A New Application-Defined Scheduling Implementation in RTLinux

Arnoldo Díaz, Ismael Ripoll, Alfons Crespo and Patricia Balbastre
Universidad Politécnica de Valencia
Camino de Vera s/n, Valencia, Spain
ardiara@doctor.upv.es, {iripoll,alfons,patricia}@disca.upv.es

Abstract

Scheduling theory has shown an impressive evolution due to the intensive research done in this area. As a result, a lot of scheduling algorithms has been proposed to support a large amount of applications. RTLinux uses a fixed-priority scheduler and although fixed-priority scheduling is suited for a large number of real-time applications, it has some drawbacks. Nevertheless, adding new scheduling policies is a complex and time-consuming task since the kernel's internal structure must be modified. In recent years, an Application Program Interface (API) to create Application-Defined Schedulers in a way compatible with POSIX has been proposed. This model allows the implementation of different scheduling policies in a portable way, without modifying the internal structure of the real-time operating system. In this document, the implementation of a new Application-Defined Scheduling model in RTLinux is presented. The Application-Defined schedulers are not implemented as special threads but as a set of primitive operations that are executed inside the kernel, avoiding unnecessary and costly context switches. Also in this paper, a new library of Application-Defined Schedulers and its API is presented. This library offers a wide range of scheduling policies. Using this library, the implementation of Real-Time systems can be done in an easy, efficient and consistent way. The use of the library makes possible the creation of systems with multiple schedulers working concurrently, every one at a different priority level. [1]

1 Introduction

In the past few years, a lot of new real-time scheduling algorithms have been proposed to support a large amount of applications, ranging from control and robotics to multimedia. Nevertheless, most commercial real-time operating systems offer only fixed-priority scheduling to schedule real-time applications. Fixed-priority scheduling is suited for large number of real-time systems, but not every application requirements are fully accomplished using a fixed scheduler alone. For example, the performance of some bandwidth servers is better when used with dynamic priorities schedulers [1][6]. Also, dynamic priority algorithms allow a better usage of the available processing resources and makes possible a wider range of applications than fixed priority ones [5]. RTLinux, by instance, uses a fixed-priority scheduler and although a dynamic-priorities one has been already implemented [4], adding new scheduling policies is a complex and time-consuming task since the kernel's internal structure must be modified.

Recently, M. Aldea and M. Gonzalez [2] proposed an application program interface (API) that enables applications to use application-defined scheduling algorithms in a way compatible with the scheduling model defined in POSIX. This API has been already implemented in MaRTE OS and RTLinux [7]. This API allows the implementation of different scheduling policies in a portable way, without modifying the internal structure of the real-time operating system.

Since the model is based on the use of a special thread to implement each application scheduler, it's necessary a double context switch in every scheduling decision. Although the API's implementations has shown acceptable overhead measures, it is evident that it can be improved. For this reason, a new more general and efficient API has been proposed [4] with similar capabilities. In the new approach, the application scheduler thread can be implemented in several ways for a better performance and the final design is left up to the implementors. Also, a notion of urgency is defined in the new model to order the threads in the kernel's ready queue. Finally, the thread's synchronization is considered through mutexes by adding the Stack Resource Protocol and the

[1]This work has been supported by the the European Union project IST-2001-35102

Priority Inheritance Policy.

In this document, the implementation of the new approach of the Application-Defined Scheduling in RTLinux is presented, altogether with a library of schedulers that make use of this model. Also, an API for the use of this library is presented. This library offers a wide range of scheduling policies. Using this library, the implementation of Real-Time systems can be done in an easy, efficient and consistent way. The use of the library makes possible the creation of systems with multiple schedulers working concurrently, every one at a different priority level. Additionally, deadline misses or overruns handlers can be defined. In the Application-Defined Schedulers Library, POSIX Trace facilities are already integrated.

This document is organized in the following way. In Section 2 an overview of the new model is presented. Section 3 shows the implementation strategy of the model in RTLinux. In Section 4, an example of the use of the new API is presented. Next, in chapter 5 the new library of schedulers and its API is introduced. Section 6 shows an example on the use of this library's schedulers API, and finally in Section 7 presents conclusions and future work.

