Ada Issue 327 Dynamic Ceiling Priorities

!standard D.03 (13) 05-03-17 AI95-00327/08
!standard D.05 (00)
!standard D.05.01 (00)
!standard D.05.02 (00)
!standard D.07 (09)
!class amendment 03-06-05
!status Amendment 200Y 04-06-24
!status WG9 approved 04-11-18
!status ARG Approved 8-0-0 04-06-13
!status work item 03-06-05
!status received 03-06-05
!priority Medium
!difficulty Medium
!subject Dynamic ceiling priorities

!summary

A mechanism is defined for dynamic ceiling priorities. This enables the ceiling priority of a protected object to be changed by assignment to the new attribute Priority within a protected operation of the object.

!problem

In Ada, the ceiling priority of a protected object is constant and thus it can only be set once, by means of pragma Priority, when the object is created. In contrast, task priorities may be set dynamically. The ability to dynamically change protected object ceiling priorities is especially required in situations where dynamic task priority change occurs, or where a library containing protected objects with inadequate ceilings is used by an application-specific set of tasks and interrupt handlers. Examples of dynamic task priority changes include:

- multi-moded systems
- systems scheduled using dynamic priorities

For multi-moded systems, a common workaround is to use the so-called "ceiling of ceilings", i.e. the highest ceiling that the protected object can have across all operating modes. Clearly this approach is error-prone, particularly during composition of sub-systems, and may have a considerable impact on blocking times. The use of ceiling of ceilings implies that a task running in a particular mode can be blocked by another one unnecessarily, due to an inflated ceiling priority resulting from the requirements of a different mode of operation. Consequently, the ceiling of ceilings reduces the schedulability of the task set.

!proposal

An implementation of dynamic ceiling priorities is proposed. The ceiling priority of a protected object would be dynamically changed by giving access to a new attribute of a protected object.

The ceiling must be changed whilst mutual exclusion for the protected object is in force, according to the ceiling locking protocol. This is the only way to ensure a consistent ceiling change without the need for a further lock to protect the ceiling change. Examples are provided to show how to use this feature.

A prototype implementation by Javier Miranda, Edmond Schonberg and Jorge Real concludes that the overheads of this feature is small and not distributed.

!wording

Add a new paragraph after D.3(13)

Bounded (Run-Time) Errors

Following any change of priority, it is a bounded error for the active priority of any task with a call queued on an entry of a protected object to be higher than the ceiling priority of the protected object.
In this case one of the following applies:

- at any time prior to executing the entry body Program_Error is raised in the calling task;

- when the entry is open the entry body is executed at the ceiling priority of the protected object;

- when the entry is open the entry body is executed at the ceiling priority of the protected object and then Program_Error is raised in the calling task;

- when the entry is open the entry body is executed at the ceiling priority of the protected object that was in effect when the entry call was queued.

Insert after the title of D.5:

This clause describes how the priority of an entity can be modified or queried at run time.

D.5.1 Dynamic Priorities for Tasks

[This changes the section number of all of the existing text.]

Delete paragraph D.5(11) (now D.5.1(11)).

Add a new section:

D.5.2 Dynamic Priorities for Protected Objects

This clause specifies how the priority of a protected object can be modified or queried at run time.

Static Semantics

The following attribute is defined for a prefix P that denotes a protected object:

P'Priority
Denotes a non-aliased component of the enclosing protected object P. This component is of type System.Any_Priority and its value is the priority of P. Reference to this attribute shall appear only inside the body of P.

The initial value of this attribute is set by pragmas Priority or Interrupt_Priority, and can be changed by an assignment.

Dynamic Semantics

If the locking policy Ceiling_Locking is in effect then the ceiling priority of a protected object P is set to the value of P'Priority at the end of each protected action of P.

Metrics

The implementation shall document the following metric:
The difference in execution time of calls to the following procedures in protected object P,

```
protected P is
  procedure Do_Not_Set_Ceiling (Pr : System.Any_Priority);
  procedure Set_Ceiling (Pr : System.Any_Priority);
private
  null;
end P;

protected body P is
  procedure Do_Not_Set_Ceiling (Pr : System.Any_Priority) is
  begin
    null;
  end;
  procedure Set_Ceiling (Pr : System.Any_Priority) is
  begin
    P'Priority := Pr;
  end;
end P;
```

Notes
Since P'Priority is a normal variable, the value following an assignment to the attribute immediately reflects the new value even though its impact on the ceiling priority of P is postponed until completion of the protected action in which it is executed.

--

Change the definition of Restriction identifier No_Dynamic_Priorities:

No_Dynamic_Priorities
  There are no semantic dependences on the package Dynamic_Priorities, and no occurrences of the attribute Priority.

