POSIX-Compatible Application-Defined Scheduling in MaRTE OS

Mario Aldea Rivas and Michael González Harbour

Departamento de Electrónica y Computadores
Universidad de Cantabria
39005-Santander, SPAIN
{aldeam,mgh}@unican.es

***Abstract**: This paper presents 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. Several application-defined schedulers, implemented as special user threads, can coexist in the system in a predictable way. This API is currently implemented on our operating system MaRTE OS. We plan to propose it for a future revision of the POSIX standard.*

1. Introduction[1]

Real-time POSIX [1][7] defines a preemptive fixed priority scheduling policy, together with two other compatible scheduling policies: a round robin within priorities policy and the sporadic server policy [5]. Although fixed priority scheduling is an excellent choice for real-time systems, there are application requirements that cannot be fully accomplished with these policies only. It is well known that with dynamic priority scheduling policies it is possible to achieve higher utilization levels of the system resources than with fixed priority policies. In addition, there are many systems for which their dynamic nature make it necessary to have very flexible scheduling mechanisms, such as multimedia systems, in which different quality of service properties need to be traded against one another.

It could be possible to incorporate into the POSIX standard new dynamic scheduling policies to be used in addition to the existing policies [14]. The main problem is that the variety of these policies is so great that it would be difficult to standardize on just a few. Different applications needs would require different policies. Instead, in this paper we propose defining an interface for application-defined schedulers that could be used to implement a large variety of scheduling policies. A preliminary version of this interface was presented in [3]. This interface is integrated into the POSIX standard and it is being submitted for consideration by the Real-Time POSIX Working Group.

The proposed interface is currently implemented in our operating system MaRTE OS (Minimal Real-Time Operating System for Embedded Applications) [4], which is a real-time kernel that follows the Minimal Real-Time POSIX.13 subset [6], providing both the C and Ada language POSIX interfaces. It allows cross-development of Ada and C real-time applications. Mixed Ada-C applications can also be developed, with a globally consistent scheduling of Ada tasks and C threads.

The paper is organized as follows: Section 2 discusses some related work on application-defined scheduling and sets the justification for our proposal. Section 3 discusses our model for application-defined scheduling. In Section 4 the C language application program interface (API) is described and in Section 5 an example of its use is shown. Section 6 presents some performance metrics showing the overhead of using our implementation. Section 7 gives our conclusions and future work.

2. Related Work and Motivation

The idea of application-defined scheduling has been used in many systems. A solution is proposed in RED-Linux [8], in which a two-level scheduler is used, where the upper level is implemented as a user process that maps several quality of service parameters into a low-level attributes object to be handled by the lower level scheduler. The parameters defined are the thread priorities, start and finish times, and execution time budget. With that mechanism some scheduling algorithms can be implemented but there may be others that cannot be implemented if they are based on parameters different from those included in the aforementioned attributes object. In addition, this solution does not address the implementation of protocols for shared resources that could avoid priority inversion or similar effects.

1. This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grants TIC99-1043-C03-03 and 1FD 1997-1799 (TAP)

A different approach is followed in the CPU Inheritance Scheduling [9], in which the kernel only implements thread blocking, unblocking and CPU donation, and the application defined schedulers are threads which donate the CPU to other threads. In this approach the only method used to avoid priority inversion is the priority inheritance. Although other synchronization policies could be implemented, the lack of an interface to trigger scheduling decisions by the use of mutexes makes it difficult or impossible to implement general synchronization protocols, which may be a limitation for special application-defined policies. In addition although this approach supports multiprocessor schedulers, it is not possible to have one single-threaded scheduler to schedule threads in other processors. Some multiprocessor architectures, for example using one general-purpose processor running the scheduler and an array of digital signal processors running the scheduled threads, may require that capability.

Another common solution is to implement the application scheduling algorithms as modules to be included or linked with the kernel (S.Ha.R.K [10], RT-Linux [12], Vassal [13]). With this mechanism the functions exported by the modules are invoked from the kernel at every scheduling point. This is a very efficient and general method but as a drawback, the application scheduling algorithms can neither be isolated from each other nor from the kernel itself, so, a bug in one of them could affect the whole system.

