Preliminary Multiprocessor Support of Ada 2012 in GNU/Linux Systems

Sergio Sáez and Alfons Crespo

Instituto de Automática e Informática Industrial,
Universidad Politécnica de Valencia,
Camino de vera, s/n, 46022 Valencia, Spain
{ssaez,alfons}@disca.upv.es

Abstract. The next release of the Ada language, Ada 2012, will incorporate several new features that address current and future software and hardware issues. One of these features is expected to be explicit support for multiprocessor execution platforms. This work reviews the enhancements at the language level required to support real-time scheduling over symmetric multiprocessor platforms, and the corresponding support at the operating system level. It analyses the preliminary support for these features within the Linux kernel and proposes a set of language extensions that will provide the required functionalities. Multiprocessor implementation aspects of other Ada language constructs such as *timing events*, *execution time clocks* and *interrupt management* are also analysed.

Keywords: Symmetric Multiprocessor Platforms, Linux kernel, Ada 2012.

1 Introduction

Real-Time and embedded systems are becoming more complex, and multiprocessor/multicore systems are becoming a common execution platform in these areas. Although schedulability analysis techniques for real-time applications executing over multiprocessor platforms are still not mature, some feasible scheduling approaches are emerging. However, in order to allow these new techniques to be applied over existing real-time multiprocessor operating systems, a flexible support has to be provided at the kernel and user-space levels.

Ada 2005 allows real-time applications to be executed on multiprocessor platforms in order to exploit task-level parallelism, but no direct support is provided to allow the programmer to control the task-to-processor mapping process. In order to achieve a predictable behaviour in a real-time multiprocessor platform, the system designer must be able to control the task-to-processor allocation process and, in general, the processor assignment of any executable unit with dynamic behaviour, such as timers and interrupt handlers.

At the last International Real-Time Ada Workshop (IRTAW 14) some of these multiprocessor issues were addressed. Ada extensions were proposed to cope with open issues in the scope of real-time application over symmetric multiprocessor platforms (SMP) [1]. However, no specific real-time multiprocessor operating system was considered during this analysis.

This paper analyses the existing support in the GNU/Linux platform to implement the multiprocessor extensions of the Ada language that are likely to be integrated in the forthcoming Ada 2012. Some features of Ada 2005 that are poorly supported in the GNU/Linux platform are also considered. The work reviews the scheduling approaches that can be used to design real-time multiprocessor systems, the programming functionalities required and some of the proposed Ada interfaces for making use of these approaches at the application level. Once the Ada RTS requirements have been analysed, the current support to implement these extensions in the GNU/Linux platform is studied and simple extensions at library and kernel level[1] are proposed for those features that are still missing. Kernel and library support for execution-time clocks, group budgets and interrupt handling over multiprocessor platforms are also considered.

The rest of this paper is organised as follows: the next section deals with multiprocessor task scheduling in real-time systems, and the requirements at the language and kernel level to address the presented approaches. Section 3 addresses execution time clocks and group budget issues over the GNU/Linux platform. Interrupt affinities and timing events are addressed in section 4. The paper finishes with some conclusions and proposals for future work.

2 Multiprocessor Task Scheduling

A real-time system is composed of a set of tasks that concurrently collaborate to achieve a common goal. A real-time task can be viewed as an infinite sequence of job executions. Each job is activated by an event, such as an external interrupt from a physical device or an internal clock event. It is assumed that a job does not suspend itself during its execution. A task is suspended only at the end of a job execution, waiting for the next activation event. Even so, the system scheduler can preempt a job execution in order to execute a higher priority task.

2.1 Multiprocessor Scheduling Approaches

In order to predictably schedule a set of real-time tasks in a multiprocessor platform several approaches can be applied. One classification based on the capability of a task to migrate from one processor to another is shown next:

Global scheduling. All tasks can be executed on any processor and after a preemption the current job can be resumed in a different processor.

Job partitioning. Each job activation of a given task can be executed on a different processor, but a given job cannot migrate during its execution.

Task partitioning. All job activations of a given task have to be executed in the same processor. No job migration is allowed.

