Extending Ada's Real-Time Systems Annex with the POSIX Scheduling Services

By: Mario Aldea Rivas and Michael González Harbour
Departamento de Electrónica y Computadores
Universidad de Cantabria
39005- Santander, SPAIN
{aldeam, mgh}@unican.es

Abstract[1]

In this paper we propose extending the scheduling model of the Ada 95 Real-Time Systems Annex with the services specified in the Real-Time POSIX standard. These services include a round robin within priorities scheduling policy, a sporadic server scheduling policy, and execution time clocks and timers. With these services the Ada Real-Time Annex will enable addressing a larger number of application requirements.

Keywords: *Ada, Real-Time, Scheduling, POSIX, Execution-time limits*

1. Introduction

The Real-Time Annex in Ada 95 [1] defines an extensible mechanism for specifying scheduling policies, and defines only one scheduling policy: `FIFO_Within_Priorities`. Although this policy is an excellent choice for real-time systems, there are application requirements that cannot be fully accomplished with this policy only:

- Many applications have a mixture of real-time and non real-time activities. The natural way of scheduling non real-time activities is by time sharing the processor, like in most general purpose operating systems. A fixed priority scheduling policy can be used, but it requires long computations to be broken into segments, and yield operations (`delay 0.0`) to be inserted at the segment boundaries to allow other non real-time tasks to make progress. This may be difficult when using standard libraries that have been designed for systems with a time sharing scheduling.
- Many applications have aperiodic activities activated through events whose arrival pattern is of unbounded nature. There is often a need to accomplish high average response times for some or all of these aperiodic activities, and at the same time guarantee that the hard real-time requirements of periodic or sporadic activities are still met. Several flexible scheduling policies exist for this purpose, such as the slack stealing algorithm [9][8], dual priorities [7], the sporadic server [6] or the deferrable server [5]. Although these policies can be implemented at the application level in a system with fixed priority scheduling [10][11], these implementations are usually inefficient and do not exploit the possibilities that a full kernel-level implementation would have.

In addition, the current Real-Time Annex does not support the measurement and limitation (budgeting) of task execution times. Real-time analysis techniques are always based on the assumption that the application developer can accurately measure the worst-case execution time (WCET) of each task. This measurement is always very difficult, because, with effects like cache misses, pipelined and superscalar processor architectures, etc., the execution time is highly unpredictable. There are models that allow calculation of WCET's for some architectures, but they are generally very complex and not widely available for all architectures.

In hard real-time systems is essential to monitor the execution times of all tasks and detect situations in which the estimated WCET is exceeded. This detection was usually available in systems scheduled with cyclic executives, because the periodic nature of its cycle allowed checking that all initiated work had been completed at each cycle. In event-driven concurrent systems the same capability should be available, and this can be accomplished with execution time clocks and timers.

1. This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grant TIC99-1043-C03-03

In recognition of all these application requirements, the Real-Time extensions to POSIX [3] have recently incorporated support for them [2]. Real-Time POSIX supports a “round robin within priorities” scheduling policy that is adequate for scheduling non real-time tasks in conjunction with real-time tasks. It also supports the sporadic server scheduling policy for efficiently scheduling aperiodic activities. In addition, it provides execution time clocks and timers that allow an application to monitor the consumption of execution time by its tasks, and to set limits for this consumption.

In this paper we propose that the next revision of the Ada language supports a scheduling model similar to the Real-Time POSIX model, so that the requirements of a larger set of applications can be met.

It could be argued that given that the POSIX standard already defines an appropriate scheduling model it would not be necessary to have the same in the Ada language standard, because the POSIX services can be accessed through the appropriate bindings. This is true for Ada platforms built on top of POSIX OS implementations, but not for bare-machine implementations, like the ones used in embedded systems, avionics, etc. For portability purposes it would be interesting to have this scheduling model supported in a homogeneous way for all the implementations that choose to support it.

It could also be argued that scheduling policies different than those standardized in POSIX could have better performance or less overhead. For example, a deferrable server might be preferred over the sporadic server. But the fact that the latter is standardized in POSIX makes it more probable that the underlaying OS supports the policy, and therefore that it can also be supported in the Ada runtime system.

