CENTRE FOR INDUSTRIAL CONTROL SCIENCE

Department of Electrical and Computer Engineering

The University of Newcastle
N.S.W. 2308, Australia

THE PRIORITY DISINHERITANCE PROBLEM

P.J. Moylan, R.E. Betz, R.H. Middleton

Technical Report EE9345
October 1993

THE PRIORITY DISINHERITANCE PROBLEM

P.J. Moylan, R.E. Betz, R.H. Middleton
Department of Electrical and Computer Engineering
The University of Newcastle
N.S.W. 2308, Australia
peter@ee.newcastle.edu.au
reb@ee.newcastle.edu.au
rick@ee.newcastle.edu.au
Fax: +61 49 60 1712

Abstract

In software systems using preemptive scheduling based on task priorities, it is desirable to include a priority inheritance mechanism. This is an arrangement by which a task's priority is temporarily increased when it is blocking a task of higher priority. Although it is easy to work out when and how to increase a task's priority, the subsequent reduction of that task's priority involves some hidden traps. It will be shown that the "obvious" solutions are flawed, in that they can reduce the priority either too early or too late.

This paper gives two classes of solutions: one for the case where time-slicing is allowed, and one where it is not. It turns out that prohibiting time-slicing among tasks of equal priority has a major impact on the theoretical results, but does not result in major changes in terms of implementation algorithms.

Keywords: Priority inheritance, scheduling, real-time systems.

1. Introduction

In software systems which used preemptive scheduling based on task priorities, it is by now well known that critical section protection can create a problem of *unbounded priority inversion*, where a high-priority task can be delayed for unreasonable amounts of time while waiting to enter a critical section. The problem occurs when (a) a low-priority task is inside a critical section; (b) as the result of an external interrupt, a high-priority task runs; (c) the high-priority task is blocked trying to enter a critical section guarded by the same lock; and (d) the low-priority task does not run because there are some intermediate-priority tasks also able to run.

An obvious solution to this problem is to use *priority inheritance*, where the priority of a task can be temporarily raised to reflect the priority of any higher-priority task which it is blocking. That is, if task T_2 is blocked by task T_1, and T_2 has a higher priority than T_1, then the priority of T_1 is raised to that of T_2 until such time as T_1 is no longer blocking T_2. This ensures that T_2 cannot be further delayed by tasks of intermediate priority.

Although this sounds simple, attractive, and easy to implement, there are some hidden traps. The most difficult implementation problem is what we will call the *disinheritance problem*: given that a task has had its priority temporarily increased, when do we lower the priority again; and to what level do we lower it? The goal of this paper is to demonstrate why this is a problem, and to indicate solutions.

Let us consider three potential approaches to the disinheritance problem:

Strategy 1. Ensure that a task's priority is always equal to the highest priority of any task which it is (directly or indirectly) blocking.

Strategy 2. Demote a task, at the time it leaves a critical section, to the priority it had when it entered that critical section. This is the rule given in [1], and we suspect – although documentation on this point is hard to find – that it is the solution adopted by the majority of implementers. This is relatively easy to implement, since the task's old priority can be stored in the lock; but it has a tendency to drop a task's priority too soon.

Strategy 3. Restore a task to its normal priority as it leaves the outermost of nested critical sections. This strategy is obviously too conservative, in that it sometimes leaves the task's priority high for too long; but it is somewhat safer than strategy 2.

Strategy 1 is, of course, the most accurate interpretation of pure priority inheritance. As will be shown in section 3, strategies 1 and 2 are not equivalent; and in fact strategy 2 is an unsatisfactory rule, in that it can lead to high-priority tasks being blocked for significant periods of time.

As it happens, reference [1] is mainly concerned with an inheritance mechanism known as the Priority Ceiling Protocol [2], for which the disinheritance problem appears to be simpler. Our concern in this paper is for (a modified version of) what [1] calls the Basic Inheritance Protocol. The difference between the two is that the Priority Ceiling Protocol has rather more stringent conditions for allowing a task to obtain a lock; this makes it safer, but sometimes overly cautious.

We have, incidentally, been unable to find significant performance differences between the Priority Ceiling Protocol and the basic inheritance mechanism studied in the present paper. There are examples where the former has better performance, and other examples where the latter is better, but no consistent trend has emerged. The basic inheritance method is a little easier to implement, and seems to have less "system overhead" than the Priority Ceiling Protocol; but even in that respect the differences are minor in terms of performance.

2. Semaphores and Locks

Suppose that binary semaphores are used as the task blocking mechanism. (This is an inessential assumption, but it helps to keep the terminology more concrete.) Ideally, any P() operation on a semaphore should raise the priority of the task which is going to execute the corresponding V(), unless of course its priority is already high enough. In practice, nobody has yet come up with a reasonable method of identifying that other task, especially where there are several potential sources of the V()[1], so that the general priority inheritance problem remains unsolved.

The one situation where a good inheritance algorithm is possible is in the case where binary semaphores are used to protect critical sections. This is the case covered in this paper, and is also what is handled by other approaches such as that in [2]. To make this clear, let us define a *lock* as a binary semaphore used only for critical section protection. The term (general) *semaphore* is then reserved for semaphores used for other purposes (resource counters, intertask synchronisation, and so on).

With this as background, a task can be in any of the following states:

(a) Running: actually executing instructions on the processor;

(b) Ready: able to run as soon as the processor is available. To simplify the notation, we also consider the running task to be a ready task;

(c) Blocked: unable to run because it is waiting to obtain a lock;

(d) Inactive: unable to run for some other reason.

In practice, an inactive task is one which is not competing for system resources because it is held up on some resource other than a lock. For the purposes of this paper, it is convenient to depart slightly from standard terminology and not use the term "blocked" for this case.

[1]Some results are possible for the special case of a producer-consumer chain; but this is just one out of several common cases.

We shall at times use the phrases “entering the system” and “leaving the system” to describe tasks which leave and enter the inactive state, respectively.

In the sequel, the following assumptions will be needed.

Assumption 1. Inactive tasks do not hold any locks.

Assumption 2. No task is ever deadlocked.

Both of these are conditions which must be satisfied by the applications programmer. The algorithms to be given in this paper do not automatically prevent deadlock. They can, with minor modifications, detect it.

3. An example

Consider a system of four tasks T_1, T_2, T_3, T_4, and two locks L_{13} and L_{14}. As a mnemonic aid (for this example only), the subscripts on the locks show which tasks share those locks, and the task subscripts show the task priorities. Suppose that T_1 executes the sequence of operations

```
Obtain(L13)...Obtain(L14)
...Release(L14)...Release(L13)
```

and consider the following sequence of events.