[1] During this work Linux kernel 2.6.31 and GNU C Library *glibc* 2.10 have been used.

Depending on whether the choice of the target processor is carried out *on-line* or *off-line*, six possible approaches arise. Static schedulability analysis techniques that ensure timely execution of the task set in all of them have not yet been developed. Neverthless, new techniques are continuously emerging with improved support for several of these approaches: new global scheduling schedulability tests for fixed [2] and dynamic priorities [3], task splitting approaches [4], etc. Since it is not expected that a commercial Real-Time Operating System (RTOS) would implement each possible on-line scheduling algorithm at the kernel level, the RTOS support has to be focused on a flexible *System Call Interface* that makes it possible to implement the required services at the application or run-time level [5,6].

Next subsections analyse the required kernel functionalities that must be provided to implement these approaches at the user-space level, possible extensions of the Ada Standard Libraries, and the available support for these functionalities that is present in the Linux kernel and glibc library.

2.2 Required Functionalities

In order to allow efficient implementations of multiprocessor scheduling approaches at the user-space level, a RTOS must provide a flexible kernel programming interface and a set of administration tools. These facilities must allow the system architect to design predictable real-time applications, despite of the current unavailability of accurate schedulability analysis techniques. Depending on the scheduling approach followed to map the real-time task set into the platform processors, the functionalities required from the RTOS may vary.

On one hand, if target processors for every active entity (task or job) are determined off-line, the resulting decisions are typically coded into the application as is done with other real-time task attributes (priority, deadline, etc.) Thus, the application code needs an adequate Application Programming Interface (API) to specify the target processor for each active entity. In the case of a task partitioning approach, the target processor can be specified before thread creation, by means of thread attributes, or during the task initialisation code by means of direct processor allocation. However, if the job partitioning approach is used, the task is responsible for setting the processor to be used by the next job at the end of the execution of the current one. Next section shows examples of both approaches using a possible Ada interface. The requirements for a full off-line global scheduling approach, i.e. a cyclic multiprocessor executive, is out of the scope of this work.

On the other hand, when the target processor for a given active entity is determined on-line, i.e. during normal system execution, the processor allocation decisions are normally carried out by a privileged task or by the system scheduler. In a user-space implementation, this privileged task acts as an Application-Defined Scheduler (ADS)[5] and has to use the API offered by the OS kernel to allocate application tasks into specific processors. In this case, although other kernel mechanisms could be required to allow this application-level scheduling, e.g. normal and execution time timers, the API to specify a target processor would be quite similar. The only difference with respect to the off-line case is that the ADS thread must be able to specify the target processor for the rest of the application threads.

It is also possible that a mixed on-line/off-line approach be required by a given real-time task set, such as an application with several operating modes. In this case, an off-line task partitioning approach could be applied on each operating mode, but an on-line mode change manager should be able to modify the target processor of any system task during a mode change. Fortunately, this mixed approach does not require any additional kernel functionality with respect to pure on-line or off-line approaches.

Finally, an additional possibility is to use different scheduling approaches on each processor of the execution platform. In such a way, the available processors could be partitioned in different mutual-exclusive processor sets, applying to each subset a different scheduling approach. This possibility only requires additional functionalities from the OS kernel if at least one of the scheduling approaches used in a processor subset is the global scheduling approach, and the global scheduler is implemented at the kernel-level. In this later case, the application should be able to specify that a given set of tasks is going to be globally scheduled but that the set only can use a subset of the available processors. Once the processors subset is specified, the OS scheduler decides *when* and *where* each task is executed within the specified processor subset. No additional API is required if the processors partition is performed and managed at user-space.

Therefore, the functionalities expected from the OS to implement the presented multiprocessor scheduling approaches can be summarised as follows:

R.1. The ability to specify the target processor of the current task or a different task.
R.2. The ability to change the execution processor immediately, or to specify the target processor for the next activation of a task.
R.3. The ability to specify a unique target processor or a subset of the available ones for a given task.