In our approach the application scheduler is invoked at every scheduling point like with the kernel modules, so the scheduler can have complete control over its scheduled threads. But in addition, our application scheduling algorithm is executed by a user thread. This fact implies two important advantages from our point of view:

a) The system reliability can be improved by protecting the system from the actions of an erroneous application scheduler. For efficiency, our interface allows execution of the application-defined scheduler in an execution environment different than that of regular application thread, for example inside the kernel. But alternatively, the interface allows the implementation to execute the scheduler in the environment of the application, to isolate it from the kernel. In this way, high priority threads that are critical cannot be affected by a faulty scheduler executing at a lower priority level.

b) The application scheduling code can use standard interfaces like those defined in the POSIX standard. In some systems part of these interfaces might not be accessible for invocation from inside the kernel.

We have designed our interface so that several application-defined schedulers can be defined, and so that they have a behavior compatible with other existing scheduling policies in POSIX, both on single processor and multiprocessor platforms. In addition, the interface needs to take into account the implementation of application-defined synchronization protocols.

The dynamic scheduling mechanism proposed for Real-Time CORBA 2.0 [11] represents an object-oriented interface to application-defined schedulers, but it does not attempt to define how that interface communicates with the operating system. The interface presented in this paper is the OS low-level interface, and thus an RT CORBA implementation could use it to support the proposed dynamic scheduling interface.

In summary, the motivation for this work is to provide developers of applications running on top of standard operating systems (POSIX) with a flexible scheduling mechanism, handling both thread scheduling and synchronization, that enables them to schedule dynamic applications that would not meet their requirements using the more rigid fixed-priority scheduling provided in those operating systems. This mechanism allows isolation of the kernel from misbehaved application schedulers. In addition, we wish to provide this mechanism both for applications developed in C or Ada.

3. Model for Application-Defined Scheduling

Figure 1 shows the proposed approach for application-defined scheduling. Each application scheduler is a special kind of thread, that is responsible of scheduling a set of threads that have been attached to it. This leads to two classes of threads in this context:

- *Application scheduler threads*: special threads used to run application schedulers.
- *Regular threads*: regular application threads

The application schedulers can run in the context of the kernel or in the context of the application. This allows implementations in which application threads are not trusted, and therefore their schedulers run in the context of

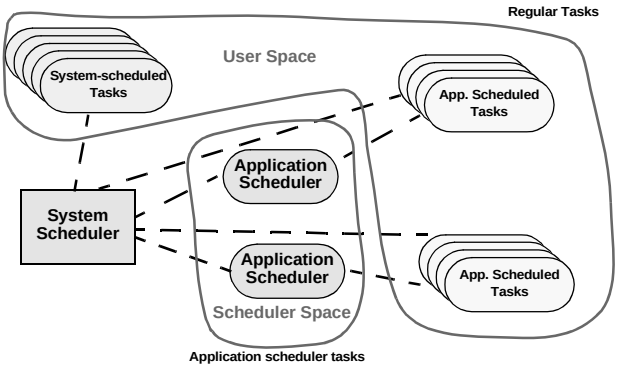

Figure 1. Model for Application Scheduling

the application, as well as implementations for trusted application schedulers, which can run more efficiently inside the kernel. Because of this duality we will model the scheduler threads as if they run in a separate context, which we call the scheduler space. The main implication of this separate space is that for portability purposes the application schedulers cannot directly share information with the kernel, nor with regular threads, except by using the POSIX shared memory objects, which is the mechanism for sharing memory among entities with different address spaces.

According to the way a thread is scheduled, we can categorize the threads as:

- *System-scheduled threads*: these threads are scheduled directly by the operating system, without intervention of a scheduler thread.
- *Application-scheduled threads*: these threads are also scheduled by the operating system, but before they can be scheduled, they need to be activated by their application-defined scheduler.