(a) T_1 is initially the only task able to run, and it successfully enters both of its critical sections;

(b) T_3 arrives, runs, and is blocked on an `Obtain(L13)`. As a result, the priority of T_1 is increased to 3, and T_1 continues to run;

(c) T_2 arrives, but does not run because T_1 has a higher priority;

(d) T_4 arrives, runs, and is blocked on an `Obtain(L14)`. As a result, the priority of T_1 is increased to 4, and T_1 continues to run;

(e) T_1 executes its `Release(T14)`, thereby unblocking T_4. The priority of T_1 also drops at this point;

(f) T_4 runs, and (as is typical for high-priority tasks) quickly completes its execution.

This is illustrated in Figure 1. For each task, the height of the graph represents its current priority. Shaded sections indicate where the task is unable to run, either because it is blocked or because there is a higher-priority task in the system.

The interesting question here is what happens after (f). At this stage T_4 is no longer wanting to run, T_3 is blocked by T_1, and T_2 and T_1 are able to run. Logically, T_1 should run at this stage, since it is blocking the highest-priority remaining task. If, however, we adopt the strategy of dropping the priority of a task, as it leaves a critical section, to the priority it had on

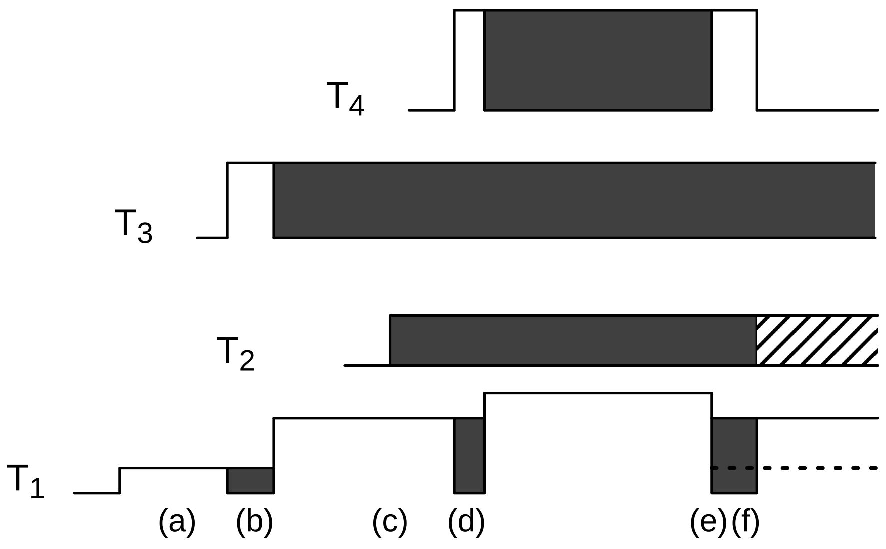

Fig 1 – An example of priority inheritance

entry to that section, then the priority of T_1 will drop to 1 at that point – as shown by the dotted line in the diagram – and T_2 will run. This can result in T_3 being blocked for an unacceptably long time.

This is for Strategy 2 of Section 1. If instead we use Strategy 3, T_1 will not drop its priority at step (e), so that T_4 is delayed until the Release(L_{13}). In either case, the results are unacceptable.

The problem we must solve, then, is to find a mechanism which allows any task which still blocks a higher-priority task to *continue* to have its inherited priority for as long as the blocking persists – but not any longer than this.

4. Defining priority inheritance

Since the “intuitively obvious” methods of defining inheritance and disinheritance can lead to undesired results, we need a more careful definition of Priority Inheritance.

For obvious reasons, we assume preemptive scheduling throughout this paper. This term has the conventional meaning: if task T has a higher (dynamic) priority than the currently running task, then there is an immediate task switch to T as soon as T enters the system, or becomes unblocked, or obtains this higher priority, as appropriate. Equivalently, a task can run only if all higher-priority tasks are blocked or inactive.

Definition 1. The set $\mathfrak{B}(T, t)$ is the set of all tasks directly blocked by task T at time t.

Definition 2. The set $\mathfrak{B}^+(T, t)$ is the set of all tasks directly or indirectly blocked by task T at time t. That is, $T_1 \in \mathfrak{B}^+(T_2,t)$ if $T_1 \in \mathfrak{B}(T_2,t)$ or if $T_1 \in \mathfrak{B}(T_3,t)$ for some $T_3 \in \mathfrak{B}^+(T_2,t)$.

Definition 3. The set $\mathfrak{B}^*(T, t)$ is the union of $\mathfrak{B}^+(T, t)$ and the singleton set {T}.

The inclusion of T in the set $\mathfrak{B}^*(T, t)$ is not, of course, intended to mean that T blocks itself. This definition is primarily to simplify some notation. As it happens, however, there is an implementation advantage in acting as if T blocks itself. It simplifies the decision as to when the priority of T should be reduced to its base priority.

Notation. A task T has two priorities: a base priority BP(T), which is assigned by the programmer, and a dynamic priority DP(T,t) which is the priority used for all scheduling decisions.

Definition 4. A preemptive scheduling method is a Priority Inheritance method if scheduling is based on the dynamic priority of a task given by

$$DP(T,t) = \max\{BP(T_j),\ T_j \in \mathfrak{B}^*(T,t)\}$$

Let us represent blocking in the system by a directed graph. Each node in the graph is a task, and there is a link from T_1 to T_2 iff T_2 is directly blocked by T_1. Clearly, we can find all instances of indirect blocking by following the links.

A cycle in the graph would indicate deadlock. If we assume that there is no deadlock, then the graph is a forest of trees. (Note: it is the programmer’s responsibility to avoid deadlock; the algorithms which we shall present in later sections do not automatically prevent deadlock, although they do allow deadlock to be detected.) The root nodes of these trees are precisely the ready or running tasks. $\mathfrak{B}(T,t)$ is the set of children of T, and $\mathfrak{B}^*(T,t)$ is the entire subtree of which T is the root node.

Note that we say that a blocked task is blocked *on* a lock and blocked *by* another task. In the traditional implementation of binary semaphores, the blocked-on information is used (either explicitly or implicitly) and the blocked-by information is discarded. When priority inheritance is used, both sets of information are potentially needed. The blocked-on information tells us which task to unblock when a lock becomes available, and the blocked-by information allows us to propagate the inherited priorities.

