nextActivation = initialTime;
  clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
                  &nextActivation, NULL);
  /**** Do some operations ****/
  ...
  while (1) {
  clock_gettime(CLOCK_MONOTONIC, &now);
  timespec_add(&nextActivation,
               &periodActivation)
  /*The thread is set to sleep until
  next activation time*/
  clock_nanosleep(CLOCK_MONOTONIC,
  TIMER_ABSTIME, &nextActivation, NULL);
  /**** Do some operations ****/
  ...
  }
  pthread_exit(NULL);
}
```

[2]https://drive.google.com/drive/folders/ 0ByXeZjrDUvExVUtnMWJyU-VpvN1k?usp=sharing

Table 3 Response time of real-time tasks with the RM and EDF policies (average time in microseconds).

Policy	context switch	task wakeup	preemptions points
`SCHED_FIFO` (RM)	11.077	2.310	14
`SCHED_EDF`	12.344	3.364	7

In Table 3, some metrics obtained during the first hyperperiod are presented. It shows that the required time to switch from one task to another one (context switch) with EDF is larger than with RM. Moreover, there is a double number of preemption points using the RM policy. In addition, each time a task needs to be activated, EDF uses more time than RM. Nonetheless, the preemption number depends on the algorithm employed to schedule the tasks. It is demonstrated by Buttazzo [6] that RM introduces more preemptions than EDF. Therefore, even if `SCHED_FIFO` requires less time to accomplish context switches, the times of `SCHED_EDF` are minimized during its execution because it does not produce many preemptions.

6. CONCLUSIONS AND FUTURE WORK

In this paper, it was shown that Linux is an appropriate OS to run hard RTSs modifying the real-time scheduling class and providing its higher priority over other classes. The integration of the EDF scheduling policy into the real-time class for uniprocessor RTSs is an aspect that we highlighted. Since EDF allows a greater processor utilization than RM, its integration into the Linux kernel allows the implementation of real-time systems that otherwise could not be implemented using the existing scheduling policies. In the conducted experiment, EDF showed a better performance than RM. We included traces into the scheduler class to ease the application debugging and thus, we provided a compatibility with the FTRACE tracer. Another contribution of this work is to integrate a new scheduling policy for developers without defining a new class in the kernel, taking advantage of the real-time class benefits and incorporating the policy complying the POSIX Thread standard to achieve application portability. The scope of this work was widened enabling the use of `SCHED_EDF` via the GLIBC library, so that no new interfaces need to be adapted. A survey of the state of the art showed no previous projects accomplished simultaneously with the features indicated: hard RTS support, POSIX-compliance standard, no adaptions in the GLIBC library, thread scheduling.

The extension of the EDF-scheduling scheme to multiprocessor systems is planned as future work; this will enlarge the support to hard RTSs that require strong computational resources. We are also interested in integrating a shared resource protocol, allowing task models be more complex and realistic.

7. ACKNOWLEDGEMENT

This work has been supported by Autonomous University of Baja California, Engineering Institute and by National Council of Science and Technology.

REFERENCES

1. Srovnal, V, Kotzian, J (2008). Development of a flight control system for an ultralight airplane. In: Proceedings of the International Multiconference on Computer Science and Information Technology (IMCSIT 2008), Wisła, Poland, IEEE Computer Society.
2. Preckshot, G G (1993). Reviewing Real-Time Performance of Nuclear Reactor Safety Systems. U.S. Nuclear Regulatory Commission.
3. (2008) The Open Group Technical Standard Base Specifications. In: Edition IEEE Standard 1003.1, Institute of Electrical and Electronic Engineers and the Open Group, 7.
4. Buttazzo, G C (2011). Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Springer Publisher.
5. Davis, RI, Burns, A (2011). A Survey of Hard Real-time Scheduling for Multiprocessor Systems. ACM Computing Surveys, 43(4) 35:1–35:44.
6. Buttazzo, GC (2005). Rate Monotonic vs. EDF: Judgment day. Real-Time Systems, 29(1) 5–26.
7. Burns, A (1991). Scheduling hard real-time systems: a review. Software Engineering Journal 6(3) 16–28.
8. Sha, L, Abdelzaher, T, Årzén, K E, Cervin, A, et al. (2004). Real Time Scheduling Theory: A Historical Perspective. Real-Time Systems, 28(2-3) 101-155.
9. Trejo, K, Angulo, C (2016). Single-Camera Automatic Landmarking for People Recognition with an Ensemble of Regression Trees. Computación y Sistemas, 20(1) 19-28.
10. Faggioli, D, Checconi, F, Trimarchi, M, Scordino, C (2009). An EDF scheduling class for the Linux kernel. In: Proceedings of the 11th Real-Time Linux Workshop, Dresden, Germany.
11. Stankovic, JA, Ramamritham, K, Spuri, M (1998). Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms. Kluwer Academic Publishers.
12. Davis, RI (2014). A Review of Fixed Priority and EDF Scheduling for Hard Real-time Uniprocessor Systems. Real-Time Systems, 11(1) 8-19.
