Reducing the Implementation Overheads of IPCP and DFP

H. Almatary, N.C. Audsley and A. Burns
Department of Computer Science,
University of York, York, UK.
email: {hmka501, 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 any shared data. This call 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. The protocols developed are verified by the use of model checking. A prototype implementation is described for POSIX pThreads (thus opening up improvements to a range of programming approaches). Experimental results demonstrate the effectiveness of the scheme, showing average case savings of 86%.**

I. Introduction

The use of priority ceiling protocols is the most effective means of providing mutual exclusion over critical sections of code on single processor systems. There are two forms of these protocols. In the original [19], a task only changes its priority when there is actual contention for the mutualy exclusive resource (ie. shared data). This protocol, however, has a number of implementation costs and hence the form that is more usually used, in for example the Ada and Java programming languages, is IPCP (Immediate Priority Ceiling Protocol)[1]. With this protocol the priority of a task is raised whenever it uses a mutually exclusive resource regardless of whether there is actual contention or not.

For tasks scheduled by the EDF scheduling policy the Deadline Floor Protocol (DFP) has recently been advocated [9]. Here a task's absolute deadline is reduced whenever it accesses a mutually exclusive region, again regardless of whether there is actual contention.

Both of these protocols are *anticipatory* [20] 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 mutually exclusive resource). Both protocols also have the significant property that they deliver mutual exclusion without the use of an actual lock as long as the application code is restricted to not self-suspend while executing in the mutually exclusive region. From this the deadlock free property can also be proved [11], [17].

The cost of the 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 mutually exclusive region by a task that will also want to use the same region may be very rare. Perhaps only a very low percentage of all protected calls actually need to be protected. Brandenburg and Anderson [6] claim that in many (soft) real-time workloads locks may be acquired many thousand times a second; a reduction in the average and worst-case 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 [15]. This potential expense has a clear impact on RTOS overheads[2].

To counter the problem of kernel overheads for simple locks, Linux, for example, supports *futexes* [12]: 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"* [20]. However, this approach still requires system calls for a contended lock, or for changing a task priority (eg. if futexes were used within an IPCP implementation).

We shall in this paper use the term *protected action* (PA) and *protected object* (PO) to indicate the code that must be executed under mutual exclusion (hence controlled by IPCP / DFP). And the concurrent entity is termed a *task* (implemented by a *thread*). Our aim is to develop protocols for IPCP and DFP that:

[1]Also called the Priority Protect Protocol in POSIX and Priority Ceiling Emulation in Real-Time Java.

[2]Some RTOSs (including some Real-Time Linux implementations) avoid this issue by running all code as supervisor, with no cross-protection domain crossing overhead for a system call. In this paper we consider the more general case of RTOSs in an environment where kernel-user protection is enforced (as will be required in safety-critical or mixed critical systems), with the inherent cross-protection domain system call overhead.

- Significantly reduce the cost of executing PAs that are not contended.
- Actually reduce the cost of contented PAs.
- Do not significantly extend the cost of other kernel operations such as context switches.

We follow the intuition of Spliet et al. [20] and Zuepke et al [21] although neither address IPCP (or DFP), or provide proof of correctness.

Following a short review of previous work, in section III we first consider non-nested protected actions. Then in section IV we extend the approach to EDF scheduling and DFP. Nested actions are then considered in section V. We verify the protocols by the use of model checking, this is covered in section VI. A prototype implementation under Linux is used to derive evidence as to the efficacy of the approach. This is described in section VII. Finally some conclusions are given in section VIII. Note, an initial description of the basic approach within the context of the Ada programming language, but without proof or implementation, is contained in a workshop paper [2].

II. Previous Work

Although almost all RTOSs and concurrent programming languages provide support for synchronisations such as mutual exclusion, there has been little work on improving the efficiency of their implementation. One exception to this is the work reported at the last RTSS by Spliet et al. [20]. They considered the improvements that can be made to the implementation of PCP (original Priority Ceiling Protocol for single processor systems), MPCP (Multiprocessor Priority Ceiling Protocol) and FMLP (the FIFO Multiprocessor Locking Protocol) [5], [7]. The testbed they used was LITMUS(RT), which is a real-time extension of the Linux kernel [7], [10]. They demonstrated: "*substantial improvements in the uncontended case (e.g., a futex implementation of the PCP lowers lock acquisition and release overheads by up to 75% and 92%, respectively), at the expense of some increases in worst-case overhead on par with Linuxs existing futex implementation*". The reader is referred to this paper for a review of how futexes were introduced into Linux.

As indicated above, Spliet et al. [20] do not address IPCP or DFP, which is the focus of the work reported here. Zuepke et al [21], [22] do address some of the issues surrounding IPCP but do not verify their informal descriptions nor apply their ideas to pThreads. In the context of L4, lazy task switching has been addressed [16].