Although an application scheduler thread can itself be application scheduled, implementations should not be required to support this feature, because usually these threads will be system scheduled.

Because the use of mutexes may cause priority inversions or similar delay effects, it is necessary that the scheduler thread knows about their use, to establish its own protocols adapted to the particular thread scheduling policy. As we show in Figure 2, two kinds of mutexes will be considered:

- *System-scheduled mutexes*. Those created with the current POSIX protocols: no priority inheritance (`PTHREAD_PRIO_NONE`), immediate priority ceiling (`PTHREAD_PRIO_PROTECT`), or basic priority inheritance (`PTHREAD_PRIO_INHERIT`). They can be used to access resources shared between application schedulers, between sets of application-scheduled threads attached to different schedulers, or even between an application scheduler and its scheduled threads (in this case the mutex and its protected state must be placed in a POSIX shared memory object).
- *Application-scheduled mutexes*: Those created with `PTHREAD_APPSCHED_PROTOCOL`. The behavior of the protocol itself is defined by the application scheduler. The kernel notifies the scheduler about the request to lock one such mutex, the execution of an unlock operation, or when a thread blocks on one of these mutexes. After the lock request operation the application scheduler can chose to grant or not the mutex to the requesting thread. The block event might not be necessary in some schedulers that implement non-blocking synchronization protocols.

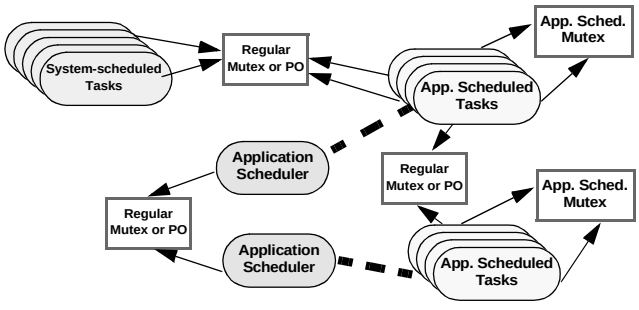

Figure 2. Model for Application-Defined Synchronization

3.1. Relations with Other Threads

Each thread in the system, whether application- or system-scheduled, has a system priority:

- For system-scheduled threads, the system priority is the priority defined in its scheduling parameters (`sched_priority` field of its `sched_param` structure), possibly modified by the inheritance of other priorities through the use of mutexes.
- For application-scheduled threads, the system priority is lower than or equal to the system priority of their scheduler thread. The system priority of an application-scheduled thread may change because of the inheritance of other system priorities through the use of mutexes. In that case, its scheduler also inherits the same system priority (but this priority is not inherited by the rest of the threads scheduled by that scheduler). In addition to the system priority, application-scheduled threads have *application scheduling parameters* that are used to schedule that thread contending with the other threads attached to the same application scheduler. The system priority always takes precedence over any application scheduling parameters. Therefore, application-scheduled threads and their scheduler take precedence over threads with lower system priority, and they are always preempted by threads with higher system priority that become ready. The scheduler always takes precedence over its scheduled threads.

If application-scheduled threads coexist at the same priority level with other system-scheduled threads, then POSIX scheduling rules apply as if the application-scheduled threads were scheduled under the FIFO within priorities policy (`SCHED_FIFO`); so a thread runs until completion, until blocked, or until preempted, whatever happens earlier. A thread running under the round-robin within priorities policy (`SCHED_RR`) runs until completion, until blocked, until preempted, or until its round robin quantum has been consumed, whatever happens earlier. Of

course, in that case the interactions between the different policies may be difficult to analyze, and thus the normal use will be to have the scheduler thread and its scheduled threads running at an exclusive range of system priorities.

In the presence of priority inheritance, the scheduler inherits the same priorities as its scheduled threads, to prevent priority inversions from occurring. This means that high priority threads that share resources with lower system-priority application threads must take into account the scheduler overhead when accounting for their blocking times.

