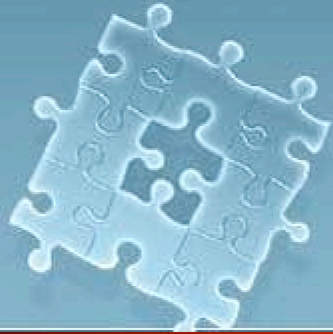


Real-Time Issues

Alan Burns, University of York, UK

Overview

- Ada 95
- Ravenscar
- Task termination
- Execution time clocks and timers
- Timing events
- Dynamic ceiling priorities for protected objects
- Additional scheduling/dispatching

Ada 95

- Flexible concurrency facilities
- Protected objects for efficient synchronisation
  - With requeue for increased expressive power
- Fixed priority scheduling
- Priority ceiling protocol
- Calendar and real-time clocks
- ATC
  - With abort deferred regions

The Ravenscar Profile

- A subset of the Ada tasking model
  - Silent on the sequential part of the language
- Restrictions designed to meet the real-time community requirements for
  - Determinism
  - Schedulability analysis
  - Memory-boundedness
  - Execution efficiency and small footprint
  - Suitability for certification

The Ravenscar Profile

- The Ravenscar Profile is a powerful catalyst for the promotion of simple and effective language-level concurrency
  - Crucial to critical applications
  - One end of the road to greater expressive power

Setting a Profile

- A profile is specified by a new pragma
  - **pragma** Profile(policy);
- Ravenscar set by
  - **pragma** Profile(Ravenscar);
  - It is equivalent to some other pragmas plus lots of restrictions

The Ravenscar Profile

- **The profile corresponds to:**

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions ( ... );
```

- Detect_Blocking is new – indicates that blocking in a protected object must be detected
- There are many new Restrictions identifiers

Ravenscar

- Profile uses a set of existing Restrictions
  - Max_Task_Entries => 0
  - Max_Protected_Entries => 1
  - No_Abort_Statements
  - No_Dynamic_Priorities
  - No_Implicit_Heap_Allocations
  - No_Task_Allocators
  - No_Task_Hierarchy
  - No_Asynchronous_Control

    (the last is now No_Dependence => Asynchronous_Task_Control)

Ravenscar

- New restriction identifiers
  - Max_Entry_Queue_Length => 1
  - No_Dependence => Ada.Calendar
  - No_Dynamic_Attachment
  - No_Local_Protected_Objects
  - No_Protected_Type_Allocators
  - No_Relative_Delay
  - No_Requeue_Statements
  - No_Select_Statements
  - No_Dependence => Ada.Task_Attributes
  - No_Task_Termination
  - Simple_Barriers

Ravenscar

- Some new features not allowed
  - No_Local_Timing_Events
  - No_Specific_Termination_Handlers
  - No_Dependence => Execution_Time.Group_Budget
  - No_Dependence => Execution_Time.Timers

Task Termination

- Old problem of tasks having a silent death
- Can monitor cause of death in Ada 2005 and distinguish between
  - Termination caused by unhandled **exception**
  - Termination caused by something nasty such as **abort**
  - Normal termination by running into final **end**
- Monitoring done by a protected procedure called when task dies
- Can specify termination procedure for
  - A specific task
  - The dependents of the current task

Package Ada.Task_Termination

```
package Ada.Task_Termination is
     pragma Preelaborable(Task_Termination);

   type Cause_Of_Termination is
                    (Normal, Abnormal, Unhandled_Exception);
   type Termination_Handler is access protected
          procedure(Cause: in Cause_Of_Termination;
                   T: in Task_Id; X: in Exception_Occurrence);

   procedure Set_Dependents_Fallback_Handler
                        (Handler: in Termination_Handler);
     function Current_Task_Fallback_Handler
                    return Termination_Handler;

   procedure Set_Specific_Handler(T: in Task_Id;
                         Handler: in Termination_Handler);
     function Specific_Handler(T: in Task_Id)
                    return Termination_Handler;
end Ada.Task_Termination;
```

Execution Time Clocks and Timers

- Monitor the task execution time
- Fire an event when a task execution time reaches a specified value
- Allocate and support budgets for groups of tasks

Monitoring Task Execution Time

- Every task has an execution time clock
- Clock starts subsequent to creation
- Clock counts up whenever the task executes
- Accuracy, metrics and implementation requirements are defined

Monitoring Task Execution Time (cont'd)

