UNIVERSIDAD POLITECNICA DE VALENCIA

Preliminary Multiprocessor Support of Ada 2012 in GNU/Linux Systems

Sergio Sáez **<ssaez@disca.upv.es>**
Alfons Crespo <alfons@disca.upv.es>

Instituto de Automática e Informática Industrial
Universidad Politécnica de Valencia

Outline

- Introduction
- Objectives
- Multiprocessor Task Scheduling
- CPU Clocks and Timers
- Interrupt Affinities
- Timing Events
- Conclusions

Introduction

Ada multiprocessor support

- Ada 2005 allows real-time applications to be executed on multiprocessor platforms.
- No direct support is provided to allow the programmer to control the task-to-processor mapping process.
- No information or control is provided to determine the execution processor of timer or interrupt handlers.

Operating System multiprocessor support

- There are no standard API to control task-to-processor assignment.
- GNU/Linux provides a specific API and system tools to control thread and interrupt processor affinities.

Main goals

A predictable behaviour of Ada real-time applications over multiprocessor platforms

To allow the Ada programmer to control the processor assignment of any executable unit

- Support for different multiprocessor task scheduling approaches.
- Control over timer and interrupt handlers execution processor.

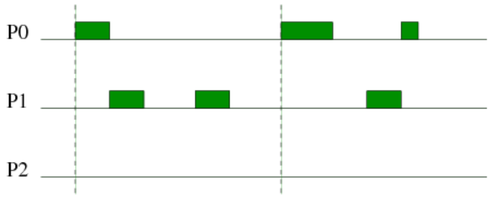
To analyse the required support from the GNU/Linux Operating System point of view

- Current kernel system call support.
- Required extension at kernel and library level.

Multiprocessor Task Scheduling

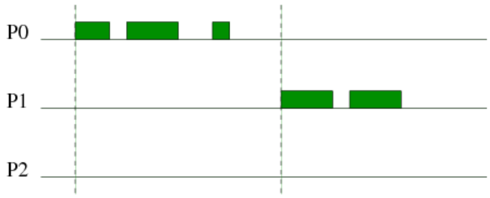
Global scheduling

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



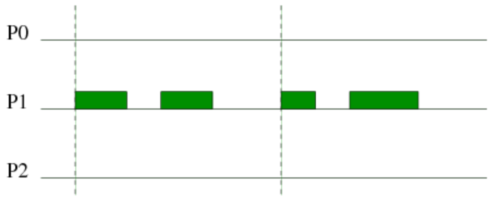
Job partitioning

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



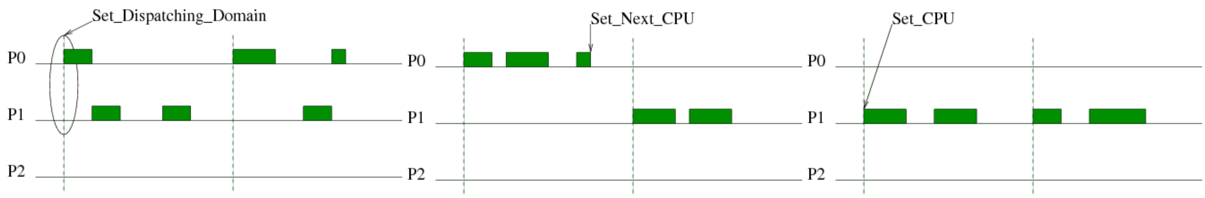
Task partitioning

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



Required functionalities

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



Ada Programming Interface

Task partitioning

```
package System.Multiprocessors is
  type CPU_Range is range 0 .. <implementation-defined>;
  subtype CPU is CPU_Range range 1 .. CPU_Range'last;
  ...
end System.Multiprocessors;

with Ada.Task_Identification; use Ada.Task_Identification;
with System.Multiprocessors; use System.Multiprocessors;
package System.Multiprocessors.Dispatching_Domains is
  ...
  procedure Set_CPU(P: CPU_Range; T : Task_Id := Current_Task);
  function Get_CPU(T : Task_Id := Current_Task) return CPU_Range;
end System.Multiprocessors.Dispatching_Domains;
```

- It allows to specify the execution processor of a given task
- If the current task invokes `Set_CPU` procedure the processor switch is performed immediately.
  - It allows the implementation of task splitting approaches.

