A Scheduling Scheme Extension of the Earliest Deadline First Policy for Hard Real-Time Uniprocessor Systems Integrated on POSIX Threads Based on Linux

Vidblain Amaro-Ortega, Arnoldo Díaz-Ramírez, Brenda Leticia Flores-Rios, Félix Fernando González-Navarro, Frank Werner, Larysa Burtseva

Abstract—In recent years, Linux has been employed to execute numerous real-time applications. However, it is limited to support soft real-time systems. For real-time systems with critical constraints, the soft real-time support and the scheduling policies offered by Linux 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]. An operating system (OS) is composed of a set of programs that act as an intermediary between applications and the computer hardware, and that manages all the available resources. It should be classified into a general purpose OS (GPOS) or a real-time OS (RTOS). In the first ones, their aim is to support the time-sharing applications and therefore to allocate equitably the available resources to the system’s tasks. Instead, an RTOS gives support to real-time applications, which incorporate policies that enable the compliance with the task’s timing constraints.

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.

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. A real-time extension is POSIX.1b, also known as the IEEE Std 1003.1b-1993, defines priority based scheduling, semaphores, real-time signals, among others. The POSIX.1c or IEEE Std 1003.1c-1995 which 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.

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 [9]. The two most known scheduling policies for hard RTSs are the Rate Monotonic (RM) and the Earliest Deadline First (EDF) [6, 10, 11, 12]. The RM algorithm, which assigns priorities inversely proportional to the task periods, is an optimal static priority assignment policy [13]. 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 [14, 15, 16, 17] and the OS scheduler level [18, 19, 20]. 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 [18], 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 [21]. 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 a 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

Most of the implementations of scheduling algorithms have been made for soft RTOSs or defining own system calls, which diverges from the POSIX standard [9, 19, 20, 22, 23]. Several OSs based on the Linux kernel offer solutions with different approaches to act as an RTOS [18, 19, 20], including OSs, which have restrictions to support the POSIX standard, managing own APIs [9, 19, 20, 22]. Different open source solutions have been proposed making Linux able to execute the tasks in a real-time mode.

SCHED_DEADLINE [9, 22, 24] 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 [18]. 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 [19] 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 [20] is based on the RT-Preempt 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 [15, 17], RTAI [14, 17] and RT-Linux [16] use a hypervisor level approach allowing Linux to behave like a 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 [14, 15, 16]. This solution turns out to be too invasive with respect to the original OS.

Commercial RTOSs, such as QNX Neutrino [25], VxWorks [26] and INTEGRITY [27], 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 [9, 19, 20, 22, 23] define own system calls 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 [19, 20] 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 \( \tau \) of \( n \) static tasks. Every task in \( \tau \) 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 \( \tau_i \). 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 \( \tau_i \); 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

\( \tau \) Task set.
\( i \) Task index, \( i = 1, 2, ..., n \).
\( j \) Activation index, \( j = 1, 2, ..., J \), 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 \( j \).

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 \( D_i = T_i \).

![Figure 1. Main parameters of a task i.](image-url)

3.2 The RTS

A 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 [28].

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 \[6, 10, 29\].

The RM and the EDF rules are frequently used as scheduling policies. They have been extensively studied in \[6, 10, 11, 12, 13, 30\]. Both policies are based on assigning priorities to the system tasks according to respective deadlines. 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 \[13\].

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. \[29\] 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 \[13\] 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:

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

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

\[
U_T = \sum_{i=1}^{n} \frac{C_i}{T_i}. \tag{2}
\]

Lehoczky et al. \[31\] 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 \(\tau\) be a set of \(n\) tasks of the periods \(T_1 \leq T_2 \leq \ldots \leq T_n\), 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:

\[
W_i(t) = \sum_{j=1}^{f} C_j \left\lfloor \frac{t}{T_j} \right\rfloor, \tag{3}
\]

where:

\[
L_i(t) = \frac{W_i(t)}{t},
\]
\[ L_i = \min_{0 \leq t \leq T_i} L_i(t), \]
\[ L = \max_{1 \leq i \leq n} L_i, \]
\[ S_i = \{ kT_k | k = 1, \ldots, \left\lfloor \frac{T_i}{T_j} \right\rfloor; j = 1, \ldots, i \}. \]

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 \( \tau_i \) 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 [13] introduced an exact EDF schedulability test for any periodic task set (6). A periodic task set is schedulable under the EDF policy if and only if:
\[ UT \leq 1. \quad (6) \]

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 RTSSs [5]. 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 [32].

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 [6]. 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: rt_sched_class holds levels from 0 to 99, and fair_sched_class from 100 to 139, where the highest priority tasks are associated with rt_sched_class [20]. In this way, the tasks with a low priority level are considered as real-time ones.

