4.a Programming real-time systems (in Ada)

Where we see how program code may be written to conform to the real-time systems theory, and we present coding patterns that ensure by-design conformance

Making code match theory /1

- Static set of tasks
  - Tasks as programmatic entities declared at the outermost scope of the program's main
    - Cannot go out of scope before the program ends
- Tasks issue jobs repeatedly
  - Task duty cycle: activation, {execution, suspension}
    - Tasks have a single source of activation (release event)
    - The job is the procedural body of the task's main loop
- Real-time attributes
  - Release time
    - Periodic: at every T time units
    - Sporadic: at least T time units between any two subsequent releases
  - Execution
    - Worst case execution time (WCET) of the job, assumed to be known
    - Deadline: D time units after release

Making code match theory /2

- Task interaction
  - Shared variables with mutually exclusive access
    - Protected objects (PO) with procedures and functions
  - Conditional synchronization solely for sporadic activation
    - PO with a single entry
  - Ceiling priority protocol for access to shared objects
    - `Ceiling_Locking` policy
- Scheduling model
  - Fixed-priority pre-emptive
    - FIFO within priorities

Protected objects /1

```
protected type Shared_Integer (Initial_Value : Integer) is
 function Read return Integer;
 procedure Write (Value : Integer);
private
 The_Integer : Integer := Initial_Value;
end Shared_Integer;
```

```
protected body Shared_Integer is
 function Read return Integer is
 begin
  return The_Integer;
 end Read;
 procedure Write (Value : Integer) is
 begin
  The_Integer := Value;
 end Write;
end Shared_Integer;
```

Concurrent

Mutually-exclusive (exclusion synchronization)

Protected objects /2

```
Buffer_Size : constant Positive := 5;
type Index is mod Buffer_Size; -- tipo modulare
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer_T is array (Index) of Any_Type;

protected type Bounded_Buffer is
   entry Get (Item :    out Any_Type);
   procedure Put (Item : in     Any_Type);
private
   First : Index := Index'First; -- 0
   Last  : Index := Index'Last;  -- 4
   In_Buffer : Count := 0;
   Buffer : Buffer_T;
end Bounded_Buffer;
```

```
s
_Type)

e pointer

ee from overflow
;

_Type)
      when In_Buffer < Buffer_Size is
   begin --  first move pointer then write
      Last := Last + 1; -- free from overflow
      Buffer(Last) := Item;
      In_Buffer := In_Buffer + 1;
   end Put;
end Bounded_Buffer;
```

Protected objects /3

```
Buffer_Size : constant Positive := 5;
type Index is mod Buffer_Size; -- tipo modulare
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer_T is array (Index) of Any_Type;

protected type Bo
   entry Get (Ite
   entry Put (Ite
private
   First : Index
   Last  : Index
   In_Buffer : Co
   Buffer : Buffe
end Bounded_Buffe
```

```
protected body Bounded_Buffer is
   entry Get (Item :     out Any_Type)
      when In_Buffer > 0 is
   begin -- first read then move pointer
      Item := Buffer(First);
      First := First + 1; -- free from overflow
      In_Buffer := In_Buffer - 1;
   end Get;
   procedure Put (Item : in      Any_Type)
   begin --  overwrite on full
      Last := Last + 1; -- free from overflow
      Buffer(Last) := Item;
      In_Buffer := In_Buffer + 1;
   end Put;
end Bounded_Buffer;
```

Guard

Language profile

- Enforced by means of a configuration directive
  - `pragma Profile (Ravenscar);`
- Equivalent to a set of restrictions plus three additional configuration directives
  - ```
    pragma Task_Dispatching_Policy
        (FIFO_Within_Priorities);
    pragma Locking_Policy (Ceiling_Locking);
    pragma Detect_Blocking;
    ```
- ISO/IEC TR 24718, Guide for the use of the Ada Ravenscar Profile in High Integrity Systems

  http://www.open-std.org/jtc1/sc22/wg9/n424.pdf
    ```

Ravenscar restrictions