By comparison we show how the protocols can be verified. This is particularly important for the nested use case where correctness is not intuitively obvious. Our implementation work is undertaken within the context of pThreads and Real-Time Linux which is a popular platform for implementing real-time systems. Moreover, we provide access to our implementation (see Appendix A).

III. IPCP, Non-nested Protected Actions

We consider first non-nested protected actions in protected objects (POs). Each task is assigned a (base) priority and each PO has a static ceiling priority which is the maximum priority of any task that accesses it.

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).

```
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 variables must be in user memory space (eg. in the task control block, TCB) so that access to them does not involve a switch to kernel mode.

- `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
- `start` – 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 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) :-
  start := true
  if to_raise then
    to_raise := false -- no preemption
  else
    K.Set_Priority(base_pri)
  end if
  start := 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 `start` one is not.

```
if not current.start 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 `start` and `to_raise`.

As the kernel is non-preemptive (in respect of the user task) then the single `start` flag is sufficient to prevent race conditions. If the `start` 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 `start` has been set then the task has, in effect, left the PO and hence no priority changes are required.

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. Evidence for this assertion is provided in section VII. Proof of the correctness of the protocol is provided in section VI where model checking is employed.

IV. DFP

The above analysis of IPCP is all within the context of fixed priority scheduling (sometimes called *Rate Monotonic Scheduling*). When EDF (Earliest Deadline First) scheduling is used other protocols are required to deliver mutual exclusion. Up to recently the main protocol for EDF was Baker's Stack Resource Protocol (SRP) [3], [4]. This behaves in a similar way to IPCP and hence an implementation similar to that for fixed priority scheduling is possible.

However, the Deadline Floor Protocol (DFP) has recently been advocated [1], [8], [9] for EDF-based systems. DFP is defined by the following:

- Each task has a relative deadline.
- Each PO also has a relative deadline, which is the minimum of the relative deadlines of all tasks that call (directly or indirectly) that PO – hence the name *deadline floor*.
- When a task calls a PO its absolute deadline may be reduced:
  - If a task calls a PO at time t, its deadline is reduced to the minimum of its current absolute deadline and t + the deadline floor of the PO.

So, for example, if a task with an absolute deadline of 10000 accesses at time 9950 a PO with a deadline floor value of 25 it deadline will be reduced to 9975 for the duration of its execution within the PO.

It has been shown that DFP has all the key properties of SRP: in particular (on a single processor) it is deadlock free, delivers mutual exclusion, and ensures that a task cannot be blocked once it starts executing. Indeed DFP has the same worst-case scheduling behaviour (i.e the same blocking term in the processor-demand schedulability analysis).

To obtain the time of the call of the PO a clock must be read. So with the above definition of the protocol the clock must be read on every access to the PO. 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; the following pseudo code describes the actions required to acquire and release the PO:

```
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.
```

It is assumed that the kernel would need to be called in order to alter the key parameter: task deadline. The call to the clock may or may not need to involve the kernel, but it is again an overhead that it would be useful to remove/minimise.

An implementation of 'standard' DFP (in the MaRTE operating system) [9] has demonstrated that DFP can be implemented more efficiently than SRP. Here we note that an even more efficient implementation would only need to call the clock or the kernel if there is an actual context switch.

The following additional variable is needed for the non-nested case:

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

The kernel would execute:

```
if not current.start 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
```

Note the call of 'clock' in the computation of `current.new_deadline`.

To complete this discussion on the efficient implementation of DFP it is necessary to prove that the late reading of the clock does not invalidate the properties of DFP.

Theorem 1: In the implementation of DFP the late reading of the clock does not break mutual exclusion.

Proof: Let task τ_i be executing and at time t call a PO with deadline floor of D^F. Assume that 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 $s + D_j < s + D^F$, that is $D_j < D^F$.

But if τ_j then accesses the PO, by definition of the deadline floor, we have $D^F \leq D_j$. This provides the contradiction, τ_j cannot access the PO, and hence mutual exclusion is preserved.

So, if τ_j preempts τ_i it cannot call any PO that is currently in use by τ_i. If it does not preempt then it will not run until, at the earliest, when τ_i leaves the PO. □

V. 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`, `to_raise = false` 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
  start := 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
  start := 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 (start 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, again model checking is used to verify correctness – see section VI.

VI. Verification of the Protocols

To verify the protocols we employed model checking using the UPPAAL tool [14]. Model checking is able to explore all possible states that a system can get into (when utilising a concurrency protocol), but only for the particular scenario being checked. So, for example, later in this section we will show that the protocol works correctly for nested PO actions to a depth of 10 – but that does not 'prove' the protocol is correct for all levels of nesting. To assert this stronger property one needs to argue that any problems with the protocol would have manifest themselves within the finite but significant levels of nesting being checked.

