A Semantics for Dynamic Ceiling Priorities in Ada

Jorge Real[*], Albert Llamosí [†] and Alfons Crespo[*]
[*]Universitat Politècnica de València (SPAIN)
[†]Universitat de les Illes Balears (SPAIN)
{jorge,alfons}@disca.upv.es   llamosi@uib.es

Abstract

Several arguments have been given to justify the exclusion of dynamic ceiling priorities for protected objects in Ada. These arguments center on semantic complexity, the risks that can be derived from an erroneous usage of dynamic ceilings, and efficiency. But dynamic ceilings are very convenient for multi-moded systems and to dynamically adapting existing program libraries containing protected objects to new applications. This paper proposes a semantics for dynamic ceilings that does not add complexity to the language semantics or inefficiency to the run time system.

1 Introduction

Ada 95 introduced dynamic priorities for tasks, thus providing for more flexible scheduling algorithms. But ceiling priorities for protected objects were restricted to being statically assigned by means of a pragma [9]. The semantic complexity of changing ceilings and efficiency considerations have been the main arguments raised to initially discard this feature.

After the standard approval in 1995, the topic of dynamic ceilings has been reopened to discussion without time restrictions. At the 8th International Real-Time Ada Workshop (IRTAW), the issue of dynamic ceilings was classified as a topic to be targeted for the next language revision, although it was not discussed in any depth [5].

Later, at the 9th IRTAW, Real and Wellings proposed an implementation that would benefit from dynamic ceilings to program mode changes [8]. Applications, in general, will only require ceilings to be changed when the mode changes or at initialisation, thus the importance of proposing a safe mode change protocol. Their paper separated two different issues: *when* to change ceilings and *how* to implement the ceiling change. The mode change algorithm guaranteed the protected object was empty (both in terms of users and waiters) at a certain point in time, after the mode change request — assuming worst-case blocking is known in advance. Therefore, it was safe to change ceilings and task priorities consistently at a given time, thus solving the first issue. About *how* to change the ceiling, it was generally felt after the workshop that dynamic ceilings would be a good feature to have in the next revision of Ada, but further discussion was needed to find the best way to do it [2].

Actually, there exists a well known work around for static ceilings, consisting in assigning the protected objects the so called *ceiling of ceilings*, i.e., the maximum ceiling priority across all operating modes. Unfortunately, the ceiling of ceilings introduces more blocking than necessary most of the time, as the restrictions from one mode are transported to all the operating modes [7]. Contrary to what can be thought, blocking operations are not necessarily fast [1, 4].

In this paper, a semantics for dynamic ceilings is proposed. The discussion is guided by the idea that objections to dynamic ceilings can also be applied to dynamic priorities for tasks. Therefore, it makes little sense to object to dynamic ceilings when Ada already has a precise semantics for dynamic priorities.

The paper is structured as follows. The section 2 summarises the main issues related to dynamic ceilings. The discussion of these issues takes place in section 3, where a semantics for dynamic ceilings is proposed. The section 4 evaluates the consequences of applying such semantics, to check whether new problems arise or not. Two other approaches are discussed in this section for comparative purposes. Finally, section 5 presents our conclusions.

2 Issues raised by dynamic ceilings

The main concerns about dynamic ceilings, as formulated in previous editions of the IRTAW [2, 5], are the following:

[*]This work is partially funded by the Spanish Government CICYT project number TIC99-1043-C03-02.

- How to deal with tasks executing protected code or queued in protected entries when the ceiling of the corresponding protected object is about to change. For instance, a task may be queued in a protected entry when we want to lower the ceiling below the task's base priority. Eventually, the barrier will be open and a violation of the Ceiling Locking Protocol (CLP) will occur. Who should be notified in this case? The queued task? The task changing the ceiling?

- What are the requirements for the ceiling change operation. In particular, what priority should a task have to be able to change a protected object's ceiling. Moreover, do we need a separate lock to implement the ceiling change operation?

The next section discusses these issues.

3 Discussion and proposal

The discussion of the issues formulated in the previous section relies on two aspects:

1. To define a precise semantics for the ceiling change operation. The semantics should be consistent with Ada's current semantics for changes in tasks' priorities.