```
No_Abort_Statements,
No_Dynamic_Attachment,
No_Dynamic_Priorities,
No_Implicit_Heap_Allocations,
No_Local_Protected_Objects,
No_Local_Timing_Events,
No_Protected_Type_Allocators,
No_Relative_Delay,
No_Requeue_Statements,
No_Select_Statements,
No_Specific_Termination_Handlers,
No_Task_Allocators,
No_Task_Hierarchy,
No_Task_Termination,
Simple_Barriers,
Max_Entry_Queue_Length => 1,
Max_Protected_Entries => 1,
Max_Task_Entries => 0,
No_Dependence => Ada.Asynchronous_Task_Control,
No_Dependence => Ada.Calendar,
No_Dependence => Ada.Execution_Time.Group_Budget,
No_Dependence => Ada.Execution_Time.Timers,
No_Dependence => Ada.Task_Attributes
```

Restriction checking

- Almost all of the Ravenscar profile restrictions can be checked at compile time
- A few can only be checked at run time
  - Potentially blocking operations in protected operation bodies
  - Priority ceiling violation
  - More than one call queued on a protected entry or a suspension object
  - Task termination

Potentially blocking operations

- Protected entry call statement
  - Only used for sporadic releases
- Delay until statement
  - Only used for periodic suspensions
- Call on a subprogram whose body contains a potentially blocking operation
- **`Pragma Detect_Blocking`** requires detection of potentially blocking operations
  - Exception **Program_Error** raised on detection at at run time
  - Blocking need not be detected if it occurs in the domain of a call to a foreign language

Other run-time checks

- Priority ceiling violation (lower than caller)
  - **Program_Error** must be raised
- More than one call waiting on a protected
  - **Program_Error** must be raised
- Task termination