2.3 Ada Programming Interface

In the last International Real-Time Ada Workshop several different Ada interfaces were proposed to cover the scheduling approaches presented above. The main differences between these proposals reside in the abstractions presented at application level. This section shows a set of similar interfaces to cope with the multiprocessor scheduling approaches and their requirements presented above.

The current behaviour of Ada applications over multiprocessor execution platforms defaults to the global scheduling approach. GNAT[2] provides a basic support for switching from global scheduling to off-line task partitioning by means of **pragma** Task_Info. Apart from this basic support, a possible interface to address task partitioning at the application level is shown by Listing 1.

[2] GPL 2009 version of GNAT compiler from AdaCore has been used in this work as the reference implementation of native Ada Run-Time System. `www.adacore.com`

Listing 1. Basic multiprocessor support

```
package Ada_System is
   type Processor is range ...; -- implementation-defined
   type Processor_Set is array (Processor) of Boolean;

   Any_Processor : constant Processor := ...;
end Ada_System;

with Ada.Task_Identification; use Ada.Task_Identification;
with Ada_System; use Ada_System;
package Ada_System_Scheduling is
    ...
   function Get_Available_Processors return Processor_Set;
   procedure Set_Processor(P: Processor; T : Task_Id := Current_Task);
   function Get_Processor(T : Task_Id := Current_Task) return Processor;
end Ada_System_Scheduling;
```

This interface provides support for requirement **R.1** allowing the current task to change its execution processor or the processor of another task. However, to support job partitioning a given task needs to establish its next target processor. If procedure Set_Processor is supposed to be a scheduling point, i.e., it changes the execution processor immediately, a different procedure will be required to perform a deferred processor assignment. Since it makes no sense to perform the complicated process involved in changing the execution processor of the current task just before getting suspended, some deferred processor assignment procedure is required to support the job partitioning approach efficiently.

Following the example of *deadline assignment and delay until* of Ada 2005, a deferred processor assignment could be established as shown by Listing 2.

Listing 2. Processor assignment and **delay until** procedure

```
procedure Delay_Until_And_Set_Processor(DT: Ada.Real_Time.Time;
                                        P: Processor; T: Task_Id:= Current_Task);
```

However, given that the use of Delay_Until_And_Set_Something procedures will be incompatible with the procedure Delay_Until_And_Set_Deadline, already defined in Ada 2005 (D.2.6) [7] we propose instead the procedure shown in Listing 3.

Listing 3. Deferred processor assignment procedure

```
procedure Set_Next_Processor(P: Processor; T : Task_Id := Current_Task);
```

This procedure makes it possible to establish the next target processor to be used when the task Task_Id is awakened again to start its next job execution. In this way, the next processor assignment can be combined with any **delay until** construct or Delay_Until_And_Set_Something procedure. Both procedures Set_*_Processor will cover requirement **R.2** described above. Listing 4 shows a periodic task with job partitioning that makes use of the deferred processor assignment and **delay until** construct.

Listing 4. Periodic task with job partitioning based on **delay until**

```
with Ada_System; use Ada_System;
with Ada_System_Scheduling; use Ada_System_Scheduling;
task body Periodic_With_Job_Partitioning is
   type List_Range is mod N;
   Processor_List: array (List_Range) of Processor
                 := (...);  -- Decided at design time
   Processor_Iter : List_Range := List_Range'First;
   Next_Processor: Processor;
   Next_Release  : Ada.Real_Time.Time;
   Period        : Time_Span := ...;
begin
   Task_Initialize ;
   Next_Release := Ada.Real_Time.Clock;
   Set_Processor(Processor_List( Processor_Iter )); -- Processor for first activation
   loop
      Task_Main_Loop;
      -- Next job preparation
      Processor_Iter := Processor_Iter 'Succ;
      Next_Processor := Processor_List(Processor_Iter );
      Next_Release := Next_Release + Period;
      Set_Next_Processor(Next_Processor);
      delay until Next_Release;
      -- Alternatively: Delay_Until_And_Set_Processor(Next_Release, Next_Processor);
   end loop;
end Periodic_With_Job_Partitioning;
```

In this example, a periodic task uses a recurrent list of the processors where its jobs will execute. The cyclic processor list is supposed to be computed at design time using an off-line job partitioning tool. In this way, task migrations between processors are determined in a predictable way at job boundaries. If Set_Processors procedure was used before **delay until** construct, then the task would perform an unnecessary processor migration just before getting suspended for the next job activation.