Ada Programming Interface (cont'ed)

Restricted global scheduling

```
package System.Multiprocessors.Dispatching_Domains is
  type Dispatching_Domain is limited private;

  function Create(First, Last: CPU) return Dispatching_Domain;

  procedure Assign_Task(DD: in out Dispatching_Domain;
                        P : CPU_Range;
                        T : Task_Id := Current_Task);
  function Get_Dispatching_Domain(T : Task_Id := Current_Task)
                                             return Dispatching_Domain;
  ...
end System.Multiprocessors.Dispatching_Domains;
```

- It allows any task to join a dispatching domain, restricting the global scheduling policy to the corresponding processor subset.
- It could be used to partition available processors for real-time and non real-time purposes.

Ada Programming Interface (cont'ed)

Job partitioning support

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

- It collides with `Delay_Until_And_Set_Deadline` procedure already present in Ada 2005.

```
procedure Set_Next_CPU(P: CPU_Range; T : Task_Id := Current_Task);
```

- It establishes the next processor to be used after the next scheduling point.
  - It could be implement to defer processor assignment until the next `delay` construction.
- This approach could also be used with other attributes as an alternative to `Delay_Until_And_Set_`*`Something`* procedures.

```
procedure Set_Next_Deadline(D: in Deadline; T: in Task_Id := Current_Task);
procedure Set_Next_Priority(P: in Priority; T: in Task_Id := Current_Task);
```

Job partitioning example

Periodic task with job partitioning based on `delay until`.

```
with Ada_System; use Ada_System;
with System.Multiprocessors.Dispatching_Domains;
use System.Multiprocessors.Dispatching_Domains;
task body Periodic_With_Job_Partitioning is
   type List_Range is mod N;
   CPU_List     : array (List_Range) of CPU_Range := (...); -- Decided at design time
   CPU_Iter     : List_Range := List_Range'First;
   Next_CPU     : CPU_Range;
   Next_Release : Ada.Real_Time.Time;
   Period       : Time_Span := ...;
begin
   Task_Initialise;
   Next_Release := Ada.Real_Time.Clock;
   Set_CPU(CPU_List(CPU_Iter)); -- Processor for first activation
   loop
      Task_Main_Loop;
      -- Next job preparation
      CPU_Iter := CPU_Iter'Succ;
      Next_CPU := CPU_List(CPU_Iter);
      Next_Release := Next_Release + Period;
      Set_Next_CPU(Next_CPU);   -- Set the processor for the next job
      delay until Next_Release; -- Delay until next job activation
   end loop;
end Periodic_With_Job_Partitioning;
```

GNU/Linux operating system support

Current functionalities

```
#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);
```

- `sched_setaffinity` allows to specify a subset of processors to be used by `pid` process among the available ones.
  - As the Linux kernel has a different `pid` for each thread, called *thread ID* (`gettid(2)`), this function can also be used for specifying the processor affinity of any thread.
- As describes by Linux manual pages:
  If the process specified by pid is not currently running on one of the CPUs specified in mask, then that process is migrated to one of the CPUs specified in mask.

Linux kernel and glibc library extensions

To allow the job partitioning approach some extensions are required at kernel and library level.

Proposed extension of the `sched_setaffinity` system call.

```
#define SCHED_SET_IMMEDIATE   1
#define SCHED_SET_DEFERRED    2

long sched_setaffinity(pid_t pid, const struct cpumask *in_mask, const long flag);
```

- If `flag` is set to `SCHED_SET_DEFERRED`, then the internal kernel function `migrate_task` is not invoked and processor migration is postponed until the thread becomes suspended.

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);
```

- To maintain backward compatibility at library level, the current library function `sched_setaffinity` will use the new implementation of the system call with `SCHED_SET_IMMEDIATE` flag activated.

CPU Clocks and Timers

- Ada 2005 introduces CPU clocks for single and groups of tasks in the `Ada.Execution_Time` package and its child packages.
- GNAT GPL 2009 does not implement CPU clocks in the native RTS for the GNU/Linux platform.
- However GNU/Linux OS implements the POSIX API for CPU clocks and timers, although *group budgets* are not supported.