  - Program behavior in that case must be documented
    - Can be silent (bad)
    - May hold the terminating task in a limbo state (unusual)
    - May call an application-defined termination handler defined with the `Ada.Task_Termination` package (C.7.3)

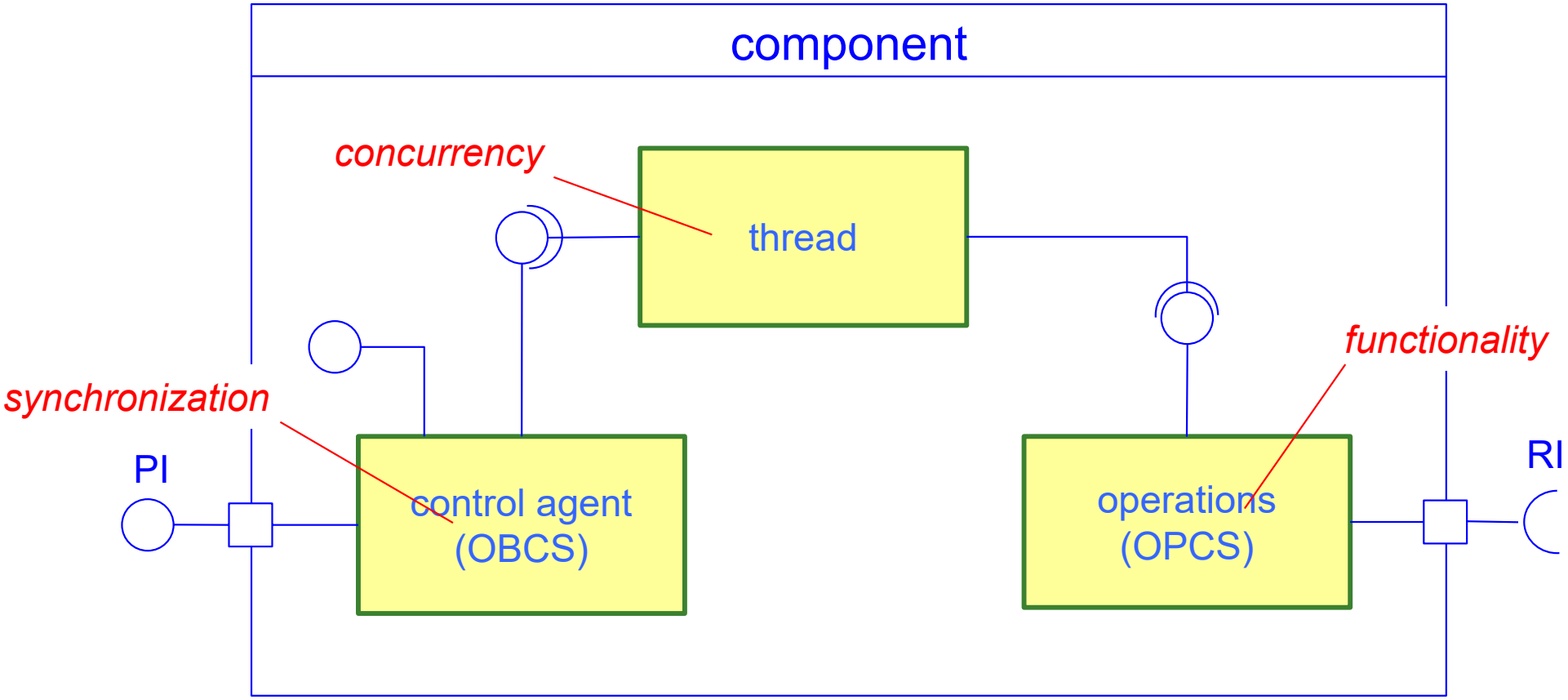
Other restrictions

- Some restrictions on the sequential part of the language may be useful in conjunction with the Ravenscar profile
  - `No_Dispatch`
  - `No_IO`
  - `No_Recursion`
  - `No_Unchecked_Access`
  - `No_Allocators`
  - `No_Local_Allocators`
- ISO/IEC TR 15942, Guide for the use of the Ada Programming Language in High Integrity Systems

An object-oriented approach

- Real-time components are objects
  - Instances of predefined classes
  - They hold an internal state and expose interfaces
    - Provided interface (PI): what can be called from the outside
    - Required interface (RI): what needs to be called outside
- Inversion of control
  - Application code is strictly sequential
  - Real-time concurrency is "injected" by predefined templates
- Based on well-defined code patterns
  - Cyclic & sporadic tasks
  - Protected data
  - Passive data

Component structure



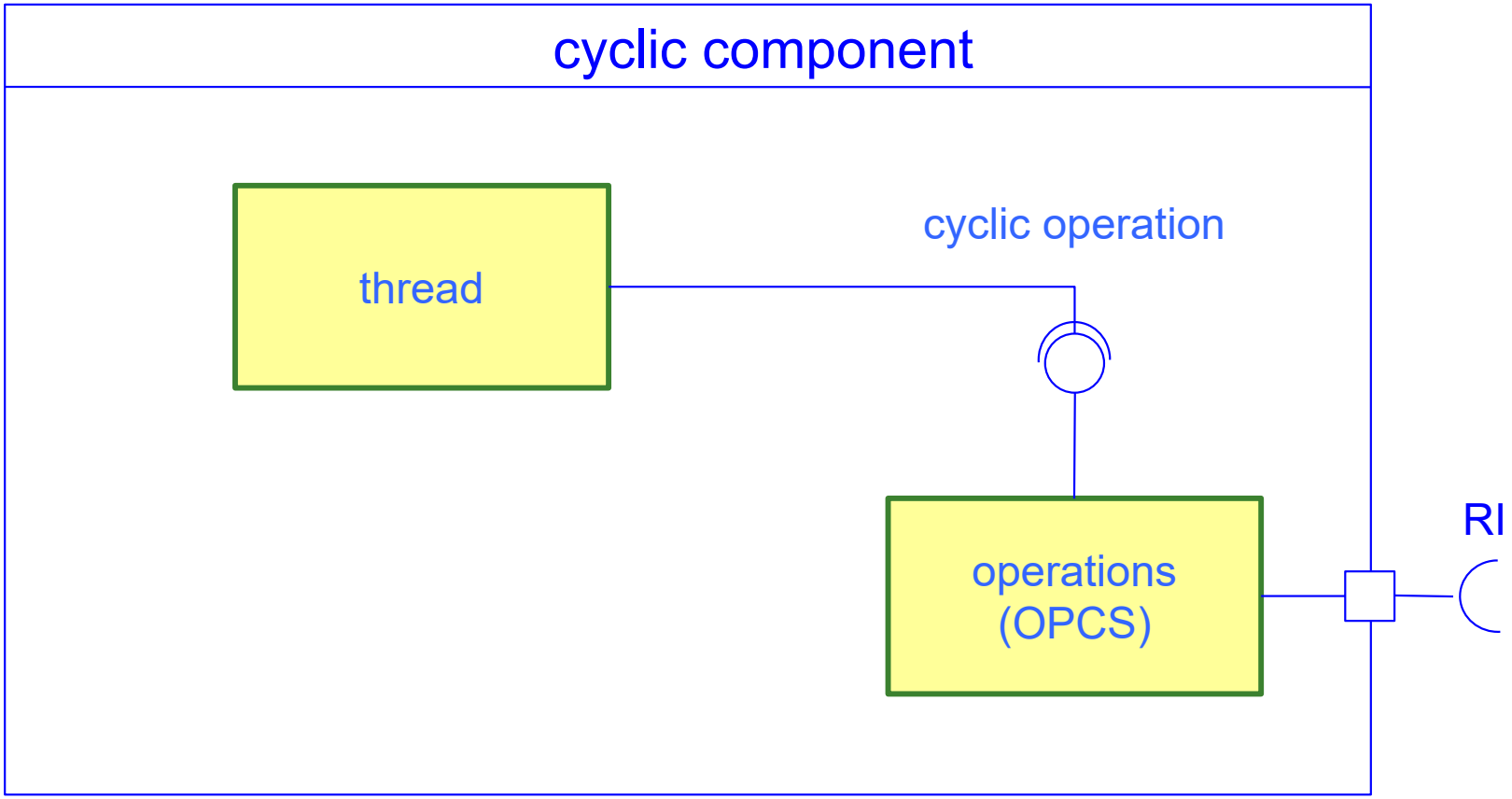
Component taxonomy

- Cyclic component
- Sporadic component
- Protected data component
- Passive component

Cyclic component

- Release event programmed from the system clock
- Attributes
  - Period
  - Deadline
  - Worst-case execution time
- The most basic cyclic code pattern does not need the synchronization agent
  - The system clock delivers the activation event
  - The component behavior is fixed and immutable

Cyclic component (basic)



Cyclic thread (spec)