3.2. Relations Between the Scheduler and its Attached Threads

When an application-defined thread is attached to its application scheduler, the latter has to either accept it or reject it, based upon the current state and the scheduling attributes of the candidate thread. Rejection of a thread causes the thread creation function to return an error.

Each application-defined scheduler may activate many application-scheduled threads to run concurrently. The scheduler may also block previously activated threads. Among themselves, concurrently scheduled threads are activated like `SCHED_FIFO` threads. As mentioned previously, the scheduler always takes precedence over its scheduled threads.

When an application-scheduled thread executes one of the following actions or experiences one of the following situations, a scheduling event is generated for its scheduler, unless the scheduling event to be generated is being filtered out (discarded).

- when a thread requests attachment to the scheduler
- when a thread blocks or gets ready
- when a thread changes its scheduling parameters
- when a thread invokes the *yield* operation
- when a thread explicitly invokes the scheduler
- when a thread inherits or uninherits a priority, due to the use of a system mutex
- when a thread does any operation on a application-scheduled mutex.

The application scheduler is a special thread whose code is usually a loop where it waits for a scheduling event to be notified to it by the system, and then determines the next application thread or threads to be activated.

Although the scheduler can activate many threads at once, it is a single thread and therefore its actions are all sequential. For multiprocessor systems this may seem to be a limitation, but for these systems several schedulers could be running simultaneously on different processors, cooperating with each other by synchronizing through regular mutexes and condition variables. For single processor systems the sequential nature of the scheduler should be no problem.

4. Interface

As said before, the proposed interface is integrated into POSIX. Some extra functionality has been added to the header files `<pthread.h>` and `<sched.h>` in order to allow the application to:

- handle scheduling events,
- manage and execute scheduling actions,
- create scheduler and scheduled threads,
- set and get application-defined scheduling parameters and the scheduling status of the scheduled threads,
- explicitly invoke the scheduler from a scheduled thread,
- and create application-scheduled mutexes.

The main aspects of this interface are described in detail in the following subsections.

4.1. Interfaces for the Scheduler threads

4.1.1. Scheduling Events

The scheduling events are stored in a FIFO queue until processed by the scheduler. The information included in these events is:

- the event code,
- the identifier of the thread that caused the event,
- and additional information associated with the event, and dependent on it: it can be an inherited (or uninherited) system priority, information related to an accepted signal stored in a `siginfo_t` object, a pointer to an application-scheduled mutex, or application-specific information.

The specific events that may be notified to a scheduler thread are shown in Table 1. An event for notifying preemption of a scheduled thread is not included. Although such event might seem to be useful to measure execution times from an application scheduler, it was difficult for the scheduler to know when a scheduled thread actually started executing. If measuring execution time is required, it is much simpler to use a POSIX execution time clock for that purpose.

4.1.2. Executing Scheduling Actions

Our interface defines an opaque type for storing a list of scheduling actions. The possible actions that can be added to one of these lists are the following:

Table 1: Scheduling Events

Event Code	Description	Additional information
POSIX_APPSCHED_NEW	A new thread has requested attachment to the scheduler	none
POSIX_APPSCHED_TERMINATE	A thread has been terminated	none
POSIX_APPSCHED_READY	A thread has become unblocked by the system	none
POSIX_APPSCHED_BLOCK	A thread has blocked	none
POSIX_APPSCHED_YIELD	A thread yields the CPU	none
POSIX_APPSCHED_SIGNAL	A signal belonging to the requested set has been accepted by the scheduler thread.	Signal-related information
POSIX_APPSCHED_CHANGE_SCHED_PARAM	A thread has changed its scheduling parameters	none
POSIX_APPSCHED_EXPLICIT_CALL	A thread has explicitly invoked the scheduler	Application message
POSIX_APPSCHED_TIMEOUT	A timeout has expired	none
POSIX_APPSCHED_PRIORITY_INHERINT	A thread has inherited a new system priority due to the use of system mutexes	Inherited system priority
POSIX_APPSCHED_PRIORITY_UNINHERIT	A thread has finished the inheritance of a system priority	Uninherited system priority
POSIX_APPSCHED_INIT_MUTEX	A thread has requested initialization of an application-scheduled mutex	Pointer to the mutex
POSIX_APPSCHED_DESTROY_MUTEX	A thread has destroyed an application-scheduled mutex	Pointer to the mutex
POSIX_APPSCHED_LOCK_MUTEX	A thread has invoked a “lock” operation on an available application- scheduled mutex	Pointer to the mutex
POSIX_APPSCHED_TRY_LOCK_MUTEX	A thread has invoked a “try lock” operation on an available application- scheduled mutex	Pointer to the mutex
POSIX_APPSCHED_UNLOCK_MUTEX	A thread has released the lock of an application-scheduled mutex	Pointer to the mutex
POSIX_APPSCHED_BLOCK_AT_MUTEX	A thread has blocked at an application-scheduled mutex	Pointer to the mutex
POSIX_APPSCHED_CHANGE_MUTEX_SCHED_PARAM	A thread has changed the scheduling parameters of an application-scheduled mutex	Pointer to the mutex

- Accept or reject a thread that has requested attachment to this scheduler
- Activate or suspend an application scheduled thread
- Accept or reject initialization of an application-scheduled mutex
- Grant the lock of an application-scheduled mutex

The list of actions will be prepared by the scheduler thread and reported to the system via a call to the `posix_appsched_execute_actions()` function. This function is the main operation in our interface. It allows the application scheduler to execute a list of scheduling actions and then it suspends waiting for the next scheduling event to be reported by the system. If desired, a timeout can be set as an additional return condition which will occur when there is no scheduling event available but the timeout expires. The system time measured immediately before the function returns can be requested if it is relevant for the algorithm.

The `posix_appsched_execute_actions()` function can also be programmed to return when a POSIX signal is generated for the thread. This possibility eases the use of POSIX timers, including CPU-time timers, as sources of scheduling events for our scheduler threads. The prototype of this function is:

```
int posix_appsched_execute_actions
  (const posix_appsched_actions_t *sched_actions,
   const sigset_t *set,
   const struct timespec *timeout,
   struct timespec *current_time,
   struct posix_appsched_event *event);
```

4.1.3. Scheduled Thread-Specific Data

When a scheduler is processing an event, the scheduling actions to execute will depend on the current scheduling status of its scheduled threads and particularly on the status of the thread which caused that event. It would be very interesting to have a mechanism for obtaining that information in a straightforward and efficient way. Because the POSIX thread identification type, `pthread_t`, is opaque, it is not possible to build a portable hashing function at the application level, and thus a list indexed by thread ids would be inefficient.

Consequently, our interface extends the POSIX "thread-specific data" functionality. Two new functions are defined (`pthread_getspecific_thread()` and `pthread_setspecific_thread()`) that permit setting and getting thread-specific data from a thread different from the owner. The scheduler thread can use those functions for attaching and retrieving the scheduling status of its scheduled threads.

4.2. Create scheduler and scheduled threads

In our interface a scheduler thread is created as such. A scheduled thread can be created attached to a particular thread. For these purposes our interface extends the thread creation attributes, the scheduling policies, and scheduling parameters defined in the POSIX standard.

A new policy "Application-defined Scheduling Policy" (`SCHED_APP`) is defined to distinguish between system-scheduled and application-scheduled threads. For a thread to be created with this policy it is necessary to define which thread is going to act as its application scheduler and, optionally, an application-defined scheduling parameters object to be interpreted by this scheduler thread. Both, the scheduler and the application-defined parameters are included as new members of the POSIX scheduling parameters structure (`sched_param`).

Whether a thread is an application scheduler or not is determined by the value of its `appscheduler` attribute. After its creation, an application scheduler thread can set some of its properties through different functions defined in the interface. The properties are the kind of timeout that is supported (relative or absolute), the clock used to determine when the timeout expires, and the event mask that allows scheduling events to be filtered out by the system, and thus are not reported to the application scheduler.