A different approach to implement the job partitioning scheme without requiring deferred processors assignments is the use of Timing_Events to implement periodic, sporadic and/or aperiodic task as proposed in [8]. In this case, the Timing_Event handler can change the target processor of the periodic task using Set_Processor before releasing it (requirement **R.1**). Listing 5 shows a periodic task using job partitioning and Timing_Events to implement a periodic release mechanism[3].

[3] Original code extracted from Real-Time Ada Framework [8].

Listing 5. Periodic task with job partitioning based on Timing_Events

```
with Ada_System; use Ada_System;
with Ada_System_Scheduling; use Ada_System_Scheduling;
package body Release_Mechanisms.Periodic is
  protected body Periodic_Release is
    entry Wait_For_Next_Release when New_Release is
    begin
       ...
      New_Release := False;
       ...
    end Wait_For_Next_Release;

    procedure Release(TE : in out Timing_Event) is
    begin
      Next := Next + S.Period;
      Set_Processor(S.Next_Processor, S.Get_Task_Id); -- Set next job processor
      New_Release := True;                            -- Activates the job
      TE.Set_Handler(Next, Release'Access);
    end Release;
end Periodic_Release;
 ...
task type Periodic_With_Job_Partitioning (S: Any_Task_State;
                                          R: Any_Release_Mechanism) is
   pragma Priority(S.Get_Priority);
end Periodic_With_Job_Partitioning;

task body Periodic_With_Job_Partitioning is
   type List_Range is mod N;
   Processor_List: array (List_Range'First .. List_Range'Last) of Processor
                 := (...); -- Decided at design time
   Processor_Iter: List_Range := List_Range'First;
   Next_Processor: Processor;
begin
   S. Initialize ;
   Set_Processor(Processor_List( Processor_Iter )); -- Processor for first activation
   loop
      S.Code;
      -- Next job preparation
      Processor_Iter := Processor_Iter 'Succ;
      Next_Processor := Processor_List(Processor_Iter );
      S.Set_Next_Processor(Next_Processor);
      R.Wait_For_Next_Release;
   end loop;
end Periodic_With_Job_Partitioning;
```

Finally, to cover requirement **R.3** and allow the construction of processor partitions to restrict global scheduling to a subset of the available processors the notion of *scheduling allocation domains* [1] has been proposed. The interface to be added to Listing 1 is similar to the one shown in Listing 6.

Listing 6. Scheduling domain management interface

```
package Ada_System_Scheduling is
  type Scheduling_Domain is limited private;

  function Create(PS : Processor_Set) return Scheduling_Domain;
  function Get_Processor_Set(SD : Scheduling_Domain) return Processor_Set;
  procedure Allocate_Task(SD: in out Scheduling_Domain;
                          T : Task_Id := Current_Task);
  procedure Deallocate_Task(SD: in out Scheduling_Domain;
                            T : Task_Id := Current_Task);
  function Get_Scheduling_Domain(T : Task_Id := Current_Task)
                                              return Scheduling_Domain;
  ...
end Ada_System_Scheduling;
```

2.4 GNU/Linux Support

Although the Linux kernel is a general purpose OS, its real-time behaviour and its expressive features are being continuously improved, making it possible to build *hard-enough* real-time applications for some environments. However, the glibc-kernel tandem on GNU/Linux systems still lacks of some of the features required by Ada 2005 (e.g. EDF scheduling[4]). This gives rise to an incomplete native implementation of the Ada RTS over this platform.