2. To evaluate whether dynamic ceilings may cause programs to lose important properties or not, such as introducing unbounded priority inversions or new sources of blocking.

This section proposes a semantics for dynamic ceilings. The section 4 will investigate the potential problems derived from the dynamic semantics proposed.

The interaction between dynamic priorities for tasks and protected operations under the CLP are dealt with in the Annex D of the Ada Reference Manual [9]:

- (D.5-10) Setting a task's base priority takes place as soon as is practical but not while the task is performing a protected action.

- (D.5-11) If a task is queued on a protected entry call, it is a bounded error to raise the task's 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 above the ceiling 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.

Paragraph D.5-10 prevents wrong priority assignments to occur, that could lead to leaving the protected object inconsistent. Certainly, raising a task's priority above the ceiling of a protected object whilst the task is using it, could give raise to a bounded error in the middle of the execution of a protected action. The ARM introduces this *deferred* semantics for `Set_Priority` so that the potential error is actually bounded. Moreover, with this dynamic semantics the task's priority and the object's ceiling only need to be compared at the beginning of the execution of the protected code, which allows implementations to be simple and efficient.

With respect to D.5-11, the argument can also be formulated the other way around to fit the problem of dynamic ceilings: *if a task is queued in a protected entry call, it is a bounded error to lower the object's ceiling below the base priority of the task.* Note this is not a new problem to Ada, as the situation can come either from lowering the ceiling (not allowed by the language) or from raising the task's priority (allowed!). Another remarkable question is that the `Program_Error` exception is raised in the task calling the protected operation, not in the task calling `Set_Priority`.

The use of requeue requires additional comments. An algorithm using requeue may fail if a calling task is aborted between the initial call and the deferred call via requeue. For instance, a read access to a hard drive may be prioritized by proximity to the current position of the head. If the disk drive is implemented with a protected object, we can have an entry where tasks place their petition, indicating what cylinder and head they want to access; then the relative priority of the petition is calculated with respect to the current position of the head, and the call is requeued to be attended in the optimum order. If the calling task is aborted between the original call and the requeued call, then the disk manager will indefinitely keep record of the petition and the system will be partially or completely blocked as no task will be willing to receive the accessed data.

If a priority check is made every time the task is requeued inside the protected object and ceilings are allowed to change, then the risk to produce such fails increases. So, when should the priority of a requeued task be compared with the ceiling? Every time the task is dequeued? Or just the first time the task executes protected code?

In other similar situations, the Ada language leaves the decision on the programmer side, providing several different semantics. For instance, we have the case of timed entry calls. An entry call can be cancelled when the timeout expires. But if the entry call is accepted and requeued, the timeout semantics is different depending on the requeue being with or without abort. Therefore, the programmer of the protected object decides the model of timed entry call to follow, according to whether not completing an entry call path (including the original call and the subsequent requeued calls) can lead to a logical error or not.

The same can be applied to ceiling checks. The check can be made the first time the task is accepted in an entry.

In the case the task is requeued with a simple requeue (i.e. without abort), the ceiling is not checked again: this avoids the protected object to end up inconsistent, as in the disk controller example above. The task will inherit the new ceiling priority, but no check will be performed (the next section analyses the potential consequences). In the case of a requeue with abort, the ceiling is checked again when the task is accepted in a new entry in the entry call path. The programmer of the protected object decides what semantics is most appropriate for the case. At run time, we should only need to check a boolean to distinguish the first call in the path from the requeued calls, and the necessarily existing boolean to distinguish between tasks requeued with or without abort.

Besides a semantics for tasks inside the protected object, we need to define the behaviour of the operation to change the ceiling itself. Let `Set_Priority` be a new operation, overriding the current homonymous operation in `Ada.Dynamic_Priorities` for assigning tasks' priorities. `Set_Priority` would also be called to change an object's ceiling, by passing the new ceiling priority and a reference to the object (not necessarily an access type, perhaps a new `'ID`-like attribute for protected objects).

