Application-Defined Scheduling (Draft 1.0, July 2002[1])

By: Mario Aldea Rivas aldeam@unican.es
Michael González Harbour mgh@unican.es

1. Introduction

The fixed priority scheduling policies defined in the current version of the Real-Time POSIX standard provide a nice combination of simplicity, predictability, and efficiency, that make them suitable for most real-time applications. However, 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 measures 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 {B14}. 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, an interface for application-defined schedulers is proposed, that could be used to implement a large variety of scheduling policies.

Introducing application-defined schedulers in POSIX has some challenges that do not appear in existing interfaces and implementations. One of the most difficult ones, is to keep the new schedulers compatible with the existing scheduling policies, while allowing implementations in which the application schedulers are not trusted and, consequently, are not permitted to execute inside the kernel address space. Other one is to use as much of the existing scheduling interface as possible, adding the fewest possible new interfaces.

A prototype of the proposed interface is currently implemented in the operating system MaRTE OS (Minimal Real-Time Operating System for Embedded Applications) {B4}, which is a real-time kernel that follows the Minimal Real-Time POSIX.13 subset {B6}, providing both the C and Ada language POSIX interfaces.

This document defines the proposed interface as a free standing extension of the POSIX standard and its real-time options. It also provides some examples of usage of the API, and performance estimations obtained from the prototype implementation.

2. Related Work and Motivation

The idea of application-defined scheduling has been used in many systems. A solution is proposed in RED-Linux {B8}, 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

1. Funded in part by the FIRST project (EU project IST2001-34140)

object. In addition, this solution does not address the implementation of protocols for shared resources that could avoid priority inversion or similar effects.

A different approach is followed in the CPU Inheritance Scheduling {B9}, 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 {B10}, RT-Linux {B12}, Vassal {B13}). 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 addition, it would be very difficult to standardize such an approach because the internal architecture of the kernel varies a lot from one implementation to another; it is a problem similar to what is found when trying to standardize device drivers.

In the approach proposed in this document 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, the application scheduling algorithm is executed by a user thread. Although less efficient, this approach implies two important advantages:

A. The system reliability can be improved by protecting the system from the actions of an erroneous application scheduler. For efficiency, the interface allows execution of the application-defined scheduler in an execution environment different than that of a 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, and therefore 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. The choice of kernel space or application space is left to the implementation.

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.

The interface has been designed 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 takes into account the implementation of application-defined synchronization protocols.

The dynamic scheduling mechanism proposed for Real-Time CORBA 2.0 {B11} 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 proposed here 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 proposal 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, the API is available both for applications developed in C or Ada.

3. Requirements

The following requirements are stated for the application-defined scheduling interface in POSIX:

- *Requirement:* The new scheduling policies shall have a behavior compatible with other existing scheduling policies in POSIX.

  Rationale: It is important to keep backwards compatibility in the standard.

- *Requirement:* It shall be possible to isolate critical parts of the application from failures in the application-defined schedulers.

  Rationale: The application-defined schedulers may not be as trusted as the predefined POSIX scheduling policies. It is important that there is a way to schedule critical tasks that do not depend on the behavior, good or bad, of the application-defined schedulers and associated threads.

- *Requirement:* Any thread shall be schedulable using an application-defined scheduler, contending with other threads of the same process. There is no such requirement for processes.

  Rationale: Interprocess application-defined schedulers seem extremely complex, and do not seem justified by application requirements. Usually processes are used to establish protection boundaries among different parts of an application, and in that context it seems that they should only be allowed to use trusted predefined scheduling policies.

- *Requirement:* When a thread requests attachment to an application scheduler it shall be possible to reject or accept such attachment

  Rationale: There is a family of scheduling algorithms that use the concept of an “acceptance test”, which is used to accept tasks if the system can guarantee that their timing requirements will be satisfied, or to reject them if this guarantee cannot be made.

- *Requirement:* It should be possible to define several application-defined schedulers that coexist in the same process.

  Rationale: Several scheduling algorithms may be required for different parts of an application, and this is easier to implement if several application-defined schedulers are possible.

- *Requirement:* The ability to nest application-defined schedulers so that one scheduler runs when another one schedules it, should be unspecified.

  Rationale: It seems that requiring multilevel application-defined schedulers may be more complex than it is worth. Therefore support for this feature is not required.

- *Requirement:* It should be possible to execute the application-defined scheduler in an execution environment different than that of regular application threads, for example inside the kernel. But the interface should also allow the implementation to execute the scheduler in the environment of the application threads.

Rationale: In some systems only trusted software is allowed inside the kernel, and thus the application-defined scheduler should run outside the kernel, even if this implies less efficiency. In other systems, for efficiency purposes, it should be possible for the implementation to execute the scheduler inside the kernel. This should be an implementation choice.

- *Requirement:* The application-defined scheduler should have the ability to determine and bound the execution time of the different threads scheduled under it run.

  Rationale: Some scheduling algorithms depend on the measurement of CPU time, for example to accommodate execution of certain activities during the available slack time.

- *Requirement:* An application-defined scheduler should have the ability to choose the clock used for its timed events.

  Rationale: Because different schedulers may have different requirements relative to the clock that they are using. For example, a scheduler may require a high resolution clock, a synchronized clock, or a monotonic clock.

- *Requirement:* There should be a mechanism to allow application schedulers to accept or reject a task that wishes to register for execution

  Rationale: There are some scheduling algorithms that use the concept of an admittance test: a task, when created is either admitted or rejected for execution depending on the current load of the system, and depending on whether or not the scheduler will be able to guarantee that all the timing requirements will be met.

- *Requirement:* If an application-scheduled thread needs to synchronize with other system-scheduled threads there needs to be a portable mechanism to bound priority inversion.

  Rationale: Unbounded priority inversion cause high priority threads to experience very long delays waiting for lower priority threads, usually as a result of synchronization. By letting the scheduler know whether a mutex is used or not, we make it possible to implement a protocol that bounds priority inversion.

- *Requirement:* The application-defined scheduler should have the ability to know when a thread running under it is using an application-scheduled mutex. In addition, it should know when such a thread inherits a priority (or uninherits it) because it is using regular mutexes.

  Rationale: Most priority inversion avoidance algorithms require knowledge of the lock and unlock operations on mutexes.

- *Requirement:* It should be possible for an application-scheduled thread to pass information to its scheduler.

  Rationale: Some scheduling algorithms may require application-specific information that has to be supplied by the scheduled threads at runtime, in addition to the normal scheduling parameters.

- *Requirement:* It should be possible to filter the specific scheduling events that the system notifies to the scheduler.

  Rationale: For efficiency purposes, if a scheduler is not going to use some of the scheduling events that can be sent by the system, it should filter them out, i.e., discard them.

- *Requirement:* It should be possible to attach application-specific data to a mutex.

  Rationale: Many scheduling algorithms require knowledge about mutex ownership to bound priority inversion. For this purpose, it might be necessary to attach application information to a mutex, in a similar way as it is done for threads with the thread-specific data interface. In this case, and for simplicity purposes, it does not seem necessary to use “keys” or different fields of data, because an application-scheduled mutex can only be used by one scheduler at a time.

- *Requirement:* It should be possible to achieve multiprocessor application scheduling. Efficient multiprocessor scheduling will require knowledge of the specific architecture, and in particular of the number or processors capable of executing application-scheduled threads simultaneously.

  Rationale: Addressing multiprocessors is a general requirement in POSIX. It is very difficult to build efficient scheduling algorithms for an unknown architecture. Architecture dependent information like the number of processors is required.

- *Requirement:* Each scheduler should be capable of activating multiple of its scheduled threads at the same time, and/or to block previously activated tasks.

  Rationale: There are architectures in which a single scheduler, perhaps executing in a special-purpose processor, schedules threads to be executed concurrently in multiple processes.

- *Requirement:* It should be possible to have a mechanism for exchanging information among scheduler threads, their scheduled threads, other scheduler threads, and/or regular threads.

  Rationale: For some applications, it is necessary that the schedulers be able to receive information to their scheduled threads, and also to send information back to them.

4. Model for Application-Defined Scheduling

In the proposed approach for application-defined scheduling, shown in Figure 1, 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 the application, as well as implementations for trusted application schedulers, which can run more efficiently inside the kernel. Because of this duality, the scheduler threads are modeled as if they run in a separate context, which is called 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. Application schedulers belonging to the same process can share data among them. This is useful for building a multithreaded application scheduler, for a multiprocessor platform.

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.

It is unspecified whether application scheduler threads can themselves be application scheduled. They can always be system scheduled.

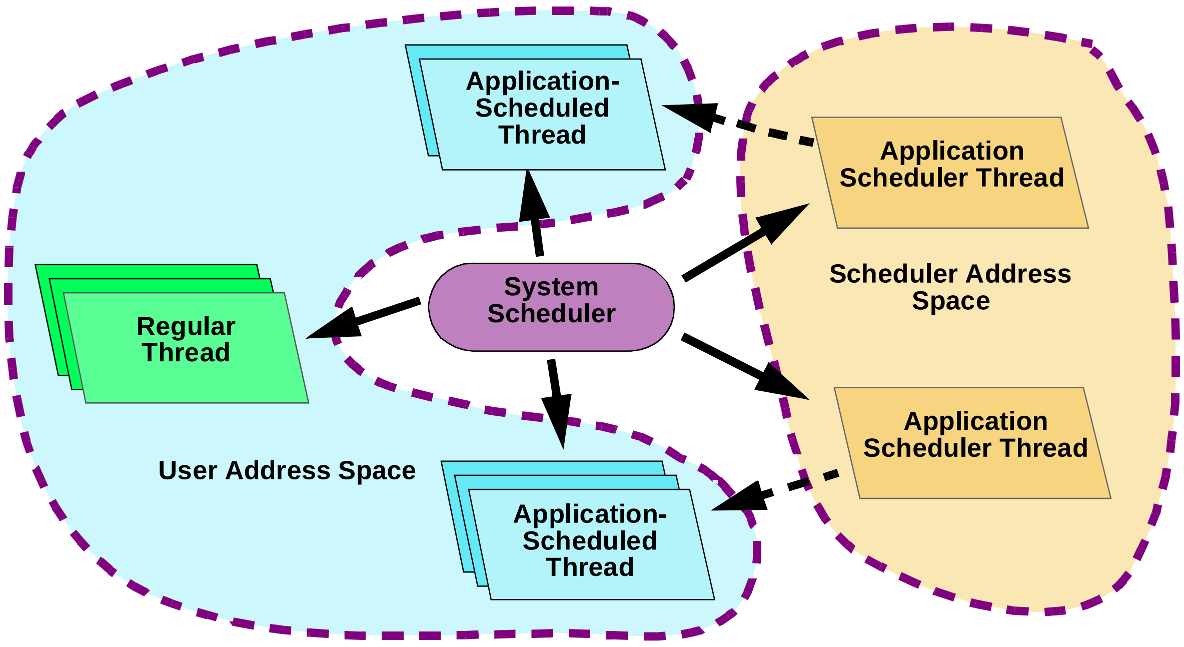
Application-Scheduled Thread
Application Scheduler Thread
Scheduler Address Space
Regular Thread
System Scheduler
Application Scheduler Thread
User Address Space
Application-Scheduled Thread

Figure 1. Model for application-defined scheduling

There are two ways in which a thread can be scheduled by an application scheduler:

- At thread creation. In this case, thread creation attributes specify the scheduler to be used, the regular and application scheduling parameters, and the scheduling policy (as application scheduled). The *pthread_create*() function creates the thread in a suspended state and waits until the scheduler either accept or rejects the new thread. If rejected, *pthread_create* fails and the thread is destroyed. To avoid this creation and destruction process, the parent thread could explicitly invoke the scheduler to reserve resources for the new thread, before creating it.
- By dynamically changing the scheduling policy to “application scheduled”. Before the change the thread must be under a policy that is not application-scheduled, and must set the scheduler thread and the application scheduling parameters attributes to the desired values. Then, the *pthread_setschedparam*() function is invoked to dynamically change the scheduling policy into application-scheduled. If the thread is rejected, the function fails.

