Priority Inheritance Spin Locks for Multiprocessor Real-Time Systems

Cai-Dong Wang, Hiroaki Takada, and Ken Sakamura
Department of Information Science,
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Abstract

When a real-time system is realized on a shared-memory multiprocessor, priority-ordered spin locks are often required to reduce priority inversions. However, simple priority-ordered spin locks can cause uncontrolled priority inversions when they are used for nested spin locks. This paper points out the problem of uncontrolled priority inversions in the context of spin locks and proposes priority inheritance spin locks, spin lock algorithms that are enhanced with the priority inheritance scheme, to solve the problem. Two algorithms of priority inheritance spin locks are presented and their effectiveness is demonstrated through performance measurements.

1 Introduction

Spin lock is one of the fundamental synchronization primitives for exclusive access to shared resources on shared-memory multiprocessors. It is usually implemented with atomic read-modify-write operations on a single word (or aligned contiguous words) of shared memory such as test_and_set, fetch_and_store (swap), or compare_and_swap. Various spin lock algorithms using these operations have been proposed [1, 2] and are widely used.

In real-time systems, each job has some timing constraints to be met. A scheduling algorithm or a run-time scheduler translates the timing constraints into a priority, and a run-time system assigns system resources to higher priority jobs. It is often the case with a multiprocessor real-time system that a spin lock is also required to pass the lock in a priority order. In order to satisfy this requirement, some priority-ordered spin lock algorithms, in which processors acquire a lock in the order of their priorities, have been proposed [3, 4, 5].

In general, shared resources that must be accessed exclusively by a processor are divided into some lock units in order to improve concurrency. When a processor accesses some shared resources included in different lock units, it must acquire multiple locks one by one. If priority-ordered spin locks are simply used for this kind of *nested spin locks*, uncontrolled priority inversions can occur. The uncontrolled priority inversion problem in nested spin locks is described in Section 2.

In order to solve this problem, this paper proposes *priority inheritance spin locks*, spin lock algorithms that are enhanced with the priority inheritance scheme. The priority inheritance scheme is a promising approach to solve uncontrolled priority inversion problems and its applications to task scheduling algorithms are actively studied [6, 7]. Priority inheritance spin locks are also required to realize bounded and scalable nested spin locks for real-time systems [8].

After describing the necessity of priority inheritance spin locks in Section 2, we present two algorithms of priority inheritance spin locks in Section 3. In Section 4, their effectiveness is evaluated through performance measurements.

```
acquire_lock(L2);
// critical section.
release_lock(L2);
```

routine (a)

```
acquire_lock(L1);
acquire_lock(L2);
// critical section.
release_lock(L2);
release_lock(L1);
```

routine (b)

Fig. 1: Example of nested spin locks

2 Priority Inversion and Priority Inheritance

2.1 The Priority Inversion Problem in Nested Spin Locks

Priority inversion in the context of spin locks is the phenomenon that a higher priority processor is forced to wait for the execution of a lower priority processor. Because priority inversion cannot be avoided unless a higher priority processor can steal the lock held by a lower priority one, how to minimize its duration is a concern. When the maximum duration of a priority inversion cannot be determined, it is called uncontrolled.

When priority-ordered spin locks are used for nested spin locks, uncontrolled priority inversions can occur. A typical case is described in the following example.

Example 1 (uncontrolled priority inversion) We assume that P_1, P_2, P_3, and P_4 are processors arranged in descending order of priority with P_1 having the highest priority, and that L_1 and L_2 are locks. These processors repeatedly execute one of the two routines presented in Figure 1. Suppose the case that when P_1 begins executing

routine (b) and tries to acquire the lock L_1, P_4 is holding L_1 and is waiting for the other lock L_2 in routine (b). If P_2 and P_3 repeatedly execute routine (a) in this situation, P_2 and P_3 can acquire L_2 alternately and P_4 must wait for L_2 all the while. Because P_1 must also wait for the executions of P_2 and P_3, this duration is a priority inversion. Obviously, the maximum duration of this priority inversion cannot be determined.