The form the protocols take is for a low priority task to be executing (within a PO or in the entry or exit code) when a higher priority task is released. Two tasks are therefore sufficient to validate the protocols – we shall return to this point later.

Models are sets of timed automata. Within these automaton clocks can be defined (`clk` in the following models, with one clock per automaton). Within each automaton a task moves

between states. A task can stay in a state as long as the state invariant is true, and can leave a state by a transition if the precondition for that transition is true. Tasks can communicate via synchronous channels.

In this section we first check the non-nested scheme and then move onto the more complex nested protocol. We only consider IPCP; the DFP can be checked by a direct application of the same models.

A. Non-Nested POs

We first define our basic approach and demonstrate that without some form of protection mutual exclusion cannot be obtained.

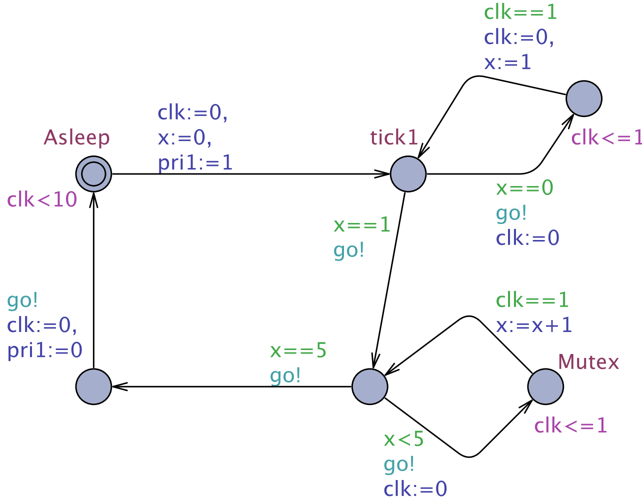

Fig. 1. Simple Task with Low Priority (Task1)

To model fixed priority scheduling, on a single processor, we force all tasks to communicate with a scheduler whenever they undertake *any* action. All actions are therefore broken down to atomic steps with interleaved synchronisation on the urgent `go` channels[3]. So in Figures 1 and 2 there are two tasks that are identical apart from their priorities. Both tasks awake from the sleep state and immediately set their priorities and move to state `tick1`. They then loop around a busy loop taking a tick of computation time each iteration (note in the diagrams the loop counter is set to just 1). They then move to the PO represented by an entry state and the `Mutex` state where they execute (within the PO) for a tick of computation. This is repeated 5 times – that is the critical section is modelled as lasting for 5 ticks with a communication with the scheduler every tick. It is therefore possible for them to be preempted while 'spinning' into and out of the `Mutex` state.

State `Mutex` is an example of a state that has all the normal attributes. It has an invariant `clk<=1` that implies the state must be left before the local clock gets a value greater then 1. It also has a precondition to leave the state: `clk==1`. Finally it has an action that is taken as the state is vacated: `x:=x+1`.

After five ticks the tasks moves away from the PO and moves to the `Asleep` state. It will leave this state at a time somewhere in the interval [0,10).

This pattern, of waking, executing, executing within `Mutex` and then sleeping again is repeated indefinitely.

[3] An urgent channel forces synchronisation immediately once both participants are prepared to communicate.

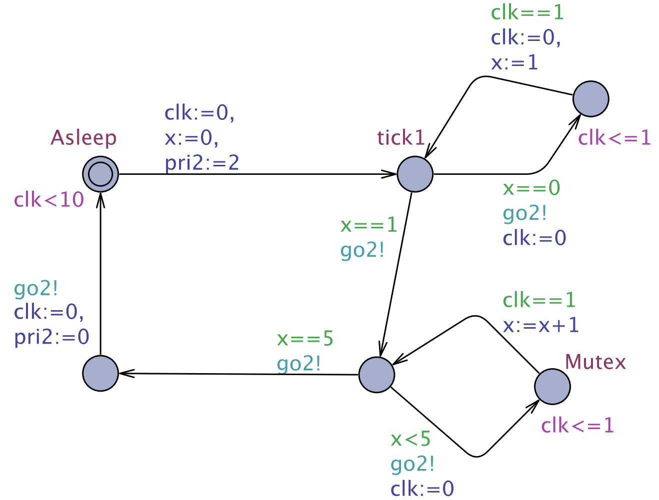

Fig. 2. Simple Task with Higher Priority (Task2)

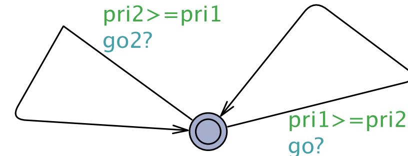

Fig. 3. The Scheduler

The scheduler (see Figure 3) simply cycles around and is prepared to synchronise on any of the go channels. But it will only communicate with the higher priority task. So a task drops its priority to 0 before it sleeps. Note the larger the integer the higher the priority in all these models.