!example

A simple example of how to use the proposed features is given in this section.

```
protected type My_Protected is
  pragma Priority(Some_Initial_Value);
  procedure Set_My_Ceiling(Pr: in Any_Priority);
  -- other protected procedures, functions and/or entries
end My_Protected;

protected body My_Protected is
  procedure Set_My_Ceiling(Pr : in Any_Priority) is
  begin
    -- Code before setting the priority
    My_Protected'Priority := Pr;
    -- Code after setting the priority
    -- The new ceiling does not take effect
    -- until the end of this procedure, but the
    -- new value is returned by any read of attribute Priority
  end Set_My_Ceiling;

  -- Rest of bodies

end My_Protected;
```

```
  PO: My_Protected;
```

In this example, the caller to Set_My_Ceiling must know the name of the protected object whose ceiling is to be changed - the call would have the form PO.Set_My_Ceiling(Pr). A more flexible scheme can easily be obtained by means of an access-to-protected-subprogram, that could be used in the implementation of a general ceiling changer:

```
  type General_Set_Ceiling is access protected procedure
    (P: in System.Any_Priority);
  Ceiling_Changing_Procedure: General_Set_Ceiling;
```

Once the access object is properly initialized, the call would take the form:

```
  Ceiling_Changing_Procedure(P);
```

The use of a protected interface could also achieve this functionality.

!discussion

Both the old and new values of the ceiling must be retained by the implementation, as an external call needs to have its priority checked against the old value until the new value takes effect, at the end of the protected action.

The interface to these features, e.g. the use of P'Priority appears to be the easiest way of achieving the required functionality.

The need for dynamic ceilings and a detailed examination of the many different ways it could be achieved has been the subject of much discussion at four IRTAWs. The following is a summary.

Two primary approaches have been considered to make this proposal:

a) To implement the ceiling change as a special operation on the protected object, not necessarily as a protected operation under the ceiling locking protocol.
b) To implement the ceiling change as a protected operation.

We will discuss now the main problems with a) and later on, how b) solves them.

-- IMPLEMENTATION AS A SPECIAL OPERATION --

The motivation for a) was to be able to change a protected object's ceiling from any priority, lower or higher than the ceiling, thus allowing to promptly change the ceilings from a high priority mode changer (in the mode change scenario). This approach was inspired by the use of ceilings in POSIX mutexes.
Let's explore it.

Three cases can be identified with respect to the relative priorities of tasks executing a protected operation, (the task assigning to the attribute Priority is referred to as the caller):

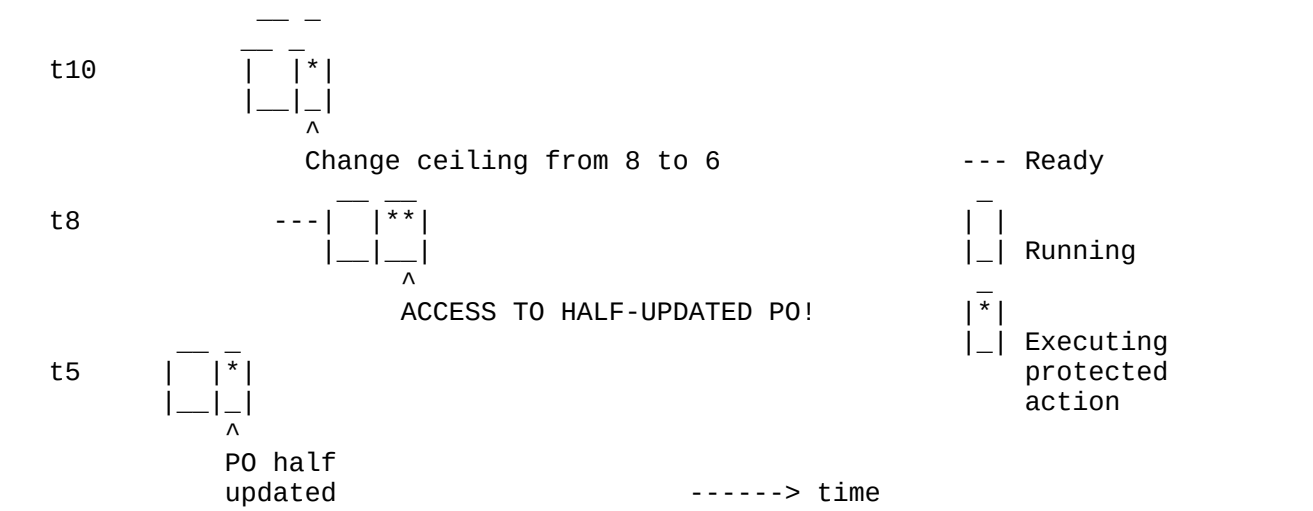
Case 1: The caller has a lower or equal priority to the current ceiling and is changing the ceiling to a higher priority. In this situation the ceiling change can take place without problems. The priority change occurs and tasks on entry queues are not affected. They may just inherit the new ceiling when they execute the protected operation, which will be higher or equal to their base priority.