4.3. Explicit Scheduler Invocation

Explicit scheduler invocation from the scheduled thread could be necessary in some scheduling algoritms (for example as a mechanism to inform the scheduler a thread has finish its work for the current activation). For this purpose, our interface defines the `posix_appsched_invoke_scheduler()` function. Calling this function will cause a scheduling event of type `POSIX_APPSCHED_EXPLICIT_CALL` to be generated for the scheduler. Optionally, a message can be attached to the event.

4.4. Application-Scheduled Mutexes

As explained above, our interface allows creating mutexes whose synchronization protocol is defined by the application scheduler. These special mutexes are created like any other POSIX mutex but specifying the value `PTHREAD_APPSCHED_PROTOCOL` for they `protocol` attribute. For this kind of mutexes two new attributes have been defined: the `appscheduler` attribute and the `appschedparam` attribute. The `appscheduler` attribute identifies the scheduler thread the mutex is attached to. The optional `appschedparam` attribute can be used for passing application-defined mutex scheduling attributes to the scheduler.

As for the application-scheduled threads, it is also important for the scheduler to have a simple mechanism to attach and retrieve the scheduling specific data associated with an application-scheduled mutex. With this purpose our interface introduces a new functionality not defined in POSIX: the mutex-specific data, and two functions: `posix_appsched_mutex_setspecific()` and `posix_appsched_mutex_getspecific()`, to get and set the value currently bound to a mutex.

5. Example of an Application-Defined Policy: EDF with CBS

The following example shows the pseudocode of an application scheduler that implements the Earliest Deadline First (EDF) scheduling policy along with the Constant Bandwidth Server (CBS) [2] scheduling policy. The latter policy allows scheduling soft real-time tasks without jeopardizing the priority guarantee of hard real-time activities. The CBS assigns each soft task a maximum bandwidth and assures that it is not overcome even in the presence of overloads. In our example the asynchronous arrival of a new job for a CBS thread causes the generation of a signal (NEW_JOB_SIGNAL) to be caught by the scheduler thread.

In order to assure that a CBS thread does not overcome its assigned bandwidth its execution time must be limited. To achieve that, a POSIX CPU-time timer is associated with each CBS thread, and the signal generated by the timer expiration is caught by the scheduler thread to perform the appropriate scheduling actions.

The EDF/CBS scheduler has a list of threads that are registered for being scheduled under it, either as EDF or

CBS threads. The state of each EDF thread can be active or timed (when it has finished its current execution and is waiting for its next period), while the CBS threads can be active or idle (a CBS thread with not pending jobs).

The `schedule_next()` function invoked by the scheduler updates the list of registered threads based upon the current time. It switches into the active state those timed threads whose activation time has been reached, and then calculates the next thread to be executed and the earliest start time of the new set of timed threads.