An important issue is the relative priority of the task changing the ceiling of a protected object with respect to the old and the new ceiling value. In our view, it would be too restrictive to impose conditions on this concern. If the priority of the task was forced to be low (e.g. lower than the lowest among the old and the new ceiling) then the ceiling change could take too long to execute, forcing a mode change manager to execute at a relatively low priority. Although mode change protocols do exist that would fit well in this scheme[1], this restriction would be very inconvenient for implementing prompt mode changes. On the other hand, a high priority for a task changing an object's ceiling could be considerd to be against the CLP. And it would, if `Set_Priority` was considered a *normal* protected operation; but it is not that normal to change the whole object state by setting a new ceiling. In POSIX, this fact is recognised in the standard when describing *pthread_mutex_setprioceiling()*, the operation for dynamically setting a mutex ceiling: when executing this operation, *the process of locking the mutex needs not adhere the priority protect protocol* [3].

According to the semantics described above, the execution of `Set_Priority` on protected objects could be performed from any priority level. The only thing to do to change a ceiling is to atomically modify the corresponding field in the protected object's descriptor. No additional checks would be needed. Therefore, this operation would be necessarily fast and will not produce a high blocking.

[1]An example is the *idle-time* protocol by Tindell and Alonso [10], where the mode change takes place when the processor becomes idle after the mode change request.

In summary, our proposal for dynamic ceilings has the following semantics:

- The operation `Set_Priority`, applied to a protected object, sets its ceiling priority to a new value given as a parameter. No additional checks are needed. The operation should be atomic, but it does not need to acquire the lock of the protected object, nor a new lock is needed. The change will have effect as soon as a task becomes ready to start to execute protected code, but will not affect a protected call currently being executed, which necessarily would have been preempted by the task executing `Set_Priority`.

- When a task starts to execute protected code, either due to a direct entry call or to an entry call that was requeued with abort, then a check is made that the base priority of the task is not above the new ceiling. If the check fails then `Program_Error` is raised in the calling task. Otherwise, the task inherits the ceiling priority and starts the execution of protected code.

- When a task has been requeued without abort and it becomes ready to start to execute protected code, then it inherits the ceiling priority and starts to execute the protected call. No additional check is made in order to avoid the entry call path to be aborted. The programmer of the protected object chooses the model depending on the impact an aborted path can have on the logical correctness of the application.

To have both semantics for requeue will allow the language to cover different application needs, more focused on temporal or in logical correctness. When both are needed, requeue operations should be abortable and the logical correctness of the program should not depend on the possibility of an entry call path to be aborted.

According to the Ada Reference Manual [9], the section 9.5.4, paragraph 16, states the utility of requeue: *If the reserved words with abort do not appear, then the call remains protected against cancellation while queued as the result of the requeue_statement*. In this case, the call path remains protected against cancellation due to a ceiling violation.

The proposal is therefore to use the existing clause *with abort* to select the way we want the call to be executed with respect to ceiling changes occurring during the call path. But, what is the interaction between the timeout and the ceiling re-check when using requeue with abort? If a task is requeued without abort, then the calling task is not really concerned about how long does it take for the entry call to be completed in all its way across the call path. If this is the case, then probably a mode change is not expected to occur or it is not a problem that some tasks take more than usual to complete. Therefore, programs using this feature do not require strict real-time performance or they are proven to have a correct timing beforehand. However, if the requeue is with abort, this means that the calling task is concerned about how the call progresses through the call path all the way. This is consistent with the use of *with abort* we propose for ceiling checks.

4 Potential problems

In the semantics defined in the previous section, the effect of the ceiling change is deferred until the next call to a protected operation. We identify now the potential conflicting situations that may arise from this fact. Nevertheless, we emphasize that a well-designed mode change protocol, with hard real-time restrictions, can avoid the conflicts mentioned below, only requiring the worst case blocking time to be known in advance.

Changing the ceiling of a protected object can obviously be in two directions: raising or lowering it.

Raising the ceiling. After raising the ceiling, tasks queued in entries of the protected object because of a direct call or a requeue with abort will eventually be dequeued, at which point they will check that their base priority is not above the new ceiling (which for sure will not be, as the ceiling has been raised), inherit the new ceiling priority and execute the protected code normally. The impact on blocking can be statically bounded.

In the case of tasks that have been requeued without abort, the ceiling check is not performed, but the new ceiling will be blindly inherited when their barriers evaluate to true. The task can thus temporarily run with a high active priority, whose impact on blocking can also be bounded.

