Efficient Implementation of IPCP and DFP

N.C. Audsley and A. Burns
Department of Computer Science,
University of York, York, UK.
email: {neil.audsley, alan.burns}@york.ac.uk

Abstract

Most resource control protocols such as IPCP (Immediate Priority Ceiling Protocol) require a kernel system call to implement the necessary control over the shared protected object. This switch can be expensive, involving a potentially slow switch from CPU user-mode to kernel-mode (and back). In this paper we look at two anticipatory schemes (IPCP and DFP - Deadline Floor Protocol) and show how they can be implemented with the minimum number of calls on the kernel. Specifically, no kernel calls are needed when there is no contention, and only one when there is. A standard implementation would need two such calls.

1 Introduction

To support mutual exclusion over the execution of protected code Ada employs the Immediate Priority Ceiling Protocol (IPCP) for ordering accesses of fixed priority dispatched tasks executing on a single processor. For tasks scheduled by the EDF scheduling policy the Deadline Floor Protocol (DFP) has recently been advocated [1, 3, 4]. Both of these protocols are *anticipatory* [7] as they make changes to the task's run-time parameters (either priority or deadline) to prevent problematic scenarios developing (e.g. two tasks executing concurrently within the same protected object).

The cost of this anticipatory action is that overheads are incurred even though the problematic scenarios may rarely develop. Indeed the probability of being preempted during the execution of a relatively short protected action by a task that will also want to use the same protected object may be very rare. Perhaps only a very low percentage of all protected calls actually need to be protected. Brandenburg and Anderson [2] claim that in many (soft) real-time workloads locks may be acquired many thousand times a second; a reduction in the average time it takes to implement such locks is therefore advantageous.

The overheads of making an anticipatory change to a task's priority (or deadline) can be expensive. Many kernels and RTOSs require this action to be performed via a system call, potentially crossing protection domains from user to kernel address space, requiring a switch from user to kernel mode in the CPU. Where such a move is required, a considerable overhead is imposed – potentially adding several thousand CPU cycles to the cost of changing priorities [6]. This potential expense has a clear impact on Ada Run-Time overheads[1].

To counter this problem Linux, for example, supports *futexes* [5]: fast user-space mutexes – *"a mechanism that supports efficient lock implementations with low average-case overheads. By exporting lock-state information to userspace, futexes avoid expensive system calls when a lock is uncontended, which is arguably the common case in well-designed systems"* [7].

In this paper we consider the development of protocols for IPCP and DFP that:

- Reduce the cost of executing protected actions that are not contended.
- Do not significantly extend the cost of executing protected actions that are contended.

We follow the intuition of Spliet et al. [7] although they did not address IPCP (or DFP).

We first consider non-nested protected objects (POs). Then we extend the approach to nested POs and then to EDF scheduling and DFP.

2 IPCP, Non-nested Protected Actions

We consider first non-nested protected actions in POs without entries. We do not, however, cover the case of POs acting as interrupt handlers (i.e. a PO with an interrupt level priority ceiling). Each task is assigned a (base) priority and each PO has a static ceiling priority. Assume the kernel has a routine for changing the priority of a task: `K.Set_Priority`. To acquire a PO the task must, via code generated by the compiler, change its priority to the ceiling of the PO by calling this routine. As indicated above this could be a relatively expensive operation involving a move to kernel-mode from user-mode (and back). A standard implementation would undertake the following (in addition there would be checks that could raise exceptions, but we do not consider these here).

[1] Some Ada run-times, eg. ORK, avoid the issue by running all code as supervisor, with no cross-protection domain crossing overhead for a system call. This also holds for some Real-Time Linux implementations (where all code runs effectively in supervisor mode). In this paper we consider the more general case of Ada running in an environment where kernel-user protection is enforced, with the inherent cross-protection domain system call overhead.

```
acquire(PO) :-
  K.Set_Priority(PO.ceiling) -- kernel knows id of executing task and PO
```

We do not use the term 'lock' as an actual OS lock may not be necessary.
To release the PO a second kernel call:

```
release(PO) :-
  K.Set_Priority(base_priority)
```

The implementation model defined is this paper has two key properties:

- If a task's execution of the protected action is uninterrupted then no changes are made to the task's priority.
- If a task is preempted during its execution of a protected action then the kernel will (belatedly) raise the priority of the task, and the task will subsequently lower its priority (via a call on the kernel) when the protected action is completed.

