Application-Defined Scheduling in Ada

Mario Aldea Rivas and Michael González Harbour

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

***Abstract**: This paper is a proposal for an application program interface (API) that would enable Ada applications to use application-defined scheduling algorithms in a way compatible with the scheduling model of the Ada 95 Real-Time Systems Annex. With this API, several application-defined schedulers, implemented by extending a tagged type, can coexist in the system in a predictable way together with their scheduled tasks, and with regular fixed priority tasks directly scheduled by the run-time system. Task synchronization through protected objects is also considered by adding the Stack Resource Policy, which can be used in a large variety of fixed and dynamic priority scheduling policies.*

Keywords: *Real-Time Systems, Scheduling, Application-defined schedulers, Protected Objects, Stack Resource Policy, Ada 95*.

1. Introduction[1]

The Real-Time Annex in Ada 95 defines only one scheduling policy: `FIFO_Within_Priorities`. Although fixed priority scheduling is an excellent choice for real-time systems, there are application requirements that cannot be fully accomplished with these policies only. It is well known that with dynamic priority scheduling policies it is possible to achieve higher utilization levels of the system resources than with fixed priority policies. In addition, there are many systems for which their dynamic nature make it necessary to have very flexible scheduling mechanisms, such as multimedia systems, in which different quality of service properties need to be traded against one another.

Another requirement for flexible scheduling schemes comes from the need to compose several applications, each with their own scheduling policy, into the same system. For example, tasks with multimedia quality of service requirements could coexist in the same system with control tasks with timing requirements that are better fulfilled with traditional fixed priority scheduling. In this context, the ability to run several schedulers composed in a hierarchical way is a natural solution.

It could be possible to incorporate into the Ada standard new dynamic scheduling policies to be used in addition to the existing policies. The main problem is that the variety of these policies is so great that it would be difficult to standardize on just a few. Different applications needs would require different policies. Instead, in this paper we propose defining an interface for application-defined schedulers that could be used to implement a large variety of scheduling policies.

Task scheduling cannot be separated from the treatment of task synchronization, because the timing properties of the scheduling policy are very much affected by the synchronization protocols used. In real-time systems it is usual that a task has to cooperate tightly with many other tasks, and a lot of shared objects and associated synchronized operations are necessary. In the past, we had proposed application scheduling synchronization protocols that were implemented in the same scheduler where the task scheduling policy was implemented [15][16]. This approach is the most general and may be useful in some particular situations, but introduces a lot of overhead. While the overhead of task scheduling is usually acceptable for each time the task is made ready or is suspended, it is not acceptable to invoke the scheduler task at each synchronized operation, given a large amount of them.

As a consequence, in this paper we propose adding the Stack Resource Policy (SRP) [13] to the Ada language, because this is a synchronization policy that can work with many fixed and dynamic priority task scheduling policies. It is not a general solution, but it is an efficient approach for most of the approaches used today such as fixed priorities, sporadic servers, EDF, CBS, etc. As we will show, the SRP

1. This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grant TIC 2002-04123-C03 and by the *Commission of the European Communities* under contract IST-2001-34140 (FIRST project)

policy will require some small interaction with the application-defined schedulers.

This paper is organized as follows: Section 2 discusses some related work and the motivation for the proposal. Section 3 describes the general overall model for application-defined scheduling. Section 4 contains some general ideas about the API. Section 5 describes the SRP for protected objects and how the interactions between synchronization and the application schedulers can be handled. Section 6 contains an example of an EDF application scheduler. Section 7 contains some conclusions and discusses further work. Finally, an Appendix is included with the proposed API.

2. Related Work and Motivation

The idea of application-defined scheduling has been used in many systems. A solution is proposed in RED-Linux [11], 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 task priorities, start and finish times, and execution time budget. With that mechanism some scheduling algorithms can be implemented but there may be others that cannot be implemented if they are based on parameters different from those included in the aforementioned attributes object. In addition, this solution does not address the implementation of protocols for shared resources that could avoid priority inversion or similar effects.

A different approach is followed in the CPU Inheritance Scheduling [7], in which the kernel only implements task blocking, unblocking and CPU donation, and the application defined schedulers are tasks which donate the CPU to other tasks. In this approach the only method used to avoid priority inversion is the priority inheritance. In addition although this approach supports multiprocessor schedulers, it is not possible to have a 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 [8], RT-Linux [12], Vassal [3]). With this mechanism the functions exported by the modules are invoked from the kernel at every scheduling point. This is a very efficient and general method but, as a drawback, the application scheduling algorithms can neither be isolated from each other nor from the kernel itself, so, a bug in one of them could affect the whole system.

