Locking Policies for Multiprocessor Ada

A. Burns and A.J. Wellings
Department of Computer Science,
University of York,
YO10 5GH, UK

Abstract

Lock-based resource sharing protocols for single processor systems are well understood and supported in programming languages such as Ada. In contrast, multiprocessor resource sharing protocols are less well developed with no agreed best practice. In this paper we consider what the next version of Ada should support. Two proposals are considered, one requiring a minor change to the current language, another requiring a more substantial change.

1 Introduction

One of the significant extensions made to the definition of Ada during the Ada 2012 revision process was the support provided for programming multi-tasking systems for execution on multiprocessor platforms. The new language features, however, did not provide complete support for this paradigm. In particular, the need to support the efficient use of protected objects accessed by tasks executing in parallel was not really addressed. This lack of support is not really surprising as appropriate locking policies for such shared object have not yet been identified. Although there has been some success in determining the necessary support for scheduling [7], the issue of how best to support multiprocessor lock-based resource control protocols is far from clear. New results are emerging; indeed, there is a plethora of proposed schemes (see [10] for a review of the literature in this area, and the work of Brandenburg [4]); a consensus is, unfortunately, far from being realised.

In this paper we consider two possible policies, one that can be achieved with only minor modifications to Ada, the other requiring a new locking policy to be defined in the language. We concentrate on fixed priority scheduling.

2 Ada 2012 Provision

Ada 2012 allows a collection of processors (CPUs) to be partitioned into a distinct set of *dispatching domains*, (DDs). A task runs in only one DD. Within a DD, a task can be either bound to a single CPU, or be allowed to execute and migrate over the entire DD. In this way, the two classic ways of scheduling tasks (partitioned and global) can be supported as well as a number of semi-partitioned schemes [5].

Real programs share data; in Ada this means having concurrent access to protected objects (POs). With a single processor, an effective sharing scheme is delivered by the pragma `Locking_Policy(Ceiling_Locking)` which assigns a ceiling priority to each PO. Whenever a task enters a PO its priority is raised to the ceiling. As a result, blocking time is minimised and deadlocks are prevented (where blocking time is the time a task has to wait until a lower priority task has completed some action). As indicated above, a similar well-accepted scheme for multiprocessor platforms and parallel (not just concurrent) access does not yet exist.

The only support that Ada 2012 gives to concurrency control for POs in multiprocessor platforms is to allow `Locking_Policy(Ceiling_Locking)` to mean that a task will spin at the ceiling priority if immediate access to the PO is not possible (as another task on a different CPU is currently executing with the PO lock).

As partitioned systems provide the greatest challenge for the definition of effective locking policies, we restrict our considerations in this paper to programs that consist of a single dispatching domain onto which all tasks are statically allocated. We note that this is the recommended structure for the Ravenscar protocol when implemented on a multiprocessor architecture.

3 Blocking Time

As noted in the previous section, an effective locking scheme must bound the impact on schedulability. With a static task allocation model, an ideal scheme would imply that for any task τ_i the progress it makes towards completing before its deadline should only be negatively impacted by tasks of a higher priority running on the same processor. Putting this another way, a task should not be blocked by lower priority tasks on the same processor or by any task on another processor.

Unfortunately this ideal cannot be fully met if resources are to be shared under mutual exclusion. So if task τ_i shares a protected object (PO_j) with task τ_k then, if τ_k has a lower priority or is allocated to another CPU, blocking will occur. If all POs are only accessed by tasks on the same processor then the priority ceiling protocol [15] (as supported by `Locking_Policy(Ceiling_Locking)`) gives a minimum blocking time. Where more than one processor is involved, two issues arise; assume PO_j is being used by τ_k, and τ_i wishes to gain access:

- should τ_i continue to spin while waiting to gain access, or should it be suspended? and
- as τ_k can be locally preempted, by a local high priority task (higher priority than the ceiling of PO_j), τ_i can be held up for a relatively long interval of time.

There is considerable evidence (see [10, 4]) that spinning is the most appropriate approach to use unless protected operations are exceedingly long (which most guidelines argue against). Moreover, suspension brings with it another scheduling issue, as the state 'suspended waiting to gain access to a protected object' is not currently part of the tasking model within Ada. Therefore, it seems appropriate to use spinning as Ada currently recommends. However, to bound the time that it takes to gain access (using spinning) it must be possible to impose some form of queue on the waiting tasks (for example, a FIFO queue – see discussion below).