2 Overview of the new ADS model

The application-defined scheduling former model is shown in Figure 1. In it, each application scheduler was a special kind of thread responsible of scheduling a set of threads attached to it. According to the way a thread is scheduled, it can be one of the following:

- *System-scheduled threads:* the operating system schedules directly these threads without intervention of the application-defined scheduler.
- *System-scheduled threads:* the operating system schedules directly these threads without intervention of the application-defined scheduler.

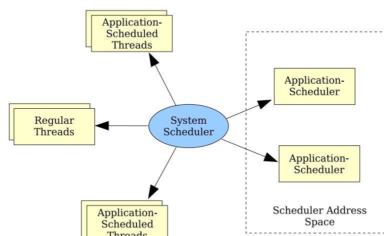

FIGURE 1: *Model for Application Scheduling*

Nevertheless, in the new model, each application scheduler is a special software module that can be implemented in several ways, without imposing any particular implementation. In the new framework, an application scheduler has the structure shown in Figure 2. The operating system may invoke a set of operations, contained in the application scheduler, every time a scheduling event occurs. In other words, every time an application-scheduled thread executes or experiences any of the scheduling events shown below, its corresponding application-defined scheduling operation is invoked. The scheduling events may be:

- a scheduled thread requests attachment to the scheduler or terminates
- a scheduled thread blocks or gets ready
- a scheduled thread changes its scheduling parameters
- a scheduled thread invokes the yield operation to give up the CPU
- a scheduled thread explicitly invokes the scheduler
- a scheduled thread inherits or uninheriths a priority, due to the use of a mutex
- a timeout expires
- a signal is generated for the scheduler
- a timed notification associated with a specific thread arrives
- a previous operation failed

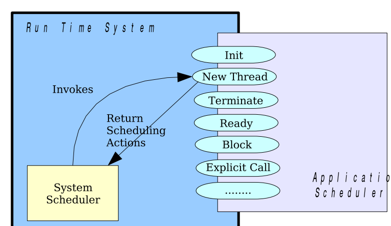

FIGURE 2: *Structure of an Application Scheduler*

To manage the scheduler events the use of a queue is suggested because when the scheduler executes there could be more than one pending scheduling event. The implementation must ensure that the application-defined scheduler always is executed before any of their scheduled threads. Once the scheduling event is processed, one or more scheduling actions may be executed. The system must ensure the sequential execution of the scheduling actions and the invocation of the scheduler primitives,

and for that purpose the use of a actions queue is also suggested.

There are some changes in the new proposed API. Now, application schedulers are created using a `posix_appsched_scheduler_opts` structure. This structure contains pointers to the primitive operations that are invoked by the system when a scheduling event occurs. An example of how could be declared this structure, using a subset of the scheduler operations, is shown next:

```
typedef struct  {

  void (*init) (void * sched_data, void *arg);

  void (*new_thread)
          (void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t actions,
            struct timespec *current_time);
  ...
  void (*thread_ready)
          (void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t actions,
            struct timespec *current_time);
  ...
  void (*thread_block)
          (void * sched_data,
            pthread_t thread,
            posix_appsched_actions_t actions,
            struct timespec *current_time);
  ...
} posix_appsched_scheduler_opts;
```

When an application scheduler is created, the structure `posix_appsched_scheduler_opts` will contain pointers only to the functions used in that particular scheduler, and null pointers to the functions not used or defined. The prototype of the function is shown next:

```
posix_appsched_scheduler_create
  (const posix_appscheduler_opts_t *sched_ops,
   sizeof_t scheduler_data_size,
   void * arg,
   size_t arg_size,
   posix_appsched_scheduler_id_t *sched_id);
```

Another improvement of the new model is the incorporation of the abstract notion of "urgency", a 64 bits unsigned integer assigned to every thread. With this new attribute, is possible to use the kernel's internal queue to order properly the application-scheduled threads in an efficient way.