To see that this behaviour leads to a break of mutual exclusion it is sufficient to ask the checker:

```
E <> Task1.Mutex and Task2.Mutex
```

i.e. is it possible for both tasks to be in their `Mutex` states at the same time. The tool's output is of course yes: `Property is satisfied`.

To demonstrate that IPCP works, Figures 4 and 5 are modified versions of the previous automaton with just the addition of the raising of each task's priority to 4 (the assumed ceiling for the PO) before entry to the `Mutex` state. Now the

request for E <> Task1.Mutex and Task2.Mutex produces (in red): Property is not satisfied.

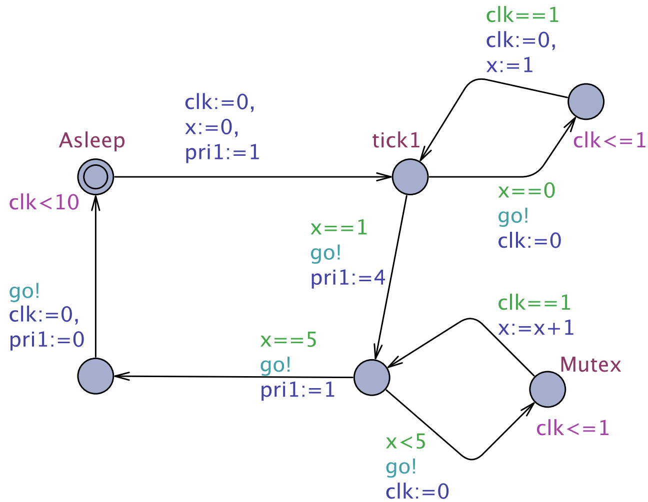

Fig. 4. Incorporation of IPCP (Task1)

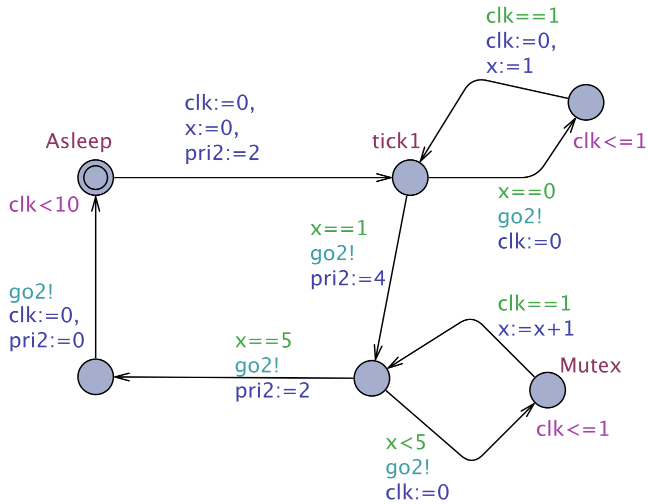

Fig. 5. Incorporation of IPCP(Task2)

We can now move to model the non-nested version of the protocol. The models are direct representations of the protocols defined in section III. Note the boolean variables start and to_raise are represented in the models by integers (start and ToRaise) that are constrained to only take on the values 0 (false) and 1 (true).

The lower priority task (only) is modified to undertake the pre and post steps before entry to the Mutex state. This is illustrated in Figures 6. Note this task makes no change to its own priority, but records the priority it should be raised to in the variable pri1new (new_pri in the protocol description).

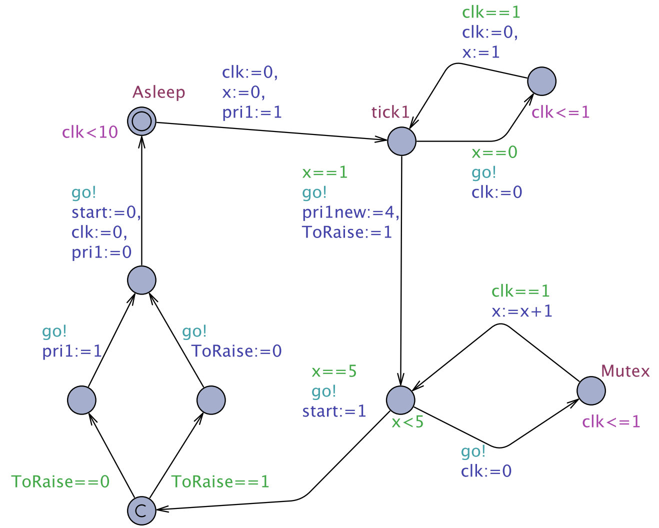

Fig. 6. Lower priority task (Task1) with pre and post steps

To model the behaviour of the kernel, when it is invoked to release a higher priority task (than the one currently executing), the actual higher priority task undertakes the operations of the kernel before it continues with its own behaviour. This is illustrated in Figures 7.