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

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

b) The application scheduling code can use standard interfaces and operating system calls. In some systems part of these interfaces might not be accessible for invocation from inside the kernel.

We have designed our interface so that several application-defined schedulers can be defined, and so that they have a behavior compatible with other existing scheduling policies, both on single processor and multiprocessor platforms.

The shadow task mechanism [14] is used in some systems for task synchronization under application-defined schedulers. This is a general mechanism, but has the drawback that it is difficult to use for synchronizing tasks of different schedulers, because they could execute without permission from their respective schedulers.

Baker presents in [13] the Stack Resource Policy (SRP) for bounding priority inversion in real time systems, independently from the scheduling policy used. The method can be applied to fixed priority or EDF schedulers, for instance. A number called the preemption level is assigned to each task, using the priority or importance of each task: the higher the priority, the higher the preemption level. Shared resources are also assigned a preemption level that is the highest of the preemption levels of all the tasks that may use that resource. And a new scheduling rule is imposed: a task can only become ready for execution if its preemption level is strictly higher than the preemption levels of the resources currently locked in the system. With this protocol, in a single processor a task can be delayed by lower priority tasks only once, during the duration of one critical section. In a fixed priority system, if the preemption level is made equal to the task priorities, the SRP protocol behaves as Ada's ceiling locking policy.

Because the SRP protocol is usable for a fair number of policies and requires very little interaction with the applica-

tion scheduler, we propose it in this paper to be provided as one of Ada's standard locking policies.

In summary, the motivation for this work is to provide developers of applications with a flexible scheduling mechanism, handling both task scheduling and synchronization, that enables them to schedule dynamic applications that would not meet their requirements using the more rigid fixed-priority scheduling currently provided in Ada. This mechanism allows isolation of the kernel from misbehaved application schedulers.

3. Overview of the scheduling model

Figure 1 shows the proposed approach for application-defined scheduling. Each application scheduler is a special software module that can be implemented inside the run-time system or as a special task, and that is responsible of scheduling a set of tasks that have been attached to it. According to the way a task is scheduled, we can categorize the tasks as:

- *System-scheduled tasks*: these tasks are scheduled directly by the run-time system and/or the operating system, without intervention of a scheduler task.
- *Application-scheduled tasks*: these tasks are also scheduled by the run-time system and/or the operating system, but before they can be scheduled, they need to be made ready by their application-defined scheduler.

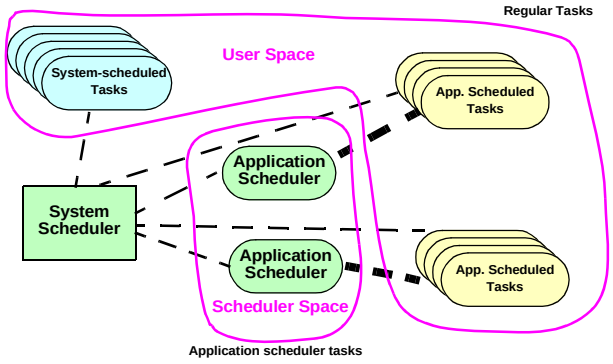

Figure 1. Model for Application Scheduling

Because the scheduler may execute in an environment different than that of the application tasks, it is an error to share information between the scheduler and the rest of the application. An API is provided for exchanging information when needed. Application schedulers may share information among them.

Because the use of protected resources may cause priority inversions or similar delay effects, it is necessary to provide a general mechanism that can bound the delays caused by synchronization when using different kinds of scheduling techniques, and also when several schedulers are being used at the same time. We have chosen the Stack Resource Policy (SRP) [13] because it is applicable to a large variety of policies, including both fixed and deadline-based priority policies. For this purpose we define a new locking policy for pragma `Locking_Policy` with the `Stack_Resource_Locking` identifier.

4. Scheduling API

4.1. Scheduling framework

The scheduling API presented in this section is designed to be compatible with the new `Round_Robin` policy proposed in [17]. In that proposal, compatible scheduling policies are allowed in the system under the `Priority_Specific Task_Dispatching_Policy`, and the specific policy is assigned to each particular priority level, with the `Priority_Policy` pragma; two values are allowed: `Fifo_Within_Priorities`, or `Round_Robin`. At each priority level, only one policy is available, thus avoiding the potentially unpredictable effects of mixing tasks of different policies at the same level.

We propose adding one more value that could be used with the `Priority_Policy` pragma: `Application_Defined`; it represents tasks that are application scheduled. If the scheduler is implemented as a special task, its base priority must be at least equal to that of its scheduled tasks.