3 Implementation

As mentioned above, the new abstract interface doesn't impose any particular implementation and this leads us to evaluate the best option. There are at least three implementation strategies and they will be discussed in this section. One of them is to leave the application-defined scheduler as a special thread, just as it is implemented right now in RTLinux and improve its performance with the use of the abstract notion of "urgency". This is done by assigning to every thread a particular numeric value, called *urgency*, and mapping to it any parameter chosen by the scheduler (deadline, laxity, ..). In this way, it is possible to use the kernel's ready queue to order the tasks at a particular priority level using a higher value as an indication of higher urgency. The implementation can be more efficient because when a task finishes its current job it's not necessary to invoke the application scheduler again to determine the next thread to execute since the kernel can choose the next task to execute by itself. Only when a new job arrives would be necessary to invoke the scheduler.

Another option is to execute the scheduler primitive operations in the kernel's context as soon as they are generated. This is perhaps the most efficient mechanism since it avoids costly and unnecessary context switches and has a small overhead.

A third option is explained next. If scheduled threads execute their primitive operations in the context of scheduled threads, double context changes are avoided with the additional advantage that is easier to implement than the previous option. When an application scheduled thread is chosen by the system to execute it invokes all pending primitive operations for its scheduler. Events associated to the execution of that thread are also executed by it. In case that there are no active scheduled threads, it will be necessary a service thread to process all pending scheduling events.

The second option, executing the scheduling operations in the kernel's context, was the one chosen to be implemented because it seems, at least in RTLinux, to be the most efficient and compact solution. It was necessary to modify the RTLinux kernel's internal structure but its good design and construction makes it feasible.

This implementation strategy has many advantages. The events and actions queues are not necessary and therefore not used since the scheduling events are processed when they occur and the scheduling actions are executed immediately. This eliminates overhead and gives a better performance. On the other hand, since the application-defined scheduler is not a thread anymore but a set of operations instead, there are not unnecessary and costly context switches, there's no need to protect the execution of the primitive operations by a mutex, in order to serialize its execution.

When an event occurs, the RTLinux kernel needs to know what primitive operation invoke. To do that, a new thread's attribute is defined:

`posix_appsched_scheduler_id_t appscheduler`

and when the application-defined scheduler is set for the thread, a pointer to its application-defined scheduler primitive operations is passed to this attribute.

To implement the new application-defined scheduling in RTLinux, some modifications have to be done to the kernel. In every scheduling point where a scheduling event related to an application-scheduled thread takes place, the corresponding primitive operation, if defined, is invoked.

For example, the `pthread_wait_np()` function has been modified as follows:

```
int pthread_wait_np(void)
{
 long interrupt_state;
 pthread_t self = pthread_self();
#ifdef CONFIG_OC_APPSCHED
 struct timespec appsched_current_time;
#endif
 rtl_no_interrupts(interrupt_state);
#ifdef CONFIG_OC_APPSCHED
 if ( regular_thread(self) &&
      self->appscheduler->thread_block} ){
        get current time
        invoke appscheduler operation
          thread_block
} else
#endif
{
 RTL_MARK_SUSPENDED (self);
}

__rtl_setup_timeout (self, self->resume_time);
rtl_schedule();
pthread_testcancel();
rtl_restore_interrupts(interrupt_state);
return 0;
}
```

Also, the function `rtl_schedule()` has been modified. Next, the pseudocode of that function is shown with the modifications done.

```
rtl_schedule()
{
 get current time
 set new_task=0
 loop through task list{
   expire all timers
   if task is application-defined scheduled
     and its activation timer has expired,
    the task->appscheduler->thread_ready()
     function is invoked, if defined

   new_task = highest priority task with
     pending signals, and if it is
     application-defined scheduled, at
     the same priority level new_task is
     the highest urgency value task
 }
 loop through task list{
    update one-shot timers if task may
       preempt new_task
 }
 newly selected task is not the old task? {
   swich to new_task
    newly selected task uses fpu? {
      save fpu registers
    }
 }
 handle the new_tasks pending signals.
}
```

4 Example of an Application-Defined Scheduler: EDF

In the next example, an EDF scheduler will be defined using the new Application-Defined Scheduling implementation. The application-defined scheduling parameters of EDF threads are contained in the structure `edf_th_t`:

```
typedef struct {
  int id;
  hrtime_t period, current_deadline;
  pthread_t thread;
}  edf_th_t ;
```

The EDF scheduler defines only the primitive operations `new_thread()`, `thread_ready()` and `thread_block()`. Since the other scheduler operations will not be used, they are set to null pointers.