2.2 Spin Lock with Priority Inheritance

In order to solve this problem of uncontrolled priority inversions, we introduce the priority inheritance scheme to spin locks. The fundamental concept of priority inheritance scheme is that when a processor makes some higher priority processors wait, its priority should be raised to the level of the highest priority processor among the waiting ones. In other words, the processor inherits the priority of the highest priority processor blocked by it. Also, priority inheritance must be transitive. For example, suppose that P_1, P_2, and P_3 are three processors in descending order of priority. When P_2 makes P_1 wait and P_3 makes P_2 wait, P_3 should inherit the priority of P_1.

With the basic priority inheritance scheme, which is the naive realization of the concept, the uncontrolled priority inversion problem illustrated in Example 1 is solved as follows. When P_1 tries to acquire L_1 and begins waiting for it, P_4, which is holding L_1, inherits the priority of P_1 because P_1 is forced to wait by P_4. Because the inherited priority is higher than the priorities of P_2 and P_3, P_4 can acquire L_2 with precedence over P_2 and P_3. As the result, P_1 need not wait for the alternate executions of routine (a) by P_2 and P_3, and the maximum duration of the priority inversion can be bounded.

When a processor releases one of the locks, its priority is necessary to be re-calculated in general. Specifically, its priority is changed to the highest one of its original priority and the priorities of the processors that is waiting for the locks held by the former one. When the processor releases the last lock it is holding, its priority is recovered to its original level.

This re-calculation can be omitted under the following two assumptions. The first assumption is that the inherited priority is used only for spin locks, and not used for task schedulings. In more specific, the inherited priority is used only when the processor tries to acquire another lock. The second assumption is that the two-phase protocol is adopted. In other words, once a processor releases a lock, it cannot acquire another lock until it releases all the locks it is holding. Under these two assumptions, once the priority of a processor is raised, it need not be lowered until it releases all the locks. These assumptions can be removed by adding re-calculation routines to the algorithms proposed in Section 3 at the cost of some runtime overhead.

The required behavior of priority inheritance spin locks can be summarized as follows.

1. Processors acquire a lock in the order of their priorities[1].

[1] A strict definition of priority-ordered spin lock is appeared in [3].

```
// global shared variables.
shared var L1, L2: Lock;

// local variables (allocated for each processor).
var I1, I2: Node;
var my_prio: integer;
var my_notify: boolean;
// my_notify is necessary only in the second algorithm.

// initialize my_prio.
acquire_first_lock(&L1, &I1);
acquire_second_lock(&L2, &I2, &L1);
// critical section.
release_lock(&L2, &I2);
release_lock(&L1, &I1);
```

Fig. 2: Usage of the algorithms

2. When a processor P_1 begins waiting for a lock, and when its priority is higher than that of the processor P_2 that is holding the lock, the priority of P_2 is raised to that of P_1.
3. When the priority of a processor P_1 is raised while waiting for a lock, and when its new priority is higher than that of the processor P_2 that is holding the lock, the priority of P_2 is raised to the new priority of P_1.

3 Priority Inheritance Spin Lock Algorithms

In this section, we present two algorithms of priority inheritance spin locks, which are based on the priority-ordered queueing spin lock algorithm proposed by Markatos [3]. With the Markatos' algorithm, processors trying to acquire a lock are linked to the waiting queue in a FIFO order[2]. In releasing the lock, a processor searches the highest priority processor in the waiting queue and passes the lock to it.

The first algorithm is a straightforward extension of the Markatos' lock. A new variable that indicates the highest priority of the processors that is waiting for the lock is prepared for each lock. The processor holding the lock *polls* the variable while it is waiting for another lock. When the processor detects that the highest priority is raised, it inherits the priority. Because any processor can poll this highest priority variable for each lock, pollings on the variable are remote memory accesses and severely increase the interconnection network traffic with a multiprocessor system without coherent cache. The second algorithm is to avoid this non-local spin and is expected to have higher performance without coherent cache.