So when this task is released (from its Asleep state) it first raises its priority to 5 (making this part of the kernel's actions effectively non-preemptive). Then it checks to see if there was a lower priority task running with an artificially low priority. It there is then the priority of this task is raised to the value it should have – that is pri1 := pri1new . After making these changes it reduces its priority to its own level (i.e. 2) and proceeds as before. Note the states marked with a C are 'committed'; the automaton cannot stay in that state. In effect they are used to implement an 'if' statement.

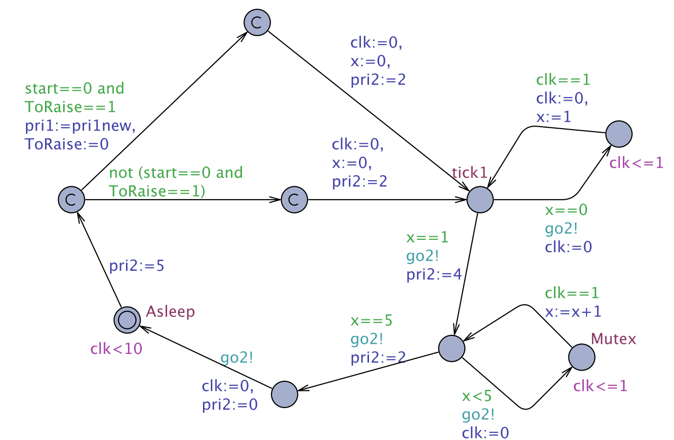

Fig. 7. Higher priority task (Task2) with kernel's functions

The model checker again tests the assertion

E <> Task1.Mutex and Task2.Mutex

and finds:

`Property is not satisfied.`

In addition we can check the effectiveness of the protocol by showing that it is possible for Task1 to be in the `Mutex` state with priority 4 and with priority 1; i.e

`E <> Task1.Mutex and pri1 == 1`

and

`E <> Task1.Mutex and pri1 == 4;`

both are proved. But any other value is not satisfiable.

The above modes have used only two automata (tasks). Of course a real system will have many tasks, but if a number of tasks are released at the same time then the kernel would just make one change to the preempted task's priority. There can only be one 'executing' task, and hence there is nothing to be gained by modelling more tasks.

B. Nested POs

We now move on to the nested case where arguable the full power of a model checker is needed. The models are necessarily more complex. We hardwire a maximum nesting level, in the associated diagrams (Figures 8 and 9) this is 10. The level the automaton is actual in is represented by the variable `level`. Each PO has a ceiling that is higher for the more nested levels, this is set to be `4 + (2*level)`. The priority that the task should have (if the protocol was not been used) is held in the array `POold[]`.

The low priority task/automaton loops round going into the `Mutex` state and then reentering this state a maximum of 10 times. So in effect there are 10 such states represented in the automaton.

There are a number of experiments that can be undertaken with this model. With the parameters represented in the Figures the priority of the high priority task is lower than the smallest ceiling and so whenever Task2 is asleep, Task1 has priority 1 (this is proven by the assertion `E <> Task2.Asleep and pri1 == 1` being satisfied, but `E <> Task2.Asleep and pri1 > 1` not being satisfied). The simple test, `E <> Task1.Mutex and Task2.Mutex`, is not satisfied.

If we now change the parameters of the model so that pri2 is 10 then this task will be allowed to call a PO with priority ceiling greater than or equal to 10, but not allowed to call one of lower ceiling priority. Note this is not prevented in the model (but would be in any implementation – as it is with Ada for example).

So let the ceiling of the PO be 12. We now have, for example:

`E <> Task1.Mutex and Task2.Mutex and level == 2`

being satisfied (as Task2 cannot call the `Mutex` in which Task1 is executing). But

`E <> Task1.Mutex and Task2.Mutex and level == 6`

is not satisfied (as there could be concurrent access to the same PO (i.e. same `Mutex`).

The verification described above, together with other checks undertaken, allows a high level of confidence to be assigned to the correctness of the protocols.

VII. Prototype Implementation

In this section we describe our implementation of the support needed within system software for the protocols introduced within the paper. The basic implementation is in terms of pThreads and Real-Time Linux. pThreads is chosen as it is used by many real-time programming languages (eg. Ada, Java) to provide the main concurrency support (and is also used within C / C++[4]). Therefore, any improvements made within pThreads due to the protocols are immediately available within common real-time programming environments. We note that one restriction of the choice of pThreads is that there is only support for IPCP (in terms of the `PTHREAD_PRIO_PROTECT` mutex attribute [18]), and not DFP, as deadline scheduling is not currently supported within standard pThreads.