A proposal for application-defined scheduling policies is currently under study for consideration in the Real-Time POSIX Standard, but it is still too preliminary to be discussed for its standardization also in the Ada Real-Time Annex.

The paper is organized as follows: Section 2 discusses the proposal for a round robin scheduling policy. Section 3 discusses the sporadic server proposal. Section 4 has a proposal for adding execution time clocks and timers. Finally, Section 5 gives our conclusions.

2. Round Robin Scheduling Policy

The POSIX scheduling model is a fixed-priority model in which there are three compatible scheduling policies defined. These policies can be set on a task by task basis, because the effects of mixing them are well defined. For example, when a round robin task is running, its execution time is limited to its time quantum. After the quantum is elapsed, the task is sent to the tail of the ready queue for its priority. If a FIFO within priorities task now comes into execution, it runs until completion (possibly preempted by higher priority tasks during its execution). It is the responsibility of the application developer to make sure that no mixture of round robin and FIFO tasks is made at the same priority level, if the round robin semantics is to be preserved.

In Ada 95 there is only one pragma for setting the task dispatching policy. It is a configuration pragma, that affects the behaviour of all tasks in the partition. In order to be able to have a mixed task dispatcher with both FIFO and round robin tasks, it is necessary to define a new partition-wide task dispatching policy, and a new pragma to specify the task dispatching policy individually for each task.

The new partition-wide task dispatching policy could be described in the `Task_Dispatching_Policy` configuration pragma as:

```
pragma Task_Dispatching_Policy
   (Fixed_Priorities);
```

The `Fixed_Priorities` dispatching policy is the same as the `FIFO_Within_Priorities` policy except that a new pragma is allowed for specifying the task-specific dispatching policy. This pragma would be:

```
pragma Individual_Task_Dispatching_Policy
   (policy);
```

This pragma would be allowed only immediately within a task definition or the declarative part of a subprogram body (for the main task). For a partition with a `FIFO_Within_Priorities` policy, the only value allowed for this pragma would be `FIFO_Within_Priorities`. For a partition with the new `Fixed_Priorities` policy, the values allowed for this pragma would be `FIFO_Within_Priorities`, or `Round_Robin_Within_Priorities` (or `Sporadic_Server`, as we will see in Section 3). The default value would be `FIFO_Within_Priorities`. The locking policy associated with the `Fixed_Priorities` partition task dispatching policy would be `Ceiling_Locking`.

The rules for the `Round_Robin_Within_Priorities` policy are the same as for `FIFO_Within_Priorities`, with the additional condition that when the implementation detects that a running task has been executing for a time interval called the round-robin quantum, it becomes the tail

of its ready queue and the head of that queue is removed and made a running task. A task under this policy that is preempted and subsequently resumes execution as a running task continues to use the unexpired portion of its round-robin interval.

This new requirement can be envisioned as a new execution resource that exists for each processor. This resource, called the round-robin resource must be acquired by a task scheduled with the `Round_Robin_Within_Priorities` policy before it can execute. It is acquired when the task is made the running task, with an interval value equal to the round robin quantum. The interval value is consumed as the task executes. While the task is preempted by a higher priority task the remaining time interval does not get consumed. When the time interval has been totally consumed, the task looses the round robin resource, and goes to the tail of the ready queue for its priority; in this case, the task at the head of the ready queue is made a running task; if its policy is `Round_Robin_Within_Priorities`, the round robin resource is granted to it, with its time interval reset to the initial round robin quantum.

The round robin quantum is implementation defined. Its value can be read via the following constant defined in package `System`:

```
Round_Robin_Quantum : constant :=
   implementation-defined-real-number;
```

Because a task might consume its round robin quantum while inside a protected operation, it is no longer true that the task can only be preempted by tasks whose active priorities are higher than the ceiling priority of the protected object. A task with the same active priority as the ceiling could cause a preemption-like effect when the quantum is consumed. This implies that the implementation must not just rely on the priority protection mechanism to implement the protected object lock in single processor systems, and should implement a real lock. Alternatively, an implementation permission could be given to defer the expiration of the round robin quantum until the end of the protected operation. In any case, for portability with POSIX-based implementations, this behavior should not be required.