To deal with the second issue we consider two schemes: Non-Preemptive Locking and a Multiprocessor Resource Sharing Protocol which we name MrsP. These are now considered.

4 Non-Preemptive Locking

If all the POs are executed non-preemptively then, obviously, a task cannot be preempted while holding a PO lock. It follows that it will hold the lock for the duration of the protected action (which we will represent by a_j for PO_j). If there is FIFO spinning then the maximum time a task can wait to gain access is $(m_j - 1)a_j$ where m_j is the number of processors on which is allocated a task that uses PO_j. This represents external blocking time for the task. Its total execution time for using the PO will be $(m_j - 1)a_j + a_j$ which is just $m_j a_j$.

So for task τ_i it has *external blocking* of $(m_j - 1)a_j$ for each resource it uses, and for any local task τ_k with priority higher than τ_i it has *local blocking* of $m_j a_j$. As non-preemption is an extreme form of ceiling locking then arbitrary task τ_i will have a single local blocking term equal to the maximum value of $m_j a_j$ over all tasks τ_s with priority lower than τ_i.

To implement this approach in Ada 2020, a minor variation to the current locking policy is needed. As well as spinning, a FIFO queue is required. This could be a refinement to the definition of `Locking_Policy (Ceiling_Locking)` or could be a new pragma. Non-preemption locking can be achieved with current Ada by just assigning the ceiling priority of all POs to be higher than the priority of all tasks. Alternatively a new explicit locking policy, `Locking_Policy(Non_Preemptive_Locking)`, could be defined.

The disadvantage of the non-preemption approach is that the high priority tasks (those that typically have the shortest deadlines) suffer significant blocking (both local and external). The amount of blocking is bounded but may be too much to guarantee that deadlines will never be missed. If guarantees can be made (blocking is manageable) then there is no better protocol than non-preemption. The next protocol to be defined is for use when non-preemption is not adequate.

We note that once true parallel execution is enabled then failures such an deadlock are possible. Pure non-preemption can lead to deadlock if resource use is nested (i.e one protected object calls another). A common deadlock prevention scheme is to statically order resources and to only allow access to resources with an order number greater than that of any currently held resource. Although this may be considered a restriction on the expressive power of the computational model, it is more expressive than those protocols that ban nested usage or require a group lock – that has the same impact as non-nesting. Within Ada the use of exceptions for 'out of order' requests allow robust programs to be developed. We have investigated the use of nested but ordered locks for general multiprocessor resource control protocols elsewhere [10] (though not for non-preemption or MrsP). The adoption of ordered locking for the protocols discussed in this paper is straightforward, though not covered further here.

5 Multiprocessor Resource Sharing Protocol - MrsP

There are a number of multiprocessor priority ceiling protocols discussed in the literature, for example [13, 14, 12, 16, 9, 1, 2, 8, 11, 3]. MrsP has the distinctive property that it deals fairly with both high and low priority tasks. In this section we first define a general model for MrsP, then an Ada-specific version is defined. Finally, an implementation scheme is described.

5.1 The MrsP Protocol

If we return to the single processor notion that all activities associated with protected object PO_j occur at the ceiling priority of PO_j, then tasks of high priority are unaffected by the existence of the PO. This is clearly not true for the non-preemptive protocol. But if spinning occurs at the ceiling level then legal preemption of the lock-holding task leads to excessive blocking (as described earlier). MrsP (defined in a non-Ada paper [6]) eliminates this excess by using a 'helping scheme' [16].

We define here the MrsP protocol for non-nested PO usage (it is easily extended to nested usage). The basic aspects of the protocol are:

1. All POs are assigned a set of ceiling priorities, one per processor (for those processors that have tasks that use the resource); for processor p_k it is the maximum priority of all tasks allocated to p_k that use the resource.

2. An access request on any PO results in the priority of the task being immediately raised to the local ceiling for the PO.

3. Accesses to a PO are dealt with in a FIFO order.

4. While waiting to gain access to the resource, and while actually using the resource, the task continues to be active and executes (possible spinning) with priority equal to the local ceiling of the PO.

5. Any task waiting to gain access to a resource must be capable of undertaking the associated computation on behalf of any other waiting task.