A. Standard pThreads / Linux

pThreads provides both lock and mutex operations to enable synchronisation between threads over shared data. Locks are basic mechanisms, whilst mutexes are more sophisticated and are used within protocols such as IPCP. Within pThreads, these protocols are attributes of a mutex – with attribute `PTHREAD_PRIO_PROTECT` corresponding to IPCP, and the more basic `PTHREAD_PRIO_INHERIT` corresponding to priority inheritance.

pThreads is specified within POSIX as an API (with suggested semantics), with the real-time extensions to POSIX including additions to pThreads to aid real-time (e.g. the `PTHREAD_PRIO_PROTECT` and `PTHREAD_PRIO_INHERIT` protocols) [13]. The implementation of pThreads was originally in userspace only, as a library. One problem with such implementations was that the kernel had no knowledge of the pThreads within a process, so that if one thread makes a blocking system call to the kernel (eg. for I/O) then the whole process would be blocked, even if other threads contained in the process were runnable.

More usually today, pThreads are implemented with the knowledge of the underlying OS, so that the system call blocking problem is largely removed. This is the implementation within standard Linux, where pThreads are mapped to kernel threads. However, whilst much of the required pThreads support can be implemented in user space and linked into applications (ie. `libpthread.so`, noting that this is part of `glibc`), some operations must be passed to the kernel –

[4]We note C, C++ are not real-time programming languages *per se*, but are used extensively for programming real-time concurrent systems.