```
task type Cyclic_Thread
   (Thread_Priority : Priority;
    Period          : Positive) is
  pragma Priority(Thread_Priority);
end Cyclic_Thread;
```

ms

Should be ~~Time~~_Span
but cannot …

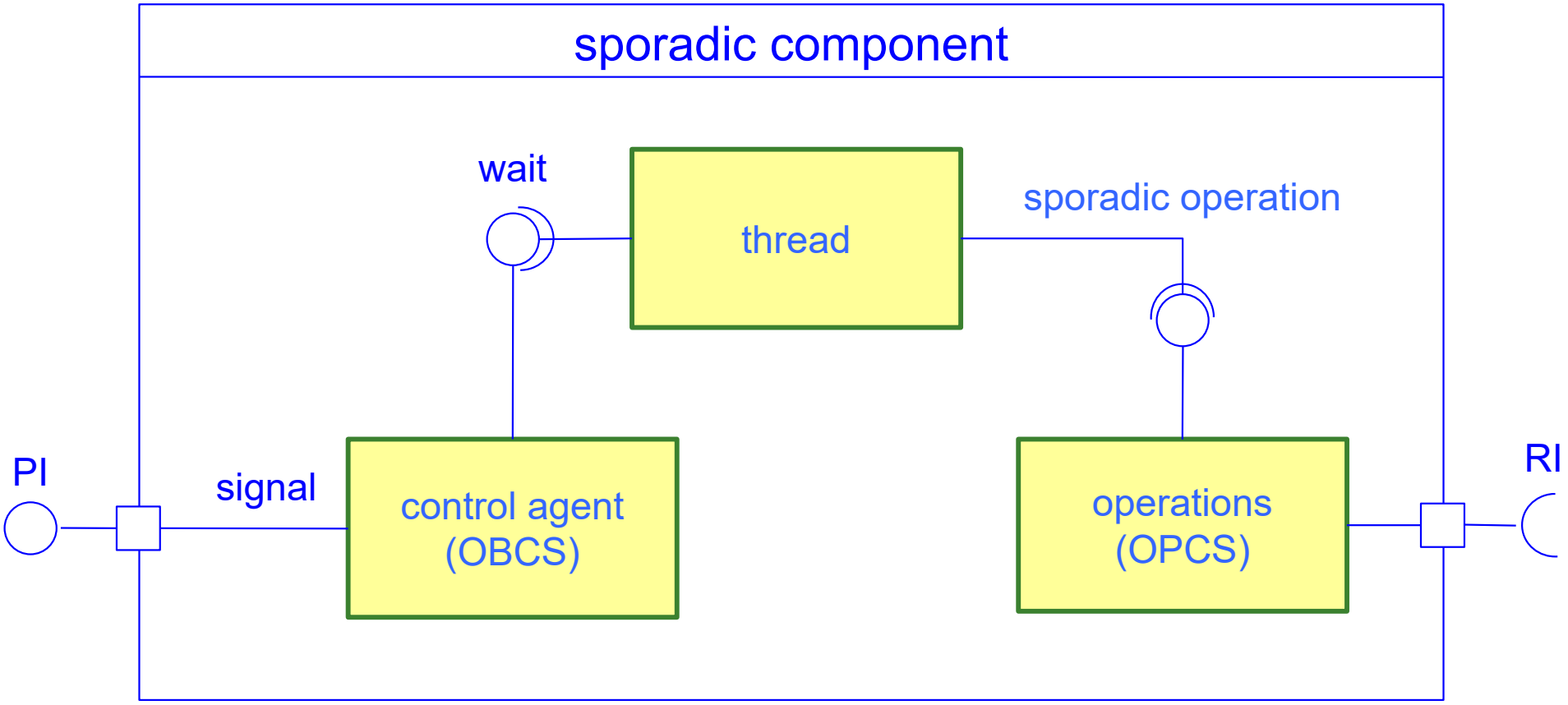
Cyclic thread (body)