4.2. Application scheduler

Application schedulers are defined by extending the `Scheduler` abstract tagged type defined in the new package `Ada.Application_Scheduling`. This type contains primitive operations that are invoked by the system when a scheduling event occurs. The type is extended by adding the data structures required by the scheduler (for example, a ready queue and a delay queue), and by overriding the primitive operations of interest to perform the scheduling decisions required to implement the desired scheduling policy.

In addition, we need to extend the `Scheduling_Parameters` tagged type defined in the same package to contain all the information that is necessary to specify the scheduling parameters of each task (such as its deadline, execution-time budget, period, and similar parameters).

To install one of these application schedulers using the priority-specific framework, we need to specify that the desired priority level, P, will be using an application-defined scheduler. We do so with the following configuration pragmas:

```
pragma Task_Dispatching_Policy
   (Priority_Specific);
pragma Priority_Policy (Application_Defined, P);
pragma Locking_Policy (Stack_Resource_Locking);
```

Then, we need to attach the scheduler to the chosen priority level through the use of the following pragma:

```
pragma Application_Scheduler (My_Scheduler,P);
```

where `My_Scheduler` is the scheduler type and `P` is the desired priority level. Before the scheduler is elaborated, no task at this priority level can run.

As we mentioned before, the scheduler may be implemented inside the run-time system, or as a user-level task. In both cases the system must ensure that the application-scheduled tasks cannot preempt their scheduler. Because the active priority of an application-scheduled task may change because of the inheritance of other priorities through the use of protected objects, it is necessary that in the user-task implementation the scheduler also inherits the same priority. In this way, the scheduler always takes precedence over its scheduled tasks. It is not necessary though that this priority is inherited by the rest of the tasks scheduled by that scheduler.

When the active priority of a task is higher than its base priority, it always takes precedence over any application scheduling parameters. Therefore, application-scheduled tasks take precedence over tasks with lower active priority, and they are always preempted by tasks with higher active priority that become ready.

Each application-defined scheduler may simultaneously make ready many application-scheduled tasks, to run concurrently. The scheduler may also block previously ready tasks. Among themselves, tasks that are concurrently made ready are scheduled like `FIFO_Within_Priorities` tasks. As mentioned previously, the scheduler always takes precedence over its scheduled tasks.

For an application-scheduled task to become ready it is necessary that its scheduler performs a ready action on it. When the application task 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 task requests attachment to the scheduler
- when a task blocks or gets ready
- when a task changes its scheduling parameters
- when a task invokes the *yield* operation (i.e., `delay 0.0` operation)
- when a task explicitly invokes the scheduler
- when a task inherits or uninherits a priority, due to the use of a protected object, a rendezvous, or a task activation.
- when a task terminates
- when a task is aborted.

In addition to these events, the scheduler can also be notified about other system-generated events such as the expiration of an execution-time timer or of a group of timers, a timed notification programmed by the own scheduler for a particular task, a timeout, or an error notification.

All the above events are reported to the scheduler by the system invoking one of its primitive operations. These operations contain information relevant to the event (such as the task Id), and an `Actions` parameter that can be used by the scheduler to specify the scheduling actions that need to be performed by the system after the operation returns. As an example, the specification for two of these primitive operations is shown next:

```
procedure Ready
  (Sched : in out Scheduler;
   Tid : in  Ada.Task_Identification.Task_Id;
   Actions : in out Scheduling_Actions);
procedure Timeout
  (Sched   : in out Scheduler;
   Actions : in out Scheduling_Actions);
```

The `Actions` parameter is passed empty by the system, and the scheduler may add multiple scheduling actions of the following set:

- accept of reject a task
- make a task ready or suspend it
- program a timed notification associated to a given task
- program a timeout to occur; this timeout will be cancelled if some other event occurs first
- program a timed notification to occur if a task overruns its execution time budget; as with the timeout, this notification will be cancelled if some other event occurs first.

The system must ensure that the execution of the scheduling actions and the invocation of the scheduler primitive operations 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 protected objects. For single processor systems the sequential nature of the scheduler should be no problem if the system provides the necessary locking.

When an application-defined task 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 task. Rejection of a task causes an exception to be raised during the elaboration of the task.

This exception and the rest of the interface offered to both the scheduler and its scheduled tasks is contained in a new package, `Ada.Application_Scheduling`, that is described in the Appendix.