3. Sporadic Server Scheduling Policy

The POSIX.1d sporadic server scheduling policy is compatible with the FIFO and round robin policies. It assigns two priority levels to each task scheduled with this policy (a normal priority and a low priority), a queue of pending replenishment operations, and a value of execution capacity. The sporadic server policy could be defined for a task with the task-specific pragma defined in Section 2 called `Individual_Task_Dispatching_Policy`, with the value `Sporadic_Server`. For tasks with this policy, a new pragma would be supported:

```
pragma Sporadic_Server_Parameters
   (Low_Priority => value,
    Replenishment_Period => value,
    Max_Pending_Replenishments => value,
    Initial_Budget => value);
```

The meaning of the arguments of this pragma is the following:

- `Low_Priority`: its expected type is `System.Priority`. It specifies the low scheduling priority for the task scheduler under a sporadic server.
- `Replenishment_Period`: its expected type is `Ada.Real_Time.Time_Span`. It specifies the replenishment period for the sporadic server.
- `Max_Pending_Replenishments`: its expected type is a positive integer value within one and an implementation-defined maximum. It specifies the maximum size of the queue of pending replenishments. As in POSIX, all implementations shall support a maximum number of at least four.
- `Initial_Budget`: its expected type is `Ada.Real_Time.Time_Span`. It specifies the initial value for the execution capacity.

The *normal* priority level of the task scheduled under a sporadic server is the one specified via pragma `Priority`, and possibly changed via `Ada.Dynamic_Priorities.Set_Priority`.

The sporadic server policy is based primarily on two parameters: the replenishment period and the available execution capacity. The replenishment period is given by the `Replenishment_Period` argument. The available execution capacity is initialized to the value given by the `Initial_Budget` argument. The sporadic server policy is identical to the `FIFO_Within_Priorities` policy with some additional conditions that cause the task's base priority to be switched between normal priority and the `Low_Priority`.

The priority assigned to a task using the sporadic server scheduling policy is determined in the following manner: if the available execution capacity is greater than zero and the number of pending replenishment operations is strictly less than `Max_Pending_Replenishments`, the base priority of the task is set to the priority specified via pragma `Priority`. Otherwise, the base priority shall be `Low_Priority`. When active, the task shall belong to the

ready queue corresponding to its base priority level, according to the mentioned priority assignment. The modification of the available execution capacity and, consequently of the assigned base priority, is done as follows:

(1) When the task at the head of the ready queue for its normal priority becomes a running task, its execution time shall be limited to at most its available execution capacity, plus the resolution of the execution time clock used for this scheduling policy. This resolution shall be implementation defined.

(2) Each time the task is inserted at the tail of the list associated with its normal priority (either because as a blocked task it became ready with a base priority equal to the normal priority or because a replenishment operation was performed) the time at which this operation is done is posted as the *activation_time.*

(3) When the running task with base priority equal to its normal priority becomes a preempted task, it becomes the head of the ready queue for its priority, and the execution time consumed is subtracted from the available execution capacity. If the available execution capacity would become negative by this operation, it shall be set to zero.

(4) When the running task with assigned priority equal to its normal priority becomes a blocked task, the execution time consumed is subtracted from the available execution capacity, and a replenishment operation is scheduled, as described below. If the available execution capacity would become negative by this operation, it shall be set to zero.

(5) When the running task with assigned priority equal to its normal priority reaches the limit imposed on its execution time, it becomes the tail of the ready queue for `Low_Priority`, the execution time consumed is subtracted from the available execution capacity (which becomes zero), and a replenishment operation is scheduled, as described below.