However, this is not the case for multiprocessor scheduling support. The scheduling scheme in the Linux kernel is based on a static priority policy for real-time processes and a variable quantum round-robin scheduling algorithm for normal processes, as proposed by POSIX 1003.1b [9]. Each processor has its own *run queue* with a separate *process queue* per priority level: the first 100 priority levels for real-time processes (SCHED_FIFO and SCHED_RR POSIX scheduling policies) and the next 40 priority levels for normal processes (SCHED_NORMAL). The lower the priority level, the higher the process priority. These real-time priority levels are exposed in reverse order to the applications[5] that can use real-time priorities in the range $[1, 99]$[6].

In spite of the per-processor run queue internal structure of the Linux kernel, processes are allowed to migrate from one processor to another. In this way, the default scheduling approach of a real-time process inside the Linux kernel is global scheduling based on static priorities. However, the Linux kernel offers a flexible interface that covers almost any requirement for implementing real-time multiprocessor scheduling approaches presented in section 2.1.

The Linux kernel interface for multiprocessor scheduling control is twofold. On one hand, an administrative tool in the form of *cpuset file system* makes it

[4] A new EDF scheduling class is being developed for new releases of Linux kernel.

[5] Real-time priority values exposed to applications put the internal priority levels upside-down: the higher the priority value, the higher the task priority.

[6] The real-time process queue with the highest priority is reserved for Linux kernel internal processes.

possible to organise the processor and memory placement for SMP and NUMA architectures [10]. These *cpusets* provide a mechanism for assigning a set of CPUs and memory nodes to a set of tasks, and for establishing a set of useful features to configure the behaviour of heterogeneous multiprocessor platforms.

On the other hand, a programming interface is also provided by means of kernel system calls. This API allows the specificantion of the set of processors that are to be used by a given thread, but they are subordinated to the configuration performed with the *cpuset* mechanism. Functions provided by the Linux kernel to support scheduling requirements presented in section 2.2 are shown in Listing 7.

Listing 7. CPU-related Linux kernel system calls

```
#define _GNU_SOURCE
#include <sched.h>
#include <linux/getcpu.h>
int  sched_setaffinity (pid_t pid,  size_t  cpusetsize,  cpu_set_t *mask);
int  sched_getaffinity (pid_t pid,  size_t  cpusetsize,  cpu_set_t *mask);
int getcpu(unsigned *cpu, unsigned *node, struct getcpu_cache *tcache);
```

These functions allow a real-time application to select between global scheduling, specifying multiple processors in its affinity map `mask`, and the task partitioning approach, using single processor affinity maps. There are a set of equivalent functions that allow the use of thread identifiers or thread attributes to specify the target task. Using this system call interface an Ada RTS can implement almost the full interface proposed in section 2.3. However, as the system call function sched_setaffinity immediately enforces the process identified by `pid` to be executed on a processor belonging to its affinity map, deferred processor assignment required by the job partitioning approach still remains unsupported. The next subsection suggest a Linux kernel extension to support Ada procedures Set_Next_Processor or Delay_Until_And_Set_Processor.

2.5 Required Linux Kernel Extensions

The required functionality to allow deferred affinity changes implies simple changes at kernel and glibc level. The suggested extension is to modify the internal system call kernel function sched_setaffinity to accept an additional flag parameter that specifies when the affinity change will be performed. If the new flag indicates *deferred change*, sched_setaffinity function avoids immediate task migration by skipping the invocation to migrate_task(). In this case, the processor switch will be automatically performed when the corresponding thread becomes suspended and removed from the current processor *run queue*. The new implementation of this system call will be faster than the previous one for deferred processor assignments and will carry almost no penalty when immediate task migration is used. The new system call function prototype is shown in Listing 8.

Listing 8. Linux kernel system calls modifications

```
#define SCHED_SET_IMMEDIATE 1
#define SCHED_SET_DEFERRED 2
long sched_setaffinity (pid_t pid, const struct cpumask *in_mask, const long flag);
```

At the glibc library level the kernel system call will be separated into two wrapper functions shown in Listing 9, for backward compatibility reasons.