```
with Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
package Ada.Execution_Time is

   type CPU_Time is private;
   CPU_Time_First : constant CPU_Time;
   CPU_Time_Last  : constant CPU_Time;
   CPU_Time_Unit  : constant :=
         implementation-defined-real-number;
   CPU_Tick : constant Time_Span;

   function Clock
     (T : Ada.Task_Identification.Task_ID
          := Ada.Task_Identification.Current_Task)
     return CPU_Time;

   -- Subprograms for + etc, < etc, Split and Time_Of.

private
   ... -- Not specified by the language.
end Ada.Execution_Time;
```

Execution Timers

- In fault tolerance and other high integrity applications there is a need to catch task overruns
- For some algorithms a fixed time is allocated to a task for an iterative process
- Basic model is to define:
  - A *timer* that is enabled, and
  - A *handler* that is called (by the run-time) when the execution time of a task become equal to a defined value
- The handler is an access to protected procedure

Execution Timers (cont'd)

```
package Ada.Execution_Time.Timers is

   type Timer(T : not null access constant
      Ada.Task_Identification.Task_Id) is tagged limited private;
   type Timer_Handler is access protected procedure (TM : in out Timer);

   Min_Handler_Ceiling : constant System.Any_Priority := implementation-defined;

   procedure Set_Handler(TM: in out Timer;
                         In_Time : Time_Span;
                         Handler : Timer_Handler);
   procedure Set_Handler(TM: in out Timer;
                         At_Time : CPU_Time;
                         Handler : Timer_Handler);
   function Current_Handler(TM : Timer) return Timer_Handler;
   procedure Cancel_Handler(TM : in out Timer; Cancelled : out Boolean);

   function Time_Remaining(TM : Timer) return Time_Span;

   Timer_Resource_Error : exception;
private
   ...
end Ada.Execution_Time.Timers;
```

Task Group Budgets

- A number of schemes, including those that use servers allow a group of tasks to share a budget
- The budget is usually replenished periodically
- The scheme supports firing a handler when budget goes to zero
  - the tasks are not prevented form executing
  - but this can be programmed
  - or priorities changes to background, or whatever…

Task Group Budgets (cont'd) (member operations)

```
package Ada.Execution_Time.Group_Budgets is
  type Group_Budget is tagged limited private;

  type Group_Budget_Handler is access protected
          procedure(GB : in out Group_Budget);

  type Task_Array is array(Natural range <>) of
                         Ada.Task_Identification.Task_ID;

  Min_Handler_Ceiling : constant System.Any_Priority :=
     implementation-defined;

  procedure Add_Task(GB: in out Group_Budget;
                     T : Ada.Task_Identification.Task_ID);
  procedure Remove_Task(GB: in out Group_Budget;
                        T : Ada.Task_Identification.Task_ID);
  function Is_Member(GB: Group_Budget;
        T : Ada.Task_Identification.Task_ID) return Boolean;
  function Is_A_Group_Member(
        T : Ada.Task_Identification.Task_ID) return Boolean;
  function Members(GB: Group_Budget) return Task_Array;
  ...
```

Task Group Budgets (cont'd) (timing operations)

```
    ...
    procedure Replenish (GB: in out Group_Budget; To : Time_Span);
    procedure Add(GB: in out Group_Budget; Interval : Time_Span);
    function Budget_Has_Expired(GB: Group_Budget) return Boolean;
    function Budget_Remaining(GB: Group_Budget) return Time_Span;

    procedure Set_Handler(GB: in out Group_Budget;
                          Handler : Group_Budget_Handler);
    function Current_Handler(GB: Group_Budget)
                             return Group_Budget_Handler;
    procedure Cancel_Handler(GB: in out Group_Budget;
                             Cancelled : out Boolean);