In order to avoid unnecessary complexity, this paper presents the pseudo codes of the algorithms when a processor acquires at most two locks at the same time, in other words, when the nesting level of locks is less than or equal to two. With this simplification, we prepares two lock acquisition routines: *acquire_first_lock* for acquiring the outer lock and *acquire_second_lock* for acquiring the

[2] Though the original Markatos' algorithm adopts a double-linked queue structure, a single-linked queue structure is sufficient to implement the algorithm. Therefore, we enhance the simplified version adopting single-linked queue with the priority inheritance scheme.

```
type Node = record
    next: pointer to Node;
    locked: (Released, Locked);
    prio: integer
end;

type Lock = record
    last: pointer to Node;
    maxprio: integer;
    notifyp: pointer to boolean
end;
// notifyp is necessary only in the second algorithm.
// last and notifyp fields should be initialized to NULL.
// maxprio field should be initialized to MIN_PRIO.

type NodePtr = pointer to Node;
type LockPtr = pointer to Lock;
```

Fig. 3: Data structures

inner lock. A typical usage of the routines is illustrated in Figure 2. The third argument of *acquire_second_lock* is the pointer to the lock that the processor is holding.

In Figure 2, the keyword *shared* indicates that only one instance of the variable is allocated in the system and shared by all processors. The other local variables are allocated for each processor. With a multiprocessor without coherent cache, the local variables should be placed on the processor's locally accessible shared memory. The *my_prio* variable is to store the current priority of the processor, and must be initialized before the processor tries to acquire the outermost lock.

3.1 The First Algorithm

Figure 3 presents the common data structures for both algorithms (one of the fields is necessary only in the second algorithm). The *Lock* record should be prepared for each lock in the system. Its *maxprio* field is the highest priority variable for the lock. When the lock is empty (in other words, no processor holds the lock), the *last* field of its *Lock* record is *NULL* and its *maxprio* field is *MIN_PRIO*, which designates the minimum priority value. A *Node* record is necessary for each nested lock for each processor.

Figure 8 and Figure 9 present the pseudo codes of the first algorithm. In these figures, we assume that a larger value represents a higher priority. The *fetch_and_store* operation reads the memory addressed by the first parameter, returns the contents of the memory as its value, and atomically writes the second parameter to the memory. The *CAS* operation, the abbreviation of compare_and_swap, first reads the memory pointed to by the first parameter and compares its contents with the second parameter. If they are equal, the function writes the third parameter to the memory atomically and returns true. Otherwise, it returns false.

Compared to the Markatos' algorithm, two invocations of the *raise_priority* procedure, which is to update the *maxprio* field of *lock* when it is lower than the *newprio* parameter, are added to the *acquire_first_lock* procedure and the *acquire_second_lock* procedure in Figure 8. The first invocation (marked with "∗1") is to raise *maxprio* for priority inheritance, when the processor begins waiting for the lock. The second one (marked with "∗2") is to set *maxprio*, when the processor succeeds to acquire the lock without waiting. In the *acquire_second_lock* procedure, the processor must check the *maxprio* field of *lock1*, which is the lock being held by the processor, while waiting for *lock*. When *maxprio* becomes higher than the priority of the processor (the *if* statement marked with "∗3"), it inherits *maxprio* of *lock1* and updates *maxprio* of *lock* for transitive priority inheritance.

The only difference of the *release_lock* procedure in Figure 9 with that of the Markatos' algorithm is the necessity of updating the *maxprio* field (two lines marked with "∗4" and "∗5"). Assigning *MIN_PRIO* to the *maxprio* field at first is necessary to avoid some racing conditions.