To obtain this more efficient implementation we define a number of variables (per task) in task user space. These state variable could be in the task control block, TCB (i.e. be user defined attributes) as long as access to these variables does not involve a switch to kernel mode. If the TCB cannot be used then some other reserved locations must be employed.

- `base_pri` – base priority of the task, may already be available
- `new_pri` – potentially higher priority for task
- `to_raise` – boolean, set to true if task should have a higher priority
- `leaving` – boolean flag to indicate task has started to leave the PO, initialised to false.

Each PO also has a variable in user space:

- `ceiling` – ceiling priority of the PO

We assume in this work that a task does not change its base priority and, similarly, that a PO does not have its priority ceiling value changed. These modifications could however be added in a straightforward way.

When a task wishes to access a PO then the rules of IPCP (as incorporated into the semantics for Ada) determine that the PO must be free (on a single processor) so the following can be executed entirely in user mode:

```
acquire(PO) :-
  new_pri := PO.ceiling -- notes new priority but does not change
  to_raise := true -- indicates priority should be raised
```

If the task gets to the release of the PO without being preempted then it just resets the to_raise flag. If the task is preempted during its execution within the PO then there must have been an event (clock or other interrupt) that itself caused a switch to CPU kernel-mode (to perform the kernel system call). During the system call the task's priority will have been raised and so on the release of the PO the task must lower its priority. To prevent a race condition (and to not utilise a potentially expensive test-and-set operation) a flag is used to indicate that the release operation has started.

```
release(PO) :-
  leaving := true
  if to_raise then
    to_raise := false -- no preemption
  else
    K.Set_Priority(base_pri)
  end if
  leaving := false
```

Within the kernel, if there is a call to release a previously suspended task, then it must execute the following code. Note the kernel must know the task that was executing (with id current). Action must be taken if the to_raise flag is set but the leaving one is not.

```
if not current.leaving and current.to_raise then
  K.Set_Priority(current, current.new_pri)
  current.to_raise := false
end if
```

Note the kernel must be able to access and modify the user-level variables leaving and to_raise.

As the kernel is non-preemptive (in respect of the user task) then the single leaving flag is sufficient to prevent race conditions. If the leaving flag has not been set, the kernel (whilst in kernel mode anyway) will raise the priority of the task. The task will then reset it to the base level (using a kernel-level call). Alternatively if leaving has been set then the task has, in effect, left the PO and hence no priority changes are required.

If we now consider POs which have entries. If there is one or more tasks suspended on a PO entry then the release code may have to undertake some post-processing to determine if any of the blocked tasks can now proceed. The checking can be done at the lower priority if there is no contention, but the kernel may need to be invoked if tasks are made runnable again.

In summary, if there is no contention, then no code is executed in kernel-mode. If there is contention then a single switch is needed (during the release PO code). By comparison the 'normal' implementation requires two separate kernel actions. As a result not only will the average execution time of non-contented accesses be reduced but also the worst-case cost of contented accesses.

3 IPCP, Nested PO

For nested PO calls we take the view that once there is a preemption then real priority changes must be implemented until the outermost call is completed. So the `to_raise` flag is only set at the outermost level. A new variable, `level`, is introduced to keep track of the nesting level, and a new PO variable is employed to capture the priority of the task as it enters the PO (`PO.old_pri`). A task starts with `level = 0` and `new_pri = base_pri`. The two code segments are now as follows.

```
acquire(PO) :-
  PO.old_pri := new_pri
  new_pri := PO.ceiling
  level := level + 1
  if level > 1 and not to_raise then
    K.Set_Priority(new_pri)
  else
    to_raise := true
  end if

release(PO) :-
  level := level - 1
  leaving := true
  new_pri := PO.old_pri
  if to_raise then
    if level = 0 then
      to_raise := false
    end if
  else
    K.Set_Priority(new_pri)
  end if
  leaving := false
```

The kernel code now makes the priority change on all relevant occasions and so does not exploit the fact that in the outer PO the task will be assigned its current priority. Further variables could be included to remove this potentially wasteful priority change. However here we focus on a simple intuitive scheme.

```
if not (leaving and level = 0) and current.to_raise then
  K.Set_Priority(current, current.new_pri)
  current.to_raise := false
end if
```

This protocol is not completely straightforward, and would therefore need to be proven by, for example, model checking before being deployed.