The `new_thread()` operation is called every time an EDF thread is created. This operation calculates the thread's deadline, sets the urgency value accordingly and activates the thread:

```
void new_thread (void * sched_data,
             pthread_t thread,
             posix_appsched_actions_t *actions,
             struct timespec *current_time)
{
 edf_th_t edf_data;
 pthread_getappschedparam(thread,
                     (void *)&edf_data, NULL);
 edf_data.current_deadline = edf_data.period +
                 timespec_to_ns(current_time);
 posix_appsched_actions_addactivate(NULL, thread,
               URG(edf_data.current_deadline));
}
```

After the thread has finished its work for the current activation, the `thread_block()` operation is invoked, and it just suspends the thread:

```
void thread_block (void * sched_data,
              pthread_t thread,
              posix_appsched_actions_t *actions,
              struct timespec *current_time)
{
  posix_appsched_actions_addsuspend(NULL,thread);
}
```

The pseudocode of the scheduled threads is shown next:

```
void *edf_thread_code(void *arg){
  int size=0;
  struct timespec t;

  clock_gettime(CLOCK_RTL_SCHED, &t);

  while (!stop) {
    timespec_add_ns(&t, (hrtime_t) *arg);
    clock_nanosleep(CLOCK_RTL_SCHED,
                    TIMER_ABSTIME,&t,NULL);
    // do some useful work
       .....
  }
  pthread_exit(NULL);
  return 0;
}
```

Finally, the pseudocode of the init_module(), where the scheduler and the scheduled threads are created, is the following:

```
int init_module(void) {
  pthread_attr_t attr;
  struct sched_param sched_param;
  pthread_t task;
  edf_th_t edf_th;
  posix_appsched_scheduler_id_t
                  *edf_scheduler_id;
  int stop=0;
  posix_appsched_scheduler_ops_t
          edf_scheduler_ops =
             {NULL, new_thread, NULL,
             thread_ready, thread_blocked,
             NULL, NULL, NULL, NULL, NULL,
             NULL, NULL, NULL, NULL, NULL};
/*
    Scheduler creation
*/
  posix_appsched_scheduler_create(
                 &edf_scheduler_ops,
                 0, NULL, 0,
                 &edf_scheduler_id);
/*
    Application Scheduled threads creation
*/
  pthread_attr_init (&attr);
  sched_param.sched_priority = 10;
  edf_th.period=30;
  pthread_attr_setappschedulerstate(&attr,
                         PTHREAD_REGULAR)
  pthread_attr_setappschedparam(&attr,
                         (void *) &edf_th,
                         sizeof(edf_th));
  pthread_attr_setschedparam (&attr,
                          &sched_param);
  pthread_attr_setappscheduler(&attr,
                          edf_sched_id);
  pthread_create (&task, &attr,
                   edf_thread_code, 0);
  return 0;
}
```

5 Application-Defined Schedulers Library

In order to improve the facilities offered by RTLinux, in the past few years some scheduling policies had been added to it. Eearlist Deadline First [4] and Constant Bandwidth Server are some examples, and if this tendency continues we will end up with a large and complex kernel. As said before, using the application-defined scheduling API it is possible to create user-defined scheduling algorithms without modifying the kernel's internal structure, leaving only the fixed-priorities scheduler inside the kernel and the rest of scheduling policies defined outside. This approach seems to be better because the RTLinux kernel remains simple and compact, while a growing number of scheduling policies can still be added.