While the definition of blocked-on is clear enough, the meaning of blocked-by needs more thought. What do we mean by saying that task T_1 is blocked by task T_2? In this paper, we take it to mean that

(a) T_1 is trying to obtain a lock L_1, but is prevented from doing so because T_2 is holding a lock L_2, where either L_2=L_1 or there is some policy in place (e.g. a grouping of locks into equivalence classes) which prevents T_1 from obtaining L_1 as long as T_2 holds L_2;

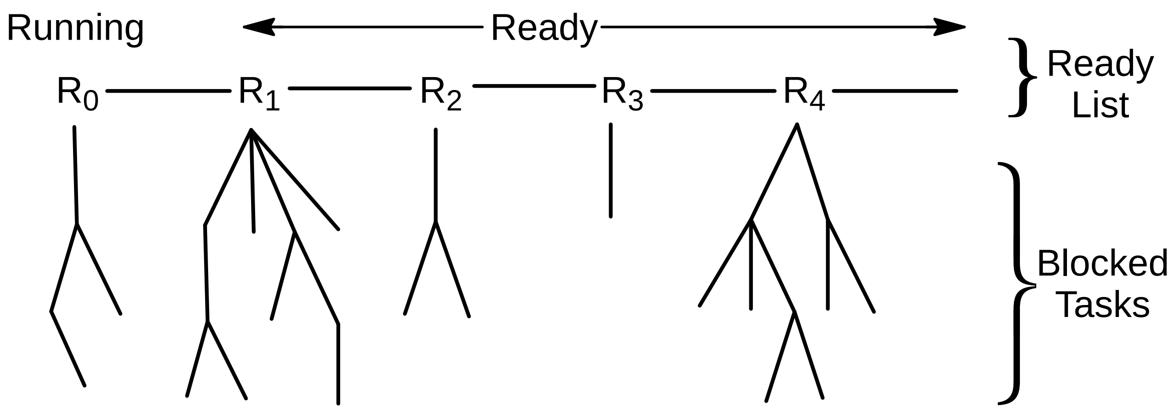

Fig 2 – The ready list, with blocked tasks attached

(b) T_1 will be granted lock L_1 when T_1 is no longer blocked by any task, and this unblocking can happen only when the blocking task releases a lock.

This description is still a little vague, because it is describing a class of algorithms rather than one explicit algorithm. To be more explicit, let us consider some of the possibilities.

Policy 1. T_1 is blocked by T_2 if T_2 holds the lock which T_1 is trying to obtain. This is the "obvious" definition of the blocked-by relation, but it is not necessarily the best.

Policy 2. T_1 is blocked by T_2 if (a) T_2 holds the lock L which T_1 is trying to obtain, and T_1 is at the head of the list of tasks blocked on L; or (b) T_2 is the predecessor of T_1 on the list of tasks blocked on L.

Policy 3. T_1 is blocked by T_2 if (a) T_2 holds the lock L which T_1 is trying to obtain, and T_2 is a ready or running task; or (b) the lock L is held by a blocked task T_3, and T_3 is blocked by T_2.

Policy 4. Suppose that we assign priorities to locks as well as to tasks; let LP(L) denote the (static) priority of lock L, as assigned by the programmer. We can define T_1 to be blocked by T_2 if (a) of all locks which are currently held by any task, T_2 holds the lock L of highest priority LP(L); (b) T_1 is attempting to obtain a lock (not necessarily the same one); and (c) $T_1 \neq T_2$, and the dynamic priority of T_1 is less than or equal to LP(L).

Policies 1-3 are equivalent from the viewpoint of the external caller, but are different in terms of internal implementation. (They differ in terms of precisely when a blocked task inherits a new priority, but all three policies assign the same priorities to all ready tasks.) Policy 2 minimises the width of the blocking trees, and thereby directly identifies the task to be unblocked when a lock is released. Policy 3, at the other extreme, eliminates transitive blocking by minimising the height of the blocking trees.

Policy 4 is different both internally and externally from the others. Its distinctive feature is that at any given time there is a *unique* "blocker"; that is, all blocked tasks are blocked by the same task. We conjecture that Policy 4 is precisely equivalent to the Priority Ceiling Protocol [1], but so far we have not found a formal proof of this conjecture.[2]

One way to keep track of all active tasks in the system is via a priority-ordered ready list, with each ready task carrying along the tree of tasks which it blocks. This is illustrated in Figure 2, where R_0 is the currently running task and R_1, R_2, ... are the ready tasks. For convenience, we picture the running task as being at the head of the ready list. This might or might not be reflected in the actual implementation.

[2]In principle, this equivalence should be easy to prove. The difficulty lies in the use of different specification methods. Our approach, via Definition 4, is to specify the properties that an implementation should have, and then to look for possible implementations. The definition in [1] is effectively an algorithmic definition, which leaves us with the difficult problem of proving the equivalence of two dissimilar algorithms.

In the next section, we shall need the following results.

Lemma 1. For any task T and time t, $DP(T,t) \geq BP(T_i)$ for all $T_i \in \mathfrak{B}^*(T,t)$.

Proof. Obvious from Definition 4. ❒

Lemma 2. For any task T and time t there is a task $T_1 \in \mathfrak{B}^*(T,t)$ with $BP(T_1)=DP(T,t)$.

Proof. This follows directly from Definition 4. ❒

Theorem 1. The dynamic priority of the running task is greater than or equal to that of any other active task.

Proof. Let DP(T,t)=p, where T is the running task; and suppose that there is some other task T_1 for which $DP(T_1,t)=q>p$. Clearly T_1 cannot be a ready task, since then T could not have been the running task, but we still have to dispose of the possibility that T_1 is a blocked task.

From Lemma 2, there is a task $T_2 \in \mathfrak{B}^*(T_1,t)$ with $BP(T_2)=q$. Let T_3 be the root node of the blocking tree containing T_2. From Definition 4, $DP(T_3,t) \geq q>p$. That is, there exists a ready task whose dynamic priority is greater than that of T, which contradicts the hypothesis that T was the running task. ❒

Note that the proof of Theorem 1 explicitly relies on the assumption that there is no deadlock.

5. A general scheduling algorithm

In this section, we present a general solution to the problem of keeping track of priorities. The basic idea is to keep, for each task, a count of the number of tasks it is directly blocking at each priority level; and to remember, for each lock, the priority of the task which is to be unblocked when the lock becomes available.

The solution is general in that it makes only weak assumptions about the ordering of the list of ready tasks. Of course we always want that list to be priority-ordered, but there is some scope for varying the ordering of tasks of equal priority. By not insisting that tasks of equal priority be ordered in any particular way, we get a solution which is suitable for systems which use time-slicing among tasks of equal priority. (This also goes part way towards providing a solution for multiprocessor systems.) We also make no assumption as to the ordering of tasks on a blocked list; some implementers would prefer the tasks to be priority-ordered, and some would prefer a FIFO order.

In a later section, we shall show that simpler and more efficient solutions are possible if we make stronger assumptions about the ordering of tasks of equal priority. Those assumptions will, however, rule out the option of time-slicing.