13. Brun, A, Guo, C, Ren, S (2015). A Note on the EDF Preemption Behavior in "Rate Monotonic Versus EDF: Judgment Day". Embedded Systems Letters, 7(3) 89-91.
14. Liu, CL, Layland, JW (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM, 20(1) 46–61.
15. RTAI home page, https://www.rtai.org/, Consulted at June 28, 2017.
16. XENOMAI home page. http://www.xenomai.org/, Consulted at June 28, 2017.
17. Yodaiken, V (1999). The RTLinux manifesto. In: Proceedings of the 5th Linux Conference, Raleigh, North Carolina, USA.
18. Koh, JH, Choi, BW (2013). Real-time Performance of Real-time Mechanisms for RTAI and Xenomai in Various Running Conditions. International Journal of Control and Automation, 6(1) 235-246.
19. Rostedt, S, Hart, DV (2007). Internals of the RT patch. In: Proceedings of the Linux Symposium, Ottawa, Canada, 161–172.
20. Calandrino, JM, Leontyev, H, Block, A, Devi, UC, et al. (2006). LITMUS-RT: A testbed for empirically comparing real-time multiprocessor schedulers. In: Proceedings of the 27th Symposium on Real-Time Systems, Rio de Janeiro, Brazil, 11–26.
21. Dellinger, M, Garyali, P, Ravindran, B (2011). CHRONOS Linux: A best-effort real-time multiprocessor linux kernel. In: 48th Design Automation Conference (DAC), NY, USA, 474–479.
22. GLIBC home page, https://www.gnu.org/software/libc/. (Consulted at June 28, 2017)
23. Faggioli, D, Trimarchi, M, Checconi, F, Bertogna, M, et al. (2009). An implementation of the earliest deadline first algorithm in Linux. In: Proceedings of the ACM symposium on Applied Computing, Honolulu, USA, 1984–1989.
24. Baltarejo, P, Pereira, N, Tovar, E (2012). Enhancing the real-time capabilities of the Linux kernel. Special Interest Group on Embedded Systems Review (SIGBED Review), 9(4) 45–48.
25. Lelli, J, Faggioli, D, Cucinotta, T (2011). An efficient and scalable implementation of global EDF in Linux. In: Proceedings of the 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Porto, Portugal, 6–15.
26. QNX Neutrino home. http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html. (Consulted at June 28, 2017).
27. VxWORKS home page. http://windriver.com/products/vxworks/. (Consulted at June 28, 2017).
28. INTEGRITY home page. http://www.ghs.com/products/rtos/ integrity.html. (Consulted at June 28, 2017).
29. Cottet, F, Delacroix, J, Kaiser, C, Mammeri, Z (2002). Scheduling in Real-Time Systems. Wiley Publisher.
30. Baruah, SK, Rosier, LE, Howell, RR (1990). Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. Real-Time Systems, 2(4) 301–324.
31. Baruah, SK, Goossens, J (2003). Rate-monotonic scheduling on uniform multiprocessors. In: Proceedings of the 23rd International Conference on Distributed Computing Systems, Providence, RI, USA, 52(7) 966–970.
32. Davis, RI, Cucu-Grosjean, L, Bertgona, M, Burns, A (2016). A review of priority assignment in real-time systems. Journal of Systems Architecture, 65(C) 64–82.
33. Lehoczky, J, Sha, L, Ding, Y (1989). The Rate Monotonic scheduling algorithm: exact characterization and average case behavior. In: Proceedings of the Symposium on Real Time Systems, Santa Monica, CA, USA, 166–171.
34. Wu, J (2016). Energy Efficient Dual Execution Mode Scheduling for Real-Time Tasks with Shared Resources. Computer Systems Science and Engineering. 31(3) 239–253.
35. Rajkumar, R, Lee, I, Sha, L, Stankovic, J (2010). Cyber-physical systems: The next computing revolution. In: Proceedings of the 47th ACM/IEEE Design Automation Conference (DAC), Anaheim, CA, USA, 731–736.
36. Schneider, R, Goswami, D, Masrur, A, Becker, M, et al. (2013). Multi-layered Scheduling of Mixed-criticality Cyber-physical Systems. Journal of System Architecture, 59(10) 1215–1230.
37. Vázquez-Santacruz, E, Cruz-Santos, W, Gamboa-Zúñiga, M (2015). Design and Implementation of an Intelligent System for Controlling a Robotic Hospital Bed for Patient Care Assistance. Computación y Sistemas, 19(3) 467–474.
38. Zhang, X, Zhang, H, Wu, Y, Dai, Guojun (2015). A SoPC design of a real-time high-definition stereo matching algorithm. Computer Systems Science and Engineering. 30(5).
39. Love, R (2010). Linux Kernel Development. Addison-Wesley Professional.
40. FTRCE home page. https://www.kernel.org/doc/Documentation/trace/ ftrace.txt. (Consulted at June 28, 2017)
41. TRACE-CMD home page.https://git.kernel.org/cgit/linux/kernel/ git/rostedt/trace-cmd.git. (Consulted at June 28, 2017)
42. KIWI home page (Consulted at June 28, 2017). http://www.gti-ia.upv.es/sma/tools/kiwi/index.php.