(6) Each time a replenishment operation is scheduled, the amount of execution capacity to be replenished, *replenish_amount*, is set equal to the execution time consumed by the task since the *activation_time*. The replenishment is scheduled to occur at *activation_time* plus `Replenishment_Period`. If the scheduled time obtained is before the current time, the replenishment operation is carried out immediately. Notice that there may be several replenishment operations pending at the same time, each of which will be serviced at its respective scheduled time. Notice also that with the rules defined for this policy, the number of replenishment operations simultaneously pending for a given task that is scheduled under the sporadic server policy shall not be greater than `Max_Pending_Replenishments`.

(7) A replenishment operation consists of adding the corresponding *replenish_amount* to the available execution capacity at the scheduled time. If as a consequence of this operation the execution capacity would become larger than `Initial_Budget`, it shall be rounded down to a value equal to `Initial_Budget`. Additionally, if the task was ready or running, and with base priority equal to `Low_Priority`, then it becomes the tail of the ready queue for its normal priority.

4. Execution-Time Clocks and Timers

4.1. The POSIX Model

The execution time clocks and timers interface defined in the proposed standard POSIX.1d [2] is based on the POSIX.1b [3] clocks and timers interface used for normal real time clocks. The new interface creates two functions to access the execution time clock identifier of the desired process or thread, respectively: *clock_getcpuclockid*() and *pthread_getcpuclockid*().

An execution time clock “id” can be used to read or set the time using the same functions *clock_gettime*() and *clock_settime*() that are used for the standard `CLOCK_REALTIME` clock, which measures real time. In addition, timers may be created using either the `CLOCK_REALTIME` or a CPU-time clock as their time base. A POSIX timer is a logical object that measures time based upon a specified time base. The timer may be armed to expire when an absolute time is reached, or when a relative interval elapses. When the expiration time has been reached, a signal is sent to the process, to notify about the timer expiration. The timer can be rearmed or disarmed at any time. In addition, it is possible to program the timer so that it expires periodically, after the first expiration.

If a timer is created using a CPU-time clock of a particular thread, and a relative expiration time is given, it can be used to notify that a certain budget of execution time has elapsed, for that thread. If the timer is armed each time a thread is activated, and the relative expiration time is set to the thread’s estimated worst-case execution time (plus some small amount to take into account the limited resolution and precision of the CPU-time clock), then the timer will only expire if the thread suffers an execution time overrun.

4.2. Proposed Ada Specification

The requirements for the Ada specification are the following:

- Should be implementable on top of POSIX CPU-time clocks and timers.
- Each task should have a CPU-time clock.
- It should be possible to read the value of a CPU-time clock. Setting the value is not considered necessary, because usually time differences between two events are calculated. Besides, setting the clock has implementation difficulties, when timers are running based on a specified clock.
- It should be possible to create a timer based on a CPU time clock.
- A CPU-time timer should have operations to arm it, disarm it, or read its value. The timer would be armed in a one-time shot. Given the usage schemes shown below, periodic timers do not seem to be necessary
- It should be possible to determine if a timer has expired, or to wait for a timer expiration with an entry call. In this way, selective abort, conditional entry calls, and other language constructs could be used based upon the timer expiration condition.

The interface to the execution time clocks and timers can be specified in a standard package, called `Ada.Real_Time.Execution_Time`, which would follow the specification shown below. The implementation of this package would encapsulate the internal aspects of the use of the POSIX services for clocks and timers, including the use of signals to notify the occurrence of timer expirations caused by an execution time overrun. Alternatively, in a bare-machine implementation, it would provide access to a full implementation of the CPU time clocks and timers.

```
with Ada.Task_Identification;
package Ada.Real_Time.Execution_Time is
   Disarmed : exception;
      -- raised by Timer.Time_Exceeded,
      -- Timer.Time_Was_exceeded, and
      -- Timer.Time_Remaining
   type Clock_ID is private;
   type CPU_Time is private;
   Time_Unit : constant := impl.def.real-num;
   function Clock_Of
      (T : Ada.Task_Identification.Task_ID)
       return Clock_ID;
   function Clock
      (C : Clock_ID)
      return CPU_Time;
   function "+"
      (Left : CPU_Time; Right : Time_Span)
      return CPU_Time;
   function "+"
      (Left : Time_Span; Right : CPU_Time)
      return CPU_Time;
   function "-"
      (Left : CPU_Time; Right : Time_Span)
      return CPU_Time;
   function "-"
      (Left : CPU_Time; Right : CPU_Time)
      return Time_Span;
   function "<"
      (Left, Right : CPU_Time)
      return Boolean;
   function "<="
      (Left, Right : CPU_Time)
      return Boolean;
   function ">"
      (Left, Right : CPU_Time)
      return Boolean;
   function ">="
      (Left, Right : CPU_Time)
      return Boolean;
   protected type Timer is
      procedure Initialize
         (C : Clock_ID);
      procedure Finalize;
      procedure Arm
         (Interval : Time_Span);
      procedure Disarm;
      entry Time_Exceeded;
      function Time_Was_Exceeded
         return Boolean;
      function Time_Remaining
         return Time_Span;
   private
      ...
   end Timer;
private
   ...
end Ada.Real_Time.Execution_Time;
```