This algorithm can be easily generalized to the case that a processor acquires more than two locks at the same time with the following method. The list of locks held by a processor should be maintained using an array or a linked list. In the generalized version of the *acquire_lock* procedure, the *maxprio* fields of all the locks in the list should be checked while waiting for another lock. If some of them are higher than the priority of the processor, it inherits the highest priority among them.

3.2 Avoiding Non-Local Spins

While a processor is waiting for a lock in the *acquire_second_lock* procedure of the first algorithm, the *maxprio* field of the holding lock is accessed repeatedly (marked with "∗3"). This accesses cause a heavy traffic on the interconnection network without coherent cache.

With the second algorithm presented in Figure 10 and Figure 11, this problem is solved by introducing a flag to notify that the *maxprio* field is modified. This notification flag (the *my_notify* variable in Figure 2) is prepared for each processor on its locally accessible shared memory. A processor waiting for a lock in the *acquire_second_lock* procedure in Figure 10 reads the *maxprio* field only when the notification flag of the processor is set (the *if* statement marked with "∗6"). Thus the non-local spin can be avoided. It also checks the *maxprio* field when it begins waiting for a lock (by assigning *TRUE* to *my_notify*). Introducing the notification flag is also advantageous when a processor acquires more than two locks at the same time, because only one memory location (i.e. the notification flag) is necessary to be checked in the waiting loop. Maintaining the list of locks held by a processor is still necessary in this case.

Also, the *raise_priority_notify* procedure is used instead of *raise_priority* (three lines marked with "∗7") in Figure 10. After updating the *maxprio* field of *lock*, the *raise_priority_notify* procedure sets the notification flag of the processor holding the lock. In order to locate the notification flag of the lock holder, a new field *notifyp* which points to the notification flag is introduced in the *Lock* record. The *notifyp* field of a lock is set when a processor succeeds to acquire the lock (two lines marked with "∗8"). The field is also necessary to be cleared to NULL at the top of the *release_lock* procedure in Figure 11 (marked with

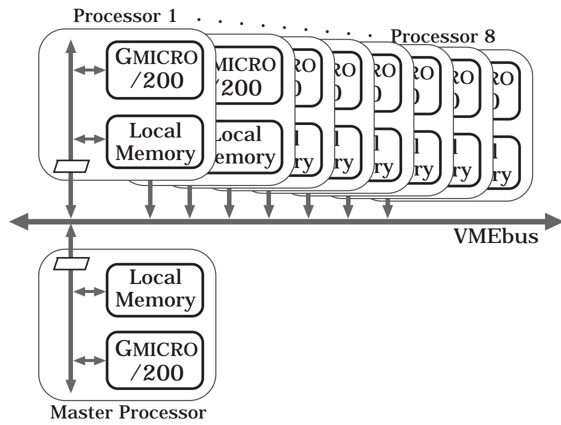

Fig. 4: Evaluation environment

acquire_lock(L_2);
// critical section.
release_lock(L_2);

routine (a)

acquire_lock(L_1);
acquire_lock(L_2);
// critical section.
release_lock(L_2);
release_lock(L_1);

routine (b)

acquire_lock(L'_1);
acquire_lock(L_2);
// critical section.
release_lock(L_2);
release_lock(L'_1);

routine (c)

acquire_lock(L''_1);
acquire_lock(L_2);
// critical section.
release_lock(L_2);
release_lock(L''_1);

routine (d)

Fig. 5: Evaluation routines

"∗9").

There is a slight chance that the notification flag of a wrong processor is set. Specifically, suppose the case that the processor holding a lock passes the lock to another one and its *notifyp* field is changed, after yet another processor reads the *notifyp* field of the lock in the *raise_priority_notify* procedure and before it writes TRUE on ∗*notifyp*. In this case, the notification flag of the processor that has already passed the lock to another is set. Although this difficulty can increase the interconnection network traffic a little, it does not cause wrong behavior.