The pseudocode of the scheduler is the following:

```
void *edf_scheduler (void *arg)
{
  ...;
  while (1) {
    schedule_next (&next_thread, &earliest_start,
                   &now);
    /* Thread activation and suspension actions*/
    if (next_thread != NULL)
      posix_appsched_actions_addactivate
         (&actions, next_thread);
    if (current_thread != NULL)
      posix_appsched_actions_addsuspend
         (&actions, current_thread);
    current_thread = next_thread;
    /* Execute scheduling actions */
    posix_appsched_execute_actions
       (&actions, &awaited_signal_set,
        &earliest_start, &now, &sched_event);
    /* Process scheduling events */
    switch (sched_event.event_code) {
    case POSIX_APPSCHED_NEW :
      Get thread EDF/CBS specific sched param;
      if (CBS thread) { Create CPU-time timer; }
      Add thread to list of scheduled threads;
      break;
    case POSIX_APPSCHED_TERMINATE :
      if (CBS thread) { Delete CPU-time timer; }
      Remove thread from scheduled threads list;
      break;
    case POSIX_APPSCHED_EXPLICIT_CALL :
      if (CBS thread) {
        thread.number_of_pending_jobs--;
        if (thread.number_of_pending_jobs == 0)
          thread.state=CBS_IDLE;
      } else { // is EDF
        thread.state=TIMED;
        Obtain next deadline & activation time;
      }  break;
    case POSIX_APPSCHED_SIGNAL :
      switch (received signal) {
      case END_OF_BUDGET_SIGNAL :
        thread.deadline +=  thread.period;
        thread.budget = thread.max_budget;
        Arm CPU-timer for 'thread.budget' secs;
        break;
      case NEW_JOB_SIGNAL :
        thread.number_of_pending_jobs++;
        if (thread.state==CBS_IDLE) {
          thread.state=ACTIVE;
          if (not enough time until deadline) {
            thread.deadline +=  thread.period;
            thread.budget = thread.max_budget;
            Arm CPU-timer for
                'thread.budget' secs;
          }
        }
      }
      break;
    case POSIX_APPSCHED_TIMEOUT :
      break; // threads will be rescheduled
    } // switch
  } // while (1)
}
```

The pseudocode of one of the application-scheduled threads is the following:

```
void * edf_or_cbs_thread (void * arg)
{
   while (1) {
      // do useful work
      ...
      // tell the scheduler that the
      // current job has finished
      pthread_appsched_invoke_scheduler ();
   }
}
```

The scheduler needs to know the period and the deadline of the EDF periodic threads and the maximum budget of the CBS threads. To store that information, the `edf_cbs_parameters_t` type is created:

```
typedef struct {
  int cbs_thread;
  struct timespec deadline, period, max_budget;
} edf_cbs_parameters_t;
```

The `posix_appsched_param` field of the `sched_param` structure assigned to an EDF or CBS thread will point to an object of this type. Using the POSIX function `pthread_getschedparam()` the scheduler will be able to get the scheduling parameters of its attached threads.

Finally, the pseudocode of a thread that creates the scheduler, a periodic EDF thread and a CBS thread is:

```
int main ()
{
  ...;

  // Set attr object for the scheduler thread
  Set policy to FIFO;
  Set the thread type to “application scheduler”;
  // Create scheduler thread
  pthread_create (attr object, edf_scheduler);

  // Set the attributes object for an EDF thread
  Set policy to application-defined;
  Set the thread type to “regular”;
  Set edf_scheduler as its application scheduler;
  Set period and deadline in the application
       defined scheduling parameters;
  // Create EDF thread
  pthread_create (attributes object, edf_thread);
```

```
  // Set the attributes object for a CBS thread
  Set policy to application-defined;
  Set the thread type to "regular";
  Set edf_scheduler as its application scheduler;
  Set period, deadline and max_budget in the
       application defined scheduling parameters;
  // Create CBS thread
  pthread_create (attributes object, cbs_thread);
}
```

6. Performance Metrics

Table 2 shows some performance metrics measured on a 1.1 GHz Pentium III, relative to a context switch between a CBS thread that consumes its execution time budget, and an EDF thread that becomes active. The first three entries in the table show the execution times of different parts of this context switch. The total is 4.1 μs, which represents an overhead of 0.82% for a 1KHz periodic thread, assuming two context switches per execution. Of course, this overhead would be lower for lower frequency threads. As a comparison, the last entry in the table shows the context switch time associated with a timer expiration, using the kernel-level fixed-priority scheduler. This time (1.1 μs) must be compared with the time needed to activate the scheduler (1.8 μs) and to activate a new thread (1.0 μs) to determine the overhead of the application-defined scheduling mechanism. We can see that this overhead is around 2.54 times the overhead of the internal kernel scheduler. In our opinion, this penalty is small compared to the benefits of being able to define application schedulers in a flexible and portable way.