The Linux scheduler keeps the control on the active tasks; it temporarily stores them into the queue structures, where the priority levels of every class are represented by a structure. The type of the structure depends strictly on the class. The active tasks are managed according to an associated policy and a priority level. Then the classes are grouped into a general structure. A representation of the run queues in the Linux scheduler architecture is shown in Figure 3.

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 [18] 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:

- 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 [9, 22, 24], 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.
Table 1. RTOS supports featured in projects.

<table>
<thead>
<tr>
<th>Project</th>
<th>Hard RTs</th>
<th>POSIX</th>
<th>GLIBC</th>
<th>Thread policy</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHED_DEADLINE</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>[9, 22, 24]</td>
</tr>
<tr>
<td>Litmus-RT</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>[19]</td>
</tr>
<tr>
<td>ChronOS</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>[20]</td>
</tr>
<tr>
<td>Xenomai</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>[15, 17]</td>
</tr>
<tr>
<td>RTAI</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>[14, 17]</td>
</tr>
<tr>
<td>RT-Linux</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>[16]</td>
</tr>
<tr>
<td>SCHED_EDF</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

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 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 relative and absolute deadlines, respectively. They are defined as timespec type, which accepts the time values with a resolution up to nanoseconds. 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:

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.

The Linux scheduler should be extended, particularly the real-time class, to consider first EDF among all policies, also to update the absolute deadline of EDF-scheduled tasks, both at every scheduling time. For instance, when a task is added to the system, the enqueue_task_rt() function is invoked. Similarly, the check_preempt_curr_rt() function was modified. It is invoked when a task is activated after being on standby. The pick_next_rt_entity() function selects the next task to be executed, according to the policies defined in the real-time class. It was modified to include a new scheduling policy.

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(). 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 5.

---

1 The new prototype trace_sched_edf_absDeadline 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 in the appendices from A to E.
4.4 Support threads through GLIBC

In Linux, the applications are executed using two ways: kernel and user modes [33]. In the first mode, the OS tasks, such as 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 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 [21] 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 6). The GLIBC API was chosen for the integration of RTSs employing the new SCHED_EDF policy.

Figure 5. A package diagram of the major changes in the code and header files.

Figure 6. Communication between the user programs and the Linux kernel through GLIBC API.
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. Figure 7, mainly to recognize the new scheduling policy SCHED_EDF and the new parameter sched_rel_deadline, which were integrated into the sched_param structure:

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

Figure 7. A package diagram of the files modified in GLIBC.

### 4.5 Tracer: FTRACE

FTRACE [34] 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.

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 [35].

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 [36] 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 8.

![Figure 8. Data collection for an end user.](image)

## 5 Experimental test

In this section, an experimental test of the EDF policy is discussed. The experiment was conducted on an Intel Pentium IV 2.8 GHz computer with 1.5 GB RAM at 400 MHz, 20 GB HD at 150Mbps. 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.

<table>
<thead>
<tr>
<th>Thread</th>
<th>$C_i$</th>
<th>$D_i$</th>
<th>$T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>290</td>
<td>700</td>
<td>700</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>600</td>
<td>600</td>
</tr>
<tr>
<td>3</td>
<td>190</td>
<td>400</td>
<td>400</td>
</tr>
</tbody>
</table>

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

Calculating the total processor utilization $U_T$ we get:

$$U_T = \frac{290}{700} + \frac{50}{600} + \frac{190}{400},$$

$$U_T = 0.972.$$  

The exact schedulability test introduced by Lehoczky [31] was used to verify whether the task set is schedulable using the RM scheduling policy:

$S_3 = \{400, 600, 700\},$

$W_1 = C_1 + C_2 + C_3 = 530 \leq 400,$

$W_2 = C_1 + C_2 + 2C_3 = 720 \leq 600,$

$W_3 = C_1 + 2C_2 + 2C_3 = 770 \leq 700.$
The test showed that Thread 1 is not capable to accomplish the execution time before corresponding deadline even in the first activation. Threads 2 and 3 show the same behavior.

Figure 9 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 $U_t \leq 1$, the task set can be correctly scheduled using the EDF policy.

Figure 10 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, one can observe how the tasks were created using both SCHED_EDF and SCHED_FIFO policies. It is noteworthy that 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:

```c
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,
```

$^2$ A part of the test program is described in appendix F.
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.

