Operating System Support for Execution Time Budgets for Thread Groups

Mario Aldea Rivas and Michael González Harbour
Universidad de Cantabria
39005-Santander, SPAIN
{mgh, aldeam}@unican.es

Abstract

The recent Ada 2005 standard introduced a number of new real-time services, with the capability of creating and managing execution time budgets for groups of tasks. This capability has many practical applications in real-time systems in general, and therefore it is also interesting for real-time operating systems. In this paper we present an implementation of thread group budgets inside a POSIX real-operating system, which can be used to implement the new Ada 2005 services. The architecture and details of the implementation are shown, as they may be useful to other implementers of this functionality defined in the new standard.

Keywords: *Real-time systems, Execution time budgets, Thread groups, CPU time, Ada 2005*.

1. Introduction[1]

In hard real-time systems it is essential to monitor the execution times of all tasks and detect situations in which the estimated worst-case execution time (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 can be accomplished with execution time clocks and timers.

This need for managing execution time is recognized in standards related to real-time systems. The POSIX standard [4] defines services for execution time measurement and budget overrun detection, and its associated real-time profiles [5] require implementations to support these services. The recent Ada 2005 standard introduced a number of new real-time services intended to provide applications with a higher degree of flexibility. In particular this standard defines capabilities for measuring the execution time of individual tasks, and the ability to detect and handle execution-time budget overruns.

As real-time applications evolve towards an increased complexity level, issues such as composability of independently developed application components and support for legacy code introduce the need for supporting different levels of hierarchy in the scheduling mechanism, leading to a hierarchical concurrency model with different layers, and with capabilities for establishing boundaries for the protection of different parts of the application. In this context of hierarchical scheduling it is often required to bound the execution time of a group of activities that are inside the same protection boundary, so that they cannot interfere with other activities in other protection boundaries by using up more resources than they should. This need introduces a requirement on the underlying implementation to support the measurement of the execution times of groups of tasks, and the handling of potential budget overruns, in a way similar to what is usually done for individual tasks.

Following this general requirement, the Ada 2005 standard defines services for execution-time budgets for groups of tasks, and is now a step forward in relation to the real-time extensions to POSIX, which still has no such service.

In this paper we propose an implementation of a mechanism to support execution-time budgets for thread groups inside a POSIX operating system. The API of this implementation could be used as a basis for a future extension to POSIX. It will also be used to implement the task group budgets defined in Ada 2005. The architecture and details of the implementation are shown, as they may be useful to other implementers of this functionality defined in the new standard. Some performance metrics are provided.

The paper is organized as follows. Section 2 discusses the current services that are available in the platform chosen for this implementation, MaRTE OS and GNAT, and that are related to thread group budgets. Section 3 introduces the

1. This work has been funded by the *Plan Nacional de I+D+I* of the Spanish Government under grant TIC2005-08665-C03 (THREAD project), by Ada Core, and by the European Union's Sixth Framework Programme under contracts FP6/2005/IST/5-034026 (FRESCOR project) and IST-004527 (ARTIST2 NoE). This work reflects only the author's views; the EU is not liable for any use that may be made of the information contained herein.

services designed to represent sets of threads. Section 4 discusses the implementation of the execution time clocks for groups of threads, while Section 5 does the same for budgets and their associated handlers. Section 6 provides some performance metrics and, finally, Section 7 gives our conclusions.

2. Background

The implementation of execution time budgets for thread groups presented in this paper has been developed in MaRTE OS [1] [2], which is a real-time operating system (RTOS) that follows the POSIX.13 [5] minimum real-time system profile, and is mostly written in Ada. It is available for the ix86 architecture as a bare machine, and it can also be configured as a POSIX-thread library for GNU/Linux. The GNAT run-time library has been adapted to run on top of MaRTE OS, which is itself being extended in a joint effort between Ada Core and the University of Cantabria with the objective of providing a platform fully compliant with Ada 2005, available for industrial, research, and teaching environments. The implementation of thread group budgets presented in this paper is part of the effort to achieve this objective.