Case 2: The caller has a lower or equal priority to the current ceiling, as in case 1, but now it is changing the ceiling to a lower priority. Here, it is possible that tasks queued on entries might have active priorities higher than the new ceiling. In this situation, the Ceiling_Locking policy could be violated and it would be

necessary to raise an appropriate exception in the queued tasks. Currently in Ada, Program_Error is raised when a task tries to call a protected operation in a protected object with a lower ceiling priority.

Case 3: The caller has a higher priority than the current ceiling. Hence, the ceiling change cannot adhere to the ceiling protocol to change the ceiling priority. The POSIX 1003.1b standard says exactly the same about the operation provided for changing the ceiling of a mutex. Cases 1 or 2 still apply with respect to the relationship between the old and new ceilings.

The worst case is represented by the third situation, where data corruption may occur. It could be the case that a task with a priority 10 wants to change the ceiling priority of a protected object with a ceiling lower than 10, say 8. The next figure shows graphically this scenario, where "ti" represents a task with priority "i". A task with a lower priority, say 5, could be executing a protected action (with an active priority 8) when it was preempted by the high priority task t10. The main problem here is that a task with a medium priority, say 8, that uses the same protected object, can be released whilst t5 is preempted by t10. When t10 completes, the medium priority task t8 starts to run when t5 is still in the middle of the protected operation. The risk of data corruption in the protected object is clear. A well designed program can avoid this situation, but it would be very difficult for the language to detect it.



Two possible solutions to this problem have been identified; unfortunately, both present important drawbacks. The first one is to use a lock on the PO that would prevent t8 in the example above to access the PO. The lock eliminates the risk for data corruption but the use of locks can cause well-known priority inversion problems. More importantly, the implementation of ceiling locking on a mono-processor does not require a lock. Hence this solution should also be rejected for efficiency reasons.

A second way to avoid the potential for data corruption is that the task executing the protected action when a ceiling change happens to occur, "immediately" inherits the new ceiling. In such case, no other task accessing the protected object can preempt it, therefore the protected action will be completed normally even in the presence of a ceiling change. A possible implementation of an assignment to the attribute Priority conforming to such semantics would be the following:

1. Change variable "ceiling" to New_Ceiling
2. if PO_In_Use and Prio(User_Task) < New_Ceiling then
3. User_Task inherits New_Ceiling
4. end if

A problem with this approach is that the runtime needs to know the identity of the task running the protected action -the User_Task-, something that is not needed in Ada but for this particular case. This approach therefore also suffers from implementation impact on efficiency, and so we have rejected it.

-- IMPLEMENTATION AS A PROTECTED OPERATION --

The second approach (the one that is advocated in this AI) proposes to implement the ceiling change as a protected operation, thereby avoiding the problems described above.

For a given protected object PO, we shall assume the existence of a predefined procedure Set_Ceiling, implemented as an assignment to the attribute Priority, with an "in" parameter of the type System.Any_Priority that sets the new ceiling priority for PO. This attribute can only be used immediately within the body of a protected procedure or entry of the affected protected object, thus its use is subject to the ceiling locking policy. This means that a task calling it must have a priority lower than or equal to the protected object's ceiling.

Accordingly, a task calling Set_Ceiling will be executed in mutual exclusion with other potential users of the protected object, therefore ensuring that no other task is executing a protected action when the ceiling is changed. This approach avoids the situation described previously, where data consistency was at risk due to the asynchronous nature of the ceiling change when it is implemented as a special operation. It also makes it unnecessary to use a lock on the protected object. If Set_Ceiling is implemented as a protected operation, the standard implementation of ceiling locking is sufficient to guarantee its execution in mutual exclusion with other users of the protected object.

With respect to tasks queued on protected entries, it could be the case that the ceiling of the protected object is lowered below the queued task's active priority, which represents a bounded error in Ada. According to D.5(11):

> "If a task is blocked on a protected entry call, and the call is queued, it is a bounded error to raise its base priority above the ceiling priority of the corresponding protected object."

In other words, if dynamic ceilings are considered, this rule would also apply to the case where a protected object's ceiling is set below the active priority of tasks queued on the protected object's entries. The situation is already considered by the language; therefore no new problems are introduced in this sense.