Listing 9. Glibc library level extensions

```
/* The old one use SCHED_SET_IMMEDIATE flag */
int sched_setaffinity (pid_t pid, size_t cpusetsize, cpu_set_t *mask);
/* The new one use SCHED_SET_DEFERRED flag */
int sched_setnextaffinity (pid_t pid, size_t cpusetsize, cpu_set_t *mask);
```

Once these simple extensions are applied to the kernel and glibc library, all the multiprocessor scheduling approaches reviewed in section 2 can be implemented, and the proposed Ada interfaces can be incorporated within the native Ada RTS for GNU/Linux platform.

3 Execution Time Clocks, Timers and Group Budgets

The Ada Reference Manual provides a language-defined package to measure execution time of application tasks in section D.14 [7]. However, GNAT GPL 2009 does not implement execution time clocks in the native RTS for the GNU/Linux platform. This section examines whether the current Linux kernel and glibc library have enough facilities to implement the Ada.Execution_Time package and its child packages Timers and Group_Budgets, and the corresponding multiprocessor extensions.

Recent Linux kernel and glibc library implement several clocks that make it possible to measure time in different ways. Among the currently supported clocks, we find CLOCK_THREAD_CPUTIME_ID that performs high-resolution measurements of the CPU-time consumed by a given thread. The interface functions shown in Listing 10 provide the *clock identifier* of a given thread and read its CPU-time clock and resolution.

Listing 10. GNU/Linux support functions for execution time clocks

```
#include <pthread.h>
#include <time.h>
int pthread_getcpuclockid(pthread_t thread, clockid_t *clock_id);
int clock_getres (clockid_t clk_id, struct timespec *res);
int clock_gettime(clockid_t clk_id, struct timespec *tp);
```

These functions and the definition of the `timespec` structure are enough to implement the package Ada.Execution_Time on the GNU/Linux platform. However, the manual pages about these functions advise about possible bogus results if the implied thread is allowed to migrate between processors with different clock sources, since the processor running frequencies could be slightly different. SMP platforms based on multicore processors should not suffer from this kind of clock drifts.

In the same way, the Linux kernel implements POSIX timers, which implement all the functionalities required by package Ada.Execution_Time.Timers. The function prototypes in C are shown in Listing 11.

Listing 11. POSIX timers in Linux kernel

```
#include <signal.h>
#include <time.h>
int timer_create(clockid_t clockid, struct sigevent *evp, timer_t *timerid);
int timer_settime(timer_t timerid, int flags, const struct itimerspec *new_value,
                  struct itimerspec * old_value);
int timer_gettime(timer_t timerid, struct itimerspec *curr_value);
int timer_delete(timer_t timerid);
```

However, the Linux kernel defines an additional notification mechanism that can be specified in struct sigevent on timer creation: SIGEV_THREAD_ID. This new mechanism makes it possible to notify a specific thread when the timer expires. As described in previous sections, a thread can have a specific processor affinity map. This notificationfacility can be used by the Ada RTS to create a set of server tasks that manage timer expirations on a per-processor or per-scheduling domain basis. With This structure we can add specific multiprocessor support in package Ada.Execution_Time.Timers in order to set the processor where the protected procedure defined by the Timer_Handler will be executed. The notification thread will directly depend on the target processor specified for the timer handler execution. Based on the previously proposed interfaces, Timer type could be extended as shown by Listing 12.

Listing 12. Extensions to execution time timers

```
with Ada_System; use Ada_System;
with Ada_System_Scheduling; use Ada_System_Scheduling;
package Ada.Execution_Time.Timers is
  ...
  procedure Set_Scheduling_Domain(TM : in out Timer;
                                  SD: access all Scheduling_Domain);
  function Get_Scheduling_Domain(TM : Timer) return Scheduling_Domain;
  procedure Set_Processor(TM : in out Timer; P: Processor);
  function Get_Processor(TM : Timer) return Processor;
end Ada.Execution_Time.Timers;
```

On the other hand, although the Linux kernel computes the execution time of the group of threads that composes a process, no additional groups of threads can be defined without disrupting process unity. So, to directly support group budgets under GNU/Linux systems, a completely new set of kernel system calls and the corresponding support will have to be added.