The central part of package `Execution_Time` is a protected object called `Timer`. This protected object has visible operations for the application tasks to initialize or finalize a CPU-time timer, to arm or disarm a timer, and to determine whether a timer has expired or not (`Time_Was_Exceeded`). In addition, `Timer` has an entry (`Time_Exceeded`) that can be used by application tasks to block until an execution time overrun is detected, or as an event that triggers the abortion of the instructions of a select statement with an abortable part. The type `CPU_Time` represents absolute values of CPU time, relative to an arbitrary start time. Operations are provided to operate between values of this type and of the type `Ada.Real_Time.Time_Span`.

A more detailed description of the operations related to the CPU clocks and timers follows:

- `Clock_Of`: Returns the clock identifier of the specified task.
- `Clock`: Returns the value of the execution time clock specified by `C`.
- `Timer.Initialize`: Allocates and initializes the resources required to operate a CPU-time timer based on the execution time clock specified by `C`. If the operation would exceed the limit of the maximum number of timers in the system, the `Storage_Error` exception is raised. The timer is initialized in the disarmed state.
- `Timer.Finalize`: Deallocates the system resources used by the timer. No other calls to the timer operations of the associated timer may be made, except another `Initialize` call. `Constraint_Error` will be raised if such attempt is detected.
- `Timer.Arm`: The timer is loaded with the value specified by `Interval` and set to the armed state. In this state the timer counts execution time and, when the CPU clock associated with the timer measures the passage of `Interval`, it is said to have expired. If the timer was already armed, it is rearmed.
- `Timer.Disarm`: The timer is set to the disarmed state. In this state no timer expirations occur.
- `Timer.Time_Exceeded`: If the timer is in the armed state but has not yet expired, the calling task is suspended. The entry is allowed to complete when the timer is in the armed state and has expired. If the timer is in the disarmed state, the `Disarmed` exception is raised.
- `Timer.Time_Was_Exceeded`: If the timer is in the armed state, the function returns `True` if the timer has expired, and `False` otherwise. If the timer is in the disarmed state, the `Disarmed` exception is raised.
- `Timer.Time_Remaining`: If the timer is in the armed state, the function returns the CPU time interval that remains until the timer will expire, or a value representing zero if the timer has expired. If the timer is in the disarmed state, the `Disarmed` exception is raised.

There are different ways of using the services offered in package `Execution_Time`, depending on the application requirements [11]:

- *Handled*: When an execution time overrun is detected, an error handling operation is performed, but the task is allowed to complete its execution. Used for testing or for highly critical tasks.
- *Stopped*: When an execution time overrun is detected, the associated task execution is stopped, to allow lower priority tasks to execute within their deadlines. The whole instance of the stopped task is aborted and is never repeated. The task itself waits until its next activation and then proceeds normally.
- *Imprecise*: This usage scheme corresponds to the case in which the task is designed using the imprecise computation model [12], in which the task has a mandatory part (generally short and for which it is easier to estimate a worst-case execution time), and an optional part that refines the calculations made by the task. Since the worst-case execution time of this optional part is usually more difficult to estimate, this part will be aborted if an execution time overrun is detected.
- *Lowered*: This usage scheme can be used to limit the effects of an execution time overrun of a particular task, on lower priority tasks, when asynchronous select statements are not allowed or are not available for an application task. In this case, when the overrun is detected, the priority of the task is lowered to a background level, lower than the priorities of all real-time tasks. When the task that overrun its execution time has the opportunity to finish its execution, it can determine that it overrun by invoking `Time_Was_Exceeded`, and then it can take a corrective action or report the error.