4 Performance Evaluation

In this section, the effectiveness of the priority inheritance spin lock algorithms proposed in the previous section is examined through performance evaluation. Their performance is compared with the simple priority-ordered spin locks without supporting priority inheritance scheme. We have used the single-linked queue version of the Markatos' spin lock algorithm for this purpose.

4.1 Evaluation Environment

A shared-bus multiprocessor system without coherent cache is used for the evaluation. The shared bus is based on the VMEbus specification, and each processor node consists of a GMICRO/200 microprocessor, which is rated approximately at 10 MIPS, and 1 MB of local memory. The local memory can be accessed from other processors through the shared bus. No coherent cache is equipped. All the program code and the data area for each processor are placed on the local memory of the processor. G lobal shared variables are placed on the local memory of the master processor, which does not execute spin locks (Figure 4).

Since the GMICRO/200 microprocessor supports compare_and_swap instruction but not fetch_and_store, the latter operation is emulated with a compare_and_swap instruction and a retry loop. As the VMEbus has only four pairs of bus request/grant lines, processors are classified into four groups by the bus request line they use. The round-robin arbitration scheme is adopted among groups and the static priority scheme is applied among processors belonging to a same group.

4.2 Evaluation Method

We have used one to eight processors for the evaluation. The original (or assigned) priority of a processor is fixed to its ID numbers. Each processor repeatedly executes one of the four routines presented in Figure 5 in random order. Routines (c) and (d) are introduced in order to expose the problem of non-local spins with the first algorithm[3]. The execution time of each routine is measured for each execution, and their distributions are obtained. Inside the critical section, a processor accesses the shared bus several number of times and waits for a while using empty loops. In case of routines (b), (c), and (d), shared bus accesses and an empty loop are also inserted between *acquire_first_lock* and *acquire_second_lock*. Without spin locks, the execution time of each routine is about 30 μs, including the overhead for measuring execution times. Each processor also waits for a random time after each execution of the routines.

Because our evaluation system has no coherent cache, the simple implementation of the first algorithm causes heavy shared-bus traffic. In order to avoid shared-bus saturation, the frequency to read the *maxprio* field in the *acquire_second_lock* routine is reduced. In more concrete, *maxprio* is checked only once for every four checkings of *me→locked*.

In real-time systems, because the worst-case behavior of the system has primary importance, the effectiveness of the algorithms should be evaluated with their maximum execution times. Because maximum execution times cannot be obtained through experiments due to unavoidable non-determinism in multiprocessor systems, however, a p-reliable time, the time within which a processor finishes execution with probability p, is adopted as the performance metric instead of a maximum execution time. In this section, we show the evaluation results when p is 0.9999 (i.e. 99.99%).

4.3 Evaluation Results

Figure 6 presents the 99.99%-reliable execution times that the *highest* priority processor executes routine (b).

[3]With routines (a) and (b) only, the effect of shared-bus traffic is not revealed, because at most one processor spins on non-local memory at the same time.

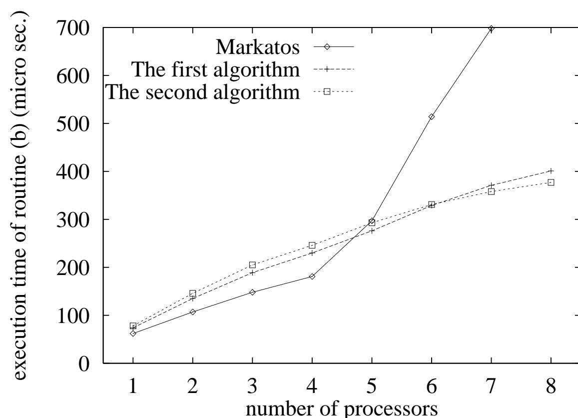

Fig. 6: 99.99%-reliable execution times of routine (b)