```
/* CPU clocks support */
#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);

/* Timer support */
#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);
```

CPU Clocks and Timers (cont'ed)

- However, Ada 2005 does not provided any control about the execution processor of the timer handler.

Proposed Execution Time Timers extension

```
with Ada_System; use Ada_System;
with System.Multiprocessors.Dispatching_Domains;
use System.Multiprocessors.Dispatching_Domains;
package Ada.Execution_Time.Timers is
  ...
  procedure Set_Dispatching_Domain(TM : in out Timer;
                                   DD: access all Dispatching_Domain);
  function Get_Dispatching_Domain(TM : Timer) return Dispatching_Domain;
  procedure Set_CPU(TM : in out Timer; P: CPU_Range);
  function Get_CPU(TM : Timer) return CPU_Range;
end Ada.Execution_Time.Timers;
```

- It allows to specify the processor or group of processors where the timer handler will be executed.
- The default processor affinity of the timer handler can be inherited from the task to be monitored.

Implementation over GNU/Linux systems

- Upon timer creation, the Linux kernel allows to specify how the caller should be notified when the timer expires within the `sigevent` structure.

```
int timer_create(clockid_t clockid, struct sigevent *evp, timer_t *timerid);
```

- A Linux-specific value of the `sigev_notify` field of this structure (`SIGEV_THREAD_ID`) allows to send the specified signal to a given thread when the timer expires.
- This notification facility can be used by the Ada RTS to create a set of server tasks that manage timer expiration on a per-processor or per-dispatching domain basis.
- The notification thread will directly depend on the target processor specified for the timer handler execution.
- Information about the handler to be executed can be attached to a real-time signal, if required.

Interrupt Affinities

- It could desirable to control in which processor an interrupt handler will be executed.
  - It will allow not only to attach real-time related interrupts to specific processors, but also to move away non-real-time interrupts from processors that are executing real-time tasks.
- Ada 2005 also lacks of support for interrupt affinities under multiprocessor platforms.

Explicit multiprocessor support for Ada Interrupts

```
with Ada_System; use Ada_System;
with System.Multiprocessors.Dispatching_Domains;
use System.Multiprocessors.Dispatching_Domains;
package Ada.Interrupts is
  ...
  procedure Set_Dispatching_Domain(Interrupt : Interrupt_ID;
                                   DD: Dispatching_Domain);
  function Get_Dispatching_Domain(Interrupt : Interrupt_ID)
                                  return Dispatching_Domain;
  procedure Set_CPU(Interrupt : Interrupt_ID; P: CPU_Range);
  function Get_CPU(Interrupt : Interrupt_ID) return CPU_Range;
end Ada.Interrupts;
```

Hardware interrupts over GNU/Linux systems

- To support hardware interrupts the package `Ada.Interrupt.Names` needs to be extended with new interrupt identifiers.
  - As HW interrupt numbers change from one system to another, a generic interrupt identifiers could be defined.

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

- In Linux systems, the processor affinity of a hardware interrupt `N` can be established by writing the processor mask value on `/proc/irq/IRQN/smp_affinity` file.
- However, no interrupt handler can be defined in an Ada application for a hardware interrupt.
- Ada interrupt handlers in GNU/linux systems are limited to POSIX signal handlers.

Signal interrupts over GNU/Linux systems

- POSIX does not allow to specify the thread within a process that will receive a given signal.
- However, the Ada RTS can use `pthread_sigmask` function to block the signals that are mapped as Ada interrupts in every application thread.

```
#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);
```

- Then synchronous wait for signals can be performed by a set of *signal server* threads, each one attached to a different processor.
- Each time an interrupt handler is attached to a given processor by means of `Set_CPU`, the signal mask of the *signal server* allocated in that processor is modified accordingly.
  - If an interrupt is not attached to a specific processor in its dispatching domain, then the signal mask of each signal server in that dispatching domain will accept that signal.

Timing Events

- Finally, the last Ada event handlers to consider are *Timing Events*, with an Ada interface similar to the ones shown previously.

Multiprocessor support for Timing Events

```
with Ada_System; use Ada_System;
with System.Multiprocessors.Dispatching_Domains;
use System.Multiprocessors.Dispatching_Domains;
package Ada.Real_Time.Timing_Events is
  ...
  procedure Set_Dispatching_Domain(TM : in out Timing_Event;
                                   DD: access all Dispatching_Domain);
  function Get_Dispatching_Domain(TM : Timing_Event) return Dispatching_Domain;
  procedure Set_CPU(TM : in out Timing_Event; P: CPU_Range);
  function Get_CPU(TM : Timing_Event) return CPU_Range;
end Ada.Real_Time.Timing_Events;
```

- To support multiprocessor platforms, an event-driven server task can be allocated on each 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.

Conclusions

- Some of the proposed extensions of Ada 2012 for multiprocessor platforms have been analysed.
- 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, the support of the presented features has been considered feasible for its implementation at Ada RTS, C library and kernel level.