4 DFP

Here we can get more advantage. With EDF scheduling the absolute deadline of each task is used to order execution. When a task enters a PO its deadline is potentially reduced to the time of entry plus the (relative) deadline floor of the PO. The deadline floor is the shortest relative deadline of all tasks that make use of the PO.

Currently the clock must be read on every acquire. Assume that `old_ deadline` holds the current absolute deadline of the task, and that `K.Set_Deadline` is the kernel routine that changes the deadline of the task.

```
acquire(PO) :-
  new_deadline := clock + PO.floor
  if old_deadline > new_deadline then
    K.Set_Deadline(new_deadline)
```

To release the PO:

```
release(PO) :-
  K.Set_Deadline(old_deadline) -- could be a null-op if deadline
                               -- did not change during acquire.
```

The efficient implementation would only need to call the clock if there is an actual context switch. We consider only the non-nested case. The following additional variable is needed:

- `new_rel_deadline` – Potentially higher relative deadline for the task

The code is straightforward and follows the form of the IPCP approach.

```
acquire(PO) :-
  new_rel_deadline := PO.floor
  to_raise := true

release(PO) :-
  leaving := true
  if to_raise then
    to_raise := false -- no preemption
  else
    K.Set_Deadline(old_deadline)
  end if
  leaving := false
```

The kernel would execute:

```
if not current.leaving and current.to_raise then
  current.new_deadline := clock + current.new_rel_deadline
  if current.old_deadline > current.new_deadline then
    K.Set_Deadline(current.new_deadline)
    current.to_raise := false
  end if
end if
```

It is easy to prove that this late access to the clock does not break the deadline floor protocol.

Proof Sketch

Let task τ_i be executing and at time t call a PO with deadline floor of D^F. The deadline of the task should be reduced to $t + D^F$. While executing within the PO (with its original absolute deadline) another task, τ_j, is released at time s ($t < s$). The modified protocol will now reduce the deadline of τ_i to $s + D^F$; the newly release task will have a deadline of $s + D_j$ – where D_j is the relative deadline of the task. If the newly released task preempts τ_i then it must have an earlier deadline, so $D_j < D^F$. But if it is then going to access the PO, by definition of the deadline floor, we have $D^F \leq D_j$. So if it preempts it cannot call any PO that is currently in use. If it does not preempt then it will not run until, at the earliest, τ_i leaves the PO.

5 Conclusion

This paper has discussed an approach that should lead to a reduction in the cost of implementing the necessary protocol for ensuring mutual exclusive access to protected actions on a single processor platform. The reduced cost will be particularly marked on implementations that require an expensive switch to kernel-mode operation in order to make the necessary changes to a task's priority (or absolute deadline).

The paper has introduced a protocol, but this would need to be proven before it could be used. Moreover an implementation is required to evaluate the actual performance of the approach to see to what extent the potential cost saving can be demonstrated.

The approach applies to both the fixed priority protocol, IPCP, and the EDF protocol, DFP. In the latter the cost saving is likely to be more as the need to read the real-time clock, for uncontended PO accesses, is removed.

References

[1] M. Aldea, A. Burns, M. Gutirrez, and M. González Harbour. Incorporating the deadline floor protocol in Ada. *ACM SIGAda Ada Letters – Proc. of IRTAW 16*, XXXIII(2):49–58, 2013.

[2] B.B. Brandenburg and J.H. Anderson. Feather-trace: A light-weight event tracing toolkit. In *Proc. OSPERT*, 2007.

[3] A. Burns. A Deadline-Floor Inheritance Protocol for EDF Scheduled Real-Time Systems with Resource Sharing. Technical Report YCS-2012-476, Department of Computer Science, University of York, UK, 2012.

[4] A. Burns, M. Gutierrez, M. Aldea, and M. González Harbour. A deadline-floor inheritance protocol for EDF scheduled embedded real-time systems with resource sharing. *IEEE Transactions on Computers*, (Online PrePrints), 2014.

[5] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *AUUG Conference Proceedings*, pages 85–97. AUUG, Inc., 2002.

[6] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07. ACM, 2007.

[7] R. Spliet, M. Vanga, B.B. Brandenburg, and S. Dziadek. Fast on average, predictable in the worst case: Exploring real-time futexes in litmus. In *Proc. IEEE Real-Time Systems Symposium*, pages 96–105. IEEE, 2014.