6. This cooperating task must undertake the outstanding requests in the original FIFO order.

So all action takes place at the ceiling priority level (hence higher priority tasks are not impacted). If there is no local preemptions then a task will wait at most $(m_j - 1)a_j$ to gain access (as described earlier). But if there is a local preemption then the waiting tasks must take over the work. In the worst case task τ_i when accessing PO_j will execute the protected actions on behalf of the other tasks that want to use PO_j (but are preempted). So rather than spin unproductively the task executes the protected actions on behalf of other task ahead of it in the queue. As long as at least one task is free to execute then progress will be made. And if all tasks are locally preempted then it is acceptable for no progress to be made as all involved processors are executing tasks with higher priorities.

5.2 Ada-Specific MrsP

The general definition of MrsP required each protected object (shared resource) to have a ceiling priority per processor. Although such a protocol could be defined for Ada, here we give a more constrained protocol that is closer to current Ada. The first rule of the protocol (defined above) is replaced by:

1. All POs are assigned a ceiling priority, it is the maximum priority of all tasks that use the resource[1].

In other words, this is the current definition of the priority ceiling for a protected object. With this definition it remains the case that all actions take place at the ceiling priority level (hence higher priority tasks are not impacted), but the actual value of the ceiling might be higher than it would be with the more expressive protocol.

So, for example, if a set of tasks ($\tau_1 \ldots \tau_6$; each with priority p_i, with $p_6 > p_5$ etc.) are statically allocated to two CPUs in the following way: τ_2, τ_4 and τ_6 to CPU(1) and τ_1, τ_3 and τ_5 to CPU(2). Assume two POs (PO_x and PO_y). Let τ_1 and τ_4 access PO_x; and τ_2 and τ_4 access PO_y. Both POs therefore have the priority ceiling p_4 (one is used by tasks on different CPUs, the other is used by co-located tasks).

Let the execution time of the protected actions be A. Each task can be analysed in the following way:

τ_1 Interference from τ_3 and τ_5; no blocking, use of PO_x costs $2A$.

τ_2 Interference from τ_4 (including $2A$ from its use of PO_x) and τ_6; no blocking, use of PO_y costs A.

τ_3 Interference from τ_5; blocking, via PO_x, of $2A$.

τ_4 Interference from τ_6; blocking, via PO_y, of A; use of PO_x costs $2A$.

τ_5 No interference or blocking; ceiling priority of the POs (p_4) less then p_5.

τ_6 No interference or blocking; ceiling priority of the POs (p_4) less then p_6.

If the full MrsP protocol had been used then τ_3 would not suffer blocking as τ_1 would spin at priority p_1 (not p_4). Note also that if non-preemption was used then τ_5 and τ_6 would also suffer blocking.

[1]This requires all tasks to be assigned priorities that are globally consistent even though the task set is partitioned amongst the available processors. So, for example, deadline monotonic priorities assignment could be applied to the complete task set before it is partitioned.

5.3 Implementation of Ada-Specific MrsP

In the context of Ada the notion of one task executing the code of a protected action on behalf of another task (that has been released from an entry barrier) is part of the standard implementation model. Hence the requirements for MrsP are not exceptional.

Ada provides for tasks to migrate between CPUs in the same dispatching domain. Here we assume that tasks are statically allocated to CPUs apart from when they migrate to implement MrsP. Hence all the functionality required (and explained in the following pseudo code) would be supported in an Ada 2012 compliant run-time support system.

The basis of the implementation scheme is to dynamically associate a set of affinities with each PO[2]. An attempt to lock a PO adds a CPU to the PO's affinities. While accessing the PO, the task inherits the PO's affinities and hence it can execute on any CPU that has a waiting (spinning) task. The following pseudo code outlines a possible implementation; in this code R is the PO, t is the accessing task and p is the CPU from which the lock request is being made (ie. t is executing on p). The code also keeps track, in R(t), of the id of the task that currently holds the lock (if there is one, otherwise R(t) is null). When the request to lock R is made, the protocol raises the priority of the t to the local ceiling and sets the affinity of the PO to include its own affinity. If the PO is already locked, it sets the current lock-holder's affinity to be that of the resource and spins at the next available position in the FIFO queue. The task continues to spin until it is released (by the action of some other task calling `Unlock`).

```
Lock (R, t, p) ->
  raise priority of t to local ceiling of R
  Affinities(R) := Affinities(R) + p
  if already locked
    get current resource user R(t)
    Affinities(R(t)) := Affinities(R)
    obtain FIFO lock on R and spin
  else
    Affinities(t) := Affinities(R)
  end if
  set current lock holder to self
  raise priority of t by 1
  -- use R

Unlock(R, t, p) ->
  Affinities(R) := Affinities(R) - p
  Release next task in FIFO queue (if there is one)
  Affinities(t) := p
  lower priority of t to its base value
```

It should be noted that all code is run at the ceiling priority of the resource, and hence the protocol does not contain any non-preemptive sections.