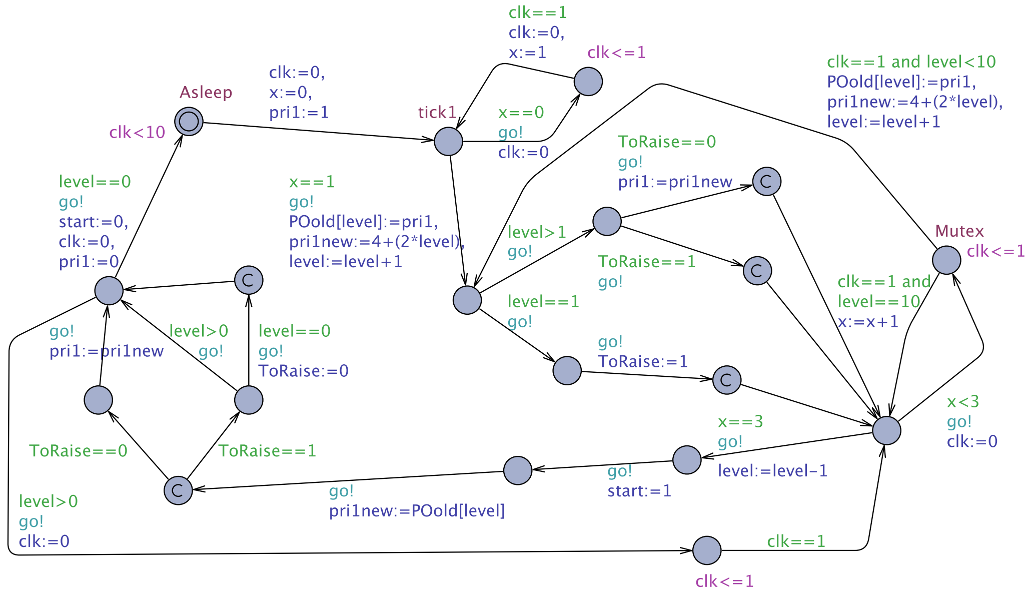

Fig. 8. Low priority task (Task1) with nested POs

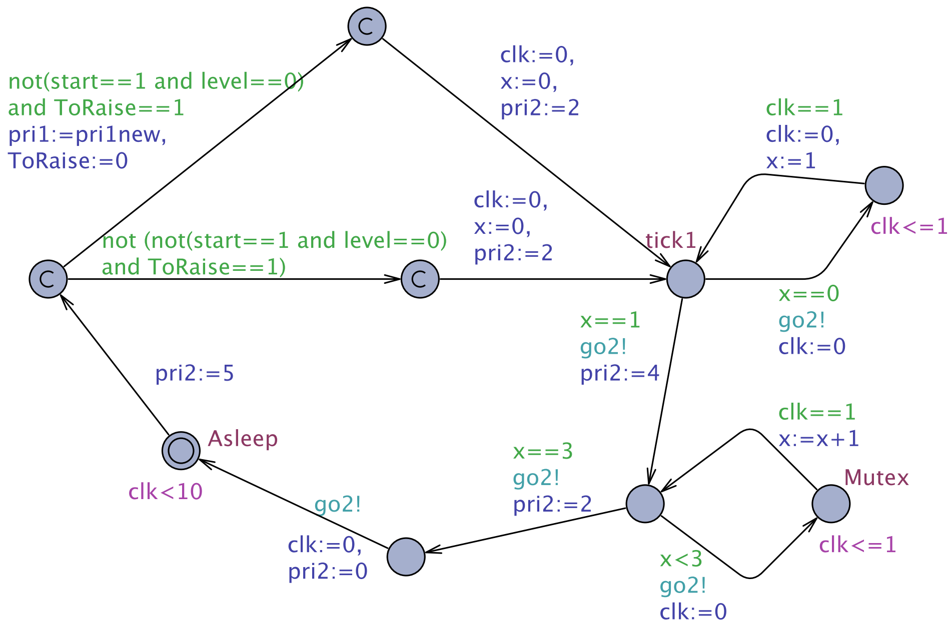

Fig. 9. Higher priority task (Task2) with nested POs

eg. operations that affect scheduling, such as priority change, must be passed to the kernel, as the kernel is responsible for scheduling kernel threads (ie. processes). Thus priority changing operations within pThreads must be passed to the kernel – eg. mutex locking and unlocking under IPCP must make a system call to alter priority.[5]

The overheads of pThread locking / unlocking system calls are reduced by implementing pThreads using *futexes* [12]: fast user-space mutexes. This removes the necessity of a system call when the lock is uncontended. However, this approach still requires system calls for changing a task priority, or for a contended lock.

[5] ie. locking operations `pthread_mutex_lock` `pthread_mutex_unlock` [13].

B. pThreads / Linux Changes

The changes required to implement the protocols proposed in this paper are now described (see Appendix A for details of where to get code patches etc.), assuming `glibc-2.21` and Linux kernel version 4.0. Changes are in three main areas: to the pThreads implementation within `glibc`, to the system

call interface within glibc, and to the kernel itself. These are now described.

1) pThreads: Changes are required to support the new per-thread flags, so york_context is added within the per thread control block (TCB) state; ie. within struct pthread:

```
struct
{   int needs_update;
    int pending_priority;
    int leaving;
} york_context;
```

Major savings in mutex costs are made by ensuring that only the per-thread flags are set when a pthread_mutex_lock or pthread_mutex_unlock call is made, and system calls to change priority are made only when required as part of the new IPCP protocol detailed in this paper.

Both pthread_mutex_lock and pthread_mutex_unlock invoke __pthread_tpp_change_priority. When called by pthread_mutex_lock. We modify it to set per-thread flags (in york_context), with the kernel responsible for (potentially) changing priority on a subsequent scheduling operation (see later). When called by pthread_mutex_unlock where a priority change has not been made by the kernel, the per-thread flags are changed appropriately. If a priority change is required __pthread_tpp_change_priority makes a sched_setscheduler system call to actually change the priority.

Modifications to __pthread_tpp_change_priority are illustrated:

```
// if called from mutex lock
if (!self->york_context.leaving)
{   self->york_context.needs_update = 1;
     self->york_context.pending_priority
                    = newpriomax;
}
// if called from mutex unlock, & priority
//  didn't change within the kernel
else if (self->york_context.leaving &&
         self->york_context.needs_update)
{   self->york_context.needs_update = 0;
    self->york_context.leaving = 0;
}
// if called from mutex unlock, and
// priority did change within the kernel
else
{    if (__sched_setscheduler
           (self->tid, self->schedpolicy,
            &sp) < 0)
         result = errno;
     self->york_context.leaving = 0;
}
```

Version	Function	ns	System Call
Standard	pthread_mutex_lock	63389	✔
	pthread_mutex_unlock (contended)	8739	✔
	pthread_mutex_unlock (uncontended)	3072	
New	pthread_mutex_lock	1979	
	pthread_mutex_unlock (contended)	8070	✔
	pthread_mutex_unlock (uncontended)	1790	
Other	dummy system call	1675	✔
	sched_setparam (kernel part)	7130	
	context switch + kernel scheduling	8805	✔

TABLE I
MICRO-BENCHMARK RESULTS (NANOSECONDS)

2) System Call Interface: To ensure that the kernel knows when pThreads are created that need to be controlled by the new implementation, the clone system call preamble is altered (within glibc) to pass a new flag (CLONE_YORK). An additional parameter is also passed, that of the location of the thread's TCB, so that the kernel has the location of its york_context (see above) for changing priorities correctly if needed. This code is architecture dependent and not described further here (see Appendix A to download full source).

3) Linux Kernel: When creating a new kernel thread as a result of the clone system call, if the CLONE_YORK flag is set, the kernel saves the location of the new thread's york_context. During a scheduling operation (called via __sched__schedule) the scheduler checks to see if the pre-empted task is a new thread (ie. with a york_context), and by checking the flags in york_context if it needs its priority changing (achieved via the kernel part of the sched_setscheduler system call). At this point threads in the scheduling queues will have the correct priority for scheduling.