To compute the dynamic priority of a task, we do not need to keep complete information about the blocking trees; it suffices to count the number of tasks in a tree at each priority level. With each task T we can associate a set of counts

T.count1[j] = the number of tasks in $\mathfrak{B}^*(T,t)$ whose base priority is j

From Definition 4, DP(T,t) is obviously the largest j for which T.count1[j] is nonzero. Unfortunately, T.count1[j] can be expensive to compute. (When a task becomes blocked, it might itself be blocking other tasks, so potentially the entire count1 array has to be propagated up the blocking tree.) We can save some overhead by instead working with

T.count2[j] = $\delta(j,BP(T))$ + (the number of tasks in $\mathfrak{B}(T,t)$ with dynamic priority j)

where $\delta(i,j)=1$ if i=j, and $\delta(i,j)=0$ otherwise. It is easy to show, from Lemma 2, that T.count2[j]=0 if and only if T.count1[j]=0, so the count2 array still encodes the information we need but is easier to compute.

Since a task's dynamic priority never drops below its base priority, the counts need not be accurate for $j \leq BP(T)$. (In practice, however, this observation does not seem to lead to any simplifications.*)* Note that we have to initialise T.count2[BP(T)] to 1 – that is, we create the fiction that a task is blocked by itself.

Appendix A gives some pseudo-code based on these ideas. The code appears to contain some redundant computations; this is because Appendix A actually describes a class of algorithms, with some of the details left unspecified. For example, one might expect that T.BlockedBy is normally equal to

T.BlockedOn.Holder for any blocked task T, but in fact that is true only for Policy 1 of Section 4. Similarly, we have left unspecified the detail of which task to unblock when several tasks are blocked on the same lock. After details such as this are decided, a certain amount of “fine tuning” of the code is possible.

The recursion in procedure `Promote` is, of course, inessential, and is present only for clarity. In practice we would replace the tail recursion by a loop.

The code shown for the `Release` operation is not completely general, since it relies on the following assumption: if a task releases a lock, and if one or more tasks were waiting for that lock, then one of those tasks will obtain the lock. This is true for Policies 1-3 of Section 4, but it is not necessarily true for Policy 4. We have chosen not to illustrate the completely general case, on the grounds that this procedure is already a complex one.

A non-obvious point in the `Release` operation is the way the “blocked by” status of some tasks changes when another task becomes unblocked. Suppose that T_1 holds a lock L, and it unblocks T_2 by releasing L: then all other tasks which were blocked on L now become blocked by T_2 instead of T_1 – see Figure 3. (This diagram is for the case of Policy 1 in Section 4; but similar diagrams can be drawn for the other “blocked by” definitions.) This operation could have been simplified by the alternative approach of unblocking *all* tasks blocked on L and then forcing them to re-compete for the lock. It turns out, however – see Section 6.4 – that that approach causes redundant task switching, thereby giving an algorithm which is not attractive for practical implementation.

The final part of the “Release” code relies on the following result.

Theorem 2. Suppose that the running task T_1 releases a lock at time t, and define $p=DP(T_1,t^-)$. Then, immediately after time t, (a) there does not exist any ready task with dynamic priority greater than p; and (b) there exists a ready task T_2 (with the possibility that $T_2=T_1$) such that $DP(T_2,t^+)=p$.

Proof. From Theorem 1, no task has dynamic priority greater than p at time t^-. From Lemma 1, we deduce that no task has base priority greater than p at time t^-. The same must be true at time t^+, since no new task has entered the system. Applying Definition 4, we have part (a) of the Theorem.

When T_1 releases a lock, this causes a rearrangement of the blocking trees, but does not alter the overall set of tasks in the system. From Lemma 2, there is some task T_3 in $\mathfrak{B}^*(T_1,t^-)$ with $BP(T_3)=p$. After the rearrangement, T_3 must be in one of the resulting trees. That is, T_3 must be in either $\mathfrak{B}^*(T_1,t^+)$ or $\mathfrak{B}^*(T_2,t^+)$, where T_2 is a some other ready task. (Typically, but not necessarily, T_2 is a newly unblocked task.) In the former case the priority of T_1 does not drop; in the latter case, the new dynamic priority of T_2 is equal to the old priority of T_1. ❐

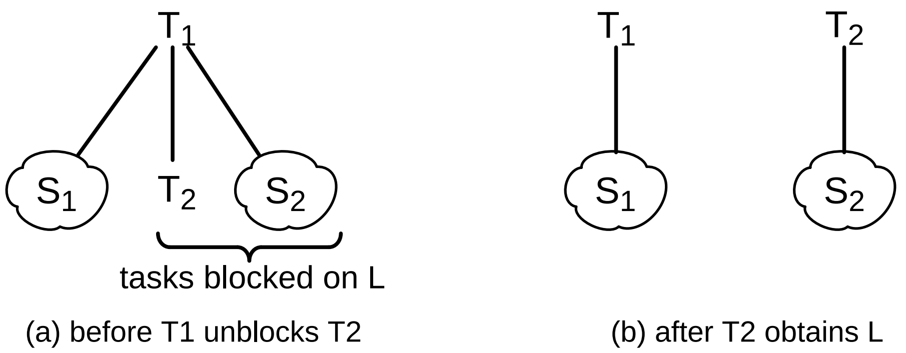

Fig 3 – Change in the blocking trees when a lock is released. This diagram is for the case where the blocker of a task is defined to be the holder of the lock on which the task is blocked. Similar diagrams can be obtained for other definitions of the “blocked by” relation.

This implies that one of two things must happen: either the running task (whose priority must have been at least as high as that of any other task in the system) keeps its old priority, and therefore continues to be eligible to be the running task; or the running task drops its priority, in which case we must perform a task switch to another task *whose dynamic priority is the same as the old priority of the original task.* We distinguish between these two possibilities by looking at the "count" fields.

It is interesting to observe that the dynamic priority of the running task remains constant – although the identity of the running task may change – except when the running task becomes inactive or when a new task, of priority higher than that of the running task, arrives. This is because when the running task becomes blocked it is always replaced by a task of the same dynamic priority, either because it already had that priority or because it gained it through inheritance; and when the running task drops its priority, there is always another task of the same priority to replace it, from Theorem 2.

This observation has an important consequence for practical implementations. As will be seen in Section 6, we can sometimes avoid having to keep an explicit record of the dynamic priority of each task. Instead, we can simply record the dynamic priority of the running task. From Theorem 2, this changes only when tasks enter or leave the system.

6. Some explicit implementations

In this section, we consider four actual implementations of the Lock and Release mechanisms, based on the different definitions of "blocked by" which were introduced in Section 4. We have coded all of these implementations, and tuned the code in each case for the best performance we could manage. The descriptions below are for the final code after tuning.