<table>
<thead>
<tr>
<th></th>
<th>context switch</th>
<th>task wakeup</th>
<th>preemption points</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHED_FIFO (RM)</td>
<td>11.077</td>
<td>2.310</td>
<td>14</td>
</tr>
<tr>
<td>SCHED_EDF</td>
<td>12.344</td>
<td>3.364</td>
<td>7</td>
</tr>
</tbody>
</table>

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

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 adaptations 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.
REFERENCES


APPENDICES

Appendix A
enqueue_task_rt(...) {
...
#define CONFIG_SCHED_EDF_POLICY
if the task policy is SCHED_EDF
    get current time
    update task absolute deadline according to the current time and its relative deadline
    generate a trace of the absolute deadline calculated
#undef
...
}

Appendix B
check_preempt_curr_rt(...) {
#define CONFIG_SCHED_EDF_POLICY
if the task policy is SCHED_EDF
    if the absolute deadline of the task woken is less than the running task one
        invoke the function resched_task() for reschedule tasks
    else
        if the priority of the task woken is higher than the running task one
            invoke the function resched_task() for reschedule tasks
#else
    if the priority of the woken task is higher than the running task one
        invoke the function resched_task() for reschedule tasks
#endif
}

Appendix C
pick_next_rt_entity(...) {
    get the rt priority levels
    get the first priority level index containing rt tasks
#define CONFIG_SCHED_EDF_POLICY
    loop through each run queue
        loop through each task in the rt array of each priority level
            if index is not a RT priority level
                exit loop
        loop through each task on the priority level queue
            if task policy is not SCHED_EDF
                exit loop
            if task relative deadline is less or equal to 0
                exit loop
        if there is a next task in the current queue
            if absolute deadline of current task in the loop is less than the next one
                next task to be executed is current
else
    next task to be executed is current
    if the current priority level rt array is empty
        find the index of the next priority level rt array
        go to the beginning of the loop
    if there is not selected a next task to be executed
        find the index of the next priority level rt array in the
        current run queue
        select the next rt array of current run queue
    #else
        select the rt array according to the index
        next task to be executed is the first stored in the array
    #endif
    return the next task to be executed

Appendix D

switched_to_rt(...) {
    ...
    if task is in the active run queue and is not the same as the
    current running one
        ...
    else
        #ifdef CONFIG_SCHED_EDF_POLICY
            ...
            if the switched task policy is SCHED_EDF
                if task absolute deadline is less than running task
                    need to re-schedule running task
            #endif
            if task priority is higher than the running task one
                need to re-schedule running task
        }

prio_changed_rt(...) {
    ...
    if task is the current running one
        ...
    else
        #ifdef CONFIG_SCHED_EDF_POLICY
            if task policy is SCHED_EDF
                if task absolute deadline is less than running task one
                    need to re-schedule running task
                    exit
            else
                #endif
                if task priority is less than running task one
                    need to re-schedule running task
        }

Appendix E

set_curr_task_rt() {
    select the running task
    ...
}
ifdef CONFIG_SCHED_EDF_POLICY
    if the task policy is SCHED_EDF
        get current time
        update task absolute deadline according to the current time and its relative deadline
        generates a trace of the absolute deadline calculated
endif
...

Appendix F

int main(...) {
    pthread_t in, pr, out;
    pthread_attr_t inAttr, prAttr, ouAttr;
    struct sched_param param1, param2, param3;
    struct timespec tIn = {0,0}, tPr = {0,0}, tOu = {0,0};
    ...
    /*Initialize the thread attributes*/
    pthread_attr_init(&inAttr);
    ...
    /*Indicate that threads have explicit policy (SCHED_EDF
    *OR SCHED_FIFO)*/
    pthread_attr_setinheritsched(&inAttr,PTHREAD_EXPLICIT_SCHED);
    ...
    /*Set to each attribute the policy (SCHED_EDF OR SCHED_FIFO)*/
    pthread_attr_setschedpolicy(&inAttr, SCHED_EDF);
    ...
    /*Get the max priority of the SCHED_FIFO policy to be assigned on
    *each thread through their attributes. Each priority must be less
    *than the max and different each one. This section is only
    *required when RM behavior is needed*/
    max_priority = sched_get_priority_max(SCHED_FIFO);
    param1.sched_priority = max_priority-1;
    ...
    /*Define for each thread its period. This section is only
    *needed when EDF behavior is needed*/
    tIn.tv_nsec = 700000000;
    tPr.tv_nsec = 600000000;
    tOu.tv_nsec = 400000000;
    param1.sched_rel deadline = tIn;
    pthread_attr_setschedparam(&inAttr, &param1);
    ...
    /* Create threads*/
    pthread_create(&in, &inAttr, (void *) &inThread, NULL);
    ...
    pthread_join(in, NULL);
    ...
    pthread_exit(NULL);
}