```
task body Cyclic_Thread is
  Next_Time : Time := <Start_Time>; --  taken at elaboration time
                                    --+ higher in the system
                                    --+ hierarchy
begin
  loop
    delay until Next_Time; -- so that all tasks start at T0
    OPCS.Cyclic_Operation; -- fixed and parameterless
    Next_Time := Next_Time + Milliseconds(Period);
  end loop;
end Cyclic_Thread;
```

Sporadic component

- Release event caused by software action
  - Signal from another task or a hardware interrupt
- Attributes
  - Minimum inter-arrival time (must be assured!)
  - Deadline
  - Worst-case execution time
- The synchronization agent of the target component is used to deliver (signal) the activation event
  - And to store-and-forward signal-related data (if any)

Sporadic component



Sporadic component (spec)

```
task type Sporadic_Thread(Thread_Priority : Priority) is
  pragma Priority(Thread_Priority);
end Sporadic_Thread;
```

```
protected type OBCS(Ceiling : Priority) is
  pragma Priority(Ceiling);
  procedure Signal;
  entry     Wait;
private
   Occurred : Boolean := False;
end OBCS;
```

A sporadic task is activated by calling the `Signal` operation

Sporadic thread (body)

```
task body Sporadic_Thread is
  Next_Time : Time := <Start_Time>;
begin
  delay until Next_Time; -- so that all tasks start at T0
  loop
    OBCS.Wait;
    OPCS.Sporadic_Operation;
    --  may take parameters if they were delivered by Signal
    --+ and retrieved by Wait
  end loop;
end Sporadic_Thread;
```

Sporadic control agent (body)

```
protected body OBCS is
  procedure Signal is
  begin
    Occurred := True;
  end Signal;
  entry Wait when Occurred is
  begin
     Occurred := False;
  end Wait;
end OBCS;
```

Other components

- **Protected component**
  - No thread, only synchronization and operations
  - Straightforward direct implementation with protected object
- **Passive component**
  - Purely functional behavior, neither thread nor synchronization
  - Straightforward direct implementation with functional package

Temporal properties

- Basic patterns only guarantee periodic activation
- They should be augmented to guarantee additional temporal properties at run time
  - Minimum inter-arrival time for sporadic events
  - Deadline for all types of thread
  - WCET budgets for all types of thread

Minimum inter-arrival time /1

- Violations of the specified separation interval may cause higher interference on lower priority tasks
- The solution is to prevent sporadic threads from being activated earlier than stipulated
  - Compute earliest (absolute) allowable activation time
  - Withhold activation (if triggered) until that time

Sporadic thread with minimum separation (spec)

```
task type Sporadic_Thread
   (Thread_Priority : Priority;
    Separation      : Positive) is
  pragma Priority(Thread_Priority);
end Sporadic_Thread;
```

ms

Minimum inter-arrival time expressed in ms

Sporadic thread (body)

```
task body Sporadic_Thread is
  Release_Time : Time;
  Next_Release : Time := <Start_Time>;
begin
  loop
    delay until Next_Release;
    OBCS.Wait;
    Release_Time := Clock;
    OPCS.Sporadic_Operation;
    Next_Release := Release_Time + Milliseconds(Separation);
  end loop;
end Sporadic_Thread;
```

These three statements notionally still form a *single* point of activation

Critique

- May incur some temporal drift as the clock is read *after* task release
  - Preemption may hit just after release before reading the clock
  - Separation may become *larger* than required
- Better read clock at place and time of task release
  - Within the synchronization agent
  - Which is protected and thus less exposed to general interference

Minimum inter-arrival time /2