This code is not described further here (see Appendix A to download full source).

C. Evaluation

The evaluation was carried out on an Intel Core i7-2630QM CPU, running at 800MHz (i.e. power saving mode). We utilise one core in the CPU only, with the CPU having access to 6GB DDR.

A micro benchmarking approach was used, with the significant operations within the protocols measured in nanoseconds. The evaluation results are given in Table I, where ticks indicate that the call involves a system call. Each result represents an average of 100 executions of the particular operation. Timings were achieved using calls to clock_gettime, noting that this can be used from both user space and kernel space without a system call.

In terms of pThreads, it is clear that the reduction in the cost of a mutex lock is significant, from 63389ns to 1979ns. However, for the former, a scheduling operation in the kernel will take 8805s, and for the latter an additional 7130ns for the sched_setparam (kernel part) call if required. This total is still significantly less than the native (a total of 9109ns rather than 63389ns) – a reduction of 86%.

The cost of unlocking mutexes is improved marginally for contended and significantly for uncontended, where the just having to set flags (rather than execute all of the standard code) realises the improvement.

VIII. Conclusion

This paper has discussed an approach that should lead to a reduction in the cost of implementing the necessary protocols 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 both for non-nested and nested actions that require mutual exclusion. The protocols developed are then verified by the use of model checking. The assertion that the protocols lead to more efficient implementation of the standard immediate priority ceiling protocol is investigated by a prototype implementation that does indeed demonstrate a significant reduction in system overheads.

The approach developed 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] N.C. Audsley and A. Burns. Efficient implementation of ipcp and dfp. *ACM SIGAda Ada Letters, Proc. of IRTAW 17, to appear*, 2015.

[3] T.P. Baker. A stack-based resource allocation policy for realtime processes. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 191–200, 1990.

[4] T.P. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, 3(1), March 1991.

[5] B. B. Brandenburg. The FMLP+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *ECRTS*, pages 61–71, 2014.

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

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

[8] 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.

[9] 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*, 64(5):1241–1253, 2015.

[10] J.M. Calandrino, H. Leontyev, A. Block, U.C. Devi, and J.H. Anderson. LITMUS(RT): A testbed for empirically comparing real-time multiprocessor schedulers. In *Real-Time Systems Symposium (RTSS)*, pages 111–126. IEEE, 2006.

[11] B. Dutertre. Formal analysis of the priority ceiling protocol. In *Proc. of the 23rd Real-Time Systems Symposium*, pages 151–160, 2002.

[12] 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.

[13] Institute of Electrical and Electronics Engineers. *POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)*. 1995.

[14] K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1/2):134 – 152, 1997.

[15] 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.

[16] Jochen Liedtke and Horst Wenske. Lazy process switching. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 15–18. IEEE, 2001.

[17] A. Burns M.Pilling and K.Raymond. Formal specification and proofs of inheritance protocols for real-time scheduling. *Software Engineering Journal*, 5(5):263–279, 1990.

[18] Open Group. pthread_mutexattr_setprotocol (The Single UNIX Specification, Version 2). http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_mutexattr_setprotocol.html, 1997. [Online; accessed 13-May-2015].

[19] L. Sha, Rajkumar R., Son S., and Chang C-H. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, 1991.

[20] 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.

[21] A. Zuepke, M. Bommert, and R. Kaiser. Fast User Space Priority Switching. In *10th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERTS)*, pages 23–29, 2014.

[22] A. Zuepke, M. Bommert, and D. Lohmann. AUTOBEST: A United AUTOSAR-OS and ARINC 653 Kernel. In *Proc. of the 20th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

Appendix A: Download of Code

This describes how to setup and run the modified pThreads and Linux kernel:

1) Download linux-4.0 kernel (http://www.kernel.org)
2) Download and apply the kernel patch (https://rtslab.wikispaces.com/Experiment+Source+Code)
3) Clone `glibc` (GLIBC270415):
   - `git clone git://sourceware.org/git/glibc.git`
   - `cd glibc`
   - `git checkout --track -b local_glibc-2.21 origin/release/2.21/master`
4) Apply glibc patch (https://rtslab.wikispaces.com/Experiment+Source+Code).
5) Build linux kernel:
   - make menuconfig; make; make install
6) Build `glibc` from another directory:
   - ../glibc/configure –prefix=$GLIBC_INSTALL_PATH
   - make; make install
7) Reboot with the new kernel.

Now you can compile a POSIX C program with our newly installed glibc using the following command:

- `gcc -g -O0 -Wl, --rpath=$GLIBC_INSTALL_PATH/lib/ -Wl, --dynamic-linker=$GLIBC_INSTALL_PATH/lib/ld-linux-x86-64.so.2 -pthread posix.c -o posix.elf`