The use of '+1' in assigning the task's priority means that the task holding the PO lock can migrate to any processor on which there is a spinning task, and then preempt that spinning task. Obviously no other task can be allocated this priority. And to be clear, the spinning task must be waiting to gain access the the same PO.

We define this new locking policy: `Locking_Policy(MRSP_Locking).`

[2]A task's affinity is the set of processor on which it is allowed to execute.

6 Global Scheduling

The MrsP approach has been defined for fully partitioned single dispatching domain (DD) systems. Here we consider how to generalise the protocol. Within a single DD if some of the involved tasks are globally scheduled (their affinities are equal the full set of CPUs) then the protocol is easily implemented as progress will always be made unless there are N higher priority tasks runnable (for N CPUs).

If there is a combination of fixed and global tasks (within the same DD) then the partitioned tasks must inherit the full set of CPUs when it is accessing a PO shared with one or more 'global' tasks.

Between DDs, the situation is less clear. What is the right means of communicating between tasks within different DDs? Perhaps only non-locking protocols should be used. It would seem inappropriate to use MrsP. One of the key properties of the Ada model, in its use of DDs, is that tasks can never migrate between DDs. The use of migration for MrsP would seem to prevent its use. One is therefore left with non-preemptive locking for POs that are shared between DDs.

7 Conclusions

To complete the definition of Ada for use on multiprocessor platforms it is necessary to define appropriate locking policies for protected objects (PO). Currently, the language recommends spinning at the ceiling priority of the PO. We argue here that, at a minimum, a queuing discipline needs to be added to this policy. And that, at least, FIFO queuing should be supported.

With FIFO spinning, non-preemptive locking (if it can deliver a schedulable system) is an acceptable policy. It is easy and efficient to implement. However, on a single processor system, non-preemption is not considered sufficient (hence the use of ceiling locking). On a multiprocessor, blocking is inevitably increased and it is therefore unlikely that non-preemption can form the basis of a general purpose solution.

The MrsP policy uses controlled migration to ensure that external and internal blocking is limited (to its minimum value). High priority tasks with short deadlines will not suffer blocking from POs with lower ceiling priorities. Clearly an implementation is required to evaluate the overheads with the policy. We recommend that such an implementation is attempted, and if successful the policy be included in Ada 2020.

References

[1] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *13th International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, pages 47–56. IEEE Computer Society, 2007.

[2] B.B. Brandenburg and J.H. Anderson. Optimality results for multiprocessor real-time locking. In *Real-Time Systems Symposium (RTSS)*, pages 49–60, 2010.

[3] B.B. Brandenburg and J.H. Anderson. Real-time resource sharing under cluster scheduling. In *Proc. EMSOFT*. ACM Press, 2011.

[4] Björn B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[5] A. Burns and A.J. Wellings. Dispatching domains for multiprocessor platforms and their representation in Ada. In J. Real and T. Vardanega, editors, *Proc. of Reliable Software Technologies - Ada-Europe 2010*, volume LNCS 6106, pages 41–53. Springer, 2010.

[6] A. Burns and A.J. Wellings. A schedulability compatible multiprocessor resource sharing protocol - MrsP. In *Under Review*, 2013.

[7] R.I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1 –35:44, 2011.

[8] D. Faggioli, G Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Proc. of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 90–99, 2010.

[9] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proc. 22nd RTSS*, pages 73–83, 2001.

[10] S. Lin, A. Burns, and A.J. Wellings. Supporting lock-based multiprocessor resource sharing protocols in real-time programming languages. *Provisionally accepted for Concurrency and Computation: Practice and Experience*, 2012.

[11] J.P. Lozi, G. Thomas, G. Muller, and J. Lawall. The remote core lock (RCL): Can migrating the execution of critical sections to remote cores improve performance? In *EuroSys11*, 2011.

[12] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[13] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proc. 9th IEEE Real-Time Systems Symposium*, pages 259–269, 1988.

[14] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for shared memory multiprocessors. In *Proc. of the 10th International Conference on Distributed Computing*, pages 116–125, 1990.

[15] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[16] H. Takada and K. Sakamura. A novel apporach to multiprogramming multiprocessor synchronization for real-time kernels. In *Proc. 18th IEEE Real-Time Systems Symposium*, pages 134–143, 1997.