    Group_Budget_Error : exception;
  private
      -- Not specified by the language.
  end Ada.Execution_Time.Group_Budgets;
```

Timing Events

- A means of defining code that is executed at a future point in time
- Does not need a task
- Similar in notion to interrupt handing (time itself generates the interrupt)
- Again a handler is used

Timing Events (cont'd)

```
package Ada.Real_Time.Timing_Events is
  type Timing_Event is tagged limited private;
  type Timing_Event_Handler
       is access protected procedure(Event : in out Timing_Event);
  procedure Set_Handler(Event : in out Timing_Event;
                        At_Time : Time;
                        Handler: Timing_Event_Handler);
  procedure Set_Handler(Event : in out Timing_Event;
                        In_Time: Time_Span;
                        Handler: Timing_Event_Handler);
  function Is_Handler_Set(Event : Timing_Event)
                          return Boolean;
  function Current_Handler(Event : Timing_Event)
                           return Timing_Event_Handler;
  procedure Cancel_Handler(Event : in out Timing_Event;
                           Cancelled : out Boolean);
  function Time_Of_Event(Event : Timing_Event) return Time;
private
  ... -- Not specified by the language.
end Ada.Real_Time.Timing_Events;
```

Example of Usage

```
protected Watchdog is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  entry Alarm_Control;
     -- Called by alarm handling task.
  procedure Timer(Event : in out Timing_Event);
     -- Timer event code.
  procedure Call_in;
     -- Called by application code every 50ms if alive.
private
  Alarm : Boolean := False;
end Watchdog;

Fifty_Mil_Event : Timing_Event;
TS : Time_Span := Milliseconds(50);

Set_Handler(Fifty_Mil_Event, TS, Watchdog.Timer'Access);
```

Example of Usage (cont'd)

```
protected body Watchdog is
  entry Alarm_Control when Alarm is
  begin
    Alarm := False;
  end Alarm_Control;

  procedure Timer(Event : in out Timing_Event) is
  begin
    Alarm := True;
  end Timer;

  procedure Call_in is
  begin
    Set_Handler(Fifty_Mil_Event, TS, Watchdog.Timer'access);
    -- Note, this call to Set_Handler cancels the previous call.
  end Call_in;
end Watchdog;
```

Dynamic Ceilings

- A new attribute for any protected object: `'Priority`
- This attribute can only be read and assigned to within the body of a protected object
- The effect of any change to the ceiling of the protected object takes effect at the end of the current protected action

Changing Ceiling Priority

```
protected type PT is
    procedure Change_Priority(Change: in Integer);
    ...
  end;

protected body PT is
    procedure Change_Priority(Change: in Integer) is
    begin
       ...               -- PT'Priority has old value here
       PT'Priority := PT'Priority + Change;
       ...               -- PT'Priority has new value here
       ...
    end Change_Priority;
    ...
  end PT;
```

- Note raw assignment

Scheduling and Dispatching

- Ada 95 provides a complete and well defined set of language primitives for fixed priority scheduling
- But fixed priority scheduling is not the only scheme of interest
- Ada 2005 defines four schemes
  - FIFO_Within_Priorities
  - Non_Preemptive_FIFO_Within_Priorities
  - Round_Robin_Within_Priorities
  - EDF_Across_Priorities

Dispatching Package

```
package Ada.Dispatching is
  pragma Pure(Dispatching);
  Dispatching_Policy_Error : exception;
end Ada.Dispatching;
```

Round Robin

- A common policy for non-real-time systems and real-time schemes requiring a level of fairness
- A simple scheme with the usual semantics
- If the defined quantum is exhausted during the execution of a protected action then the task involved continues executing until the protected action is complete

Round Robin (cont'd)