Direct change of the application scheduler is not allowed because if the new scheduler rejects the thread after the old scheduler has detached it, the thread would be left in an uncertain state, with no scheduler. Therefore, to change the scheduler a thread would first have to switch to another policy, SCHED_FIFO for example, and from there request attachment to the new scheduler. If rejected, the thread would continue with the SCHED_FIFO policy in that case.

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 is shown 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

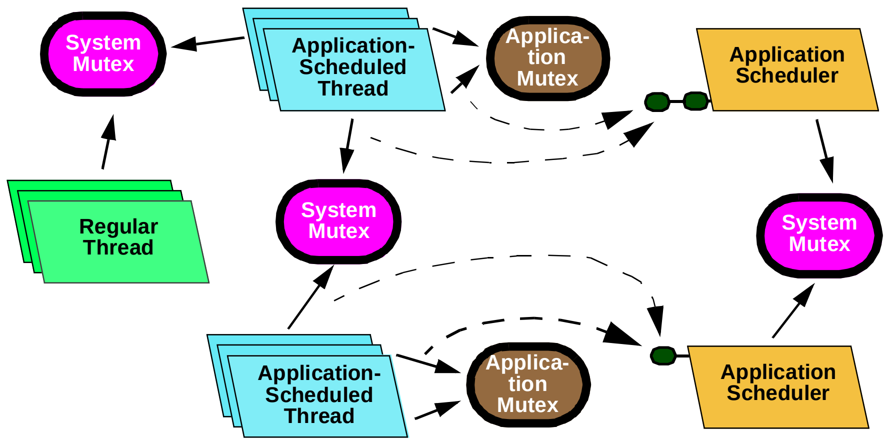

Figure 2. . Model for Application-Defined Synchronization

can be used to access resources shared between application schedulers, or between sets of application-scheduled threads attached to different schedulers.

- *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.

4.1 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 tasks, to prevent priority inversions from occurring. This means that high priority tasks that share mutexes with lower system priority application threads must take into account the scheduler overhead when accounting for their blocking times.

4.2 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.

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

- when a thread requests attachment to the scheduler
- when a thread terminates or requests de-attachment from the scheduler
- when a thread blocks (except at an application-scheduled mutex)
- when a thread is unblocked by the system and would become ready
- when a thread changes its scheduling parameters
- when a thread invokes the *pthread_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 threads to be activated. The scheduling events are stored in a FIFO queue until processed by the scheduler. For most scheduling events, unless the event is masked, after the event is generated the associated thread is suspended, to allow the scheduler to make a decision; for the thread to become active again it has to be explicitly activated by the scheduler, via the “execute actions” operation. For other events, the thread continues in the same state as before. The specific events that may be notified, and the state of the associated thread after the event are shown in Table 1.

Table 1. Application scheduling events and the state of the associated thread

Application Scheduling Events	State of associated thread after the event
POSIX_APPSCHED_NEW	Suspended
POSIX_APPSCHED_TERMINATE	No change

Table 1. Application scheduling events and the state of the associated thread (Continued)

POSIX_APPSCHED_READY	Suspended
POSIX_APPSCHED_BLOCK	No Change
POSIX_APPSCHED_YIELD	Suspended
POSIX_APPSCHED_SIGNAL	Not applicable (no associated thread)
POSIX_APPSCHED_CHANGE_SCHED_PARAM	No change
POSIX_APPSCHED_EXPLICIT_CALL	Suspended
POSIX_APPSCHED_EXPLICIT_CALL_WITH_DATA	Suspended
POSIX_APPSCHED_TIMEOUT	Not applicable (no associated thread)
POSIX_APPSCHED_PRIORITY_INHERIT	Suspended
POSIX_APPSCHED_PRIORITY_UNINHERIT	Suspended
POSIX_APPSCHED_INIT_MUTEX	Suspended
POSIX_APPSCHED_DESTROY_MUTEX	Suspended
POSIX_APPSCHED_LOCK_MUTEX	Suspended
POSIX_APPSCHED_TRY_LOCK_MUTEX	Suspended
POSIX_APPSCHED_UNLOCK_MUTEX	Suspended
POSIX_APPSCHED_BLOCK_AT_MUTEX	Suspended
POSIX_APPSCHED_CHANGE_MUTEX_SCHED_PARAM	Suspended

The description of the different events appears next:

- POSIX_APPSCHED_NEW. A thread has requested attachment to this scheduler; this can be a newly created thread (via *pthread_create*()), or an existing thread that was not running under this scheduler (via *pthread_setschedparam*()).
- POSIX_APPSCHED_TERMINATE. A thread attached to this scheduler has been terminated (via an explicit or implicit *pthread_exit*(), or by cancellation via *pthread_cancel*()), or it has changed its scheduling parameters and should no longer run under this scheduler (via *pthread_setschedparam*()). The thread is not suspended, because it is no longer going to run under the present scheduler.
- POSIX_APPSCHED_READY. A thread attached to this scheduler that was blocked has become unblocked by the system.
- POSIX_APPSCHED_BLOCK. A thread attached to this scheduler has blocked (except at an application-scheduled mutex). The thread is not suspended because it is already blocked by the system. Once it is unblocked, a POSIX_APPSCHED_READY event will be generated.
- POSIX_APPSCHED_YIELD. A thread attached to this scheduler has invoked the *sched_yield*() operation.
- POSIX_APPSCHED_SIGNAL. A blocked signal belonging to the set of signals for which the scheduler is waiting has been accepted by the scheduler thread.
- POSIX_APPSCHED_CHANGE_SCHED_PARAM. The scheduling parameters of a thread attached to this scheduler have been changed, but the thread continues to run under this scheduler. The change includes either the regular scheduling parameters (`schedpolicy` and `schedparam` attributes, via *pthread_setschedparam*()) or the application-defined scheduling parameters. (`appsched_param`, via *pthread_setappschedparam*()). Because this opera-

tion may be invoked asynchronously by some thread different than the one changing its parameters, there is no change to the activated/suspended state of the involved threads.
- POSIX_APPSCHED_EXPLICIT_CALL. A thread attached to this scheduler has explicitly invoked the scheduler via *posix_appsched_invoke_scheduler*().
- POSIX_APPSCHED_EXPLICIT_CALL_WITH_DATA. A thread attached to this scheduler has explicitly invoked the scheduler, with a message containing scheduling information, and possibly requesting a reply message, via *posix_appsched_invoke_withdata*().
- POSIX_APPSCHED_TIMEOUT. A timeout requested by the scheduler has expired.
- POSIX_APPSCHED_PRIORITY_INHERIT. A thread attached to this scheduler has inherited a new system priority due to the use of system mutexes.
- POSIX_APPSCHED_PRIORITY_UNINHERIT. A thread attached to this scheduler has finished the inheritance of a system priority that was inherited due to the use of system mutexes.
- POSIX_APPSCHED_INIT_MUTEX. A thread attached to this scheduler has requested initialization of an application-scheduled mutex.
- POSIX_APPSCHED_DESTROY_MUTEX. A thread attached to this scheduler has destroyed an application-scheduled mutex.
- POSIX_APPSCHED_LOCK_MUTEX. A thread attached to this scheduler has invoked the “lock” operation on an available application- scheduled mutex.
- POSIX_APPSCHED_TRYLOCK_MUTEX. A thread attached to this scheduler has invoked the “try lock” operation on an available application- scheduled mutex.
- POSIX_APPSCHED_UNLOCK_MUTEX. A thread attached to this scheduler has released the lock of an application-scheduled mutex.
- POSIX_APPSCHED_BLOCK_AT_MUTEX. A thread attached to this scheduler has blocked at an unavailable application-scheduled mutex.
- POSIX_APPSCHED_CHANGE_MUTEX_SCHED_PARAM. A thread attached to this scheduler and currently holding the lock on an application-scheduled mutex has changed the scheduling parameters of that 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.

4.3 Sharing Information Between the Schedulers and Their Scheduled Threads

There is an explicit “invoke scheduler” family of operations that can be used by an application-scheduled thread to directly invoke its scheduler, pass information to it, and obtain information back. Generally, the information to be shared by the scheduler and its associated threads is small, and therefore this mechanism does not introduce much overhead.

Because the scheduler may run in a context different than its scheduled threads, possibly with a different address space, there is no mechanism to directly share memory among them. The mechanism used in POSIX to share memory between processes that are placed in different address spaces, shared memory objects is not useful for this case because it is designed for different processes, and in this case the scheduler and its threads are in the same process.

5. Interface for Application-Defined Scheduling

All the interfaces defined in this standard are mandatory if the standard is supported. The implementation shall conform to at least one of the POSIX.13 realtime profiles.

5.1 Data Definitions

5.1.1 Errors

The following symbolic name representing an error number shall be defined in <`errno.h`>:

[EREJECT]	The thread requesting attachment to an application-defined scheduler has been rejected by that scheduler.
[EPOLICY]	The scheduling policy or the scheduler state attribute of the calling thread is not valid for this operation.
[EMASKED]	The operation cannot be executed because the associated scheduling event is currently masked by the application scheduler.

5.1.2 Minimum and Configurable Values

The constants specified in the Table 1-1 shall be defined in <`limits.h`> with the values shown. These are symbolic names for the most restrictive value for certain features in this Standard. A conforming implementation shall provide values at least this large. A portable application shall not require a larger value for correct operation:

Table 1-1: Minimum Values

Constant	Description	Value
_POSIX_APPSCHEDPARAM_MAX	The minimum size in bytes for the bound on the size of the application scheduling parameters (see 5.1.3)	32
_POSIX_APPSCHEDINFO_MAX	The minimum size in bytes for the bound on the size of the data exchanged between an application-scheduled thread and its scheduler (see 5.10)	32
_POSIX_APPMUTEXPARAM_MAX	The minimum size in bytes for the bound on the size of the mutex scheduling parameters (see 8.1.2)	16

The constants defined in Table 1-2 are related to the constants that appear in Table 1-1. If the actual value is determinate, it shall be represented by the constant, defined in <`limits.h`>. If the actual bound is indeterminate, it shall be provided by the *sysconf*() function using the corresponding name value, defined in <`unistd.h`> and specified in Table 1-3.

Table 1-2: Run-time Invariant Values (possibly Indeterminate)

Constant	Description	Minimum Value
POSIX_APPSCHEDPARAM_MAX	The bound in bytes for the size of the application scheduling parameters (see 5.1.3)	_POSIX_APPSCHEDPARAM_MAX
POSIX_APPSCHEDINFO_MAX	The bound in bytes for the size of the data exchanged between an application-scheduled thread and its scheduler (see 5.10)	_POSIX_APPSCHEDINFO_MAX
POSIX_APPMUTEXPARAM_MAX	The bound in bytes for the size of the mutex scheduling parameters (see 8.1.2)	_POSIX_APPMUTEXPARAM_MAX

Table 1-3: Configurable System Variables

Variable	Description	name Value
POSIX_APPSCHEDPARAM_MAX	The bound in bytes for the size of the application scheduling parameters (see 5.1.3)	_SC_POSIX_APPSCHEDPARAM_MAX
POSIX_APPSCHEDINFO_MAX	The bound in bytes for the size of the data exchanged between an application-scheduled thread and its scheduler (see 5.10)	_SC_POSIX_APPSCHEDINFO_MAX
POSIX_APPMUTEXPARAM_MAX	The bound in bytes for the size of the mutex scheduling parameters (see 8.1.2)	_SC_POSIX_APPMUTEXPARAM_MAX

5.1.3 Scheduling Policy and Attributes

The following scheduling policy shall be defined in `<sched.h>`. This policy shall not be used to schedule processes, i.e., it shall not be used in a call to *sched_setscheduler*():

Symbol	Description
SCHED_APP	Application-defined scheduling policy

The following new thread attributes are defined for specifying application scheduling parameters:

Attribute type	Attribute Name	Description
pthread_t	`appscheduler`	Scheduler thread to which the application-defined scheduler is attached
n/a	`appsched_param`	Application-defined scheduling parameters

For a thread to be created as a application-scheduled thread under a specified scheduler, the scheduler must have been created beforehand. Then, at the thread creation time, the thread creation policy is set to the value SCHED_APP, and the scheduling parameters are set with the appropriate values in the `appscheduler` and `appsched_param` attributes. These attributes can also be set or queried dynamically, after the thread has been created.