```
task body Sporadic_Thread is
  Release_Time : Time;
  Next_Release : Time := <Start_Time>;
begin
  loop
    delay until Next_Release;
    OBCS.Wait(Release_Time);
    OPCS.Sporadic_Operation;
    Next_Release := Release_Time + Milliseconds(Separation);
  end loop;
end Sporadic_Thread;
```

Recording release time /1

```
protected type OBCS(Ceiling : Priority) is
  pragma Priority(Ceiling);
  procedure Signal;
  entry Wait(Release_Time : out Time);
private
   Occurred : Boolean := False;
end OBCS;
```

Recording release time /2

```
protected body OBCS is
  procedure Signal is
  begin
    Occurred := True;
  end Signal;

  entry Wait(Release_Time : out Time) when Occurred is
  begin
    Release_Time := Clock;
    Occurred := False;
  end Wait;
end OBCS;
```

Deadline miss /1

- May result from
  - Higher priority tasks executing more often than expected
    - Can be prevented with inter-arrival time enforcement
  - Overruns in the same or higher priority tasks
    - Programming error in the functional code
    - Inaccurate WCET analysis

Deadline miss /2

- May be detected with the help of **timing events**
  - A mechanism for requiring some application-level action to be executed at a given time
  - Under the Ravenscar Profile, timing events can only exist at library level
- Timing events are statically allocated
- Minor optimization possible for periodic tasks
  - Which however breaks the symmetry of code patterns

Timing events

- Lightweight mechanism for defining code to be executed at a specified time
  - Does not require an application-level task
  - Analogous to interrupt handling
- The code is defined as an event handler
  - An (**access** to) a **protected procedure**
- Directly invoked by the runtime

Ada.Real_Time.Timing events

```
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 : in Time;
                         Handler : in Timing_Event_Handler);
  ...
  procedure Cancel_Handler (Event     : in out Timing_Event;
                            Cancelled : out Boolean);
  ...
end Ada.Real_Time.Timing_Events;
```

Cyclic thread with deadline miss detection (spec)

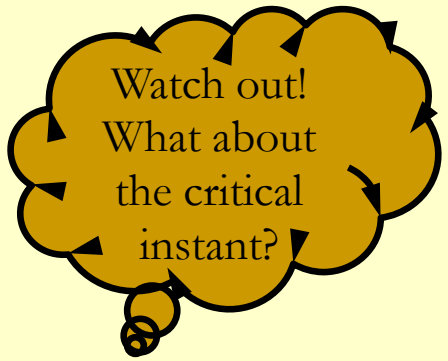
```
task type Cyclic_Thread
   (Thread_Priority : Priority;
    Period          : Positive;
    Deadline        : Positive) is
  pragma Priority(Thread_Priority);
end Cyclic_Thread;
```

Thread body

```
Deadline_Miss : Timing_Event; -- static, local per component
task body Cyclic_Thread is
  Next_Time : Time := <Start_Time>;
  Canceled  : Boolean := False;
begin
  loop
    delay until Next_Time;
    Set_Handler(Deadline_Miss,
                Next_Time + Milliseconds(Deadline),
                Deadline_Miss_Handler); -- application-specific
    OPCS.Cyclic_Operation;
    Cancel_Handler(Deadline_Miss, Canceled);
    Next_Time := Next_Time + Milliseconds(Period);
  end loop;
end Cyclic_Thread;
```

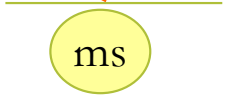
Thread body (streamlined)