4.3. Application-scheduled tasks

Application-scheduled tasks may choose to be scheduled by an application-defined scheduler just by setting their priority to the appropriate value. In addition, they must specify their own scheduling parameters, that will be used by the scheduler to schedule that task contending with the other tasks attached to the same application scheduler. We set the scheduling parameters through the following pragma:

```
pragma Application_Defined_Sched_Parameters
   (My_Parameters'Access);
```

where `My_Parameters` is an object of the extended scheduling parameters type, containing specific values for each parameter. These parameters may be changed dynamically with the `Set_Parameters` call. The application scheduling parameters have no effects in tasks with other scheduling policies.

Some scheduling algorithms require the ability to explicitly invoke the scheduler from the application task, perhaps passing information to the scheduler and obtaining reply information back. The API for this explicit invocation is

```
procedure Invoke
   (Msg   : in Message_To_Scheduler'Class);
procedure Invoke
   (Msg   : in  Message_To_Scheduler'Class;
    Reply : out Reply_From_Scheduler'Class);
```

where the message passed to the scheduler is of an application-defined type extended from `Message_To_Scheduler`, and the reply message used in the second version of the call is extended from `Reply_From_Scheduler`.

5. Protected Objects

A new `Locking_Policy` is defined for the SRP, identified with the `Stack_Resource_Locking` name. This is a configuration pragma that affects the whole partition. Under this locking policy, the priority ceilings of the protected objects continue to exist, and are assigned via the usual `pragma Priority`. In addition, a new pragma may be used to assign a preemption level to each task and protected object:

```
pragma Preemption_Level (Level);
```

This pragma specifies the preemption level relative to tasks or protected objects of the same priority level. The priority of the task or the protected object always takes precedence over it. This is equivalent to defining the actual preemption level as:

$$actual = prio \times 2^n + level$$

where n is the number of bits used to represent the preemption level.

With this approach, under the SRP any task or protected object that does not specify the preemption level behaves exactly as with the `Ceiling_Locking` policy. This is an important property because we can reuse all of the concurrent software developed for that policy and make it work together with other tasks scheduled under an application-defined policy.

The system scheduler is in charge of verifying the SRP rule: a task cannot be made ready if its (actual) preemption level is strictly higher than the preemption level of the system, which is the maximum of the (actual) preemption levels of the protected objects currently locked. An error event is sent to the application scheduler if it tries to violate this rule.

Some scheduling policies such as the sporadic server (SS) of the constant bandwidth server (CBS) rely on execution time budgets to bound the amount of bandwidth given to a certain task. When the budget is exhausted, the scheduler makes a scheduling decision, usually making ready some other task as a result. The problem with these approaches is that the suspension of the current task due to the expiration of its execution time budget should not be allowed to preempt a critical section, i.e., should not be made while holding the lock on a protected object. One common approach to solve this problem is to only lock the protected object if there is enough budget to ensure that the protected operation will not be interrupted. This approach requires knowledge of the duration of each protected operation by the scheduler, and is difficult to implement in practice.

A second alternative is to defer the expiration of the budget until the protected operation finishes. This is the approach that we propose with our application scheduling API, because it is easier to implement. Of course this approach means that during the analysis the extra time that the budget expiration may be delayed has to be accounted for.

To implement the deferral of the budget expiration we propose a mechanism by which any of the scheduling events that may be notified to the application scheduler by the system may be deferred while a protected operation of a priority ceiling above some given value is in progress. A

mask of events to be deferred during protected operations is specified by the scheduler. This approach is quite efficient, because in the normal case in which critical sections are short there will be no events pending when the protected operation finishes, and the overhead is only increased by a simple check. If events are pending, they are reported to the scheduler task at the end of the protected operation. The API to set this mask is the following:

```
procedure Set_Protected_Event_Mask
  (Mask    : in Event_Mask;
   Ceiling : in System.Any_Priority);
procedure Get_Protected_Event_Mask
  (Mask    : out Event_Mask;
   Ceiling : out System.Any_Priority);
```

The mask may include the timed notification events, and this make them an extremely useful way of programming future actions for the scheduler that do not interfere with the synchronization through protected objects.

6. Example: EDF Scheduler

In this example we show the implementation of a basic EDF scheduler for periodic tasks. A real implementation of an EDF scheduler would need to take into account additional event arrival patterns (aperiodic, sporadic, etc.) and the possibility of task suspension. The purpose of the simplified example is just to illustrate how to use the API proposed in this paper.