If the scheduling policy of the thread is not SCHED_APP, these attributes have no effect.

The default value for the `appscheduler` attribute is unspecified. If at the time of the thread creation the scheduling policy is SCHED_APP and the `appscheduler` attribute does not refer to a valid application scheduler thread, the corresponding *pthread_create*() operation shall fail with an error of [EINVAL].

The `appsched_param` attribute is a variable-size buffer containing application-defined scheduling parameters. The maximum size of this attribute shall be represented by the variable POSIX_APPSCHEDPARAM_MAX (see 5.1.2). The default value for the `appsched_param` attribute shall be a buffer of zero bytes.

5.1.4 Scheduler Thread State Attribute

A new attribute, `appscheduler_state`, is defined to represent whether a thread is an application-scheduler thread or a regular thread. This is a thread-creation attribute that can be queried dynamically, but cannot be modified dynamically.

A value of PTHREAD_APPSCHEDULER for the `appscheduler_state` attribute shall cause threads created with that attribute to be created as application scheduler threads. Application scheduler threads shall not share variables or memory with other regular application threads, and shall not synchronize with such threads using mutexes or condition variables; otherwise the results are undefined. Application scheduler threads may share variables and cooperate via mutexes and condition variables with other application scheduler threads in the same process.

A value of PTHREAD_REGULAR for the `appscheduler_state` attribute shall cause the thread to be created as a normal application thread as specified elsewhere in this standard.

These symbols shall be defined in `<pthread.h>`. The default value shall be PTHREAD_REGULAR.

5.1.5 Scheduling Events

The following datatypes shall be defined in `<sched.h>`:

```
union posix_appsched_eventinfo {
     int sched_priority;
     siginfo_t siginfo;
     pthread_mutex_t *mutex;
     void *info;
     int user_event_code;
};
struct posix_appsched_event {
     int event_code;
     pthread_t thread;
     union posix_appsched_eventinfo event_info;
     size_t info_size;
};
```

The structure *posix_appsched_event* represents a scheduling event that the system notifies to the application-defined scheduler. It contains the code of the specific event that has occurred (see Table 2) in the *event_code* member; the thread identifier of the thread that caused the event in the *thread* member; the information associated with that event in the *event_info* member; and the size of the information, if necessary, in the *info_size* member.

The union *posix_appsched_eventinfo* represents the different kinds of information that may be attached to a scheduling event. Table 2 shows the information associated with each scheduling event. All the symbols shall be defined in `<sched.h>`.

Table 2. Application scheduling events and their associated information

Application Scheduling Events (`event_code`)	**Additional Information (`event_info`)**
POSIX_APPSCHED_NEW	NULL pointer
POSIX_APPSCHED_TERMINATE	NULL pointer
POSIX_APPSCHED_READY	NULL pointer
POSIX_APPSCHED_BLOCK	NULL pointer
POSIX_APPSCHED_YIELD	NULL pointer
POSIX_APPSCHED_SIGNAL	*siginfo_t* value delivered with the signal
POSIX_APPSCHED_CHANGE_SCHED_PARAM	NULL pointer
POSIX_APPSCHED_EXPLICIT_CALL	User event code
POSIX_APPSCHED_EXPLICIT_CALL_WITH_DATA	Pointer to the message set by the thread[a]
POSIX_APPSCHED_TIMEOUT	NULL pointer

Table 2. Application scheduling events and their associated information (Continued)

POSIX_APPSCHED_PRIORITY_INHERIT	Inherited system priority
POSIX_APPSCHED_PRIORITY_UNINHERIT	Uninherited system priority
POSIX_APPSCHED_INIT_MUTEX	Pointer to the app. scheduled mutex
POSIX_APPSCHED_DESTROY_MUTEX	Pointer to the app. scheduled mutex
POSIX_APPSCHED_LOCK_MUTEX	Pointer to the app. scheduled mutex
POSIX_APPSCHED_TRY_LOCK_MUTEX	Pointer to the app. scheduled mutex
POSIX_APPSCHED_UNLOCK_MUTEX	Pointer to the app. scheduled mutex
POSIX_APPSCHED_BLOCK_AT_MUTEX	Pointer to the app. scheduled mutex
POSIX_APPSCHED_CHANGE_MUTEX_SCHED_PARAM	Pointer to the app. scheduled mutex

[a] In this case, *info_size* shall be the size of the information; in other cases, *info_size* shall be zero

5.1.6 Scheduling Events Sets

The *posix_appsched_eventset_t* type shall be defined in `<sched.h>`. Values of this type represent sets of scheduling event codes. Several functions used to manipulate objects of this type are defined in 5.8. No comparison or assignment operators are defined for the type *posix_appsched_eventset_t*.

5.1.7 Scheduling Actions

The *posix_appsched_actions_t* datatype shall be defined in `<sched.h>` to represent a list of scheduling actions that the scheduler will later request to be executed by the system. The possible actions are of the following kinds:

- 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

No comparison or assignment operators are defined for the type *posix_appsched_actions_t*.

5.1.8 Rationale for Data Definitions

The new thread attributes for application scheduling, `appscheduler` and `appsched_param`, were initially proposed as additional fields of the *sched_param* structure that is commonly used to store the scheduling parameters of a thread. However, this solution has the disadvantage that code requiring use of that structure needs recompilation. In particular, because the widely used C library (`libc`) uses that structure it was considered that, to ease backwards compatibility, new thread attributes should be used instead.

The minimum size for the scheduling parameters attribute appsched_param is set to 32 because there are many scheduling algorithms that require specifying up to four time parameters (for example, period, execution time, soft deadline, and hard deadline). Each of these parameters requires eight bytes if stored in a *struct timespec* datatype.

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 would be 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.

5.2 Interface for the Creation of the Scheduler

5.2.1 Synopsis

```
#include <pthread.h>

int pthread_attr_setappschedulerstate
        (pthread_attr_t *attr,
         int appschedstate);

int pthread_attr_getappschedulerstate
        (const pthread_attr_t *attr,
         int *appschedstate);
```

5.2.2 Description

The *pthread_attr_setschedulerstate*() and *pthread_attr_getschedulerstate*() functions are used to set and get the `appscheduler_state` attribute in the object pointed to by *attr*. The *pthread_attr_setschedulerstate*() function shall set this attribute to the value specified by *appschedstate*, which shall be either PTHREAD_REGULAR or PTHREAD_APPSCHEDULER. These symbols are described in 5.1.4.

An application-defined scheduler itself shall be scheduled by the system as a thread under the SCHED_FIFO policy, with the system priority determined by the *sched_priority* member of its `schedparam` attribute.

5.2.3 Returns

Upon successful completion, *pthread_attr_setschedulerstate*() and *pthread_attr_getschedulerstate*() shall return a value of 0. Otherwise an error number shall be returned to indicate the error.

The *pthread_attr_getschedulerstate*() function stores the `appscheduler_state` attribute in the variable pointed to by *appschedstate* if successful.

5.2.4 Errors

If any of the following conditions occur, the *pthread_attr_setschedulerstate*() function shall return the corresponding error number:

[EINVAL]	The value of *appschedstate* was not valid.

5.2.5 Cross-References

pthread_create(), *pthread_getappscheduler*().

5.3 Dynamically Getting the `Appscheduler` Attribute

5.3.1 Synopsis

```
#include <pthread.h>
int pthread_getappschedulerstate (pthread_t thread, int *appschedstate);
```

5.3.2 Description

This function allows the `appscheduler_state` attribute to be retrieved dynamically, after the thread has been created. The value of the attribute shall be returned in the variable pointed to by *appschedstate*.

5.3.3 Returns

The *pthread_getappschedulerstate*() function stores the `appscheduler_state` attribute of the thread specified by *thread* in the variable pointed to by *appschedstate* and returns zero, if successful.

5.3.4 Errors

If any of the following conditions are detected, the *pthread_getappschedulerstate*() function shall return the corresponding error number:

[ESRCH]	The value specified by *thread* does not refer to an existing thread.

5.3.5 Cross-References

pthread_attr_setschedulerstate().

5.4 Interfaces for Creating Application-Scheduled Threads

5.4.1 Synopsis

```
#include <pthread.h>
int pthread_attr_setappscheduler
        (pthread_attr_t *attr,
         pthread_t scheduler);
int pthread_attr_getappscheduler
        (const pthread_attr_t *attr,
         pthread_t *scheduler);
```

```
int pthread_attr_setappschedparam
        (pthread_attr_t *attr,
         const void *param,
         size_t paramsize);

int pthread_attr_getappschedparam
        (const pthread_attr_t *attr,
         void *param,
         size_t *paramsize);
```

5.4.2 Description

The *pthread_attr_setappscheduler*() and *pthread_attr_getappscheduler*() functions are used to set and get the `appscheduler` attribute in the object pointed to by *attr*. For those threads with the scheduling policy SCHED_APP, this attribute represents the identifier of its scheduler thread. The attribute is described in 5.1.3. If successful, the *pthread_attr_setappscheduler*() function shall set this attribute to the value specified by *scheduler*, which shall be a valid application scheduler thread.

The *pthread_attr_setappschedparam*() and *pthread_attr_getappschedparam*() functions are used to set and get the `appsched_param` attribute in the object pointed to by *attr*. For those threads with the scheduling policy SCHED_APP, this attribute represents the application-specific scheduling parameters. The attribute is described in 5.1.3. If successful, the *pthread_attr_setappschedparam*() function shall set the size of the `appsched_param` attribute to the value specified by *paramsize*, and shall copy the scheduling parameters occupying *paramsize* bytes and pointed to by *param* into that attribute.

The *pthread_attr_getappschedparam*() function shall copy the contents of the `appsched_param` attribute into the memory area pointed to by *param*. This memory area shall be capable of storing at least a number of bytes equal to the size of the `appsched_param` attribute; otherwise, the results are undefined.

5.4.3 Returns

Upon successful completion, *pthread_attr_setappscheduler*(), *pthread_attr_setappschedparam*(), *pthread_attr_getappscheduler*(), and *pthread_attr_getappschedparam*() shall return a value of 0. Otherwise an error number shall be returned to indicate the error.

The *pthread_attr_getappscheduler*() function stores the `appscheduler` attribute in the variable pointed to by *scheduler* if successful.

The *pthread_attr_getappschedparam*() function stores the size of the `appsched_param` attribute in the variable pointed to by *paramsize*, and copies the contents of the `appsched_param` attribute into the memory area pointed to by *param*.

5.4.4 Errors

If any of the following conditions occur, the *pthread_attr_setappscheduler*() function shall return the corresponding error number:

[EINVAL]	The value of *scheduler* was not valid.

If any of the following conditions occur, the *pthread_attr_setappschedparam*() function shall return the corresponding error number:

[EINVAL]	The value of *paramsize* was smaller than zero, or was larger than POSIX_APPSCHEDPARAM_MAX.

If any of the following conditions are detected, the *pthread_attr_setappschedparam*() function shall return the corresponding error number:

[EINVAL]	The value of *param* was invalid.

5.4.5 Cross-References

pthread_create().

5.5 Interfaces for Dynamic Access to Application Scheduling Parameters

5.5.1 Synopsis

```
#include <pthread.h>

int pthread_setappscheduler
        (pthread_t thread,
         pthread_t scheduler);

int pthread_setappschedparam
        (pthread_t thread,
         const void *param,
         size_t paramsize);

int pthread_getappscheduler
        (pthread_t thread,
         pthread_t *scheduler);

int pthread_getappschedparam
        (pthread_t thread,
         void *param,
         size_t *paramsize);
```

5.5.2 Description

The *pthread_setappscheduler*() and *pthread_getappscheduler*() functions are used to dynamically set and get the `appscheduler` attribute of the thread identified by *thread*. For a thread with the scheduling policy SCHED_APP, this attribute represents the identifier of its scheduler thread. This attribute is described in 5.1.3.