```
Deadline_Miss : Timing_Event; -- static, local per component
task body Cyclic_Thread is
  Next_Time : Time := <Start_Time>;
  Canceled  : Boolean := False;
begin
  loop
    -- setting again cancels any previous event
    Set_Handler(Deadline_Miss,
                Next_Time + Milliseconds(Deadline),
                Deadline_Miss_Handler); -- application-specific
    delay until Next_Time;
    OPCS.Cyclic_Operation;
    Next_Time := Next_Time + Milliseconds(Period);
  end loop;
end Cyclic_Thread;
```



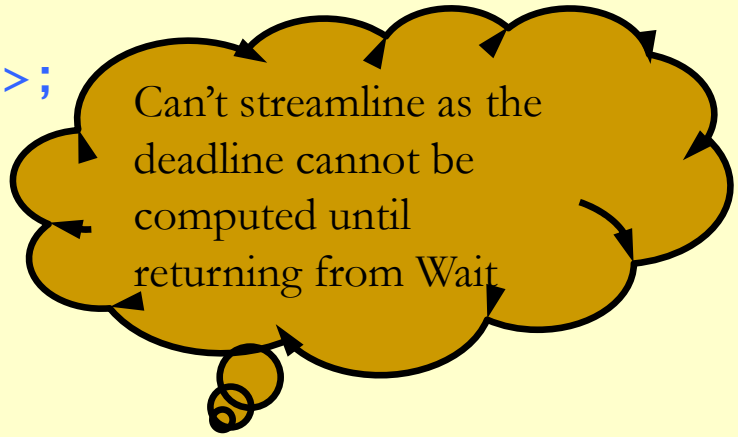
Sporadic thread with deadline miss detection (spec)

```
task type Sporadic_Thread
   (Thread_Priority : Priority;
    Separation      : Positive;
    Deadline        : Positive) is
  pragma Priority(Thread_Priority);
end Sporadic_Thread;
```



Thread body

```
Deadline_Miss : Timing_Event; -- static, local per component
task body Sporadic_Thread is
  Release_Time    : Time;
  Next_Release    : Time := <Start_Time>;
  Canceled        : Boolean := False;
begin
  loop
    delay until Next_Release;
    OBCS.Wait(Release_Time);
    Set_Handler(Deadline_Miss,
                Release_Time + Milliseconds(Deadline),
                Deadline_Miss_Handler); -- application-specific
    OPCS.Sporadic_Operation;
    Cancel_Handler(Deadline_Miss, Canceled);
    Next_Release := Release_Time + Milliseconds(Separation);
  end loop;
end Sporadic_Thread;
```



Execution-time overruns

- Tasks may execute for longer than stipulated owing to
  - Programming errors in the functional code
  - Inaccurate WCET values used in feasibility analysis
    - Optimistic vs. pessimistic
- WCET overruns can be detected at run time with the help of **execution-time timers**
  - Not included in Ravenscar
  - Extended profile

Execution-time timers

- A user-defined event can be fired when a CPU clock reaches a specified value
  - An event handler is automatically invoked by the runtime at that point
  - The handler is an (**access** to) a **protected procedure**
- Basic mechanism for execution-time monitoring

Ada.Execution_Time.Timers /1

```
with System;
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 : in Time_Span;
                          Handler : in Timer_Handler);
   procedure Set_Handler (TM      : in out Timer;
                          At_Time : in CPU_Time;
                          Handler : in Timer_Handler);
   ...
end Ada.Execution_Time.Timers;
```

Ada.Execution_Time.Timers /2

- Mechanism built on execution time clocks
- Needs an interval timer
  - To update at every dispatching point
  - To raise «zero events» that signify execution-time overruns
- Sensible handling of those zero events requires other sophisticated features

Cyclic thread with WCET overrun detection (spec)

```
task type Cyclic_Thread
   (Thread_Priority : Priority;
    Period          : Positive;
    WCET_Budget     : Positive) is
  pragma Priority(Thread_Priority);
end Cyclic_Thread;
```

Thread body

```
task body Cyclic_Thread is
  Next_Time : Time := <Start_Time>;
  Id : aliased constant Task_ID := Current_Task;
  WCET_Timer : Timer(Id'access);
begin
  loop
    delay until Next_Time;
    Set_Handler(WCET_Timer,
                Milliseconds(WCET_Budget),
                WCET_Overrun_Handler); -- application-specific
    OPCS.Cyclic_Operation;
    Next_Time := Next_Time + Milliseconds(Period);
  end loop;
end Cyclic_Thread;
```

Summary

- We have seen how one particular programming language is able to capture all design and execution aspects that descend from the real-time systems theory that we seen so far
- We have seen how design and code patterns may be used to make sure that the application program conforms with the required semantics

Selected readings

- T. Vardanega, J. Zamorano, J.A. de la Puente (**2005**), *On the Dynamic Semantics and the Timing Behavior of Ravenscar Kernels*
  DOI: 10.1023/B:TIME.0000048937.17571.2b