6.1. The Inherit1 algorithm

This is the most basic version of the priority inheritance, based on Policy 1 in Section 4: the blocker of a task is defined to be the task holding the lock which the current task is trying to obtain.

The implementation is almost identical to that shown in Appendix A. The main difference is that it turns out to be unnecessary to maintain any "Blocked By" information, since that information is already available from the "Holder" field of a Lock record. For this and similar reasons, the final code is actually a little simpler than shown in Appendix A.

This algorithm is compact and reasonably efficient, and has the advantage that it is not too hard to see from the code how the priority inheritance is working. It is not, however, the most efficient implementation we have been able to obtain.

6.2. The Inherit2 algorithm

We get a rather different implementation, although the same external behaviour, when we use the "blocked by" definition of Policy 2 in Section 4.

The obvious way to keep a task list in this case is to use a doubly linked list, where the "previous" pointer does double duty as a "blocked by" pointer. Superficially, this approach is attractive because tasks are kept on this list in the order in which they are going to be unblocked.

In practice, problems arise because the priority inheritance implies that from time to time we have to reposition tasks on this list. It turns out that this causes major problems in updating the "count" information when the priority of a task changes. These problems can be solved, but at the cost of some significant execution time overheads. We therefore conclude that this is not an attractive algorithm for practical implementation.

6.3. The Inherit3 algorithm

This algorithm, which is based on Policy 3 in Section 4, is built around the notion that the blocker of a task is its ultimate blocker. For example, if T_1 holds a lock which T_2 is trying to obtain, and T_2 is holding a lock which T_3 is trying to obtain, then we say that both T_2 and T_3 are blocked by T_1. That is, we have defined transitive blocking out of existence! This has the interesting consequence that the blocking trees all have unit height, so that

(a) task promotion is very simple, since it does not have to ripple up the tree;

(b) the dynamic priority of a blocked task is always equal to its base priority.

The only real trouble spot in implementing this is in the redistribution, as illustrated in Figure 3, which must be done when a lock is released. In principle we have to recompute the blocker of every task which was blocked by the task holding the lock. While this is not too hard to do – and is certainly less complex than the corresponding problem in Section 6.2 – it is an overhead which we would prefer to avoid if possible.

The solution is to avoid maintaining an explicit representation of the “Blocked By” relation. Instead, a task finds its blocker by checking the holder of the lock on which it is blocked, by checking further if that task is also blocked, and so on until an unblocked task is found. Although this does require looping, the loop is a tight one and in practice very few executions of the loop are needed.

Furthermore, this removes the need to have any explicit record of a task’s dynamic priority. Theorem 2 tells us the dynamic priority of the next task to run, and it is a simple matter to find that task, again by tracing through the “Blocked On” information. Given this, it turns out that we can also do without the “count” arrays of Section 5.

The relevant parts of the code are given in Appendix B. From our tests, this is the most efficient of the algorithms described in this section.

6.4. The Inherit3a algorithm

For completeness, we have tested an alternative implementation in which a task releasing a lock unblocks *all* tasks which it was blocking, after which those tasks re-compete for the locks which they wanted. The way to implement this is to make the main body of procedure `Obtain` a loop, so that a task continues trying to get the desired lock until it is successful.

The beauty of this approach is that it is no longer necessary to keep any form of blocked lists, since all tasks are presumed to be ready until proved otherwise. The data structures simplify considerably; for example, there is no need to maintain the “count” information of Section 5.

Even better, the disinheritance problem becomes a non–problem! This is because the looping in procedure `Obtain` amounts, in effect, to a continuous re-computation of all dynamic priorities.

With all these advantages, the implementation code looks very simple. Nevertheless, the performance of this algorithm is actually quite poor. In practice, it leads to excessive task switching: tasks wake up in procedure `Obtain`, discover that the locks they want are unavailable, and immediately re-block themselves. For this reason, we cannot recommend the Inherit3a algorithm as a serious contender for practical implementation.

6.5. The Inherit4 algorithm

We have also tested an implementation of the policy listed as Policy 4 in Section 4; as mentioned earlier, we suspect that Policy 4 is equivalent to the Priority Ceiling Protocol [1], although we have not yet been able to prove this. The resulting code is remarkably similar to that for the Inherit3 algorithm, the only difference of real substance being the test for whether a task should be blocked.

Because of this difference, however, it turns out to be possible to discard the concept of a Lock. Instead of specifying a lock, the caller of procedure `Obtain` simplify specifies what level of protection is being requested. For example, a call `Obtain(3)` specifies that all tasks of priority 3 or lower should be blocked on any call to `Obtain`. This results in a rather coarse discrimination, but it does tend to reduce the overall level of task switching, because it gives special preference to tasks which have already passed their calls to `Obtain`.

7. Simplifications in the absence of time-slicing

Although the algorithms in the preceding section work whether or not time-slicing is permitted, time-slicing can sometimes cause a degradation in system performance. The reason for this is that a task can obtain a lock, and then be replaced by another task (of the same priority) which can also obtain locks. This expands the set of tasks with the potential to block higher-priority tasks. It also means that the tasks which participate in the priority inheritance mechanism can be scattered through a “ready list” or similar structure, rather than concentrated at the head of the kernel lists.

We can reasonably conjecture that the abolition of time-slicing could reduce the number of tasks which affect the inheritance process; and, indeed, this turns out to be the case.

The following theorems provide a theoretical basis for the practical simplifications which can be made. (The theorems could also be used to check timing properties, since they imply limits on the extent to which tasks can be held up by other tasks. That aspect, however, lies beyond the scope of the present paper.) In Section 8 we shall show how the results can be turned into practical implementations.

Let us define the *Active Task Set* (ATS) to be the set of all tasks which have entered the system and commenced execution (and which have not yet left the system). Note that this set necessarily includes the running task, every task which holds a lock, and every task which is blocked on a lock. (A task cannot be blocked on a lock unless it has first run.) It does not, however, include all of the ready tasks. A task which has entered the system, been placed on the ready list, but not yet selected for execution is not in the Active Task Set.

Definition 5. A preemptive scheduling method is a Strict Priority Inheritance method if (a) it is a Priority Inheritance method, (b) tasks in the Active Task Set are always given preference over tasks of the same priority which are not yet in the Active Task Set, and (c) a task cannot be blocked on a lock unless that lock is already held by some other task.

Restriction (b) in this definition is essentially a condition on where in the ready list a task is placed after it becomes unblocked or changes its priority. (Note that this rules out the possibility of time-slicing among tasks of equal priority.) Restriction (c) rules out strategies like the Priority Ceiling Protocol [2] which can deny a lock to a task even though that lock is free.