If successful, the *pthread_setappscheduler*() function shall set the `appscheduler` attribute to the value specified by *scheduler*. For this function to succeed, the scheduling policy of the calling thread shall not be SCDHED_APP, because directly changing the scheduler is not permitted.

The *pthread_setappschedparam*() and *pthread_getappschedparam*() functions are used to dynamically set and get the `appsched_param` attribute of the thread identified by *thread*. For a thread with the scheduling policy SCHED_APP, this attribute represents its application-defined scheduling parameters. This attribute is described in 5.1.3.

If successful, the *pthread_setappschedparam*() function shall set the size of the `appsched_param` attribute to the value specified by *paramsize*, and it shall copy the scheduling parameters occupying *paramsize* bytes and pointed to by *param* into that attribute. In addition, if the scheduling policy of *thread* is SCHED_APP, it shall generate a POSIX_APPSCHED_CHANGE_SCHED_PARAM event for the application scheduler of that thread, unless that event is masked in that application scheduler.

The memory area represented by *param* in the call to *pthread_getappschedparam*() shall be capable of storing at least a number of bytes equal to the size of the `appsched_param` attribute; otherwise, the results are undefined.

5.5.3 Returns

Upon successful completion, *pthread_setappscheduler*(), *pthread_getappscheduler*(), *pthread_setappschedparam*(), and *pthread_getappschedparam*() shall return a value of 0. Otherwise an error number shall be returned to indicate the error.

The *pthread_getappscheduler*() function stores the `appscheduler` attribute in the variable pointed to by *appsched*, if successful.

The *pthread_getappschedparam*() function stores the size of the `appsched_param` attribute in the variable pointed to by `paramsize`, and copies the contents of the `appsched_param` attribute into the memory area pointed to by *param*, if successful.

5.5.4 Errors

If any of the following conditions occur, the *pthread_setappscheduler*() function shall return the corresponding error number:

[EINVAL]	The value of *scheduler* was not valid.
[EPOLICY]	The scheduling policy of the calling thread is SCHED_APP

If any of the following conditions occur, the *pthread_setappschedparam*() function shall return the corresponding error number:

[EINVAL]	The value of *paramsize* was less than zero, or was larger than POSIX_APPSCHEDPARAM_MAX.

If any of the following conditions are detected, the *pthread_setappschedparam*() function shall return the corresponding error number:

[EINVAL]	The value of *param* was invalid.

If any of the following conditions are detected, the *pthread_setappscheduler*(), *pthread_getappscheduler*(), *pthread_setappschedparam*(), and *pthread_getappschedparam*() functions shall return the corresponding error number:

[ESRCH]	The value specified by *thread* does not refer to an existing thread.

5.5.5 Cross-References

pthread_create(), *pthread_attr_setappscheduler*, *pthread_attr_setappschedparam*().

5.6 Interfaces for the Scheduler Thread: Scheduling Actions

5.6.1 Synopsis

```
#include <sched.h>

int posix_appsched_actions_init(
        posix_appsched_actions_t *sched_actions);

int posix_appsched_actions_destroy(
        posix_appsched_actions_t *sched_actions);

int posix_appsched_actions_addaccept(
        posix_appsched_actions_t *sched_actions,
        pthread_t thread);

int posix_appsched_actions_addreject(
        posix_appsched_actions_t *sched_actions,
        pthread_t thread);

int posix_appsched_actions_addactivate(
        posix_appsched_actions_t *sched_actions,
        pthread_t thread);

int posix_appsched_actions_addsuspend(
        posix_appsched_actions_t *sched_actions,
        pthread_t thread);

int posix_appsched_actions_addacceptmutex(
        posix_appsched_actions_t *sched_actions,
        const pthread_mutex_t *mutex);

int posix_appsched_actions_addrejectmutex(
        posix_appsched_actions_t *sched_actions,
        const pthread_mutex_t *mutex);

int posix_appsched_actions_addlockmutex(
        posix_appsched_actions_t *sched_actions,
        pthread_t thread,
        const pthread_mutex_t *mutex);
```

5.6.2 Description

A scheduling actions object is of type *posix_appsched_actions_t* (defined in `<sched.h>`) and is used to specify a series of actions to be performed by the *posix_appsched_execute_actions*() function. The order of the actions added to the object shall be preserved.

The *posix_appsched_actions_init*() function initializes the object referenced by *sched_actions* to contain no scheduling actions to perform. After successful initialization, the number of actions that may be successfully added to the actions object shall be at least equal to _POSIX_THREAD_THREADS_MAX.

The effect of initializing an already initialized actions object is undefined.

The *posix_appsched_actions_destroy*() function destroys the object referenced by *sched_actions*; the object becomes, in effect, uninitialized. An implementation may cause *posix_appsched_actions_destroy*() to set the object referenced by *sched_actions* to an invalid value. A destroyed scheduling actions object can be reinitialized using *posix_appsched_actions_init*(); the results of otherwise referencing the object after it has been destroyed are undefined.

The *posix_appsched_actions_addaccept*() function adds a thread-accept action to the object referenced by *sched_actions*, that will serve to notify that the thread identified by *thread* has been accepted by the scheduler thread to be scheduled by it. When the *posix_appsched_execute_actions*() function is invoked with this scheduling actions object, any thread waiting for such notification either on a *pthread_create*() or a *pthread_setschedparam*() function shall successfully complete the function. If no thread was waiting for such notification, the action has no effects.

The *posix_appsched_actions_addreject*() function adds a thread-reject action to the object referenced by *sched_actions*, that will serve to notify that the thread identified by *thread* has not been accepted by the scheduler thread to be scheduled by it, possibly because the thread contained invalid application scheduling attributes, or because there are not enough resources for the new thread. When the *posix_appsched_execute_actions*() function is invoked with this scheduling actions object, any thread waiting for such notification either on a *pthread_create*() or a *pthread_setschedparam*() function shall complete the function with an error code of [EREJECT]. If no thread was waiting for such notification, the action has no effects.

The *posix_appsched_actions_addactivate*() function adds a thread-activate action to the object referenced by *sched_actions*, that will cause the thread identified by *thread* to be activated when the *posix_appsched_execute_actions*() function is invoked with this scheduling actions object. If the thread was already active at the time the thread-activate action is executed, then the thread shall continue to be active. If the thread was suspended at a *posix_mutex_trylock*() operation then this action shall be an indication that the corresponding mutex is not available.

The *posix_appsched_actions_addsuspend*() function adds a thread-suspend action to the object referenced by *sched_actions*, that will cause the thread identified by *thread* to be suspended when the *posix_appsched_execute_actions*() function is invoked with this scheduling actions object. If the thread was already suspended at the time the thread-suspend action is executed, then the thread shall continue to be suspended.

The *posix_appsched_actions_addacceptmutex*() function adds a mutex-accept action to the object referenced by *sched_actions*, that will serve to notify that the mutex identified by *mutex* has been accepted by the scheduler thread to be scheduled by it. When the *posix_appsched_execute_actions*() function is invoked with this scheduling actions object, any thread waiting for such notification on a *pthread_mutex_init*() function shall successfully complete the function. If no thread was waiting for such notification, the action has no effects.

The *posix_appsched_actions_addrejectmutex*() function adds a mutex-reject action to the object referenced by *sched_actions*, that will serve to notify that the mutex identified by *mutex* has not been accepted by the scheduler thread to be scheduled by it, possibly because the mutex contained invalid scheduling attributes, or because there are not enough resources for the new mutex. When the *posix_appsched_execute_actions*() function is invoked with this scheduling actions object, any thread waiting for such notification on a *pthread_mutex_init*() function shall complete the call with an error code of [EREJECT]. If no thread was waiting for such notification, the action has no effects.

The *posix_appsched_actions_addlockmutex*() function adds a mutex-lock action to the object referenced by *sched_actions*, that will cause the lock on the mutex identified by *mutex* to be granted to the thread indicated by *thread* when the *posix_appsched_execute_actions*() function is invoked with this scheduling actions object.

5.6.3 Returns

Upon successful completion, *pthread_appsched_actions_init*(), *pthread_appsched_actions_destroy*(), *pthread_appsched_actions_addaccept*(), *pthread_appsched_actions_addreject*(), *pthread_appsched_actions_addactivate*(), *pthread_appsched_actions_addsuspend*(), *posix_appsched_actions_addacceptmutex*(), *posix_appsched_actions_addrejectmutex*(), and *posix_appsched_actions_addlockmutex*() shall return a value of 0. Otherwise an error number shall be returned to indicate the error.

5.6.4 Errors

If any of the following conditions occur, the *pthread_appsched_actions_init*() function shall return the corresponding error number:

[ENOMEM]	There is insufficient memory to initialize the actions object.

If any of the following conditions occur, the *pthread_appsched_actions_addaccept*(), *pthread_appsched_actions_addreject*(), *pthread_appsched_actions_addactivate*(), *pthread_appsched_actions_addsuspend*(), *posix_appsched_actions_addacceptmutex*(), *posix_appsched_actions_addrejectmutex*(), and *posix_appsched_actions_addlockmutex*() functions shall return the corresponding error number:

[ENOMEM]	There is insufficient memory to add a new action to the actions object.

If any of the following conditions is detected, the *pthread_appsched_actions_destroy*(), *pthread_appsched_actions_addaccept*(), *pthread_appsched_actions_addreject*(), *pthread_appsched_actions_addactivate*(), *pthread_appsched_actions_addsuspend*(), *posix_appsched_actions_addacceptmutex*(), *posix_appsched_actions_addrejectmutex*(), and *posix_appsched_actions_addlockmutex*() functions shall return the corresponding error number:

[EINVAL]	The value specified by *sched_actions* is invalid.

5.6.5 Cross-References

pthread_create(), *pthread_setschedparam*(), *posix_appsched_execute_actions*(), *pthread_mutex_init*(), *pthread_mutex_lock*(), *pthread_mutex_timedlock*(), *pthread_mutex_trylock*().

5.7 Interfaces for the Scheduler Thread: Execute Scheduling Actions

5.7.1 Synopsis

```
#include <signal.h>
#include <sched.h>

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);
```

5.7.2 Description

The *posix_appsched_execute_actions*() shall execute the scheduling actions pointed to by *sched_actions* (as described in 5.1.7 and 5.6) in the same order as they were added to the actions object, and then it shall cause the application scheduler thread calling the function to wait for the next scheduling event notified by the system. If no event is available in the scheduling event queue of the scheduler thread, then the scheduler thread shall block. If *sched_actions* is **NULL**, then no scheduling actions shall be executed, but the function shall wait for the next scheduling event notified by the system, as in the case in which scheduling actions were executed.

The threads referenced in the actions object pointed to by *sched_actions* shall refer to threads scheduled by the calling application scheduler for the function to succeed. If mutex-lock action is specified in *sched_actions* and the associated thread is not waiting in a *pthread_mutex_lock*() or *pthread_mutex_timedlock*() for the specified mutex at the time of the call, the function shall fail. Detection of one of these error conditions shall be made as the action is performed; thus, if an error is detected, previous actions will have been executed, and none of the further actions shall be executed.

If *set* is not **NULL**, the *posix_appsched_execute_actions*() function shall enable an additional scheduling event which will occur when there is no scheduling event available in its queue and one of the blocked signals belonging to *set* is accepted by the calling thread; the event generated shall have the code POSIX_APPSCHED_SIGNAL, its *thread* member shall be unspecified, and the information associated with the signal shall be returned in the *siginfo* member of the *event_info* member of *event*. If *set* is **NULL**, there shall be no acceptance of signals during the call.