4 Interrupt Affinities

In a general purpose OS like GNU/Linux system the interrupt management rarely can be performed at user-space level. What is intended as interrupt management in Ada RTS on UNIX systems is to manage the *software interrupts* or *signals* as defined by the POSIX standard.

4.1 Hardware Interrupts and POSIX Signals

Adding multiprocessor support to POSIX signals management is somehow problematic, since the standard specifies that signals are sent to a process as a whole. Because of this, to establish on which thread and in which processor is a signal handler to be executed does not have a straightforward solution. A possible interface and an implementation that will allow the specification of the processor or execution domain where a signal handler would be executed is presented bellow.

In order establish an interrupt affinity the package Ada.Interrupts can be extended as follows:

Listing 13. Explicit multiprocessor support for Ada Interrupts

```
with Ada_System; use Ada_System;
with Ada_System_Scheduling; use Ada_System_Scheduling;
package Ada.Interrupts is
   ...
  procedure Set_Scheduling_Domain(Interrupt : Interrupt_ID;
                                            SD: Scheduling_Domain);
  function Get_Scheduling_Domain(Interrupt : Interrupt_ID)
                                                  return Scheduling_Domain;
  procedure Set_Processor(Interrupt : Interrupt_ID; P: Processor);
  function Get_Processor(Interrupt : Interrupt_ID) return Processor;
end Ada.Interrupts;
```

The proposed implementation on a GNU/Linux system is similar to that proposed for handlers of execution time timers in section 3. One thread is attached to each processor in the system where a signal handler can be executed[7]. Then, all the signals are blocked on all threads allowing these signal handler threads to catch pending signals by means of the POSIX function sigwaitinfo. The prototypes of the functions involved are shown in Listing 14.

Listing 14. POSIX functions to mask and wait for signals

```
#include <pthread.h>
#include <signal.h>
int pthread_sigmask(int how, const sigset_t *newmask, sigset_t *oldmask);
int sigwaitinfo (const sigset_t *set, siginfo_t *info);
```

The value of the `sigset_t` set for a given thread represents the set of signals that have been assigned to the processor on which the thread is allocated. Signals

[7] Some processors could be reserved for non-real-time applications.

allocated only to a scheduling domain as a whole will be present on the signal sets of every signal handler thread of that domain.

Although hardware interrupt handlers cannot be attached at user level in GNU/Linux systems, it could be interesting to have a programming interface that allow an Ada real-time application to establish the processor affinity of such interrupts. As an example, a real-time application could be interested in allocating all its real-time tasks in a subset of the available processors and to move non real-time related hardware interrupts to another processors.

The Ada interface will remain as shown in Listing 13, but to support hardware interrupts the package Ada.Interrupt.Names needs to be extended with new interrupt identifiers. As interrupt lines (numbers) change from one system to another, a generic interrupt identifiers could be defined as shown by Listing 15.

Listing 15. Hardware interrupt Ada names

```
package Ada.Interrupts.Names is
  ...
  HW_Interrupt_0 : constant Interrupt_ID := ...;
  HW_Interrupt_1 : constant Interrupt_ID := ...;
  ...
end Ada.Interrupt.Names;
```

A Linux kernel over an Intel x86 platform will require at least 224 hardware interrupt identifiers, although some of them are reserved for internal kernel use. The Linux kernel offers a simple interface through its `proc` virtual file system [11] to change the affinity of a hardware interrupt. The processor affinity mask of an interrupt `#` can be established by writing the corresponding hexadecimal value on `/proc/irq/IRQ#/smp_affinity` file.

4.2 Timing Events

The standard package Ada.Real_Time.Timing_Events is also strongly related to interrupt management. It allows a user-defined protected procedure to be executed at a specified time. This protected procedure is normally executed at Interrupt_Priority 'Last in real-time applications. When such applications execute in a multiprocessor platform, with more and more code being moved to timing event handlers, it will be useful to allow the programmer to specify the processor where a timing event handler will be executed. A possible extension to support this functionality in Timing_Event type is shown in Listing 16.

