Priority Inversion in Multi Processor Systems due to Protected Actions

Gustaf Naeser
Department of Computer Science and Engineering,
Mälardalen University,
Sweden
gustaf.naeser@mdh.se

The use of multiple processors give rise to new issues regarding old problems of the Ada95 design.

In this paper we discuss how serviceing of protected actions, due to unspecified dispatching, can lead to unbounded priority inversion in systems with multiple processors. We also discuss different ways to resolve the problem.

Additional Key Words and Phrases: Ada95, priority inversion, multiple processors.

1. INTRODUCTION

One way to increase the performance in a system is to add more processors to share the work load. However, going from single to multiple processors can in Ada95 introduce unwanted scheduling effects, like priority inversion. This is surprising since the language was designed to support multiple processors.

Priority inversion attracted attention when Ada95 was designed. Areas like synchronisation and semaphores [5, 8], elaboration order of packages [7], and server task priorities [4] were investigated. Study of the properties of to protected objects in Ada 95 has mostly resolved around priority inheritance [1, 9] and the priority ceiling protocol [6,10]. However, the semantics of the Ada95 language for protected actions makes unbounded priority inversion possible.

In this paper we show how the protected action functionality, where queued protected calls are serviced, allows for executions where lower prioritised tasks may be scheduled and run before a higher prioritised task.

2. PRIORITY INVERSION

The definition of priority inversion given in the reference manual [2], D2.2(14), *"Priority inversion is the duration for which a task remains at the head of the highest priority ready queue while the processor executes a lower priority task. [...]"*, has some weaknesses,

a) the definition does not describe all situations that should be considered inversions, *b*) it seems to describe a single processor environment, and *c*) it relies on not completely defined rules for when tasks are made ready to run, e.g. when relaesed from protected actions. We believe that a better wording for priority inversion would be in line with those used in [3], *"Priority inversion has occured [...] when progress of a task is blocked by the actions of a lower priority task."* and [5],

"Priority inversion is the term used to describe the situation when a higher priority task's execution is delayed by lower priority tasks.". To state that lower priority tasks only can priority inverse tasks in the ready queue disqualifies valid priority inversion situations presented in this paper. We propose that any situation where a task of lower priority is running when a task of higher priority *could* be running should be treated as priority inversion.

To illustrate how priority inversions can occur, we outline a sample system and an example execution. Consider a system configuration with

(1) one protected object with an entry, PO_P which is guarded with barrier B^{PO_P}, and an entry PO_V which unlocks the barrier,

(2) two processors, and

(3) three tasks, T_h, T_{l_1} and T_{l_2}, the first with high and the two others with low priority priority.

Consider an execution where B^{PO_P} is false and T_{l_1} and T_{l_2} queue up waiting for access to PO_P. When T_h executes PO_V it will start a protected action which will service all queued calls queued on the protected object, c.f. 9.5.1(7) in [2], *"After performing an operation on a protected object other than a call on a protected function, but prior to completing the associated protected action, the entry queues (if any) of the protected object are serviced (see 9.5.3)."*. Priority inversion may occur depending on how the tasks are released from the protected action. The release mechanism is in 9.5.3(22) described as *"An implementation may perform the sequence of steps of a protected action using any thread of control; it need not be that of the task that started the protected action. If an entry_body completes without requeuing, then the corresponding calling task may be made ready without waiting for the entire protected action to complete."*.

Priority inversion occurs if the thread used for servicing tasks is that of T_h and T_h is not given a new thread, or made ready in the ready queue, when its call to the object is completed. I.e., there is nothing preventing the release mechanism for the protected action to be implemented so the task starting the protected action, T_h in the example above, service the queued calls to PO_P on behalf of the blocked tasks, T_{l_1} and T_{l_2}. Since the relation between the release order of the queued tasks and the servicing task is not specified, serviced tasks T_{l_1} and T_{l_2} may be released when their protected calls have been serviced, i.e., ahead of T_h.

This kind of behaviour does not lead to priority inversion in a single processor system since T_h will still have the highest priority, executing PO_V, and will still have the highest priority when leaving the object to resume execution of its task code. However, in a multi processor system priority inversion will occur at that moment T_{l_1} can continue its execution on an other processor while T_h will have to service the queued entry call of the other task.

In theory T_h can be starved by a steady supply of calls to the protected object if we suppose that the execution time of the protected entry is longer than the time the lower priority tasks use on the second processor to call the protected object entry again, hence we have a possibly unbounded priority inversion.

2.1 Related issues in the language

The Ada95 Rationale identifies in section 9.1 a related problem when it discusses the unbounded priority inversion that can occur if blocking is allowed within protected operations, *"By disallowing blocking within a protected operation and by also using the ceiling priority mechanism, unbounded priority inversion can be avoided. The generality that might be gained by allowing blocking would inevitably result in an increase in implementation complexity, run-time overhead, and unbounded priority inversion."*, but the release problem is not mentioned.

3. SOLUTIONS

There are a number of plausible ways to solve the inversion problem caused by the protected actions. The solutions outlined below trade ease of implementation against performance in different combinations. Some of the solutions propose drastic changes to the language specification.