Note that we propose that the new ceiling should take effect at the end of the protected action, instead of at the end of the protected operation that executes the Set_Ceiling call. This is intended to reduce the risk of Program_Error for queued tasks that have already successfully performed the initial entry call with respect to the ceiling check. Those tasks for whom the barrier becomes open as part of the same protected action as is changing the ceiling will be successfully executed at the old ceiling priority. Thus the only tasks that will be affected are those that remain blocked after the entire protected action has been completed.

Paper [1] contains further discussions on the motivation behind the proposal and other examples (for an earlier version of the feature). Dynamic ceilings have been implemented in an existing Ada compiler. The report on this experience can be found in [2].

References:
---

[1] J. Real, A. Crespo, A. Wellings, and A. Burns. Protected Ceiling Changes. Proceedings of the 11th International Real-Time Ada Workshop. Ada Letters XXII(4). December 2002.

[2] J. Miranda, E. Schonberg, M. Masmano, J. Real and A. Crespo. Dynamic Ceiling Priorities in GNAT. Proceedings of the 12th International Real-Time Ada Workshop. To appear in Ada Letters, December 2003.

!corrigendum D.3(13)

@dinsa
@xbullet<When a task calls a protected operation, a check is made that its active priority is not higher than the ceiling of the corresponding protected object; Program_Error is raised if this check fails.>
@dinss
@i<@s8<Bounded (Run-Time) Errors>>

Following any change of priority, it is a bounded error for the active priority of any task with a call queued on an entry of a protected object to be higher than the ceiling priority of the protected object.
In this case one of the following applies:

@xbullet<at any time prior to executing the entry body Program_Error is raised in the calling task;>
@xbullet<when the entry is open the entry body is executed at the ceiling priority of the protected object;>
@xbullet<when the entry is open the entry body is executed at the ceiling priority of the protected object and then Program_Error is raised in the calling task; or>
@xbullet<when the entry is open the entry body is executed at the ceiling priority of the protected object that was in effect when the entry call was queued.>

!corrigendum D.5(1)

@dinsb
This clause specifies how the base priority of a task can be modified or queried at run time.
@dinss
This clause describes how the priority of an entity can be modified or queried at run time.

@fa<@s10<@b<D.5.1 Dynamic Priorities for Tasks>>>

[This changes the subclause of all of the existing text.]

!corrigendum D.5(11)

@ddel
If a task is blocked on a protected entry call, and the call is queued, it is a bounded error to raise its base priority above the ceiling priority of the corresponding protected object. When an entry call is cancelled, it is a bounded error if the priority of the calling task is higher than the ceiling priority of the corresponding protected object. In either of these cases, either Program_Error is raised in the task that called the entry, or its priority is temporarily lowered, or both, or neither.

!corrigendum D.5.2(01)

@dinsc
This clause specifies how the priority of a protected object can be modified or queried at run time.

@i<@s8<Static Semantics>>

The following attribute is defined for a prefix P that denotes a protected object:

@xhang<@xterm<P'Priority>
Denotes a non-aliased component of the enclosing protected object P. This component is of type System.Any_Priority and its value is the priority of P. Reference to this attribute shall appear only inside the body of P.>

The initial value of this attribute is set by pragmas Priority or Interrupt_Priority, and can be changed by an assignment.

@i<@s8<Dynamic Semantics>>

If the locking policy Ceiling_Locking is in effect then the ceiling priority of a protected object P is set to the value of P'Priority at the end of each protected action of P.

@i<@s8<Metrics>>

The implementation shall document the following metric:

@xbullet<The difference in execution time of calls to the following procedures in protected object P,>

```
@xcode<   @b<protected> P @b<is>
     @b<procedure> Do_Not_Set_Ceiling (Pr : System.Any_Priority);
     @b<procedure> Set_Ceiling (Pr : System.Any_Priority);
  @b<private>
     @b<null>;
  @b<end> P;

  @b<protected body> P @b<is>
     @b<procedure> Do_Not_Set_Ceiling (Pr : System.Any_Priority) @b<is>
     @b<begin>
        @b<null>;
     @b<end>;
     @b<procedure> Set_Ceiling (Pr : System.Any_Priority) @b<is>
     @b<begin>
        P'Priority := Pr;
     @b<end>;
  @b<end> P;>
```

@xindent<@s9<NOTES@hr
38 Since P'Priority is a normal variable, the value following an assignment to the attribute immediately reflects the new value even though its impact on the ceiling priority of P is postponed until completion of the protected action in which it is executed.>>

!corrigendum D.7(9)

@drepl
@xhang<@xterm<No_Dynamic_Priorities>
There are no semantic dependences on the package Dynamic_Priorities.>
@dby
@xhang<@xterm<No_Dynamic_Priorities>
There are no semantic dependences on the package Dynamic_Priorities, and no occurrences of the attribute Priority.>

!ACATS test

Create ACATS test(s) for this feature.