The simplifications which this definition allows all derive ultimately from the following result.

Theorem 3. For all time, and for each priority level p, there exists at most one task in the Active Task Set whose base priority is p.

Proof. The result is vacuously true initially, when the Active Task Set is empty. Now suppose that it is true up to time t, and consider what happens if a new task enters the ATS at time t. (Observe that the proposition cannot be invalidated by a task's leaving the ATS, nor is it affected by any changes of state of tasks already in the ATS.) Suppose that there exist $T_1 \in ATS$ and $T_2 \notin ATS$ with $BP(T_1)=BP(T_2)$. If T_1 is ready or running, then by Definition 5 T_2 cannot run, and therefore cannot enter the ATS. If T_1 is blocked, let T_3 be the root node of the blocking tree in which T_1 resides; then $DP(T_3,t) \geq BP(T1)$ from Lemma 1. That is, T3 is a ready task in the ATS which takes precedence over T2, and again T2 cannot enter the ATS. ❐

Note that the membership of the ATS changes only when the running task leaves the ATS, or when a new task arrives whose priority is strictly greater than that of any task already in the ATS. Priority inheritance and disinheritance do not in themselves cause any change in the ATS, they simply cause changes of state of tasks already in the ATS. This follows from the comments following Theorem 2.

Theorem 4. For each priority level p,

(a) There exists at most one task T_1 which holds a lock and for which $BP(T_1)=p$;

(b) There exists at most one blocked task T_2 for which $BP(T_2)=p$.

Proof. Obvious from Theorem 3, since all blocked tasks and all lock-holders are in the Active Task Set. ❐

Theorem 5. For each time t, each lock L, and each priority level p, there exists at most one task with base priority p which is blocked on L.

Proof. Obvious from Theorem 4(b). ❐

The foregoing results are results about base priorities. In terms of finding an efficient implementation, it is useful to have comparable results which talk about dynamic priorities.

Lemma 3. If T_1 and T_2 are two tasks in the ATS for which $DP(T_1,t)=DP(T_2,t)$, then either $T_2 \in \mathfrak{B}^*(T_1,t)$ or $T_1 \in \mathfrak{B}^*(T_2,t)$.

Proof. From Lemma 2, there must exist $T_3 \in \mathfrak{B}^*(T_1,t)$ and $T_4 \in \mathfrak{B}^*(T_2,t)$ such that $BP(T_3)=BP(T_4)$. From Theorem 3, this is possible only if $T_3=T_4$. That is, the blocking trees for T_1 and T_2 have a node in common, and this is possible only if one is a subtree of the other. ❐

Theorem 6. For each time t and each priority level p, there is at most one ready or running task in the Active Task Set whose dynamic priority is p.

Proof. Suppose T_1 and T_2 are two ready (or running) tasks in the ATS, with $DP(T_1,t)=DP(T_2,t)$. From Lemma 3, and the fact that a ready task must be the root node of a blocking tree, we deduce that $T_1=T_2$. ❐

The significance of this result is that it tells us how to deal with tasks which become unblocked or which change their priorities: such tasks may always be placed at the head of the appropriate section of the ready list, without any risk that this will cause inappropriate queue-jumping. Furthermore the result implies that there is a *unique* ready task of highest priority in the ATS, which simplifies the decision about whether to perform a task switch.

Normally there will, of course, be other ready tasks which are not in the Active Task Set. Those tasks can simply be placed at the tail of the relevant part of the ready list as they arrive. They will not be considered for execution until there is no task in the ATS of equal or higher priority – at which time it is appropriate that a new task enter the ATS. In other words, nothing special need be done about separating ready tasks in the ATS from those which are not in the ATS.

As a matter of implementation detail, it may be convenient to have a separate array for the ready tasks in the ATS, so that the conventional ready list is used only for ready tasks which have not yet entered the ATS. When the currently running task becomes inactive, the ready list can be examined to see whether to move a new task into the ATS. Apart from this special case, there is never any need to look at the tasks in the ready list.

Theorem 7. The dynamic priority of the running task is strictly greater than that of any other task in the Active Task Set.

Proof. This follows directly from Theorem 1 and Theorem 6. ❐

This innocent-looking result has some important implications for what happens when the running task becomes blocked. It implies that the newly blocked task immediately goes to the head of the appropriate blocked list. In addition, it implies that *priority inheritance always propagates to the top of a blocking tree*. To see what this means, consider the case of a task T_1 which becomes blocked by another task T_2. It is possible that T_2 is already blocked by a task T_3, which is in turn blocked by T_4, and so on. The action of the priority inheritance mechanism is that the priority of T_1 should be passed along this chain until (a) a ready task is reached, or (b) a task is found whose priority is already greater than or equal to that of T_1. Theorem 7 shows that case (b) cannot happen.

Theorem 8. For each time t, each task T, and each priority level p, there exists at most one $T_p \in \mathfrak{B}(T,t)$ with $DP(T_p,t)=p$.

Proof. Suppose that there exist two tasks $T_{p1},T_{p2} \in \mathfrak{B}(T,t)$ with the same dynamic priority p. From Lemma 3, and the fact that distinct elements of $\mathfrak{B}(T,t)$ are in separate branches of the blocking tree, the only possibility is $T_{p1}=T_{p2}$. ❐

This result becomes useful if we need to record the set of tasks blocked by a given task. In the presence of time-slicing, a linked list or something similar would be needed. With time-slicing prohibited, it is sufficient to have an array.

Theorem 9. Suppose that we are using Policy 1, 2, or 3 of Section 4. Then for each time t, each lock L, and each priority level p, there exists at most one task with dynamic priority p which is blocked on L.

Proof. From Theorem 7, a task which becomes blocked has a priority different from that of any task which is already blocked. With Policy 3, a task never changes its priority after it has become blocked, so there is nothing more to prove.

With Policy 1, a blocked task can change its priority by inheritance, but – again from Theorem 7 – the new priority must be greater than that of any task which is already blocked.

With Policy 2 a more careful analysis is needed. It would appear at first sight that, if blocked task T inherits a new priority, then the predecessor of T on the blocked list will also inherit that priority. In fact the sequence of events is (a) T inherits a new priority, higher than that of any other task blocked on L; (b) as a result of this inheritance, T moves to the head of the blocked list, and now has no predecessor in the list. Inheritance then flows to the holder of L rather than to some other task blocked on L. ❒

These last two results say that the representation of blocked lists can be simplified: instead of a complicated list structure, we can just use arrays of blocked tasks. For Policy 3 we can go a little further: for each p, there is at most blocked task with dynamic priority p, and this holds globally rather than on a per-lock basis.