3.1 Collected task release

The first solution gives a well defined behaviour of the protected action by releasing all tasks involved in the protected action when the whole action completes. In the example execution of Section 2 all tasks in the action, tasks T_h, T_{l_1} and T_{l_2}, are released and can continue execution when the protected action completes and at that time T_h will have priority precedence to get a processor.

This solution to the problem requires minimal change to Ada semantics. The only change is that the order in which tasks are released is enforced rather than left open to the implementation. However, there is still the risk that tasks with lower priority not involved in the protected action can execute before tasks in a protected action and hence priority inversion still exists in the system and it will be hard to calculate the maximum priority inversion as required in D2.2(14) [2].

In a single processor system this solution would be no different from the dynamic behaviour described in [2].

3.2 Servicing threads

A second solution is to make T_h ready when its protected call is complete but let the thread used by T_h continue servicing the calls to the the protected object. This solution avoids the priority inversion since tasks can be made ready when their protected calls are completed.

On single processor systems this solution can, in comparison to the collected task release solution, lead to additional task switches in the case where T_h should be run directly after the completion of the action. In the odd case that the last task serviced should continue execution after the servicing the thread can be used by that task.

3.3 No servicing

The third solution would be to take the task switching performance penalty full on and remove the servicing of calls. Every tasks leaves the protected object when the execution of its protected call is complete. This solution has the advantages that it is much easier to implement and has much easier semantics, but there is a

performance penalty since a task switch will be required for each queued call.

4. DISCUSSION

The different solutions outlined in the previous section suit different system setups. The current definition of protected actions works in single processor systems and the mechanism minimises the need for tasks switches by using the running thread for the execution of all calls in an action. However, as we have shown the mechanism can introduce problems in multi processor systems. Of the proposed solutions the first, collected release, is closest to the current behaviour and would require the least modifications.

In a multi processor system any solution taking advantage of multiple threads is desired, hence the second and third solutions should be preferred, action threads and no protected actions. The action threads have the advantage that a lower number of task switches are needed but there might be implementation reasons to prefer the solution where task switches are always made, as elaborated below.

One of the reasons for the introduction of protected objects was that the cost of task switching in a solution using tasks would be too high. If the RTOS is implemented in software it will also cause task switches when it needs to work. Hence the task switching will be a factor when choosing the best solution for software RTOS, i.e. the solution with the least number of task switches should be preferred. A hardware RTOS would not need to consider the task switches to and from the RTOS and hence the solutions where additional switches are imposed will still be fair.

In systems with heterogenous processors the solution with no protected actions might be the most attractive one since it yields most control to the RTOS. This control allows the RTOS to dispatch the tasks more effectively. The other two solutions will make the whole protected action execute on the same processor which, in a heterogenous system, may not be the most resource effective execution.

The complexity of the first two solutions, that use protected actions, is higher than that of the one without. Hence the solution without might be preferable if system analysis and verification is desired.

5. CONCLUSIONS

There is an obvious danger of introducing unbounded priority inversion when a single processor real-time kernel implementation, following the Ada 95 language specification, is extended to handle multiple processors. In this paper we have shown an example of how the servicing of protected actions can cause priority inversion. The source of the problem is that interpretation of when tasks are made ready and added to the the ready queue is left open to implementations of the servicing of protected actions in Ada95. Since the problem only appears in systems with multiple processors, and the use of multiple processors is becoming an attractive alternative to boost system performance we believe that this, and any, if any, similar vague definitions in the standard, can become a problem. The release problem and different solutions are detailed and discussed.

REFERENCES

[1] J. Abu-Ras, "Optimal Mutex policy in Ada 95", Ada Letters, Volume XV, Issue 6, 1995.

[2] "The Consolidated Ada Reference Manual", Springer–Verlag, LNCS 2219, 2001.

[3] Ö. Babaoğlu, K. Marzullo, and F. Schneider, "A Formalization of Priority Inversion", Technical Report UBLCS-93-4, 1993.

[4] D. Cornhill, and L. Sha, "Priority inversion in Ada", Ada Letters, Volume VII, Issue 7, 1987.

[5] S. Davari, and L. Sha, "Sources of unbounded priority inversions in real-time systems and a comparative study of possible solutions", *ACM SIGOPS Operating Systems Review*, Volume 26, Issue 2, 1992.

[6] J. Goodenough, and L. Sha, "The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks", *International Workshop on Real-time Ada Issues*, 1988.

[7] L. Lander, S. Mitra, and T. Piatkowski, "Priority inversion in Ada programs during elaboration", *Proceedings of the seventh Washington Ada symposium on Ada*, 1990.

[8] G. Levine, "The control of priority inversion in Ada", Ada Letters, Volume VIII, Issue 6 , 1988.

[9] D. Locke, L. sha, R. Rajkumar, J. Lehoczky, and G. Burns, "Priority inversion and its control: An experimental investigation", *International Workshop on Real-time Ada Issues*, 1988.

[10] R. Rajkumar, L. Sha, and P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors", *Real Time Systems Symposium*, IEEE, 1988.