The EDF scheduler would be programmed in a package with the following specification. It uses a priority queue that is written in package `Queues`; the priority is the deadline (earliest deadline first):

```
with Ada.Application_Scheduling, Queues,
   Ada.Task_Identification, Ada.Real_Time;
use Queues, Ada.Application_Scheduling,
   Ada.Real_Time;

package EDF_Scheduling is

   -- EDF scheduling specific parameters
   type EDF_Parameters is new
     Scheduling_Parameters with
   record
      Period            : Time_Span;
      Relative_Deadline : Time_Span;
   end record;

   -- Explicit Invocation Data
   type Message is new Message_To_Scheduler
     with null record;

   -- EDF scheduler

   type EDF_Scheduler is new Scheduler with
   record
      Ready_Queue  : Queue;
      Current_Task :
        Ada.Task_Identification.Task_Id;
   end record;

   procedure Init (Sched : out EDF_Scheduler);

   procedure New_Task
     (Sched : in out EDF_Scheduler;
      Tid : in Ada.Task_Identification.Task_Id;
      Actions : in out Scheduling_Actions);

   procedure Terminate_Task
     (Sched : in out EDF_Scheduler;
      Tid : in Ada.Task_Identification.Task_Id;
      Actions : in out Scheduling_Actions);

   procedure Explicit_Call
     (Sched : in out EDF_Scheduler;
      Tid : in Ada.Task_Identification.Task_Id;
      Msg : in Message_To_Scheduler'Class;
      Reply : out Reply_From_Scheduler'Class;
      Actions : in out Scheduling_Actions);

   procedure Task_Notification
     (Sched : in out EDF_Scheduler;
      Tid : in Ada.Task_Identification.Task_Id;
      Actions : in out Scheduling_Actions);

   procedure Error
     (Sched : in out EDF_Scheduler;
      Tid : in Ada.Task_Identification.Task_Id;
      Cause : in  Error_Cause;
      Actions : in out Scheduling_Actions);

   pragma Application_Scheduler
     (EDF_Scheduler, P, Stack_Resource_Policy);
end EDF_Scheduling;
```

The body of the package contains the scheduling operations:

```
with Ada.Task_Attributes;
package body EDF_Scheduling is

   use Ada.Task_Identification;

   type EDF_Attributes is record
      Period : Time_Span;
      Deadline : Time;
      Ready_Time : Time;
   end record;

   package EDF_Attr is new Ada.Task_Attributes
     (EDF_Attributes,
      (To_Time_Span(0.0),Time_Last,Time_Last));

   procedure Init (Sched : out EDF_Scheduler) is
   begin
      Make_Queue_Empty (Sched.Ready_Queue);
      Sched.Current_Task := Null_Task_Id;
   end Init;
   -- Do_Scheduling contains actions that are
   -- common to all scheduling operations
```

```
procedure Do_Scheduling
  (Sched : in out EDF_Scheduler;
   Actions : in out Scheduling_Actions) is
begin
   if Sched.Current_Task/=Sched.Current_Task
   then
      if Head_Of_Queue(Sched.Ready_Queue)/=
        Null_Task_Id
      then
         Add_Activate(Actions, Head_Of_Queue
                      (Sched.Ready_Queue));
         Sched.Current_Task :=
           Head_Of_Queue(Sched.Ready_Queue);
      end if;
      if Sched.Current_Task/=Null_Task_Id
      then
         Add_Suspend (Actions,Current_Task);
      end if;
   end if;
end Do_Scheduling;

procedure New_Task
  (Sched   : in out EDF_Scheduler;
   Tid     : in     Task_Id;
   Actions : in out Scheduling_Actions)
is
   Param : EDF_Parameters;
   Current_Time : Ada.Real_Time.Time:=
     Ada.Real_Time.Clock;
   Task_Data : EDF_Attributes;
begin
   Get_Parameters (Tid, Param);
   -- Check if task can be accepted
   if Schedulability_Test_OK (Param) then
      Add_Accept (Actions, Tid);
      -- Calculate task data relevant to EDF
      Task_Data.Period :=  Param.Period;
      Task_Data.Ready_Time:=
        Current_Time;
      Task_Data.Deadline := Current_Time +
        Param.Relative_Deadline;
      EDF_Attr.Set_Value(Task_Data);
      Add_To_Queue (Sched.Ready_Queue, Tid);
   else
      Add_Reject (Actions, Tid);
   end if;
   Do_Scheduling (Sched,Actions);
end New_Task;

procedure Terminate_Task
  (Sched   : in out EDF_Scheduler;
   Tid     : in     Task_Id;
   Actions : in out Scheduling_Actions) is
begin
   -- Just remove it from the list of tasks
   Remove_From_Queue (Sched.Ready_Queue, Tid);
   Do_Scheduling(Sched,Actions);
end Terminate_Task;

procedure Task_Notification
  (Sched   : in out EDF_Scheduler;
   Tid     : in     Task_Id;
   Actions : in out Scheduling_Actions) is
begin
   -- Ready time reached: add to ready queue
   Add_To_Queue (Sched.Ready_Queue, Tid);
   Do_Scheduling(Sched,Actions);
end Task_Notification;

procedure Explicit_Call
  (Sched   : in out EDF_Scheduler;
  Tid     : in     Task_Id;
  Msg     : in     Message_To_Scheduler'Class;
  Reply   : out    Reply_From_Scheduler'Class;
  Actions : in out Scheduling_Actions)
is
   Task_Data : EDF_Attributes;
begin
   -- Task finishes its current job
   Task_Data:=EDF_Attr.Value;
   -- Calculate new deadline and ready time
   Task_Data.Deadline:=
     Task_Data.Deadline+Task_Data.Period;
   Task_Data.Ready_Time:=
     Task_Data.Ready_Time+Task_Data.Period;
   EDF_Attr.Set_Value(Task_Data);
   --  Remove task from ready queue
   Remove_From_Queue (Sched.Ready_Queue, Tid);
   --  Program task notif. by ready time
   Add_Timed_Task_Notification
     (Actions, Tid, Task_Data.Ready_Time);
   Do_Scheduling(Sched,Actions);
end Explicit_Call;

procedure Error
  (Sched   : in out EDF_Scheduler;
   Tid     : in Task_Id;
   Cause   : in  Error_Cause;
   Actions : in out Scheduling_Actions) is
begin
   Report_Error(Cause); -- for debugging
end Error;

end EDF_Scheduling;
```