Two of the new Ada 2005 real-time services are closely related to the thread group budgets and are already available in MaRTE OS and GNAT [3]:

- *Timing events* are defined in Ada 2005 as an effective and efficient mechanism to execute user-defined time-triggered procedures without the need to use a task. They are very efficient because the event handler may be executed directly in the context of the interrupt handler, avoiding the need for a server task.
- Execution time clocks and timers are defined in Ada 2005 as a standardized interface to obtain the execution time consumption of a task, together with a mechanism that allows creating handlers that are triggered when the execution time of a task reaches a given value, providing the means to execute a user-defined action when the execution time assigned to a specific task expires.

Timing events have been implemented in MaRTE OS through a service that we call “timed handlers”, which are not only useful to implement their Ada counterpart, but are also useful to other applications as a general-purpose RTOS mechanism.

MaRTE OS supports the execution-time clocks and timers defined in POSIX.1, which would be appropriate to implement their couterparts in Ada. However, the timers defined in POSIX to detect execution time overruns use an operating system signal to notify about their expiration. Signals are a very scarce resource inside an RTOS. Besides, the signal is usually handled through a thread that is waiting to accept the signal, but this is a mechanism that introduces relatively high overheads, mainly due to the need for the handler to be a thread, with the associated costs in context switches. This leads to the same reason for introducing the new "timing events" mechanism for regular time management.

As a consequence, the Ada implementation of execution time clocks and timers has been achieved in MaRTE through the "timed handler" mechanism, which allows a direct handling of the event inside the hardware timer interrupt handler, thus avoiding the use of a signal and the subsequent double context switch that would be necessary otherwise.

To implement thread group budgets inside MaRTE OS we will follow an approach similar to that followed for execution time budgets for individual threads, creating the appropriate execution time clocks for thread groups and extending the "timed handler" mechanism to also support these new clocks.

3. Thread sets

Before creating the execution time clocks for thread groups or sets, it is necessary to specify a mechanism to represent the groups themselves. Instead of defining a mechanism specific to execution-time clocks, we have chosen to create an independent RTOS object that represents a group of threads. In this way, we will be able to address future extensions that require handling groups of threads using these same objects. Examples of such new services might be related to the requirements for supporting hierarchical scheduling, for instance to suspend or resume a group of threads atomically.

A thread set is implemented by a record that may be extended in the future to add functionality. This record has the following fields:

- *Set* : A list of the threads belonging to the set.
- *Iterator*: A reference to the current thread in the list, used when iterating through `marte_threadset_first` and `marte_threadset_next`.

A restriction has been made so that a thread can belong to only one thread set. This restriction is also made in the Ada 2005 standard, and its rationale is that in the hierarchical scheduling environment for which thread groups are useful, threads only belong to one specific scheduling class, and therefore to one specific set. This restriction allows a more efficient implementation, because at each context switch only one of the *Consumed_Time* fields of the set to which the running thread belongs needs to be updated.

Threads can be added/removed to/from a thread set dynamically.

Every thread has a pointer in its thread control block (TCB) to the set it belongs to. This field is *null* if the thread doesn't belong to any thread set.

The C language API to manage thread sets from the application level is the following:

```
// create an empty thread set
int marte_threadset_create
       (marte_threadset_id_t *set_id);
// destroy a thread set
int marte_threadset_destroy
       (marte_threadset_id_t set_id);
// empty an existing thread set
int marte_threadset_empty
       (marte_threadset_id_t set_id);
// add a thread to a set
int marte_threadset_add
       (marte_threadset_id_t set_id,
         pthread_t thread_id);
// delete a thread from a set
int marte_threadset_del
       (marte_threadset_id_t set_id,
         pthread_t thread_id);
// check thread membership
int marte_threadset_ismember
       (marte_threadset_id_t set_id,
         pthread_t thread_id);
// reset the iterator and get the first thread id
int marte_threadset_first
       (marte_threadset_id_t set_id,
         pthread_t *thread_id);
// advance the iterator and get next thread id
int marte_threadset_next
       (marte_threadset_id_t set_id,
         pthread_t *thread_id);
// check whether the iterator can be advanced
int marte_threadset_hasnext
       (marte_threadset_id_t set_id)
// get the set associated with the given thread
int marte_threadset_getset
       (marte_threadset_id_t *set_id)
         pthread_t thread_id);
```