If *timeout* is not **NULL**, the *posix_appsched_execute_actions*() function shall enable an additional scheduling event which will occur when there is no scheduling event available in its queue but the timeout specified by *timeout* expires; the event generated shall have the code POSIX_APPSCHED_TIMEOUT, and its *thread* member shall be unspecified. The value of *timeout* shall be interpreted according to the way in which the timeouts have been configured, as described for the scheduler attributes (see 5.9). If *timeout* is **NULL**, there shall be no pending timeout for the call.

If *current_time* is not **NULL**, the *posix_appsched_execute_actions*() function shall return in the variable pointed to by that argument the value of the clock specified by the `clockid` attribute of the application scheduler (see 5.9), as measured immediately before the function returns.[1]

If the thread calling *posix_appsched_execute_actions*() is not an application scheduler thread, then the function shall fail.

5.7.3 Returns

If successful, *posix_appsched_execute_actions*() shall return zero. Otherwise an error number shall be returned to indicate the error.

5.7.4 Errors

If any of the following conditions occur, the corresponding error number shall be returned by, *posix_appsched_execute_actions*():

1. Notice that the value returned may not represent the time at which it is used by the scheduler, since preemptions may occur in between.

[EINVAL]	One or more of the threads specified by the *sched_actions* scheduling actions object does not refer to a thread associated with this scheduler at the time of the *posix_appsched_execute_actions*() call.
	A mutex-lock action was specified in *sched_actions* and the associated thread was not waiting for the specified mutex at the time of the call.
[ESRCH]	One or more of the threads identifiers specified by the *sched_actions* scheduling actions object did not correspond to a thread that existed at the time of the *posix_appsched_execute_actions*() call.
[EPOLICY]	The calling thread is not an application scheduler.

If any of the following conditions is detected, the corresponding error number shall be returned by, *posix_appsched_execute_actions*():

[EINVAL]	The value of one or more of the arguments is invalid.

5.7.5 Cross-References

posix_appsched_actions_init(), *pthread_create*(), *pthread_setschedparam*(), *pthread_mutex_lock*(), *pthread_mutex_timedlock*(), *pthread_mutex_trylock*().

5.8 Interfaces for the Scheduler Thread: Scheduling Events Set Manipulation

5.8.1 Synopsis

```
#include <sched.h>

int posix_appsched_emptyset
   (posix_appsched_eventset_t *set);

int posix_appsched_fillset
   (posix_appsched_eventset_t *set);

int posix_appsched_addset
   (posix_appsched_eventset_t *set, int appsched_event);

int posix_appsched_delset
   (posix_appsched_eventset_t *set, int appsched_event);

int posix_appsched_ismember
   (const posix_appsched_eventset_t *set, int appsched_event);
```

5.8.2 Description

The *posix_appsched_emptyset*() function shall eliminate all the scheduling event codes from the set pointed to by *set*.

The *posix_appsched_fillset*() function shall add all the scheduling event codes to the set pointed to by *set*.

The *posix_appsched_addset*() function shall add the scheduling event code represented by *appsched_event* to the set pointed to by *set*.

The *posix_appsched_delset*() function shall eliminate the scheduling event code represented by *appsched_event* from the set pointed to by *set*.

The *posix_appsched_ismember*() function shall return one if the scheduling event code represented by *appsched_event* is a member of the set pointed to by *set*. Otherwise, it shall return zero.

5.8.3 Returns

Upon successful completion, the *posix_appsched_ismember*() function shall return a value of one if the specified event code is a member of the specified set, or a value of zero if it is not. Upon successful completion, the other functions shall return a value of zero. For all of the above functions, if an error is detected, an error number is returned.

5.8.4 Errors

If any of the following conditions is detected, the *posix_appsched_addset*(), *posix_appsched_addset*(), and *posix_appsched_ismember*() functions shall return the following error number:

[EINVAL]	The value of *appsched_event* is invalid.

5.8.5 Cross-References

posix_appschedattr_seteventmask(), *posix_appschedattr_geteventmask*().

5.9 Interfaces for the Scheduler Thread: Scheduler attributes

5.9.1 Synopsis

```
#include <sched.h>

int posix_appschedattr_setclock (clockid_t clockid);

int posix_appschedattr_getclock (clockid_t *clockid);

int posix_appschedattr_setflags (int flags);

int posix_appschedattr_getflags (int *flags);

int posix_appschedattr_seteventmask
   (const posix_appsched_eventset_t *set);

int posix_appschedattr_geteventmask
   (posix_appsched_eventset_t *set);

int posix_appschedattr_setreplyinfo (const void *reply, int reply_size);

int posix_appschedattr_getreplyinfo (void *reply, int *reply_size);
```

5.9.2 Description

The *posix_appschedattr_setclock*() and *posix_appschedattr_getclock*() functions respectively set and get the `clockid` attribute of the scheduler thread that calls the function. This is the clock that shall be used to report the current time in the *posix_appsched_execute_actions*() function, if requested, and for the timeout of the same function. The default value of the `clockid` attribute shall be CLOCK_REALTIME. If successful, the *posix_appschedattr_setclock*() function shall set the value of the attribute to *clockid*.

The *posix_appschedattr_setflags*() and *posix_appschedattr_getflags*() functions respectively set and get the `flags` attribute of the scheduler thread that calls the function. The default value of the `flags` attribute is with no flags set. If successful, the *posix_appschedattr_setflags*() function shall set the value of the attribute to *flags*. The following flags shall be defined in `<sched.h>`:

POSIX_APPSCHED_ABSTIMEOUT:
: If this flag is set in `flags`, the value of the *timeout* parameter in a call to *posix_appsched_execute_actions*() shall represent an absolute time at which the timeout expires, according to the clock specified by the `clockid` attribute. If the time specified by *timeout* has already passed at the time of the call and there are no scheduling events to be reported, the timeout expires immediately and thus the function does not block. If this flag is not set in `flags`, the *timeout* parameter shall represent a relative time interval, after which the timeout expires. This interval shall be measured using the clock specified by the `clockid` attribute.

The *posix_appschedattr_seteventmask*() and *posix_appschedattr_geteventmask*() functions respectively set and get the `eventmask` attribute of the scheduler thread that calls the function. This is the set of scheduling events that shall be masked, i.e., not reported to this scheduler thread. The default value of the `eventmask` attribute is with an empty set, i.e., all events are reported to the scheduler. If successful, the *posix_appschedattr_seteventmask*() function shall set the value of the attribute to *set*.

The *posix_appschedattr_setreplyinfo*() and *posix_appschedattr_getreplyinfo*() functions respectively set and get the `replyinfo` attribute of the scheduler thread that calls the function. This is a variable-size memory area containing information that shall be used to return information to an application scheduled thread that has requested it via a *posix_appsched_invoke_withdata*() function. The default value of the `replyinfo` attribute shall be a memory area of zero bytes. The maximum size of the `replyinfo` attribute is specified by the POSIX_APPSCHEDINFO_MAX variable (see 5.1.2). If successful, the *posix_appschedattr_setreplyinfo*() function shall set the value of the attribute equal to the memory area starting at *reply* and of size *reply_size*.

The *posix_appschedattr_getreplyinfo*() functions has undefined results if the memory area pointed to by *reply* is smaller than the size of the `replyinfo` attribute.

If the thread calling any of these functions is not a scheduler thread, then the function shall fail.

5.9.3 Returns

Upon successful completion, *posix_appschedattr_setclock*(), *posix_appschedattr_setflags*(), and *posix_appschedattr_seteventmask*() shall return zero. Otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_appschedattr_getclock*(), shall set the value of the `clockid` attribute in the variable pointed to by *clockid* and shall return zero. Otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_appschedattr_getflags*(), shall set the value of the `flags` attribute in the variable pointed to by *flags* and shall return zero. Otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_appschedattr_geteventmask*(), shall set the value of the `eventmask` attribute in the variable pointed to by *set* and shall return zero. Otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_appschedattr_getreplyinfo*(), shall set the value of the variable pointed to by *reply_size* to the size of the `replyinfo` attribute, shall copy the contents of the attribute into the memory area pointed to by *reply*, and shall return zero. Otherwise, an error number shall be returned to indicate the error.

5.9.4 Errors

If any of the following conditions is detected, the *posix_appschedattr_setclock*(), *posix_appschedattr_setflags*(), *posix_appschedattr_seteventmask*(), and *posix_appschedattr_setreplyinfo*() functions shall return the following error number:

[EINVAL]	The value of one of the arguments is invalid.

If any of the following conditions occurs, the *posix_appschedattr_setclock*(), *posix_appschedattr_setflags*(), *posix_appschedattr_seteventmask*(), *posix_appschedattr_setreplyinfo*(), *posix_appschedattr_getclock*(), *posix_appschedattr_getflags*(), *posix_appschedattr_geteventmask*(), and *posix_appschedattr_getreplyinfo*() functions shall return the following error number:

[EPOLICY]	The calling thread is not an application scheduler.

5.9.5 Cross-References

posix_appsched_emptyset(), *posix_appsched_fillset*(), *posix_appsched_addset*(), *posix_appsched_delset*(), *posix_appsched_ismember*(), *posix_appsched_execute_actions*(), *posix_appsched_invoke_withdata*().

5.10 Interfaces for the Scheduled Thread: Explicit Scheduler Invocation

5.10.1 Synopsis

```
#include <sched.h>

int posix_appsched_invoke_scheduler (int user_event_code);

int posix_appsched_invoke_withdata
   (const void *msg, size_t msg_size, void *reply, size_t *reply_size);
```

5.10.2 Description

The *posix_appsched_invoke_scheduler*() or *posix_appsched_invoke_withdata*() functions are used by an application-scheduled thread to explicitly invoke its application scheduler.

If successful, the *posix_appsched_invoke_scheduler*() function shall generate a scheduling event with code POSIX_APPSCHED_EXPLICIT_CALL, a *thread* member equal to the thread id of the calling thread, and an *event_info* member with its *user_event_code* member equal to *user_event_code*. This event shall be inserted in the scheduling events queue of the scheduler thread of the calling thread, and then the calling thread shall become suspended. The function call will return after the scheduler thread activates the calling thread via a *posix_appsched_execute_actions*() call.

If successful, the *posix_appsched_invoke_withdata*() function shall generate a scheduling event with code POSIX_APPSCHED_EXPLICIT_CALL_WITH_DATA, a *thread* member equal to the thread Id of the calling thread, and an *info_size* member equal to *msg_size*. In addition, if *msg_size* is larger than zero, the function shall make available to the scheduler thread a memory area whose contents are identical to the memory area pointed to by *msg* and of size *msg_size*, and shall set the *event_info* member of the event with its *info* member pointing to that area of memory[1]. This event shall be inserted in the scheduling events queue of the scheduler thread of the calling thread, and then the calling thread shall become suspended. When the scheduler thread activates the calling thread via a *posix_appsched_execute_actions*() call, if the *reply* argument is non NULL, the `replyinfo` attribute of the scheduler thread is copied into the memory area pointed to by *reply*, and its size is copied into the variable pointed to by *reply_size*. If the size of that memory area is smaller than the size of the `replyinfo` attribute, results are undefined. The `replyinfo` attribute is set by the scheduler thread via a call to *posix_appsched_setreplyinfo*(). Its size is limited to the variable POSIX_APPSCHEDINFO_MAX (see 5.1.2).

The *posix_appsched_invoke_withdata*() function shall fail if the size specified by *msg_size* is larger than the variable POSIX_APPSCHEDINFO_MAX (see 5.1.2).

The calling thread shall be an application-scheduled thread for these functions to succeed. The scheduler thread shall have the corresponding scheduling event unmasked, for these functions to succeed.

5.10.3 Returns

Upon successful completion, *posix_appsched_invoke_scheduler*() and *posix_appsched_invoke_withdata*() shall return zero. Otherwise, an error number shall be returned to indicate the error.

5.10.4 Errors

If any of the following conditions occurs, the *posix_appsched_invoke_scheduler*() and *posix_appsched_invoke_withdata*() functions shall return the following error number:

[EPOLICY]	The calling thread is not an application-scheduled thread.
[EMASKED]	The operation cannot be executed because the associated scheduling event is currently masked by the application scheduler.