```
with System;
with Ada.Real_Time;
package Ada.Dispatching.Round_Robin is
  Default_Quantum : constant Ada.Real_Time.Time_Span :=
                             implementation-defined;
  procedure Set_Quantum(Pri : System.Priority;
                        Quantum : Ada.Real_Time.Time_Span);
  procedure Set_Quantum(Low,High : System.Priority;
                        Quantum : Ada.Real_Time.Time_Span);
  function Actual_Quantum (Pri : System.Priority)
                           return Ada.Real_Time.Time_Span;
  function Is_Round_Robin (Pri : System.Priority)
                           return Boolean;
end Ada.Dispatching.Round_Robin;
```

Deadlines and EDF Scheduling

- The deadline is the most significant notion in real-time systems
- EDF – Earliest Deadline First is the scheduling policy of choice in many domains
- It makes better use of processing resources
- EDF or fixed-priority?
  - a long and detailed debate
  - but in reality both are needed

Support for Deadlines

- Introduction of a new library package
- Relative deadline means relative to task's release
  - complete talk in 30 minutes
- Absolute deadline means point on time line
  - complete talk by 12.30
- Usually deadline means absolute deadline

Support for Deadlines (cont'd)

```
with Ada.Task_Identification;
use Ada.Task_Identification;
with Ada.Real_Time;
package Ada.Dispatching.EDF is
   subtype Deadline is Ada.Real_Time.Time;
   Default_Deadline : constant Deadline :=
     Ada.Real_Time.Time_Last;
   procedure Set_Deadline(D : Deadline;
                          T : Task_ID := Current_Task);
   procedure Delay_Until_And_Set_Deadline
               (Delay_Until_Time : Ada.Real_Time.Time;
                TS : Ada.Real_Time.Time_Span);
   function Get_Deadline
               (T : Task_ID := Current_Task) return Deadline;
end Ada.Dispatching.EDF;
```

Catching a Deadline Overrun

```
loop
  select
     delay until Deadlines.Get_Deadline;
     -- Deal with deadline overrun.
  then abort
     -- Code.
  end select;
  -- Set next release condition
  -- and next absolute deadline.
end loop;
```

EDF Scheduling

- Need to define EDF ordered ready queues
- Need to support preemption-level locking for effective use of protected objects
  - Ideally uses existing PO locking
  - Ideally can be used with fixed priority scheduling

Baker's Protocol

- Under Fixed Priority scheduling, priority is used for:
  - Dispatching
  - Controlled access to resources e.g. POs
- Under Baker's protocol
  - Dispatching is controlled by absolute deadline
  - Preemption levels used for resources

Baker's Protocol

- Basic algorithm
  - A newly released task can preempt the currently executing task iff:
    - Its deadline is earlier
    - Its preemption-level is greater than that of the highest locked resource

Bounding Blocking

- If preemption levels are assigned according to relative deadline then we can have:
  - Deadlock free execution
  - Maximum of one block per invocation
- Hence same properties as priority ceiling protocol for FP systems
  - i.e., Ada’s existing model for POs

Dispatching Rules for EDF

- Use a task's base priority to represent preemption level
- Assigned PO ceiling priorities (preemption levels) in the usual way
  - execution within a PO is at ceiling level
- Order ready queues by absolute deadline

Which Queue to Join?

- Define a ready queue at priority level p as being busy if a task has locked a PO with ceiling p – denote this task as T(p)
- A newly released task S is added to highest priority busy ready queue p such that deadline of S is earlier than T(p) and base priority of S is greater than p
- If no p exist put S on `Priority'First`

Properties

- Task S is always placed on a priority level below that of the ceiling priority of any PO it uses
- Implements Baker’s protocol
- Splitting the priority range into FP and EDF allows both to work together

Example

- Following slide has one cyclic task of a simple system of 5 tasks with preemption levels 1..5
- Dispatched by:

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
```

Example (cont'd)

```
protected X is – one of 3 POs
  pragma Priority(5);
  -- Definitions of subprograms.
private
  -- Definition of internal data.
end X;

task A is
  pragma Priority(5);      -- Period and
end A;                     -- relative deadline equal to 10ms.

task body A is
    Next_Release: Ada.Real_Time.Time;
begin
    Next_Release := Ada.Real_Time.Clock;
    loop
      -- Code, including call(s) to X.
      Next_Release := Next_Release +
         Ada.Real_Time.Milliseconds(10);
      delay until Next_Release;
    end loop;
end A;
```