4. Execution time clocks for thread groups

To implement execution time clocks for groups of threads we add the following information to the object that represents a thread set:

- *Consumed_Time*: CPU-time consumed for all the task in the group. Every time a thread of a given set leaves the CPU, the time consumed by this task since its last activation is added to the *Consumed_Time* of its thread set, even if there is no timed event associated with it, because the value of the execution-time clock may be read at any time by the application.
- *Group_Timed_Event* : A reference to the internal RTOS execution time event, used by the scheduling mechanism. A set can be associated with at most one such event.

The API to obtain an execution-time clock from a thread set is:

```
// destroy a thread set
int marte_getgroupcpuclockid
       (marte_threadset_id_t set_id,
         clockid_t *clock_id);
```

The returned id represents a clock that can be read and set through the standard POSIX API for clocks, i.e., using functions `clock_gettime`, `clock_settime`, ... They can also be used as the base for POSIX timers and MaRTE OS timed events as any other clock defined in the system. They can not however be used as the base for the `clock_nanosleep` operation, as is also the case with the single-thread CPU-time clocks. POSIX leaves this behavior as unspecified and Ada does not define execution time as a type that can be used in the equivalent delay statements.

POSIX requires type `clockid_t` to be defined as an arithmetic type, and therefore clock ids are implemented using a unsigned number of 32 bits. The value stored in a clock id can have different interpretations:

- Special values for the regular calendar-time clock `CLOCK_REALTIME`, the execution time clock of the current thread `CLOCK_THREAD_CPUTIME_ID`, and the monotonic clock `CLOCK_MONOTONIC`.
- A pointer to a thread control block when the clock is a thread CPU-time clock of a particular thread.
- A pointer to a thread set when it is a thread group clock.

5. Timed events based on a group clock

Group clocks can be used as the base of timers and timed handlers. When a timer or a timed handler is armed, a MaRTE OS timed event is enqueued in the system event queues. Time-based events in MaRTE OS are of two kinds: standard time and execution time. They are kept in separate priority queues because they cannot be compared with each other for ordering. Events based on group clocks are a special case of execution time events. An execution time event has the following information:

- *CPU_Time*: The event will expire when the execution time consumed by the associated task reaches this value
- *Group_Expiration_Time*: The event will expire when the *Consumed_Time* field of the task set associated with the event reaches this value. This field is only used in events based on a group clock.
- *Is_Based_On_Group_Clock*: This is a boolean used to identify events based on group clocks
- *Base_Clock*: A clock id representing the clock used as the timing base of the event. It could be a thread CPU-time clock or a group clock.

- *Task_Where_Queued* : A pointer to the task that has queued the event.

Execution time events are kept in a queue associated with the task on which the event is based on, and stored as the *CPU_Time_Timed_Event_Queue* in the task control block. Every time a new thread gets the CPU, the events at the head of the standard-time events queue and of the running task´s *CPU_Time_Timed_Event_Queue* queue are compared. The hardware timer is programmed to expire at the most urgent of the two.

Events based on group clocks are special CPU-time events that "jump" between the *CPU_Time_Timed_Event_Queue* of the threads in the group. Each time the system schedules a task included in a thread set that has an event associated, the following actions are performed in the `Do_Scheduling` internal kernel operation:

```
-- Set CPU_Time of the event according to the
-- time consumed by T
T.Set.Group_TE_Ac.CPU_Time := T.Used_CPU_Time +
   (T.Set.Group_TE_Ac.Group_Expiration_Time -
    T.Set.Consumed_Time);
-- Move Group_TE_Ac from one task to another
if T.Set.Group_TE_Ac.Task_Where_Queued /= null
then
   -- Dequeue from the list it was queued
   Dequeue (T.Set.Group_TE_Ac,
            T.Set.Group_TE_Ac.Task_Where_Queued.
               CPU_Time_TEs_Q);
end if;
-- Enqueue in T's list
Enqueue_In_Order (T.Set.Group_TE_Ac,
                  T.CPU_Time_TEs_Q);
T.Set.Group_TE_Ac.Task_Where_Queued := T;
```