But if the task is executing protected code when the ceiling is changed by a high priority task, the inheritance of the new ceiling will not be effective until the next call to the protected object, which can lead to priority inversion. This is because we have defined the `Set_Priority` operation as a simple change to the ceiling value, with no additional checks. Figure 1 shows this situation. Tasks τ_1, τ_2, τ_3 and τ_4 run with priorities 1, 2, 3 and 4, respectively, 4 being the highest. Initially, τ_1 is running protected code (dark box) when τ_4 raises the object's ceiling from 1 to 3. At that point, when τ_4 completes, the highest priority ready task is τ_3, which starts to execute and afterwards tries to lock the protected object (upwards arrow). As τ_1 owns the lock, τ_3 will become blocked and leave the processor for τ_2 and other intermediate priority tasks, leading to priority inversion over τ_3.

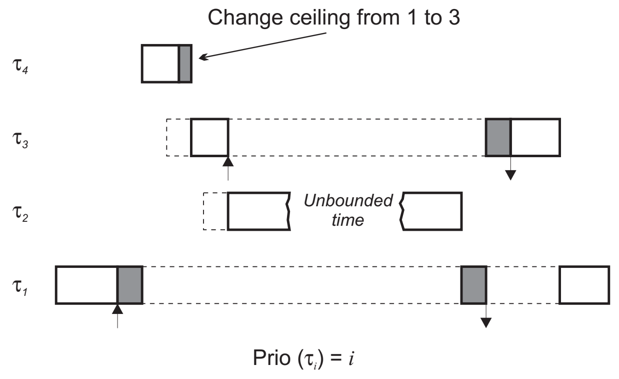

Figure 1. Unbounded priority inversion when raising ceilings whilst task τ_1 is executing protected code.

The problem in this situation is originated by τ_4 changing the ceiling at the wrong time, when τ_1 was still inside the object. If the worst-case time a task can be inside the protected object is known in advance, then τ_4 can delay its execution accordingly and the situation will not occur. Such mode change protocol is implemented by means of offsets in [6].

Lowering the ceiling. A task τ can be inside the protected object when its ceiling is lowered, either running protected code or queued. If τ is queued due to a direct entry call or a call requeued with abort and it has not inherited the ceiling yet and its base priority is higher than the new ceiling, then `Program_Error` will be raised to the task at the point of inheriting the ceiling. This is coherent with Ada's current semantics.

But if the task was executing protected code or requeued without abort, then it would use the protected object with a priority higher than the ceiling (the old ceiling priority). What would be the effect on the application? For tasks with a priority lower than τ's base priority, this would not affect their timing as the interference produced by τ would still be the same. Tasks with a higher priority than τ's active priority (the high previous ceiling) would be able to preempt it and execute normally. Finally, tasks whose priority is between the new (low) ceiling and the old (high) ceiling may suffer extra blocking, as τ is running with an active priority higher than the new ceiling. This blocking can also be bounded.

The behaviour in this case is that the effect of lowering the ceiling is deferred until no task is running a protected action, which adheres to the dynamic semantics of Ada for changing task priorities.

In summary, the proposed semantics allows for a very simple implementation. The side effects on blocking can be managed from the application, choosing the right times to change the ceilings and, in the case of mode changes, also choosing the right time to activate new-mode tasks.

To try to solve the problems of blocking from the compiler side, will necessarily add complexity at run time. For example, we can use immediate inheritance to solve the blocking. In this case, the implementation of `Set_Priority(New_Ceiling,PO'ID)` would follow the scheme:

```
1.  Change variable "ceiling" to New_Ceiling
2.  if PO is in use and Prio(Caller) < New_Ceiling then
3.     Caller inherits New_Ceiling
4.  end if
```

Under this implementation, the priority inversion is limited to the duration of the protected operation, as no intermediate-priority tasks are allowed to interfere. But this solution presents serious drawbacks. First, the implementation of `Set_Priority` becomes more complex and potentially has to change a task's active priority, which can be too costly. Second, this semantics is not orthogonal with the semantics for tasks priorities, where the change of priority is deferred to avoid changes during the execution of protected actions.