Even though the use of the new API makes easier the implementation of an unlimited set of scheduling policies, the creation of a Library of schedulers that use this model has a lot of advantages. It lets the final user implement real-time systems in an easy and consistent way, with a wide range of scheduling algorithms available, and without the need of implement them. Also, since the Application-Defined Scheduling model in RTLinux works with almost no overhead, every library's scheduler is efficient. The integration of POSIX Trace and Fault Tolerance in the application-defined schedulers is easier. It's important to note that POSIX Trace facilities are already integrated in the application-defined schedulers library. In this section, some of the API of this Application-Defined Scheduler Library is presented.

The appschedlib_policy_t data type defines the scheduling policy to be used:

```
typedef enum {APPSCHEDLIB_RM, APPSCHEDLIB_EDF,
              APPSCHEDLIB_RM_DS,
              APPSCHEDLIB_RM_SS,
              APPSCHEDLIB_EDF_CBS,
              APPSCHEDLIB_EDF_IRIS, ...
              } appschedliblib_policy_t
```

To set the scheduling policy, the library's `appschedlib_set_schedpolicy()` function must be used. This function has the protoype shown next:

```
int appschedlib_set_schedpolicy(
                 appschedlib_sched_t * scheduler,
                 appschedliblib_policy_t policy,
                 int priority)
```

where `scheduler` is the schedulers identifier, policy the scheduling policy to be used, and priority the RTLinux system priority of the scheduled threads. Since there could be more than one scheduler working at the same time at its own priority level, the API checks that every schedulers uses a different priority value, otherwise it returns an error when a scheduler policy is defined with an already used priority level.

There are functions to get the policy and priority of a scheduler:

```
appschedlib_get_schedpolicy(
            appschedlib_sched_t * scheduler,
            appschedliblib_policy_t * policy)

appschedlib_get_schedpriority(
            appschedlib_sched_t * scheduler,
            int * prio)
```

In some applications it may be necessary the use of function handlers when some unexpected event occurs, as a deadline miss. It's possible the definition of scheduler handlers for overrun and deadline misses, and also is possible to define thread specific handlers. Once the scheduling policy is set, it's possible to define the scheduler's overrun and a deadline-miss handler, through the use of the following function:

```
int appschedlib_set_schedhandlers(
          appschedlib_sched_t * scheduler,
          void (* overrun_handler)(void),
          void (* deadline_miss_handler)(void))
```

where overrun_handler and deadline_miss_handler are pointers to handlers functions, if defined.

An specific scheduling policy may use one ore more Aperiodic Servers, like the Deferrable or Constant Bandwidth Servers. Every server must be defined along with their period and budget with this function:

```
appschedlib_set_serverparam(
            appschedlib_sched_t * scheduler,
            appschedlob_server_t* server,
            int period, hrtime_t budget)
```

Threads could be System Scheduled or Application-Defined-Library Scheduled. Application-Defined-Library Scheduled threads are created using the `pthread_create()` POSIX function, but before they are executed they must be attached to a scheduler or server. To attach a thread to a server, the next function is defined:

```
pthread_attr_set_appschedlib_scheduler_np(
                  pthread_attr_t *attr,
                  appschedlib_sched_t scheduler)
```

The thread's system priority will be the same of the scheduling policy. The thread's system priority doesn't has to be set after it's been attached to its scheduler, and if a user sets a system priority value for an application-defined-library scheduled thread other than the scheduler's one, the correct behavior of the system is not guaranteed.

To set the aperiodic server by which a given thread is scheduled, the following function must be used:

```
pthread_attr_set_appschedlib_server_np(
                    pthread_attr_t *attr,
                    appschedlib_server_t server)
```

Additional attributes can be defined for the application-defined library scheduled thread, like period, deadline or promotion time. The thread's period is defined by the use of the next function:

```
pthread_attr_set_appschedlib_period_np(
                   pthread_attr_t * attr,
                   hrtime_t period)
```

Functions to retrieve thread's attributes are also defined. For example, to get the thread's promotion time:

```
pthread_get_appschedlib_promotiontime_np(
                  pthread_t * thread,
                  hrtime_t * promotiontime)
```

In the next section, an example of the use of this library is shown.