The application tasks are created like in the following example. The Specification of the task with the package is below:

```
with EDF_Scheduling, Ada.Real_Time;
use EDF_Scheduling, Ada.Real_Time;
package EDF_Task is

   My_Parameters : aliased EDF_Parameters :=
     (Period => To_Time_Span(0.001_500),
      Relative_Deadline =>
         To_Time_Span(0.001_000));

   task EDF_Scheduled_Task is
      pragma Application_Defined_Sched_Parameters
        (My_Parameters'access);
      pragma Priority (10);
      pragma Preemption_Level (34);
   end EDF_Scheduled_Task;
end EDF_Task;
```

And the body of the package appears next. It can be seen that the task notifies the scheduler that an instance has been completed with an explicit invocation:

```
with Ada.Application_Scheduling;
package body EDF_Task is

   -- Scheduled task body
   task body EDF_Scheduled_Task is
      Null_Message : Message;
   begin
      loop
         --Useful work
         -- (maybe using protected objects)
         Ada.Application_Scheduling.Invoke
            (Null_Message);
      end loop;
   end EDF_Scheduled_Task;

end EDF_Task;
```

7. Conclusions

In this paper we have defined a new API for application-defined scheduling. With this API, several application-defined schedulers, implemented by extending a tagged type, can coexist in the system in a predictable way together with their scheduled tasks, and with regular fixed priority tasks directly scheduled by the run-time system. Task synchronization through protected objects is also considered by adding the Stack Resource Policy, which can be used in a large variety of fixed and dynamic priority scheduling policies. Interactions between the application-defined schedulers and the protected objects are facilitated by being able to defer the notification of scheduling events to the application scheduler.

A previous version of this API has been implemented in MaRTE OS, which is a free software implementation of the POSIX minimal real-time operating system, intended for small embedded systems [2]. It is written in Ada but provides both the POSIX Ada and the C interfaces. An implementation of the API presented in this paper is proposed as future work.

Acknowledgments

This paper contains many ideas that were contributions of the participants at the IRTAW-2003 workshop which took place in Viana do Castelo, in September 2003. The authors want to thank all those who contributed.

References

[1] 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

[2] 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.

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

[4] IEEE Std 1003.1-2001. *Information Technology -Portable Operating System Interface (POSIX)*. Institute of Electrical and electronic Engineers.

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

[6] 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.

[7] B. Ford and S. Susarla, “CPU Inheritance Scheduling”. *Proceedings of OSD*I, October 1996.

[8] 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.

[9] E.W. Giering and T.P. Baker (1994). The GNU Ada Runtime Library (GNARL): Design and Implementation. *Wadas’94 Proceedings*.

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

[11]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.

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

[13]Baker T.P., “Stack-Based Scheduling of Realtime Processes”, Journal of Real-Time Systems, Volume 3, Issue 1 (March 1991), pp. 67–99.