Listing 16. Multiprocessor support for timing events

```
with Ada_System; use Ada_System;
with Ada_System_Scheduling; use Ada_System_Scheduling;
package Ada.Real_Time.Timing_Events is
  ...
  procedure Set_Scheduling_Domain(TM : in out Timing_Event;
                                  SD: access all Scheduling_Domain);
  function Get_Scheduling_Domain(TM : Timing_Event) return Scheduling_Domain;
  procedure Set_Processor(TM : in out Timing_Event; P: Processor);
  function Get_Processor(TM : Timing_Event) return Processor;
end Ada.Real_Time.Timing_Events;
```

To support a multiprocessor platform, an event-driven server task can be allocated on every available processor and execution domain. When procedure Set_Handler was invoked, the timing event information will be queued on the appropriate server task that will finally execute the handler code.

5 Conclusions

Some of the proposed extensions of Ada 2012 have been analysed, mainly in relation to multiprocessor platforms. The proposed Ada interfaces have been reviewed and the required support from the underlying execution platform has been studied. Existing support for the required features at Linux kernel and GNU C Library level have been analysed, and simple extensions proposed to support unaddressed requirements. Also simple Ada interfaces and implementations have been proposed to allocate any kind of execution units (timer and interrupt handlers) to specific platform processors.

After this analysis, a preliminary support of presented features has been considered feasible and we have started the corresponding implementation at the library and kernel level. We will presentExperimental results of the ongoing implementation in the near future.

Acknowledgement. This work was partially supported by the Spanish Ministerio de Ciencia y Tecnología (MCYT) under grant TIN2008-06766-C03-02 and ITEA2 06038.

References

1. Burns, A., Wellings, A.: Multiprocessor systems session summary. In: 14th International Real-Time Ada Workshop, IRTAW-14 (2009)
2. Baruah, S.K., Fisher, N.: Global fixed-priority scheduling of arbitrary-deadline sporadic task systems. In: Rao, S., Chatterjee, M., Jayanti, P., Murthy, C.S.R., Saha, S.K. (eds.) ICDCN 2008. LNCS, vol. 4904, pp. 215–226. Springer, Heidelberg (2008)
3. Baruah, S.K., Baker, T.P.: Schedulability analysis of global EDF. Real-Time Systems 38(3), 223–235 (2008)
4. Andersson, B., Bletsas, K.: Sporadic multiprocessor scheduling with few preemptions. In: 20th Euromicro Conference on Real-Time Systems, pp. 243–252 (2008)
5. Aldea Rivas, M., González Harbour, M.: POSIX-compatible application-defined scheduling in MaRTE OS. In: 14th Euromicro Conference on Real-Time Systems, pp. 67–75 (2002)
6. Aldea Rivas, M., Miranda González, F.J., González Harbour, M.: Implementing an application-defined scheduling framework for ada tasking. In: Llamosí, A., Strohmeier, A. (eds.) Ada-Europe 2004. LNCS, vol. 3063, pp. 283–296. Springer, Heidelberg (2004)

7. Taft, S., Duff, R., Brukardt, R., Ploedereder, E., Leroy, P. (eds.): Ada 2005 Reference Manual: Language and Standard Libraries. Springer, Heidelberg (2005) ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1
8. Wellings, A.J., Burns, A.: Real-time utilities for Ada 2005. In: Abdennahder, N., Kordon, F. (eds.) Ada-Europe 2007. LNCS, vol. 4498, pp. 1–14. Springer, Heidelberg (2007)
9. IEEE Std 1003.1b-1993: IEEE Standard for Information Technology: Portable Operating Sytem Interface (POSIX). Part 1, system application program interface (API) — amendment 1 — realtime extension (1994)
10. Derr, S., Jackson, P., Lameter, C., Menage, P., Seto, H.: Cpusets. Technical report ftp.kernel.org Documentation/cgroups/cpusets.txt
11. Molnar, I., Krasnyansky, M.: SMP IRQ affinity. Technical report ftp.kernel.org Documentation/IRQ-affinity.txt