Theorem 10. There does not exist $T_1 \in \mathfrak{B}^+(T,t)$ with $DP(T_1,t)=BP(T)$.

Proof. If such a T_1 did exist, then from Lemma 2 we could find a $T_2 \in \mathfrak{B}^*(T_1,t)$ such that $BP(T_2)=BP(T)$. This is impossible from Theorem 3. ❒

The utility of this result is that we sometimes want to use the “task blocking itself” fiction to ensure that a task reverts to its base priority when it is no longer blocking anything (in the same way that $\delta(BP(T),j)$ was used in the definition of count2(T,j). This eliminates a special case test during the disinheritance operation.

Another way to look at dynamic priorities is to assign dynamic priorities to locks. If we define an *active lock* as a lock on which one or more tasks are blocked, and if we define the (dynamic) priority of a lock to be the maximum base priority of the tasks blocked on it, then it can be shown that there is at most one active lock per priority level. We have found, however, that in actual implementation the overheads of maintaining the “active lock” information are not justified from the benefits gained from the information, so there seems to be little point in further pursuing that issue.

8. Implementations of the no-time-slicing methods

The implementations discussed in this section are adaptations of the corresponding algorithms in Section 6. Basically, we have taken the original algorithms and improved them by use of the theorems in Section 7.

8.1. The Inherit1N algorithm

This algorithm manages to make extensive use of a “bitset” representation of blocked tasks. The ATS is represented by an array, with ATS[j] holding the (unique) task of priority j. In view of the results in Section 7, it is almost an arbitrary decision to base this representation on base priorities or dynamic priorities; we found it slightly more efficient to use base priorities. Given this representation, a set of blocked tasks can be represented by a packed array of bits, one for each member of the ATS.

In our tests, we used 16 distinct priority levels, so that a set could be represented in a 16-bit word. In this case the highest-priority task in a set can be found in just four steps, so this is a fast operation.

It turns out to be convenient to keep track of both the tasks blocked by each task, and the tasks blocked on each lock. The implementation is then similar to that shown in Appendix A, but with all calculations replaced by fast “bitset” operations.

The final algorithm is relatively simple and quite fast. It is not the most efficient implementation that we found – the Inherit3N algorithm turned out to be faster – but is very close to being the most efficient.

8.2. The Inherit2N algorithm

As in the case where time-slicing is permitted, an implementation based on Policy 2 in Section 4 turns out not to be a good contender for implementation. Even with the use of a “bitset” representation of tasks, the computation of task dependencies is excessively complex, mostly because of the rearrangement of blocked sets which occurs when a task changes its dynamic priority. We have therefore abandoned this approach.

8.3. The Inherit3N algorithms

Starting from Policy 3 in Section 4, and using the simplifications which arise from the “Active Task Set” concept, one can derive at least two distinct algorithms which can be used as good implementations. For consistency with Section 6 we shall consider both of these under the same heading; but in fact they lead to substantially different implementation code.

The first approach is to maintain explicit representations of both the “Blocked On” and

"Blocked By" relations, using the bitset method of representing a set. The mechanism for obtaining a lock is straightforward and obvious, given that it is known that the blocking trees are limited to having height 0 or 1.

Releasing a lock is a more complicated operation, because of the redistribution operation illustrated in Figure 3. In the present case, this redistribution can be done as follows. Let T_1 be the task releasing the lock, and let T_2 – whose identity is initially not known – be the task that will obtain the lock. Define Set1 to be the set of all tasks that will remain blocked by T_1, and T_1 itself; define Set2 to be the set of all tasks that will be blocked by T_2, and T_2 itself; and define Set3 to be the set of all other tasks, originally blocked by T_1, whose new blocker has not yet been determined. Initially Set1 contains only T_1, Set2 is empty, and Set3 contains everything else. Now we can enter a loop where in each iteration we perform the following operations.

(a) Let Set4 be the singleton set containing the task of highest base priority from Set3;

(b) If that task is blocked, follow the "blocked on" chain, transferring all tasks so found from Set3 to Set4, until a task T_3 is reached which is not in Set3;

(c) Depending on whether T_3 is in Set1 or in Set2, transfer all of Set4 to Set1 or Set2.

It should be clear that Set3 will be emptied after only a small number of iterations of this loop, and that the identity of T_2 will be found as a side-effect of the calculation. Although this calculation looks moderately complex, the "bitset" representation of sets of tasks makes it execute very quickly.

The second approach, which seems in practice to be faster, is to search the chain of blocked tasks, in the same way as was done in Section 6.3, at the time that a task switch is about to be done. That is, we keep no information about the "blocked by" relationships. Not only is this a fast algorithm; as a surprise bonus, it turns out to be identical in implementation whether or not time-slicing is permitted.

8.4. Other approaches

For completeness, we have implemented and tested no-timeslicing versions of the algorithm of Section 6.4 – in which all tasks are unblocked and made to re-compete when a lock becomes available – and that of Section 6.5, which is either equivalent to or similar to the Priority Ceiling Protocol.

As expected, the approach of unblocking everything turns to have excessive overheads in terms of redundant task switching, and therefore this approach is not worth pursuing further.

Our variant of the Priority Ceiling Protocol falls a little outside the scope of this paper, because it does not satisfy Definition 5, so we have not investigated it in great detail. The main observation was that the best implementation we could find was the same regardless of whether time-slicing was permitted.

9. Practical considerations and conclusions

The algorithms labelled Inherit3 and Inherit3N have been implemented in the PMOS [3,4] multitasking kernel, and they have proved to work well with very little overhead. Time-slicing in PMOS is controlled by a compile-time constant, and there was no difficulty in using conditional compilation to support both variants of the algorithm.

One unexpected difficulty was that it was not as easy as we thought to ensure that Assumption 1 was satisfied. A difficult arises in code like

```
Obtain(L);
procedure call;
Release(L);
```

where the called procedure becomes inactive by, for example, performing a `Wait` on a general semaphore. The problem here is that the caller does not always know the internal details of the procedure's execution; and this can lead to situations where a task which is supposed to be waiting is allowed to execute. The reason this can happen is that removing a task from the Active Task Set does not automatically remove it from the "blocked by" chain, so that procedure `RunNextTask` in Appendix B could cause a task switch to a task which is not supposed to be active. Any re-design of the software to check for this condition is likely to add significantly to the kernel overhead.

We could of course accept the extra overhead, and design the algorithms to work even in the absence of Assumption 1; but it is doubtful whether this is worthwhile. It makes more sense to assert that code which violates Assumption 1 should be redesigned.

A secondary problem is that a task may exit or crash while holding a lock. This tends to be an issue during system shutdown operations.