If any of the following conditions occurs, the *posix_appsched_invoke_withdata*() function shall return the following error number:

1. In an implementation in which the scheduler is in the same address space as its threads, copying the information is not needed, and the function may just copy the pointer and size arguments.

[EINVAL]	The value of *msg_size* is less than zero or is larger than POSIX_APPSCHEDINFO_MAX.

If any of the following conditions is detected, the *posix_appsched_invoke_withdata*() function shall return the following error number:

[EINVAL]	The value of *msg* is invalid.

5.10.5 Cross-References

posix_appsched_setreplyinfo(), *posix_appsched_execute_actions*().

5.11 Access to Specific Data of Other Threads

5.11.1 Synopsis

```
#include <pthread.h>

int pthread_setspecific_for
   (pthread_key_t key, pthread_t thread, const void *value)

int pthread_getspecific_from
   (pthread_key_t key, pthread_t thread, void **value)
```

5.11.2 Description

If successful, the *pthread_setspecific_for*() function shall associate a thread-specific *value* with a *key* obtained via a previous call to *pthread_key_create*(), on behalf of the thread specified by *thread*. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread[1].

If successful, the *pthread_getspecific_from*() function shall store in the variable pointed to by *value* the value currently bound to the specified *key* on behalf of the thread specified by *thread*.

The effect of calling *pthread_setspecific_for*() or *pthread_getspecific_from*() with a *key* value not obtained from *pthread_key_create*() or after *key* has been deleted with *pthread_key_delete*() is undefined.

Both *pthread_setspecific_for*() or *pthread_getspecific_from*() may be called from a thread-specific data destructor function. However, calling *pthread_setspecific_for*() from a destructor may result in lost storage or infinite loops.

5.11.3 Returns

If successful, the *pthread_getspecific_from*() function stores in the variable pointed to by *value* the thread-specific data value associated with the given key; if no thread-specific data value is

1. Because sharing data between a scheduler thread and a regular thread has undefined results, the thread-specific data interface should not be used to exchange or share information between a scheduler thread and its scheduled threads.

associated with key, then the value **NULL** is returned. In both cases, the function shall return zero. If the function fails , an error number shall be returned to indicate the error

If successful, the *pthread_setspecific_for*() function shall return zero. Otherwise, an error number shall be returned to indicate the error.

5.11.4 Errors

If any of the following conditions occur, the *pthread_setspecific_for*() function shall return the corresponding error number:

[ENOMEM]	Insufficient memory exists to associate the value with the key.

If any of the following conditions are detected, the *pthread_setspecific_for*() function shall return the corresponding error number:

[EINVAL]	The key value is invalid.

If any of the following conditions are detected, the *pthread_setspecific_for*() and *pthread_getspecific_from*() functions shall return the corresponding error number:

[ESRCH]	No thread could be found corresponding to that specified by the given thread Id.

5.11.5 Cross-References

pthread_key_create(), *pthread_setspecific*(), *pthread_getspecific*(), *pthread_key_delete*().

5.11.6 Rationale

Thread-specific data is a very useful mechanism for attaching information to a particular thread, and in particular, to attach scheduling information. However, the information in the thread-specific data currently defined in POSIX.1 is only accessible to the thread to which it is attached. In the context of application-defined scheduling, it is useful for the application scheduler to access the thread-specific data of its scheduled threads. This data is not usually shared by different threads, but just set and used by the application scheduler itself.

It could be argued that an independent data structure could be created using the thread id as an index or with a hash function that would make access efficient. However, the pthread_t type is defined as an opaque type for which there are no comparison operators (just a *pthread_equal*() function for comparison), and therefore any data structure using a pthread_t index and built at the application level is necessarily very inefficient.

For these reasons the thread-specific data interface has been extended with the functions defined in this subclause.

6. Modifications to Existing Thread Functions

6.1 Thread Creation Scheduling Attributes

The SCHED_APP policy, described in 5.1.3, shall be supported by *pthread_attr_setschedpolicy*().

6.2 Dynamic Thread Scheduling Parameters Access

If *pthread_setschedparam*() is called for a thread whose scheduling policy is SCHED_APP to set that policy to a different one, a POSIX_APPSCHED_TERMINATE event shall be generated for the current application scheduler of that thread, with a *thread* member equal to the *thread* argument; if that event is masked in that application scheduler it is ignored.

If *pthread_setschedparam*() is called for a thread whose scheduling policy is SCHED_APP and the *policy* argument continues to be SCHED_APP, a POSIX_APPSCHED_CHANGE_SCHED_PARAM event shall be generated for the current application scheduler of that thread, with a *thread* member equal to the *thread* argument; if that event is masked in that application scheduler it is ignored.

If *pthread_setschedparam*() is called for a thread whose scheduling policy is different than SCHED_APP to set that policy to SCHED_APP, a check shall be made to determine if the `appscheduler` attribute of that thread is a valid application scheduler. If not, the function shall fail. If it is valid, a POSIX_APPSCHED_NEW event shall be generated for that application scheduler with a *thread* member equal to the *thread* argument, and the calling thread will be suspended until the application scheduler of *thread* executes an "accept" or "reject" action on that thread, via *posix_appsched_execute_actions*(). If the action is "reject", the *pthread_setschedparam*() function shall fail. The function shall also fail if the priority of *thread* is larger than the priority of the application scheduler specified in the `appscheduler` attribute of that thread. The *pthread_setschedparam*() function shall fail if the POSIX_APPSCHED_NEW scheduling event is masked by the corresponding application scheduler.

The following new error conditions are defined for *pthread_setschedparam*():

[EINVAL]	The `appscheduler` attribute of *thread* does not refer to a valid application scheduler at the time of the call.
	The priority of *thread* is larger than the priority of the application scheduler specified in the `appscheduler` attribute of that thread.
[EREJECT]	The application scheduler has rejected attachment of the requested thread.
[EMASKED]	The operation cannot be executed because the POSIX_APPSCHED_NEW scheduling event is currently masked by the application scheduler.

6.3 Thread Creation

If the attributes object used in a call to *pthread_create*() has its scheduling policy set to SCHED_APP, a check shall be made to determine if the `appscheduler` attribute of that attributes object is a valid application scheduler. If not, the function shall fail. If it is valid, after the thread is created, a POSIX_APPSCHED_NEW event shall be generated for that application scheduler with a *thread* member equal to the thread Id of the newly created thread, and the calling thread will be suspended until the application scheduler executes an "accept" or "reject" action on that thread, via *posix_appsched_execute_actions*(). If the action is "reject", the *pthread_create*() function shall fail. The *pthread_create*() function shall fail if the POSIX_APPSCHED_NEW event is masked by the corresponding application scheduler. The function shall also fail if the priority of the new thread, stored in the *sched_priority* member of the `schedparam` attribute in *attr*, is larger than the priority of the application scheduler specified in the `appscheduler` attribute.

The following new error conditions are defined for *pthread_create*():

[EINVAL]	The `appscheduler` attribute of *attr* does not refer to a valid application scheduler at the time of the call.
	The priority of the *sched_priority* member of the `schedparam` attribute in *attr* is larger than the priority of the application scheduler specified in the `appscheduler` attribute.
[EREJECT]	The application scheduler has rejected attachment of the requested thread.
[EMASKED]	The operation cannot be executed because the POSIX_APPSCHED_NEW scheduling event is currently masked by the application scheduler.

6.4 Thread termination and cancellation

After execution of the cleanup handlers and of the destructor functions of any thread-specific data by an explicit or implicit call to *pthread_exit*() or by a call to *pthread_cancel*(), if the terminating thread has a scheduling policy of SCHED_APP, a POSIX_APPSCHED_TERMINATE event shall be generated for the application scheduler of the terminating thread with a *thread* member equal to its thread Id. The terminating thread will be permitted to finish its termination operation concurrently with the processing of the generated event by the application scheduler. If the event is masked in that application scheduler, it is just ignored.

The termination of a scheduler thread (either because it calls *pthread_exit*(), or because the execution of its body terminates, or because it is cancelled) which has scheduled threads attached to it has undefined results. Applications should make sure that a scheduler thread only terminates if it has no attached threads.

6.4.1 Rationale

Because thread-specific data is destroyed when the thread terminates, the application scheduler processing a POSIX_APPSCHED_TERMINATE event should not attempt using it. If it is necessary for the scheduler to perform operations on the terminating thread just before its termination, it could install a cancellation handler that could invoke the scheduler explicitly. Cancellation handlers are executed before destroying thread-specific data.

7. Use of Regular Mutexes by Application-Scheduled Threads

When an application-scheduled thread locks or unlocks a regular mutex (i.e., one that has not been created with the PTHREAD_APPSCHED_PROTOCOL protocol) its priority is changed according to the mutex protocol. In addition, if the mutex uses the priority inheritance or priority ceiling protocols, a scheduling event of type POSIX_APPSCHED_PRIORITY_INHERIT or POSIX_APPSCHED_PRIORITY_UNINHERIT is queued to the scheduler thread at the time a priority is inherited or uninherited. The scheduler thread shall inherit (or uninherit) the same active system priority or priorities of its scheduled threads.

If the application-scheduled thread tries to lock an already locked mutex, it is queued in the mutex queue in the same way as for any other thread. Besides this queuing, a scheduling event of type POSIX_APPSCHED_BLOCK is sent to the scheduler thread, as for any other blocking situation.

The events of type POSIX_APPSCHED_PRIORITY_INHERIT and POSIX_APPSCHED_PRIORITY_UNINHERIT can be used by the application scheduler to keep the thread that locked the mutex in execution. Otherwise, regular threads that use the same mutex could suffer priority inversion due to the execution of other application-scheduled threads.

8. Management of Application-Scheduled Mutexes

8.1 Data definitions

8.1.1 Application-Scheduled Mutex Protocol

The following mutex protocol constant shall be defined in <`pthread.h`>:

Table 3:

Symbol	Description
PTHREAD_APPSCHED_PROTOCOL	Application-defined protocol

When this protocol is set for a mutex, the mutex shall only be used by threads scheduled under the scheduler referenced by the `appscheduler` attribute of the mutex. In addition, scheduling events are notified to the scheduler thread under the following circumstances:

- when a thread invokes the lock operation of the mutex,
- when a thread invokes the trylock operation of the mutex,
- when a thread unlocks the mutex,
- when a thread blocks waiting for the mutex,
- when a thread changes the scheduling parameters of the mutex.

8.1.2 Application-Scheduled Mutex Attributes

All application-scheduled mutexes shall support the `appscheduler` attribute that represents the scheduler thread that will schedule threads using that mutex. The default value of this attribute is unspecified. If at the time of the mutex creation the protocol is PTHREAD_APPSCHED_PROTOCOL and the `appscheduler` attribute does not refer to a valid application scheduler thread, the corresponding *pthread_mutex_init*() operation shall fail with an error of [EINVAL].

In addition, all application-scheduled mutexes shall support the `appschedparam` attribute, which represents an area of memory containing the application-specific scheduling parameters used for that mutex. The default value of this attribute is a memory area of zero bytes. The maximum size of this attribute in bytes shall be represented by the variable POSIX_APPMUTEXPARAM_MAX (see 5.1.2).

8.2 Application-Scheduled Mutex Attributes Manipulation

8.2.1 Synopsis

```
#include <pthread.h>
```

```
int pthread_mutexattr_setappscheduler
     (pthread_mutexattr_t *attr,
      pthread_t scheduler);

int pthread_mutexattr_getappscheduler
     (const pthread_mutexattr_t *attr,
      pthread_t *scheduler);

int pthread_mutexattr_setappschedparam
     (pthread_mutexattr_t *attr,
      const void *param,
      size_t param_size);

int pthread_mutexattr_getappschedparam
     (const pthread_mutexattr_t *attr,
      void *param,
      size_t *paramsize);
```

8.2.2 Description

The *pthread_mutexattr_setappscheduler*() and *pthread_mutexattr_getappscheduler*() functions shall respectively set and get the `appscheduler` attribute of the mutex attributes object specified by *attr*. This attribute is described in 8.1.2. If successful, the *pthread_mutexattr_setappscheduler*() shall set the attribute equal to *scheduler*. For this function to succeed, *scheduler* must be a valid application scheduler at the time of the call.