6 Application-Defined Scheduler Library at work: EDF and CBS

To show how the Application-Defined Schedulers Library works, the pseudo-code of the `init()` function of an example program is shown. This program uses an EDF with CBS scheduling policy. It also defines two threads, one scheduled by the EDF scheduler and the other one by the aperiodic server.

```
#include  <appschedlib.h>

int init_module(void) {

 pthread_attr_t attr;
 struct sched_param sched_param;
 pthread_t edf_thread, cbs_thread;
 appschedlib_sched_t *edf_cbs_sched_id;
 appschedlib_server_t *cbs_server_id;

 //  set edf_cbs scheduling policy
 //             with prioity = 10
 appschedlib_set_schedpolicy(&edf_cbs_sched_id,
                      APPSCHEDLIB_EDF_CBS,10);

 //   set server parameters: period and budget
 appschedlib_set_serverparam(&edf_cbs_sched_id,
                         &cbs_server_id,20,5);

 // create edf thread
 pthread_attr_init (&attr);

 // define thread's period (deadline = period)
 pthread_attr_set_appschedlib_period_np(
    &attr,15);

 // set application-defined library scheduler
 //                         for this thread
 pthread_attr_set_appschedlib_scheduler_np(
    &attr, edf_cbs_sched_id);

 // create thread
 pthread_create (&edf_thread, &attr,
                 edf_thread_code, 0);

 // create cbs thread
 pthread_attr_init (&attr);

 // set application-defined library server
 //                     for this thread
 pthread_attr_set_appschedlib_server_np(&attr,
                             cbs_server_id);
 // create thread
 pthread_create (&cbs_thread, &attr,
                 cbs_thread_code, 0);

  return 0;
}
```

7 Conclusions and future work

The Application-Defined Scheduling model allows users to implement a lot variety of scheduling algorithms without modifying the real-time operating system structure. A new more efficient model has been recently proposed and since a particular implementation of the model is not imposed, the best implementation strategy has to be chosen for every real-time operating system. The aim of this paper is to show the implementation of this new model in RTLinux. In this implementation, the scheduling events are executed in the kernel's context as soon as they are generated, and the scheduling actions are executed immediately, avoiding costly context switches and making possible the implementation of any scheduling policy in an efficient and easy way, with almost no execution overhead.

Also, a new Library of Application-Defined Schedulers and its API is presented. With this library, is possible to use a large amount of scheduling algorithms in RTLinux, in an efficient, easy and consistent way, without modifying the real-time operating system structure.

Further work includes the a test suite for this implementation, including conformance, functional, stress and performance tests.

References

[1] L. Abeni and G. Buttazzo, 1998, *Integrating Multimedia Applications in Hard Real-Time Systems*, In IEEE Real-Time Systems Symposium.

[2] M. Aldea and M. Gonzalez, 2002, *POSIX-Compatible Application-Defined Scheduling in MaRTE OS*, In Proceedings of 14th Euromicro Conference on Real-Time System, pp.67-75.

[3] M. Aldea and M. Gonzalez, 2004, *A New Generalized Approach to Application-Defined Scheduling*, In Proceedings of 16th Euromicro Conference on Real-Time System.

[4] P. Balbastre and I. Ripoll, 2001, *Integrated Dynamic Priority Scheduler for RTLinux* , Third Real-Time Linux Workshop.

[5] C.L. Liu and J.W. Layland, 1973, *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, Journal of the ACM, pp.44-61.

[6] L. Marzario, G. Lipari, P. Balbastre, A. Crespo, 2004, *IRIS: A New Reclaiming Algorithm for Server-Based Teal-Time System*, RTAS.

[7] J. Vidal, I. Ripoll, A. Crespo and P. Balbestre, 2003, *Application-Defined Scheduler Implementation in RTLinu*, Fifth Real-Time Linux Workshop.