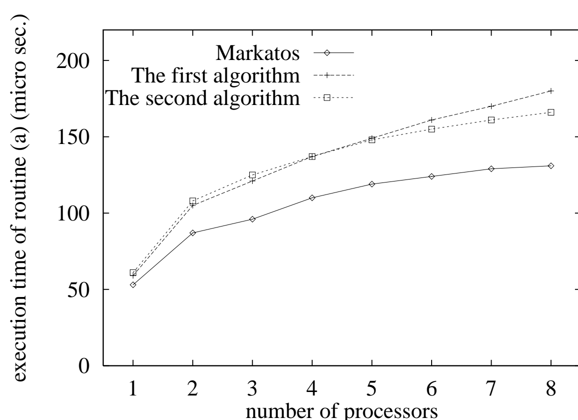

Fig. 7: 99.99%-reliable execution times of routine (a)

When the number of processors is large, the execution time with Markatos' locks is much slower than those with our algorithms due to uncontrolled priority inversions. When the number of processors is small, our algorithms are slower because of the overhead for maintaining the *maxprio* field of each lock. Our second algorithm is a bit faster than the first one when the number of processors is more than six, but the difference is very small. Though it is not measured in our experiments, the shared-bus traffic is expected to be much larger with the first algorithm.

Figure 7 presents the 99.99%-reliable execution times that the highest priority processor executes routine (a). As easily imagined, there are little difference in the behavior of routine (a) with three algorithms. This graph confirms the conjecture.

Finally, in order to examine the average performance of the algorithms, we present the average execution times of routine (b) in Figure 12. From this graph, our algorithms are slower than Markatos' lock in average performance. We can say that priority inheritance spin locks are not appropriate in case that improving average performance is the primary concern.

5 Conclusion and Future Work

When a real-time system is realized on a shared-memory multiprocessor, priority-ordered spin locks are often required to reduce priority inversions. But, simple application of a priority-ordered spin lock algorithm to nested spin locks causes uncontrolled priority inversions, which are very harmful for satisfying the timing constraints imposed on real-time jobs.

In order to solve the problem, we have proposed priority inheritance spin locks. Two algorithms of priority inheritance spin locks are presented: one for coherent cache multiprocessors and the other for multiprocessor systems without coherent cache. Performance evaluations to demonstrate their effectiveness have been conducted, and some affirmative results have been obtained.

In this paper, we have proposed the algorithms based on the Markatos' spin lock algorithm. We also plan to extend other priority-based spin lock algorithms, especially the PR-lock [5] and the bubble lock [9], to support the priority inheritance scheme.

Our motivation of this study is the application to a scalable real-time operating system kernel for function-distributed multiprocessors [10]. It also remains as a future work to implement a real-time kernel with the algorithms and to evaluate them in more realistic conditions.

References

[1] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, pp. 6–16, Jan. 1990.

[2] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Computer Systems*, vol. 9, pp. 21–65, Feb. 1991.

[3] E. P. Markatos, "Multiprocessor synchronization primitives with priorities," in *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[4] T. S. Craig, "Queuing spin lock algorithms to support timing predictability," in *Proc. Real-Time Systems Symposium*, pp. 148–157, Dec. 1993.

[5] T. Johnson and K. Harathi, "A prioritized multiprocessor spin lock," Tech. Rep. TR-93-005, Department of Computer Science, University of Florida, 1993.

[6] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Computers*, vol. 39, pp. 1175–1185, Sept. 1990.

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

[8] H. Takada and K. Sakamura, "Real-time scalability of nested spin locks," in *Proc. 2nd Real-Time Computing Systems and Applications*, pp. 160–167, Oct. 1995.

[9] N. Sakiyama, H. Takada, and K. Sakamura, "Bubble lock: Another priority-orderd spin lock algorithm," in *Collection of Position Papers for the 2nd Youth Forum in Computer Science and Engineering (YUFORIC)*, Oct. 1995.

