Synchronous Task Control and Synchronous Barriers

A. Burns and A.J. Wellings
Department of Computer Science,
University of York, York, UK.
email: {alan.burns, andy.wellings}@york.ac.uk

Abstract

This paper looks at two features of Ada that support synchronisation between tasks. First, an ambiguity with Synchronous Task Control is outlined. Second, the question of whether Synchronous Barriers should be allowed in the Ravenscar profile is addressed.

1 Introduction

The ARG has asked the IRTAW community to consider two issues that have arisen recently as part of their maintenance of the Ada language. First we consider the extent to which Synchronous Task Control should be used by concurrent tasks. And then the issue of whether, or not, Synchronous Barriers are a natural part of the Ravenscar profile is considered.

2 Synchronous Task Control

The definition of this library routine is as follows.

```
package Ada.Synchronous_Task_Control is
  pragma Preelaborate(Synchronous_Task_Control);
  type Suspension_Object is limited private;
  procedure Set_True(S : in out Suspension_Object);
  procedure Set_False(S : in out Suspension_Object);
  function Current_State(S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True(S : in out Suspension_Object);
private
     ... -- not specified by the language
end Ada.Synchronous_Task_Control;
```

The semantics for this primitive are mainly straightforward. A Suspension Object (SO) is either `true` or `false`; it is initialised to `false`. A call of `Suspend_Until_True` suspends the task if the SO is `false`. When another task calls `Set_True` then the suspended task is released for execution and the SO becomes `false`. Neither `Set_True` or `Set_False` cause blocking to the calling task, although a call of `Set_True` may cause a released task to preempt.

A simple use of a suspension object is to give condition synchronisation – task A must not proceed beyond some point until task B has completed some action. The first task suspends on a SO; the second task releases it. Suspension objects are included in Ravenscar as they allow for efficient suspend and resume actions, for example in controlling the release of an event-triggered task and in the implementation of a bounded buffer shared between two tasks [1]. They can also be used to program patterns that would naturally use multiple entries and entry queues in full Ada (such use is illustrated by the example given in Section 3).

There are a number of constraints defined on SOs: The operations `Set_True` and `Set_False` are atomic with respect to each other and with respect to `Suspend_Until_True`. Also, significantly, `Program_Error` is raised upon calling `Suspend_Until_True` if another task is already waiting on that suspension object. Hence, there cannot be a queue of suspended tasks.

This final constraint rules out the use of SOs as general binary semaphores. If three or more tasks wish to enter a critical section then the following will fail (where `Sem` is a SO initialised to `true`):

```
Suspend_Until_True(Sem);
   -- critical section
Set_True(Sem);
```

The first task to execute this code will succeed and enter the critical section, the second will block as `Sem` is now `false`, but the third will have exception `Program_Error` raised.

The question arises: is this use of SOs acceptable when there are only two tasks involved? With the current definition it would not because concurrent calls to `Suspend_Until_True` are not defined to be atomic with respect to each other. To clarify this potential ambiguity with respect to the expected behaviour of suspension objects three possible modifications could be made:

1. Disallow concurrent calls.
2. Allow concurrent calls, and define them to be atomic.
3. Allow concurrent calls, define them to be atomic and remove the restriction as to there being at most one suspended task (per SO).

The first approach is in keeping with the original motivation for SOs, namely that they are not to be shared between tasks. Each SO is really private to just one task (the one that can be suspended upon it) although, of course, other tasks will call `Set_True`. If this approach is taken then the language definition would need to say what would happen if there were concurrent calls. Presumable it could be defined as a bounded error. But care must be taken not to introduce a distributed overhead; perhaps allow an implementation to not signal the fault.

The second approach represents the smallest change to the language. It would allow two task critical sections to be programmed. But it would perhaps be misleading as the pattern cannot be generalised to more than two tasks.

The third approach is more radical, and arguably changes the basic abstraction. Now a SO is like a binary semaphore and can be used to program general mutual exclusion.

The IRTAW may wish to formulate an opinion on which of these approaches it supports.

3 Synchronous Barriers

Synchronous barriers are a relatively new feature of Ada, they allow a set of tasks to all wait at a barrier until the final member of the set arrives. They are then all are made runnable again and proceed (with one of then having a flag set). When synchronous barriers were introduced into the language it was decided not to include them in the Ravenscar profile.

It has been observed that engineers found the new restriction in Ravenscar against Synchronous Barriers painful, so it is reasonable to ask if it really should be prohibited.

At one level a synchronous barrier is just a protected object with many entries. As such a PO is not part of Ravenscar then it can be argued that it follows that synchronous barriers are similarly excluded.