[14]Paolo Gai and Giorgio Buttazzo, "Mutual Exclusion in Operating Systems with Application-Defined Scheduling", Workshop on Advanced Real-Time Operating System Services, Porto, Portugal, July 2003.

[15]Mario Aldea Rivas and Michael González Harbour. "A POSIX-Ada Interface for Application-Defined Scheduling". International Conference on Reliable Software Technologies, Ada-Europe 2002, Vienna, Austria, in Lecture Notes in Computer Science No. 2361, pp.136-150, June 2002.

[16]Mario Aldea Rivas and Michael González Harbour. “POSIX-Compatible Application-Defined Scheduling in MaRTE OS” Proceedings of 14th Euromicro Conference on Real-Time

Systems, Vienna, Austria, IEEE Computer Society Press, pp. 67-75, June 2002

[17]A. Burns, M. González Harbour and A.J. Wellings. “A Round Robin Scheduling Policy for Ada”. Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe-2003, Tolouse, France, in Lecture Notes in Computer Science, LNCS 2655, June, 2003, ISBN 3-540-40376-0.

Appendix

The following is the proposed specification of the application-scheduling package:

```
with Ada.Task_Identification, Ada.Real_Time,
  Ada.Real_Time.Execution_Time,
  Ada.Real_Time.Execution_Time.Group_Timers,
  System;
package Ada.Application_Scheduling is

   ---------------------------------------------
   -- Expiration Execution-time Timers
   ---------------------------------------------

   protected type Timer_Expiration is
      procedure Announce
        (T : in out Ada.Real_Time.
         Execution_Time.Timer);
      procedure Notify
        (T : in out Ada. Real_Time.Execution_Time.
                    Group_Timers.Group_Timer);
   private
      -- not specified by the language
   end Timer_Expiration;

   type Timer_Expiration_Ref is access
     Timer_Expiration;

   --------------------
   --Scheduling Actions
   --------------------

   type Scheduling_Actions is private;

   procedure Add_Accept
     (Sched_Actions : in out Scheduling_Actions;
      Tid : in Ada.Task_Identification.Task_Id);
   procedure Add_Reject
     (Sched_Actions : in out Scheduling_Actions;
      Tid : in Ada.Task_Identification.Task_Id);
   procedure Add_Ready
     (Sched_Actions : in out Scheduling_Actions;
      Tid : in Ada.Task_Identification.Task_Id);
   procedure Add_Suspend
     (Sched_Actions : in out Scheduling_Actions;
      Tid : in Ada.Task_Identification.Task_Id);
   procedure Add_Timeout
     (Sched_Actions : in out Scheduling_Actions;
      At_Time : in Ada.Real_Time.Time);
   procedure Add_Timed_Task_Notification
     (Sched_Actions : in out Scheduling_Actions;
      Tid : in Ada.Task_Identification.Task_Id;
      At_Time : in Ada.Real_Time.Time);
   procedure Add_Timer_Expiration
     (Sched_Actions : in out Scheduling_Actions;
      Handler : in Timer_Expiration_Ref);

   --------------------------------
   -- Explicit Scheduler Invocation
   --------------------------------

   type Message_To_Scheduler is abstract tagged
     null record;
   type Reply_From_Scheduler is abstract tagged
     null record;

   procedure Invoke
     (Msg   : in Message_To_Scheduler'Class);
   -- callable from application-scheduled task
   procedure Invoke
     (Msg   : in  Message_To_Scheduler'Class;
      Reply : out Reply_From_Scheduler'Class);
   -- callable from application-scheduled task

   Task_Not_Application_Scheduled : exception;
   -- may be raised by Invoke

   --------------------
   -- Scheduler type
   --------------------

   type Scheduler is abstract tagged private;
   type Error_Cause is
    (SRP_Rule_Violation, Invalid_Action_For_Task);

   procedure Init (Sched : out Scheduler)
      is abstract;
   procedure New_Task
     (Sched : in out Scheduler;
      Tid : in  Ada.Task_Identification.Task_Id;
      Actions : in out Scheduling_Actions);
   procedure Terminate_Task
     (Sched : in out Scheduler;
      Tid : in  Ada.Task_Identification.Task_Id;
      Actions : in out Scheduling_Actions);
   procedure Ready
     (Sched : in out Scheduler;
      Tid : in  Ada.Task_Identification.Task_Id;
      Actions : in out Scheduling_Actions);
   procedure Block
     (Sched : in out Scheduler;