5. Conclusions

The scheduling model defined in the real-time POSIX standard is richer than the Ada 95 model and, thus, is able to meet the requirements of a larger set of real-time applications. Currently real-time POSIX supports two scheduling policies that are not supported in Ada: round robin within priorities and the sporadic server. The first is very useful in applications with a mixture of real-time and non real-time tasks, which are very common in practice. The second policy is very useful for processing unbounded aperiodic events with short response times, while guaranteeing a given bandwidth for preserving the schedulability of lower priority tasks.

In addition, real time POSIX supports execution time clocks and timers, that allow assigning execution time budgets to each task, thus making it possible to rely on the results of schedulability analysis.

In this paper we have made proposals to include all these policies and scheduling services in the next revision of the Real-Time Systems Annex.

References

[1] S. Tucker Taft, and R.A. Duff (Eds.) "*Ada 95 Reference Manual. Language and Standard Libraries*". International Standard ISO/IEC 8652:1995(E), in Lecture Notes on Computer Science, Vol. 1246, Springer, 1997.

[2] IEEE Standard 1003.1d:1999, "Standard for Information Technology -Portable Operating System Interface (POSIX)- Part d: Additional Realtime Extensions". The Institute of Electrical and Electronics Engineers, 2000.

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

[4] M. González Harbour, J.J. Gutiérrez García, and J.C Palencia Gutiérrez. "Implementing Application-Level Sporadic Server Schedulers in Ada 95". Proceedings of the 1997 Ada-Europe International Conference on Reliable Software Technologies, in Lecture Notes in Computer Science, Vol. 1251, Springer, June 1997.

[5] J. P. Lehoczky, L. Sha and J. K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments", Proceedings IEEE Real-Time System Symposium, San Jose, California, pp. 261-270 (1987).

[6] L. Sha, B. Sprunt and J. P. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems", The Journal of Real-Time Systems 1, pp. 27-69 (1989).

[7] Robert Davis and Andy Wellings "Dual Priority Scheduling" Proceedings IEEE Real-Time System Symposium, pp. 100-109, 1995.

[8] R. I. Davis, K. W. Tindell and A. Bums, "Scheduling Slack Time in Fixed Priority Pre-emptive Systems", Proceedings IEEE Real-Time Systems Syposium, pp. 222- 231 (December 1993).

[9] John P. Lehoczky and Sandra Ramos-Thuel. "An optimal algorithm for scheduling soft aperiodic tasks in fixed-priority preemptive systems". In Real-Time Systems Symposium, pages 110-123, Dec.1992

[10] A. Espinosa, V. Julián, C. Carrascosa, A. Terrasa, and A. García-Fornes. "Programming Hard Real-Time Systems with Optional Components in Ada". Proceedings of the 1998 Ada Europe International Conference on Reliable Software Technologies, Uppsala, Sweden; in Lecture Notes on Computer Science, Vol. 1411, Springer, June 1998.

[11] M. González Harbour, M. Aldea Rivas, J.J. Gutiérrez García, and J.C. Palencia Gutiérrez, "Implementing and using Execution Time Clocks in Ada Hard Real-Time Applications". Proceedings of the 1998 Ada Europe International Conference on Reliable Software Technologies, Uppsla, Sweden; in Lecture Notes on Computer Science, Vol. 1411, Springer, June 1998.

[12] J. Liu, K.J. Lin, W.K. Shih, A. Chuang-Shi Yu, J.Y. Chung, and W. Zhao. "Algorithms for Scheduling Imprecise Computations". IEEE Computer, pp. 58-68, May 1991.