Table 2. Performance on a 1.1 GHz Pentium III

Description	Time (μs)
Scheduler activation after CPU-timer expiration	1.8
Scheduling algorithm time	1.3
EDF thread activation	1.0
Total context switch time	4.1
Context switch after timer expiration	1.1

7. Conclusions and Further Work

We have defined a new API for application-defined scheduling. There are two versions of the API, one in C and the other one in Ada. Both are designed in the context of a POSIX operating system. The main design requirements have been to have a compatible behavior with other existing POSIX fixed priority scheduling policy, to be able to isolate the scheduler from the kernel and from other application schedulers, to be able to run both on single processor and multiprocessor systems, and to be able to describe application-defined synchronization protocols.

The proposed API has been implemented in MaRTE OS, which is a free software implementation of the POSIX minimal real-time operating system, intended for small embedded systems. It is written in Ada but provides both the POSIX Ada and the C interfaces. Using this implementation we have programmed and tested several scheduling policies, such as a priority-based round robin scheduler, EDF, and more complex dynamic scheduling policies. We have also tested some application-defined synchronization protocols, such as the full Priority Ceiling Protocol. Preliminary performance data show a good level of efficiency.

MaRTE OS, including the application-defined scheduling services defined in this paper can be found at:
`http://marte.unican.es`

References

[1] ISO/IEC 9945-1 (1996). *ISO/IEC Standard 9945-1:1996. Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) [C Language]*. Institute of Electrical and electronic Engineers.

[2] L. Abeni and G. Buttazzo. "Integrating Multimedia Applications in Hard Real-Time Systems". *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998

[3] M. Aldea Rivas and M. González Harbour. "POSIX-Compatible Application-Defined Scheduling in MaRTE OS". Proceedings of the Work in Progress Session, 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands, June 2001.

[4] M. Aldea and M. González. "MaRTE OS: An Ada Kernel for Real-Time Embedded Applications". Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe-2001, Leuven, Belgium, *Lecture Notes in Computer Science, LNCS 2043*, May, 2001.

[5] POSIX.1d (1999). *IEEE Std. 1003.d-1999. Information Technology -Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API) Amendment: Additional Realtime Extensions [C Language]*. The Institute of Electrical and Electronics Engineers.

[6] POSIX.13 (1998). *IEEE Std. 1003.13-1998. Information Technology -Standardized Application Environment Profile-POSIX Realtime Application Support (AEP).* The Institute of Electrical and Electronics Engineers.

[7] POSIX.5b (1996). *IEEE Std 1003.5b-1996, Information Technology—POSIX Ada Language Interfaces—Part 1: Binding for System Application Program Interface (API)—Amendment 1: Realtime Extensions*. The Institute of Electrical and Engineering Electronics.

[8] Y.C. Wang and K.J. Lin, "Implementing a general real-time scheduling framework in the red-linux real-time kernel". *Proceedings of IEEE Real-Time Systems Symposium*, Phoenix, December 1999.

[9] Bryan Ford and Sai Susarla, “CPU Inheritance Scheduling”. *Proceedings of OSD*I, October 1996.

[10]P. Gai, L. Abeni, M. Giorgi, G. Buttazzo, "A New Kernel Approach for Modular Real-Time Systems Development", *IEEE Proceedings of the 13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

[11]OMG. *Real-Time CORBA 2.0: Dynamic Scheduling*, Joint Final Submission. OMG Document orbos/2001-06-09, June 2001.

[12]Yodaiken V., “An RT-Linux Manifesto”. *Proceedings of the 5th Linux Expo*, Raleigh, North Carolina, USA, May 1999.

[13]George M. Candea and Michael B. Jones, “Vassal: Loadable Scheduler Support for Multi-Policy Scheduling”. *Proceedings of the Second USENIX Windows NT Symposium*, Seattle, Washington, August 1998.

[14]F. Mueller, V. Rustagi, and T.P. Baker. “MiThOS - A Real-Time Micro-Kernel Threads Operating System”. Proceedings of the IEEE Real-Time Systems Symposium, December 1995.