      Tid : in  Ada.Task_Identification.Task_Id;
      Actions : in out Scheduling_Actions);
   procedure Yield
     (Sched : in out Scheduler;
      Tid : in  Ada.Task_Identification.Task_Id;
      Actions : in out Scheduling_Actions);
   procedure Abort_Task
     (Sched : in out Scheduler;
      Tid : in  Ada.Task_Identification.Task_Id;
      Actions : in out Scheduling_Actions);
   procedure Change_Sched_Param
     (Sched : in out Scheduler;
      Tid : in  Ada.Task_Identification.Task_Id;
      Actions : in out Scheduling_Actions);
```

```
procedure Explicit_Call
  (Sched : in out Scheduler;
   Tid : in  Ada.Task_Identification.Task_Id;
   Msg : in Message_To_Scheduler'Class;
   Reply : out Reply_From_Scheduler'Class;
   Actions : in out Scheduling_Actions);
procedure Task_Notification
  (Sched : in out Scheduler;
   Tid : in  Ada.Task_Identification.Task_Id;
   Actions : in out Scheduling_Actions);
procedure Timeout
  (Sched : in out Scheduler;
   Actions : in out Scheduling_Actions);
procedure Execution_Timer_Expiration
  (Sched : in out Scheduler;
   Expired_Timer : in out Ada.Real_Time.
       Execution_Time.Timer;
   Actions : in out Scheduling_Actions);
procedure Group_Timer_Expiration
  (Sched : in out Scheduler;
   Expired_Group : in out Ada.Real_Time.
      Execution_Time.Group_Timers.Group_Timer;
   Actions : in out Scheduling_Actions);
procedure Priority_Inherit
  (Sched : in out Scheduler;
   Tid : in  Ada.Task_Identification.Task_Id;
  From_Tid: in Ada.Task_Identification.Task_Id;
   Inherited_Prio : in  System.Any_Priority;
   Actions : in out Scheduling_Actions);
procedure Priority_Uninherit
  (Sched : in out Scheduler;
   Tid : in  Ada.Task_Identification.Task_Id;
  From_Tid: in Ada.Task_Identification.Task_Id;
   Uninherited_Prio : in  System.Any_Priority;
   Actions : in out Scheduling_Actions);
procedure Error
  (Sched : in out Scheduler;
   Tid : in Ada.Task_Identification.Task_Id;
   Cause : in  Error_Cause;
   Actions : in out Scheduling_Actions)
   is abstract;
-- Non-abstract operations have a null body

--------------
-- Event masks
--------------

type Event_Code is
  (New_Task,  Terminate_Task, Ready, Block,
   Yield, Change_Sched_Param, Explicit_Call,
   Task_Notification, Timeout,
   Execution_Timer_Expiration,
   Priority_Inherit, Priority_Uninherit);

type Event_Mask is private;

procedure Fill     (Mask : in out Event_Mask);
procedure Empty    (Mask : in out Event_Mask);
procedure Add      (Event : in Event_Code;
                    Mask : in out Event_Mask);
procedure Delete   (Event : in Event_Code;
                    Mask : in out Event_Mask);
function Is_Member (Event : in Event_Code;
                    Mask : in Event_Mask)
                   return Boolean;

   -- The event mask filters out the scheduling
   -- events that are not needed by the scheduler
   procedure Set_Event_Mask
     (Mask : in Event_Mask);
   procedure Get_Event_Mask
     (Mask : out Event_Mask);

   -- The protected event mask delays the
   -- specified scheduling events when generated
   -- while the system ceiling is larger than or
   -- equal to the specified Ceiling
   procedure Set_Protected_Event_Mask
     (Mask    : in Event_Mask;
      Ceiling : in System.Any_Priority);
   procedure Get_Protected_Event_Mask
     (Mask    : out Event_Mask;
      Ceiling : out System.Any_Priority);

   ------------------------------------
   --Change Task Scheduling Parameters
   ------------------------------------

   type Scheduling_Parameters is
     abstract tagged null record;
   type Sched_Params_Ref is access
     Scheduling_Parameters'Class;

   procedure Set_Parameters
     (Tid : in Ada.Task_Identification.Task_Id;
      Param : in Scheduling_Parameters'Class);
   -- callable from any task
   -- This is a dispatching point

   procedure Get_Parameters
     (Tid : in  Ada.Task_Identification.Task_Id;
      Param : out Scheduling_Parameters'Class);
   -- callable from any task

   -- Scheduling Policy is changed by
   -- setting the priority

private

   -- not defined by the language

end Ada.Application_Scheduling;
```