Example (cont'd)

```
task A is
  pragma Priority(5);
  pragma Relative_Deadline(10);
end A;

task body A is
    Next_Release: Ada.Real_Time.Time;
begin
    Next_Release := Ada.Real_Time.Clock;
    loop
      -- Code, including call(s) to X.
      Next_Release := Next_Release +
         Ada.Real_Time.Milliseconds(10);
      Deadlines.Set_Deadline(Next_Release +
         Ada.Real_Time.Milliseconds(10));
      delay until Next_Release;
    end loop;
end A;
----------
pragma Task_Dispatching_Policy
               (EDF_Across_Priorities);
```

Example (cont'd)

```
task body A is
    Next_Release: Ada.Real_Time.Time;
begin
    Next_Release := Ada.Real_Time.Clock;
    loop
        -- code, including call(s) to X
      Next_Release := Next_Release +
         Ada.Real_Time.Milliseconds(10);
      Deadline.Delay_Until_And_Set_Deadline
         (Next_Release,
         Ada.Real_Time.Milliseconds(10));
end A;
```

Mixed Dispatching

- Ada now allows different dispatching policies to be used together in a controlled and predictable way
- Protected object can be used to communicate across policies

```
pragma Priority_Specific_Dispatching(policy_identifier,
  first_priority_expression, last_priority_expression);
```

High Priority

FIFO

FIFO

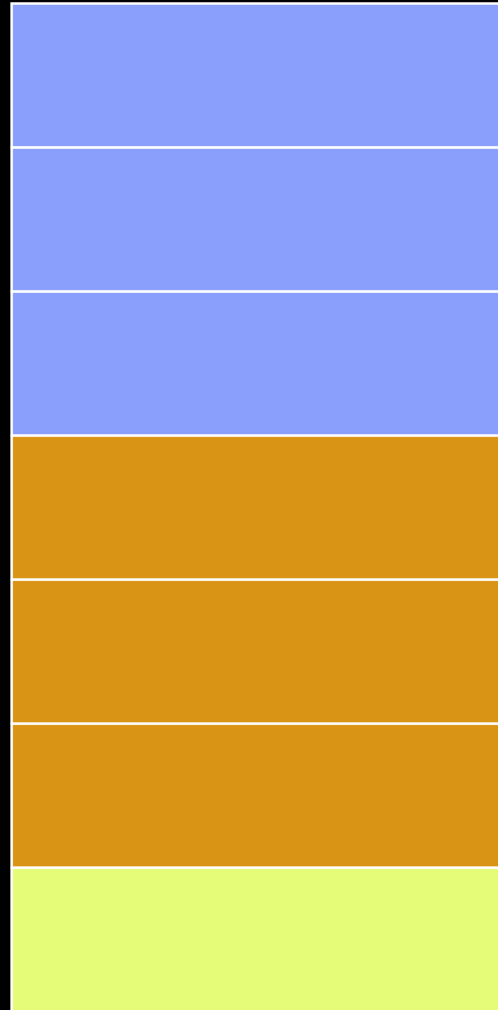
FIFO

EDF

EDF

EDF

RR



Low Priority

Splitting the Priority Range

```
pragma Priority_Specific_Dispatching
          (Round_Robin_Within_Priority,1,1);

pragma Priority_Specific_Dispatching
          (EDF_Across_Priorities,2,10);

pragma Priority_Specific_Dispatching
          (FIFO_Within_Priority,11,24);
```

Conclusions

- Ada 2005 significantly extends the facilities available for programming real-time systems
  - Ravenscar, execution time control, timing events, dispatching
- The requirements for these changes have come from the series of International Real-Time Ada Workshops
- Ada is now considerably more expressive in this area than any other programming language in the galaxy