Our compromise solution to these problems, as implemented in PMOS, is to keep track of all locks held by a task. This means, among other things, that violations of Assumption 1 can be detected at run-time, and this helps to pinpoint sections of code which were badly designed. Our experience has been that this type of violation tends to turn up almost immediately during program testing. That is, it is unlikely that a programming error involving the violation of Assumption 1 would remain undetected past the initial testing stage.

Apart from this issue, the implementation of priority inheritance turns out to be easy, and its run-time overheads – as compared to a preemptive task switching strategy with fixed priorities – are so small as to be completely negligible.

10.References

[1] Ragunathan Rajkumar, *Synchronization in real-time systems: a priority inheritance approach*, Kluwer 1991.

[2] Lui Sha, Ragunathan Rajkumar, John P. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronisation”, *IEEE Trans Computers* **39** (9), Sept 1990, 1175-1185.

[3] P.J. Moylan, “The PMOS Definition Modules”, *Tech Rept* EE9107, University of Newcastle, February 1991.

[4] P.J. Moylan, “The PMOS Real-Time Kernel”, *18th IFAC/IFIP workshop on Real-Time Programming*, Bruges, June 1992, session B3.

Appendix A – Code for the general case

```
PROCEDURE Promote (T: Task;  p: PriorityLevel);

   (* Called when T is (directly or indirectly) blocking a task  *)
   (* of priority p; thus the dynamic priority of T must be      *)
   (* increased if necessary to match this.  If T is itself      *)
   (* blocked, the priority increase ripples up the blocking     *)
   (* tree.                                                      *)

   VAR oldp: PriorityLevel;

   BEGIN
      INC (T.count[p]);  oldp := T.DP;
      IF oldp < p THEN
         T.DP := p;
         IF T.Blocked THEN
            update the position of T in T.BlockedOn.BlockedList;
            DEC (T.BlockedBy.count[oldp]);
            Promote (T.BlockedBy, p);
         ELSE
            update the position of T in the ready list;
         ENDIF;
      ENDIF;
   END Promote;
```

Fig 4 – The priority inheritance mechanism

```
PROCEDURE Obtain (L: LOCK);

   (* Obtains lock L, waiting if necessary.  *)

   VAR Blocker: Task;

   BEGIN
      IF the current task should be blocked THEN
         let Blocker be the task which caused the blocking;
         CurrentTask.Blocked := TRUE;
         CurrentTask.BlockedBy := Blocker;
         CurrentTask.BlockedOn := L;
         Add CurrentTask to L.BlockedList;
         Promote (Blocker, CurrentTask.DP);
         SelectAnotherTask;
      ELSE
         L.Locked := TRUE;
         L.Holder := CurrentTask;
      ENDIF;
   END Obtain;
```

Fig 5 – Implementation of the Lock operation when time-slicing is allowed

```
PROCEDURE Release (L: LOCK);

   (* Releases lock L. This can lead to a change in dynamic  *)
   (* priorities, and/or a task switch.                      *)

   VAR j, k: PriorityLevel;  T, U: Task;

   BEGIN
      IF L.BlockedList is empty THEN
         L.Locked := FALSE;
      ELSE
         choose a task T to unblock;
         remove T from L.BlockedList;
         L.Holder := T;  T.Blocked := FALSE;

         (* Some of the tasks which were blocked by   *)
         (* CurrentTask are now blocked by T.         *)

         j := T.DP;  DEC (CurrentTask.count[j]);
         FOR each task U in L.BlockedList DO
            k := U.DP;  U.BlockedBy := T;
            DEC (CurrentTask.count[k]);  INC (T.count[k]);
         ENDFOR;

         (* Remark: if we consistently choose the T of highest  *)
         (* DP, the above operations will have no effect on     *)
         (* T.DP.  With some other strategies – for example,    *)
         (*  unblocking tasks in FIFO order – it is possible    *)
         (* that T must be promoted at this stage.              *)

         IF T was not the highest–priority task in L.BlockedList
            THEN check the T.count array,
                  and increase T.DP if appropriate;
         ENDIF;

         (* Does CurrentTask need to be demoted?   *)

         IF (CurrentTask.DP=j) AND (CurrentTask.count[j]=0) THEN
            decrement j until CurrentTask.count[j] > 0;
            CurrentTask.DP := j;
            insert CurrentTask into the ready list, and perform
               a task switch to T;
         ELSE
            insert T into the ready list;
         ENDIF;
      ENDIF;
   END Release;
```

Fig 6 – Releasing a lock

Appendix B – The Inherit3 Algorithm

```
PROCEDURE RunNextTask;

   (* Performs a task switch to the first task on the active    *)
   (* list whose dynamic priority is CurrentPriority, if that   *)
   (* task is not blocked.  If it is blocked, we switch to its  *)
   (* blocker instead.                                          *)

   VAR T: Task;  L: Lock;

   BEGIN
      T := ActiveList[CurrentPriority].head;
      LOOP
         L := T^.WaitingFor;
         IF (L = NIL) OR NOT L^.Locked THEN
            EXIT (*LOOP*);
         END(*IF*);
         T := L^.Holder;
      END (*LOOP*);
      TaskSwitch (T);
   END RunNextTask;

(*****************************************************************)

PROCEDURE Obtain (L: Lock);

   (* Obtains lock L, waiting if necessary.  *)

   VAR Blocker: Task;  M: Lock;

   BEGIN
      IF L^.Locked THEN

         (* Work out the blocker of the current task. *)

         M := L;
         REPEAT
            Blocker := M^.Holder;  M := Blocker^.WaitingFor;
         UNTIL (M = NIL) OR NOT M^.Locked;

         CurrentTask^.WaitingFor := L;
         TaskSwitch (Blocker);
      END (*IF*);

      (* When execution resumes at this point (via *)
      (* RunNextTask), L^.Locked will be FALSE.    *)

      CurrentTask^.WaitingFor := NIL;
      L^.Locked := TRUE;
      L^.Holder := CurrentTask;

   END Obtain;
```

```
(****************************************************************)

PROCEDURE Release (L: Lock);

   (* Releases lock L.  This implicitly changes the ”blocked     *)
   (* by”  status of all tasks blocked by the current task, but  *)
   (* it turns out to be faster to let procedure RunNextTask to *)
   (* recheck the blocking status than to keep track of      *)
   (* ”blocked by” information.  All we need to know here is     *)
   (* whether we’ve inherited a priority greater than our base  *)
   (* priority.  If so, we must have been blocking another task *)
   (* which should run now – we can trust RunNextTask to work   *)
   (* out the identity of that task.                            *)

   BEGIN
      L^.Locked := FALSE;  L^.Holder := NIL;
      IF CurrentTask^.BP < CurrentPriority THEN
         RunNextTask;
      END (*IF*);
   END Release;
```