The *pthread_mutexattr_setappschedparam*() and *pthread_mutexattr_getappschedparam*() functions shall respectively set and get the `appschedparam` attribute of the mutex attributes object specified by *attr*. This attribute is described in 8.1.2. If successful, the *pthread_mutexattr_setappschedparam*() function shall set the size of the attribute to *paramsize* and shall copy the memory area pointed to by `param` of length *paramsize* into the attribute. If *paramsize* is larger than POSIX_APPMUTEXPARAM_MAX, the function shall fail.

The *pthread_mutexattr_getappschedparam*() function shall copy the contents of the `appschedparam` attribute into the memory area pointed to by *param*. This memory area shall be capable of storing at least a number of bytes equal to the size of the `appschedparam` attribute; otherwise, the results are undefined.

8.2.3 Returns

Upon successful completion, *pthread_mutexattr_setappscheduler*(), *pthread_mutexattr_setappschedparam*(), *pthread_mutexattr_getappscheduler*(), and *pthread_mutexattr_getappschedparam*() shall return a value of 0. Otherwise an error number shall be returned to indicate the error.

The *pthread_mutexattr_getappscheduler*() function stores the `appscheduler` attribute in the variable pointed to by *scheduler* if successful.

The *pthread_attr_getappschedparam*() function stores the size of the `appschedparam` attribute in the variable pointed to by *paramsize*, and copies the contents of the `appschedparam` attribute into the memory area pointed to by *param*.

8.2.4 Errors

If any of the following conditions occur, the *pthread_mutexattr_setappscheduler*() function shall return the corresponding error number:

[EINVAL]	The value of *scheduler* was not valid.

If any of the following conditions occur, the *pthread_mutexattr_setappschedparam*() function shall return the corresponding error number:

[EINVAL]	The value of *paramsize* was smaller than zero, or was larger than POSIX_APPMUTEXPARAM_MAX.

If any of the following conditions are detected, the *pthread_mutexattr_setappschedparam*() function shall return the corresponding error number:

[EINVAL]	The value of *param* was invalid.

8.2.5 Cross-References

pthread_mutex_init().

8.3 Dynamically Changing the Application-Scheduled Mutex Attributes

8.3.1 Synopsis

```
#include <pthread.h>

int pthread_mutex_getappscheduler
     (const pthread_mutex_t *mutex,
      pthread_t *scheduler);

int pthread_mutex_setappschedparam
     (pthread_mutex_t *mutex,
      const void *param,
      size_t param_size);

int pthread_mutex_getappschedparam
     (const pthread_mutex_t *mutex,
      void *param,
      size_t *paramsize);
```

8.3.2 Description

The *pthread_mutex_getappscheduler*() function is used to dynamically get the `appscheduler` attribute of the mutex specified by *mutex*. This attribute is described in 8.1.2.

The *pthread_mutex_setappschedparam*() and *pthread_mutex_getappschedparam*() functions are used to dynamically set and get the `appschedparam` attribute of the mutex attributes object specified by *mutex*. This attribute is described in 8.1.2.

If successful, the *pthread_mutex_setappschedparam*() function shall acquire the lock on the mutex. Then it shall set the size of the `appschedparam` attribute to the value specified by *paramsize*, and it shall copy the scheduling parameters occupying *paramsize* bytes and pointed to by *param* into that attribute. In addition, if the protocol of *mutex* is PTHREAD_APPSCHED_PROTOCOL and if the POSIX_APPSCHED_CHANGE_MUTEX_SCHED_PARAM event is not masked in the application scheduler of that mutex, the function shall generate a POSIX_APPSCHED_CHANGE_MUTEX_SCHED_PARAM event for that application scheduler and shall suspend until the scheduled thread activates it via a *posix_appsched_execute_actions*() call. Finally, the function shall release the lock on the mutex.

For the *pthread_mutex_setappschedparam*() function to succeed, the calling thread must be an application-scheduled thread scheduled by the same thread as the mutex.

The memory area represented by *param* in the call to *pthread_getappschedparam*() shall be capable of storing at least a number of bytes equal to the size of the `appschedparam` attribute; otherwise, the results are undefined.

8.3.3 Returns

Upon successful completion, *pthread_mutex_getappscheduler*(), *pthread_mutex_setappschedparam*(), and *pthread_mutex_getappschedparam*() shall return a value of 0. Otherwise an error number shall be returned to indicate the error.

The *pthread_mutex_getappscheduler*() function stores the `appscheduler` attribute in the variable pointed to by *scheduler*, if successful.

The *pthread_mutex_getappschedparam*() function stores the size of the `appschedparam` attribute in the variable pointed to by `paramsize`, and copies the contents of the `appschedparam` attribute into the memory area pointed to by *param*, if successful.

8.3.4 Errors

If any of the following conditions occur, the *pthread_mutex_setappschedparam*() function shall return the corresponding error number:

[EINVAL]	The value of *paramsize* was less than zero, or was larger than POSIX_APPMUTEXPARAM_MAX.
[EPOLICY]	The calling thread is not an application-scheduled thread scheduled by the same thread as the mutex.

If any of the following conditions are detected, the *pthread_mutex_setappschedparam*() function shall return the corresponding error number:

[EINVAL]	The value of *param* was invalid.

If any of the following conditions are detected, the *pthread_mutex_getappscheduler*(), *pthread_mutex_setappschedparam*(), and *pthread_mutex_getappschedparam*() functions shall return the corresponding error number:

[EINVAL]	The value specified by *mutex* does not refer to an initialized mutex object.

If any of the following conditions are detected, the *pthread_mutex_setappschedparam*() function shall return the corresponding error number:

[EDEADLK]	The current thread already owns the mutex.

8.3.5 Cross-References

pthread_mutex_init(), *pthread_mutexattr_setappscheduler*, *pthread_mutexattr_setappschedparam*().

8.3.6 Rationale

The capability of switching a mutex from one application scheduler to another one is not provided because it can be implemented by destroying the mutex and initializing it again, with new attributes.

8.4 Mutex-Specific Data

8.4.1 Synopsis

```
int posix_appsched_mutex_setspecific
     (pthread_mutex_t *mutex,
      const void *value);
int posix_appsched_mutex_getspecific
     (const pthread_mutex_t *mutex,
      void **value);
```

8.4.2 Description

In successful, the *posix_appsched_mutex_setspecific*() function shall associate a mutex-specific *value* with the mutex specified by *mutex*. This value is typically a pointer to a block of dynamically allocated memory that has been reserved for use by the calling thread[1].

In successful, the *posix_appsched_mutex_getspecific*() function shall store in the variable pointed to by *value* the value currently bound to the specified *mutex*.

8.4.3 Returns

If successful, the *posix_appsched_mutex_getspecific*() function stores in the variable pointed to by *value* the mutex-specific data value associated with *mutex*; if no mutex-specific data value is associated with it, then the value **NULL** is returned. In both cases, the function shall return zero. If the function fails, an error number shall be returned to indicate the error.

If successful, the *posix_appsched_mutex_setspecific*() function shall return zero. Otherwise, an error number shall be returned to indicate the error.

1. Because sharing data between a scheduler thread and a regular thread has undefined results, the mutex-specific data interface should not be used to exchange or share information between a scheduler thread and its scheduled threads.

8.4.4 Errors

If any of the following conditions are detected, the *posix_appsched_mutex_setspecific*() and *posix_appsched_mutex_getspecific*() functions shall return the corresponding error number:

[EINVAL]	The value specified by *mutex* does not refer to an initialized mutex object.

8.4.5 Cross-References

pthread_setspecific(), *pthread_getspecific*(), *pthread_setspecific_for*(), *pthread_getspecific_from*().

8.4.6 Rationale

POSIX.1 defines a thread-specific data mechanism that is very useful for attaching information to a particular thread, and in particular, to attach scheduling information. However, the standard does not define a similar functionality for mutexes, which would be very useful in the context of writing application schedulers. For this reason the mutex-specific data interface has been introduced.

8.5 Modifications to existing functions: Initializing and Destroying a Mutex

If the mutex attributes object used in a call to *pthread_mutex_init*() has its `protocol` attribute set to POSIX_APPSCHED_PROTOCOL, a check shall be made to determine if the `appscheduler` attribute of that attributes object is a valid application scheduler. If not, the function shall fail. If it is valid, after the mutex is initialized, a POSIX_APPSCHED_INIT_MUTEX event shall be generated for that application scheduler with a *mutex* member equal to *mutex*, and the calling thread shall be suspended until the application scheduler executes an “mutex-accept” or “mutex-reject” action on that mutex, via *posix_appsched_execute_actions*(). If the action is “mutex-reject”, the *pthread_mutex_init*() function shall fail. The *pthread_mutex_init*() function shall fail if the POSIX_APPSCHED_INIT_MUTEX event is masked by the corresponding application scheduler.

The following new error conditions are defined for *pthread_mutex_init*():

[EINVAL]	The `appscheduler` attribute of *attr* does not refer to a valid application scheduler at the time of the call.
[EREJECT]	The application scheduler has rejected attachment of the requested mutex.
[EMASKED]	The operation cannot be executed because the POSIX_APPSCHED_INIT_MUTEX scheduling event is currently masked by the application scheduler.

If the mutex specified in a call to *pthread_mutex_destroy*() has its `protocol` attribute set to POSIX_APPSCHED_PROTOCOL and the POSIX_APPSCHED_DESTROY_MUTEX event is not masked in the application scheduler of that mutex, after destroying the mutex one such event shall be generated for that application scheduler with a *mutex* member equal to *mutex*, and the calling thread shall be suspended until the application scheduler activates it via a “thread-activate” action in a call to *posix_appsched_execute_actions*().

8.6 Modifications to existing functions: Locking and Unlocking a Mutex

If the mutex specified in a call to *pthread_mutex_lock*(), has its `protocol` attribute set to POSIX_APPSCHED_PROTOCOL, before the mutex is locked, a POSIX_APPSCHED_LOCK_MUTEX event shall be generated for the application scheduler of the mutex with a *mutex* member equal to *mutex*, and the calling thread shall be suspended until the application scheduler executes a "mutex-lock" action on that mutex, via *posix_appsched_execute_actions*(). After that action the function shall return with the mutex locked. The *pthread_mutex_lock*() function shall fail if the POSIX_APPSCHED_LOCK_MUTEX event is masked by the corresponding application scheduler.

The *pthread_mutex_timedlock*() function shall behave the same as *pthread_mutex_lock*(), with the additional requirement that if the timeout expires and the POSIX_APPSCHED_READY event is not masked by the application scheduler, the thread discontinues its wait on the mutex, a POSIX_APPSCHED_READY is generated for that scheduler, and the thread is suspended until activated via a "thread-activate" action executed by *posix_appsched_execute_actions*().

The following new error condition is defined for *pthread_mutex_lock*() and *pthread_mutex_timedlock*():

[EMASKED]	The operation cannot be executed because the POSIX_APPSCHED_LOCK_MUTEX scheduling event is currently masked by the application scheduler.

If the mutex specified in a call to *pthread_mutex_trylock*(), has its `protocol` attribute set to POSIX_APPSCHED_PROTOCOL, before the mutex is locked, a POSIX_APPSCHED_TRYLOCK_MUTEX event shall be generated for the application scheduler of the mutex with a *mutex* member equal to *mutex*, and the calling thread shall be suspended until the application scheduler executes either a "mutex-lock" or a "thread-activate" action on that mutex, via *posix_appsched_execute_actions*(). If the action is a "mutex-lock", the function shall return with the mutex locked; if it is a "thread-activate" the function shall return with an error of [EBUSY]. The *pthread_mutex_lock*() function shall fail if the POSIX_APPSCHED_TRYLOCK_MUTEX event is masked by the corresponding application scheduler.