Another approach would be one with simple (no immediate) priority inheritance. In this case, the implementation of `Set_Priority` would be the simple one (just change the ceiling), but when a task calls a protected object, then it performs the checks in line 2 in the previous algorithm. This has the advantage of keeping a simple implementation of `Set_Priority` and eliminating extra blocking only when it occurs, as under the `Ceiling_Locking` policy, a task willing to access a protected object will always gain access to it. Unfortunately, the disadvantage is that we need to incorporate the checks in line 2 in the code for all the calls to protected objects (although the check will only be executed when the task does not gain access to the protected object).

Despite the attractive these two approaches may seem, there exists a separation between the problems of the language designer and those of the programmer; and both worlds must be separated to keep them simpler. Therefore, the simple semantics proposed in section 3, combined with an appropriate algorithm for changing ceilings at the right time, provides simplicity, minimal changes to the language and the required flexibility to avoid the excessive blocking introduced by the ceiling of ceilings.

5 Conclusions

Ada 95 does not provide dynamic ceiling priorities for protected objects. Their semantic complexity and the potential implementation inefficiency have been the main concerns not to include them in the standard. Nevertheless, there are situations in which it would be very convenient to have this feature, as in the case of mode changes or dynamically adapting existing program libraries with protected objects. We have proposed a semantics for changing ceilings which is intended to fit well in Ada's current semantics and not to introduce a significant implementation overhead.

A very simple `Set_Priority` operation has been proposed that acts as a deferred ceiling change, not interfering with protected code being executed when `Set_Priority` is called. The ceiling change is atomic and can be called from any priority. The ceiling check and inheritance is performed before starting to execute protected code. As an exception, for applications that may impose it, calls being requeued without abort do not check for ceiling violations but they blindly inherit the ceiling before starting to execute. An extra priority inversion may arise from the existence of this feature as shown in section 4, but it can be bounded as far as the workload is also bounded. It is a programmer's decision what model to choose depending on requirements of the application being developed.

Other approaches have also been discussed in section 4, but all of them will necessarily introduce additional overhead or produce lack of orthogonality in the language.

As the main conclusion, dynamic ceilings should be considered in further revisions of Ada, due to their potential benefits, the standard support of dynamic ceilings in POSIX and the relative maturity acquired on the subject.

References

[1] A. Burns, A. J. Wellings, C. Bailey, and E. Fyfe. The Olympus Attitude and Orbital Control System: A Case Study in Hard Real-Time System Design and Implementation. In *Ada sans frontieres. Proceedings of the 12th Ada-Europe Conference, Lecture Notes in Computer Science 688*, pages 19–35. Springer-Verlag, 1993.

[2] J. A. de la Puente. Session summary: New language features and other language issues. In *Proceedings of IRTAW9, Ada Letters*, volume XIX, nr 2, pages 19–20, 1999.

[3] IEEE. Portable operating system interface: Amendment 2: Threads extension [C language]. IEEE/1003.1c, IEEE, 1995.

[4] C. Locke, D. Vogel, and T. Mesler. *Building a Predictable Avionics Platform in Ada: A Case Study*. Proceedings of the IEEE 12th Real Time Systems Symposium, 1991.

[5] O. Pazy and M. Kamrad. Session summary: Outstanding issues. In *Proceedings of IRTAW8, Ada Letters*, volume XVII, nr 5, pages 11–15, 1997.

[6] J. Real. *Protocolos de Cambio de Modo para Sistemas de Tiempo Real* (Mode Change Protocols for Real Time Systems). Ph.D. thesis, Universidad Politécnica de Valencia, 2000. In spanish.

[7] J. Real and A. Wellings. The ceiling protocol in multi-moded real-time systems. *Reliable Software Technologies - Ada Europe 99, Lecture Notes in Computer Science*, 1622:275–286, 1999.

[8] J. Real and A. Wellings. Dynamic ceiling priorities and Ada 95. *Ada Letters*, XIX(2):41–48, July, 1999.

[9] S. Tucker Taft and Robert A. Duff (eds.). Ada 95 Reference Manual. Language and Standard Libraries. Springer, Lecture Notes on Computer Science, vol 1246, International Standard ISO/IEC-8652:1995(E), 1995.

[10] K. Tindell and A. Alonso. A very simple protocol for mode changes in priority preemptive systems. Technical report, Universidad Politécnica de Madrid, 1996.