Dequeue and enqueue operations are very fast, because the number of CPU-time events associated to a task usually will be very small, either one or two: a CPU-time event and a "group event". Consequently the number of extra operations required at each context switch to manage these clocks is kept small, and the implementation can efficiently schedule the threads with an acceptable overhead, as can be seen in the following performance metrics section.

6. Performance metrics

The support for group budgets has already been implemented in MaRTE OS. Execution time accounting introduces a small overhead: enabling this service in MaRTE OS increments the context switch time by less than 5%. Group execution time accounting increments the context switch time by another 4%, representing a total of 9% increment with respect to a system with no CPU-time accounting in an x86 architecture.

The overheads of the budget overrun detection are also relatively small. Table 1 shows a comparison of the overheads of two detection mechanisms, as measured in a 3.4GHz Pentium IV. The first one is implemented using a regular POSIX timer that sends a signal when the budget expires, and a handler thread that blocks waiting to accept the signal. The second mechanism is implemented using the new timed handler service. We can see that the overhead of the second mechanism is much smaller.

Table 1. Overhead of budget overrun notification mechanism

Metric	Time (μs) (using timer and auxiliary thread)	Time (μs) (using timed handlers)
From user´s thread to handler	1.1	0.4
From handler to user's thread	0.8	0.7
Total time:	1.9	1.1

7. Conclusion

As the complexity of real-time systems evolves, hierarchical scheduling and partitioning are mechanisms used to cope with it, by helping in establishing protection boundaries and easing the composability of independently-developed application components. One of the requirements of this partitioning is the time protection among the different groups of tasks in the hierarchy, which can be achieved by using thread group budgets as those specified in the new Ada 2005 standard.

This paper has presented an implementation of the support needed to provide such budgeting services in a real-time operating system called MaRTE OS. The paper describes the architecture and details of the implementation, together with the rationale for the main design decisions, so that this information can be used by other implementers of this functionality, either as part of Ada run-time systems, or as part of a general-purpose RTOS. The implementation has proven to be straightforward, and the overheads introduced are small, both in the context switch times and in the budget overrun notification mechanism.

As future work, the functionality defined in Ada 2005 for group budgets will be implemented. It is anticipated that support for the Ada group budgets will be a simple package built on top of the MaRTE OS implementation described in this paper.

References

[1] Aldea Rivas M. and González Harbour M. *MaRTE OS: Minimal Real-Time Operating System for Embedded Applications*. Universidad de Cantabria. `http://marte.unican.es/`

[2] Aldea Rivas M. and González Harbour M. *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, ISBN:3-540-42123-8, pp. 305,316.

[3] Aldea Rivas M. and Ruiz J.F.. *Implementation of new Ada 2005 real-time services in MaRTE OS and GNAT*. International Conference on Reliable Software Technologies, Ada-Europe-2007, Switzerland.

[4] IEEE Std. 1003.1:2004 Edition, *Information Technology — Portable Operating System Interface (POSIX)*. The Institute of Electrical and Electronics Engineers.

[5] IEEE Std. 1003.13-2003. *Information Technology - Standardized Application Environment Profile- POSIX Realtime and Embedded Application Support (AEP)*. The Institute of Electrical and Electronics Engineers.

[6] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, Erhard Ploedereder, Pascal Leroy (Eds.) *Ada-2005 Reference Manual. Language and Standard Libraries*. International Standard ISO/IEC 8652/1995(E) with Technical corrigendum 1 and Amendment 1. Springer, Number 4348 in Lecture Notes in Computer Science, Springer-Verlag (2006).