To show that this is really an issue of ease-of-use and efficiency, rather than expressive power the following (based on an email from Tucker Taff) shows that the required behaviour for a synchronous barrier can be obtained from existing Ravenscar features (protected objects and suspension objects).

```
with Ada.Synchronous_Task_Control;
use  Ada.Synchronous_Task_Control;

package Raven_Sync_Barriers is
   pragma Preelaborate(Raven);
   subtype Barrier_Limit is Positive range 1 .. 1000;
   type Synchronous_Barrier (Release_Threshold : Barrier_Limit) is limited private;
   procedure Wait_For_Release(The_Barrier : in out
         Synchronous_Barrier; Notified    :    out Boolean);
private
   type Susp_Obj_Array is array(Positive range <>) of Suspension_Object;
   protected type Protected_Barrier (Release_Threshold :  Barrier_Limit) is
      procedure Arrive(Sleep : out Boolean; BL : out Barrier_Limit; SOA : in out Susp_Obj_Array);
      --  Keep track of the arrival index for each task
   private
      Num_Arrived : Natural range 0 .. Barrier_Limit'Last :=0;
      --  Number of tasks that have arrived at barrier
   end Protected_Barrier;

   type Synchronous_Barrier(Release_Threshold :Barrier_Limit) is limited record
      Suspended_Tasks:Susp_Obj_Array (2 .. Release_Threshold);
      Barrier : Protected_Barrier (Release_Threshold);
   end record;
end Raven_Sync_Barriers;

package body Raven_Sync_Barriers is
   protected body Protected_Barrier is
      procedure Arrive(Sleep : out Boolean; BL : out Barrier_Limit; SOA : in out Susp_Obj_Array) is
      begin
        Num_Arrived := Num_Arrived + 1;
        if Num_Arrived < Release_Threshold then
          Sleep := True;
        else
          -- last task in, so wake all the others up
          for I in SOA'Range loop
             Set_True(SOA(I));
          end loop;
          --  Reset the barrier
          Num_Arrived := 0;
          Sleep := False;
        end if;
        -- set index for task to suspend on, only needed if Sleep is True
        BL := Num_Arrived + 1;
      end Arrive;
   end Protected_Barrier;

   procedure Wait_For_Release (The_Barrier : in out Synchronous_Barrier;
                               Notified    :    out Boolean) is
      Go_To_Sleep : Boolean;
      Arrival_Index : Barrier_Limit;
   begin
      The_Barrier.Barrier.Arrive(Go_To_Sleep, Arrival_Index, The_Barrier.Suspended_Tasks);
      if Go_To_Sleep then
          Suspend_Until_True(The_Barrier.Suspended_Tasks (Arrival_Index));
          Notified := False;
      else
          Notified := True;  --  This is the lucky task
      end if;
   end Wait_For_Release;
end Raven_Sync_Barriers;
```

An example of the use of this code would be:

```
with Raven_Sync_Barriers; use Raven_Sync_Barriers;
package bar_test is
```

```
   Z : Synchronous_Barrier(3);
   task type TT(Me : Positive);
   T1 : TT(1);
   T2 : TT(2);
   T3 : TT(3);
end bar_test;

with Ada.Text_IO; use Ada.Text_IO;
package body bar_test is
   task body TT is
      Notified : Boolean := False;
   begin
      Put_Line ("Number" & Positive'Image(Me) &
        "␣is␣about␣to␣wait␣for␣barrier");
      Wait_For_Release(Z, Notified);
      Put_Line ("Number" & Positive'Image(Me) &
        "␣returned␣from␣wait,␣Notified␣=␣" & Boolean'Image(Notified));
   end TT;
end bar_test;
```

which produces:

```
Number 1 is about to wait for barrier
Number 3 is about to wait for barrier
Number 2 is about to wait for barrier
Number 2 returned from wait, Notified = TRUE
Number 1 returned from wait, Notified = FALSE
Number 3 returned from wait, Notified = FALSE
```

Interestingly a second execution of the program produces a different ordering:

```
Number 2 is about to wait for barrier
Number 1 is about to wait for barrier
Number 3 is about to wait for barrier
Number 2 returned from wait, Notified = FALSE
Number 1 returned from wait, Notified = FALSE
Number 3 returned from wait, Notified = TRUE
```

The above demonstrates that the required behaviour for a synchronous barrier can be programmed with existing Ravenscar features. But if the underlying platform directly supports a barrier primitive then clearly more efficient code can be generated if such barriers are directly supported by the language. The question for IRTAW is whether this potential efficiency gain is sufficient to warrant the inclusion of synchronous barriers in Ravenscar.

4 Conclusions

This paper has provided the background for two questions that IRTAW has been asked to address: How should the ambiguity surrounding the definition of Synchronous Task Control be removed, and should Synchronous Barriers be included in Ravenscar.

References

[1] ISO/IEC. Information technology - programming languages - guide for the use of the Ada Ravenscar Profile in high integrity systems. Technical Report TR 24718, ISO/IEC, 2005.