The following new error condition is defined for *pthread_mutex_trylock*():

[EMASKED]	The operation cannot be executed because the POSIX_APPSCHED_TRYLOCK_MUTEX scheduling event is currently masked by the application scheduler.

If the mutex specified in a call to *pthread_mutex_unlock*(), has its `protocol` attribute set to POSIX_APPSCHED_PROTOCOL, and the POSIX_APPSCHED_UNLOCK_MUTEX event is not masked in the application scheduler of that mutex, after releasing the mutex but before granting it to another thread one POSIX_APPSCHED_UNLOCK_MUTEX event shall be generated for the application scheduler with a *mutex* member equal to *mutex*, and the calling thread shall be suspended until the application scheduler activates it via a "thread-activate" action in a call to *posix_appsched_execute_actions*().

9. OS Implementation Considerations

Each scheduler thread needs a list of all its associated threads, which are scheduled by it. This is a dynamic list that grows when a new thread joins the scheduler, and shrinks when a scheduled thread is terminated or abandons the scheduler.

This list could be used to determine the active system priority of the scheduler thread, taking into account the priorities inherited by the scheduled threads.

Each scheduler thread needs a FIFO queue of scheduling events. There is no need to define a constant for the length of this queue, because it is bounded by two times the maximum number of threads in the system, plus one. This bound is originated because each application-scheduled thread in the system can cause at most one scheduling event before executing again, and it will only execute after the scheduler has processed the generated event. For system scheduled threads, the only events that are relevant are the priority inheritance or uninheritance, so there may be in the worst case two events per such task. And in addition, a single timeout or signal event could be generated.

10. Example of using the defined interface

The following subsections show the code of an implementation of the Earliest Deadline First (EDF) scheduling policy for periodic threads {B15}. In this policy, each thread defines a period and a relative deadline as two time intervals. The thread executes periodically, one execution instance at each period. The n^{th} instance is activated at time t_0+nT, when t_0 is the initial time, and T is the period. The n^{th} instance has to complete before its absolute deadline, which is its activation time plus the relative deadline: t_0+nT+D, where D is the relative deadline. The EDF policy gives the highest priority to the active thread that has the earliest absolute deadline. Because the absolute deadline changes for every task instance, this is a dynamic priority scheduling policy, in which the priority is the absolute deadline.

The EDF scheduler has a list of threads that are registered for being scheduled under it. The state of each EDF thread can be *active* (when it is ready for executing), *blocked* (if the thread is blocked at some OS service) or *timed* (when it has finished its current execution and is waiting for its next period).

The state of the scheduler itself may be *idle* (no active or timed threads), *waiting* (there are timed threads but no active threads), or *running* (there are active threads).

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.

10.1 EDF Scheduler Thread Interface: File “edf_sched.h”

```
#include <time.h>

#define MAX_THREADS 30

struct my_sched_param {
  struct timespec deadline;
  struct timespec period;
};

void *edf_scheduler (void *arg);
```

10.2 EDF Threads and Main Program

```
#include <pthread.h>
#include <sched.h>
#include <unistd.h>
#include "edf_sched.h"

/*  Scheduled thread */
void * periodic (void * arg)
{
  while (1) {
    /*do useful work */
    posix_appsched_invoke_scheduler (0);
  }
}

int main ()
{
  pthread_attr_t attr;
  struct sched_param param;
  struct my_sched_param user_param;
  pthread_t sched, t1;

  /* Creation of the scheduler thread */
  pthread_attr_init (&attr);
  pthread_attr_setschedpolicy (&attr, SCHED_FIFO);
  pthread_attr_setappschedulerstate (&attr, PTHREAD_APPSCHEDULER);
  param.sched_priority = 5;
  pthread_attr_setschedparam (&attr, &param);
  pthread_create (&sched, &attr, edf_scheduler, NULL);
  pthread_attr_destroy(&attr);

  /* Creation of one scheduled thread */
  pthread_attr_init (&attr);
  pthread_attr_setschedpolicy (&attr, SCHED_APP);
  pthread_attr_setappscheduler (&attr, sched);
  user_param.period.tv_sec =1;
  user_param.period.tv_nsec =0;
  user_param.deadline.tv_sec = 0;
  user_param.deadline.tv_nsec = 800000000;
  pthread_attr_setappschedparam (&attr,
                               (void *) &user_param,
                               sizeof (struct my_sched_param));
  param.sched_priority = 3;
  pthread_attr_setschedparam (&attr, &param);
  pthread_create (&t1, &attr, periodic, NULL);

  /* create more threads, do more work... */
}
```

10.3 EDF Scheduler Thread Implementation

```
#include <pthread.h>
#include <sched.h>
#include <unistd.h>
#include "edf_sched.h"
#include "timespec_operations.h"

struct thread_data_t {
  pthread_t thread_id;
  enum {ACTIVE, BLOCKED, TIMED} th_state;
  struct timespec period,deadline; /* relative times */
  struct timespec next_deadline, next_start; /* absolute times */
} thread_data[MAX_THREADS];

int num_of_threads = 0;

pthread_key_t edf_key;

typedef enum {IDLE, WAITING, RUNNING} sch_state_t;

void schedule_next (sch_state_t *sch_state,
                    struct thread_data_t **current_thread,
                    struct timespec *earliest_start,
                    const struct timespec *now)
{
  int t, deadline_th, start_th;
  struct timespec earliest_deadline;
  struct timespec large_time={2147483647,0};

  /* find_next_thread to run and next thread to activate*/
  deadline_th = -1;
  start_th = -1;
  if (num_of_threads>0) {
    earliest_deadline = large_time;
    *earliest_start = large_time;
    for (t=0; t<num_of_threads; t++) {
      /* find threads to wake up */
      if (thread_data[t].th_state==TIMED) {
        if (smaller_or_equal_timespec(&thread_data[t].next_start,now)) {
          thread_data[t].th_state=ACTIVE;
        }
      }
      /* find_next_thread to run */
      if (thread_data[t].th_state==ACTIVE) {
        if (smaller_timespec(&thread_data[t].next_deadline,
                             &earliest_deadline)) {
          earliest_deadline=thread_data[t].next_deadline;
          deadline_th = t;
        }
      }
      /* find next thread to activate*/
      if (thread_data[t].th_state==ACTIVE ||
          thread_data[t].th_state==TIMED) {
        if (smaller_timespec(&thread_data[t].next_start,
                             earliest_start))
          {
            *earliest_start=thread_data[t].next_start;
            start_th = t;
          }
      }
    }
  }
  /* process current thread to run and current thread to wakeup */
  if (start_th == -1) {
    *sch_state = IDLE;
  } else {
    if (deadline_th == -1) {
      *sch_state = WAITING;
    } else {
      *sch_state = RUNNING;
      *current_thread = &thread_data[deadline_th];
    }
  }
}

void add_to_list_of_threads (pthread_t thread_id,
                             const struct timespec *now)
{
  struct my_sched_param param;
  size_t param_size;

  if (num_of_threads<MAX_THREADS) {
    pthread_getappschedparam (thread_id, &param, &param_size);
    thread_data[num_of_threads].period = param.period;
    thread_data[num_of_threads].deadline = param.deadline;
    thread_data[num_of_threads].next_start = *now;
    add_timespec(&thread_data[num_of_threads].next_start,
                 &param.deadline,
                 &thread_data[num_of_threads].next_deadline);
    num_of_threads++;
    thread_data[num_of_threads].th_state=ACTIVE;
    thread_data[num_of_threads].thread_id=thread_id;
    pthread_setspecific_for (edf_key, thread_id,
                             &thread_data[num_of_threads]);
  }
}

void eliminate_from_list_of_threads (pthread_t thread_id)
{
  int found=-1;
  int t;

  // At this point 'pthread_getspecific_from' cannot be used because
  // the thread is already terminated, so a search has to be performed.
  for (t=0;t<num_of_threads;t++) {
    if (pthread_equal (thread_data[t].thread_id, thread_id)) {
      found=t;
      break;
    }
  }
  if (found>=0) {
    num_of_threads--;
    for (t=found;t<num_of_threads;t++) {
      thread_data[t]=thread_data[t+1];
    }
  }
}

void make_ready (pthread_t thread_id)
{
  struct thread_data_t *td;

  pthread_getspecific_from (edf_key, thread_id, (void **)&td);
  td->th_state=ACTIVE;
}

void make_blocked (pthread_t thread_id)
{
  struct thread_data_t *td;

  pthread_getspecific_from (edf_key, thread_id, (void **)&td);
  td->th_state=BLOCKED;
}

void make_timed (pthread_t thread_id)
{
  struct thread_data_t *td;

  pthread_getspecific_from (edf_key, thread_id, (void **)&td);
  td->th_state=TIMED;
  incr_timespec(&td->next_start, &td->period);
  incr_timespec(&td->next_deadline, &td->period);
}

void *edf_scheduler (void *arg)
{
  posix_appsched_actions_t actions;
  struct posix_appsched_event sched_event;
  struct timespec earliest_start,now;
  struct thread_data_t *new_thread, *current_thread = NULL;
  sch_state_t sch_state;

  /* Init scheduler */
  posix_appschedattr_setclock (CLOCK_REALTIME);
  posix_appschedattr_setflags (POSIX_APPSCHED_ABSTIMEOUT);
  clock_gettime(CLOCK_REALTIME, &now);
  posix_appsched_actions_init (&actions);
  pthread_key_create (&edf_key, NULL);
  while (1) {
    schedule_next (&sch_state, &new_thread, &earliest_start, &now);
    /* schedule thread */
    switch (sch_state) {
    case IDLE :
      posix_appsched_execute_actions
         (&actions,NULL,NULL,&now,&sched_event);
      break;
    case WAITING :
      posix_appsched_execute_actions (&actions,NULL,&earliest_start,
                                      &now,&sched_event);
      break;
    case RUNNING :
      /* Set scheduling actions */
      if (new_thread != current_thread) {
        // Activate new thread
        posix_appsched_actions_addactivate
           (&actions, new_thread->thread_id);
        if (current_thread->th_state == ACTIVE) {
          // Suspend old thread
          posix_appsched_actions_addsuspend (&actions,
                                             current_thread->thread_id);
        }
        current_thread = new_thread;
      }
      posix_appsched_execute_actions
        (&actions, NULL, &earliest_start, &now, &sched_event);
    }
    posix_appsched_actions_reset (&actions);

    /* Process scheduling events */
    switch (sched_event.event_code) {
    case POSIX_APPSCHED_NEW :
      add_to_list_of_threads (sched_event.thread,&now);
      posix_appsched_actions_addaccept (&actions, sched_event.thread);
      break;
    case POSIX_APPSCHED_TERMINATE :
      eliminate_from_list_of_threads (sched_event.thread);
      break;
    case POSIX_APPSCHED_READY :
      make_ready (sched_event.thread);
      break;
    case POSIX_APPSCHED_BLOCK :
      make_blocked (sched_event.thread);
      break;
    case POSIX_APPSCHED_EXPLICIT_CALL :
      make_blocked (sched_event.thread);
      break;
    case POSIX_APPSCHED_TIMEOUT :
      /* no action needed */
      break;
```

```
        }
    }
}
```

References

{B1} 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.

{B2} 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

{B3} 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.

{B4} 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.

{B5} 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.

{B6} 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.

{B7} 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.

{B8} 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.

{B9} Bryan Ford and Sai Susarla, "CPU Inheritance Scheduling". *Proceedings of OSD*I, October 1996.

{B10} 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.

{B11} OMG. *Real-Time CORBA 2.0: Dynamic Scheduling*, Joint Final Submission. OMG Document orbos/2001-06-09, June 2001.

{B12} Yodaiken V., "An RT-Linux Manifesto". *Proceedings of the 5th Linux Expo*, Raleigh, North Carolina, USA, May 1999.

{B13} 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.

{B14} 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.

{B15} C.L. Liu and J.W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment”. Journal of the ACM, Vol. 20, No. 1, 1973, pp. 46-61.