[10] H. Takada and K. Sakamura, "Towards a scalable real-time kernel for function-distributed multiprocessors," in *Proc. of 20th IFAC/IFIP Workshop on Real Time Programming*, Nov. 1995.

```
   procedure raise_priority(lock: LockPtr,
                                        newprio: integer): boolean;
       var prio: integer;
   begin
     retry:
       prio := lock→maxprio;
       if newprio > prio then
           if CAS(&(lock→maxprio), prio, newprio) then
               return TRUE
           end;
           goto retry
       end;
       return FALSE
   end;

   procedure acquire_first_lock(lock: LockPtr, me: NodePtr);
       var pred: NodePtr;
   begin
       me→next := NULL;
       // enqueue myself.
       pred := fetch_and_store(&(lock→last), me);
       if pred ≠ NULL then
           // when the queue is not empty
           me→locked := Locked;
           me→prio := my_prio;
           pred→next := me;
*1         raise_priority(lock, my_prio);
           repeat until me→locked = Released
       else
           // succeed to acquire the lock without waiting.
*2         raise_priority(lock, my_prio)
       end
   end;

   procedure acquire_second_lock(lock: LockPtr,
                                        me: NodePtr, lock1: LockPtr);
       var pred: NodePtr;
   begin
       me→next := NULL;
       // enqueue myself.
       pred := fetch_and_store(&(lock→last), me);
       if pred ≠ NULL then
           // when the queue is not empty
           me→locked := Locked;
           me→prio := my_prio;
           pred→next := me;
*1         raise_priority(lock, my_prio);
           repeat
*3             if lock1→maxprio > my_prio then
                       // lock1→maxprio is non-local access.
                   my_prio := lock1→maxprio;
                   me→prio := my_prio;
                   raise_priority(lock, my_prio)
               end
           until me→locked = Released
       else
           // succeed to acquire the lock without waiting.
*2         raise_priority(lock, my_prio)
       end
   end;
```

Fig. 8: The first algorithm

```
   // move entry to the top of the waiting queue of lock.
   // pred is the predecessor of entry.
   // oldtop is the top of the queue before the move.
   procedure move_to_top(lock: LockPtr, entry, pred,
                                       oldtop: NodePtr);
       var succ: NodePtr;
   begin
       succ := entry→next;
       if succ = NULL then
           pred→next := NULL;
           if CAS(&(lock→last), entry, pred) then
               entry→next := oldtop;
               return
           end;
           repeat succ := entry→next until succ ≠ NULL
       end;
       pred→next := succ;
       enry→next := oldtop
   end;

   procedure release_lock(lock: LockPtr, me: NodePtr);
       var top, entry, pred: NodePtr;
       var hentry, hpred: NodePtr;
   begin
*4     lock→maxprio = MIN_PRIO;
       top := me→next;
       if top = NULL then
           if CAS(&(lock→last), me, NULL) then
               // the queue becomes empty.
               return
           end;
           repeat top := me→next until top ≠ NULL
       end;
       // search the higest priority processor
       hentry := top;
       hpred := NULL;
       pred := top;
       entry := pred→next;
       while entry ≠ NULL do
           if (entry→prio > hentry→prio) then
               hentry := entry;
               hpred := pred;
           end;
           pred := entry;
           entry := pred→next
       end;
       // now, hentry is the higest priority processor.
       if hentry ≠ top then
           move_to_top(lock, hentry, hpred, top)
       end;
*5     raise_priority(lock, hentry→prio);
       hentry→locked = Released
   end;
```

Fig. 9: The first algorithm (cont.)

```
    procedure raise_priority_notify(lock: LockPtr,
                                    newprio: integer);
        var notifyp: pointer to boolean;
    begin
        if raise_priority(lock, newprio) then
            notifyp := lock→notifyp;
            if notifyp ≠ NULL then
                // set the notification flag.
                *notifyp := TRUE
            end
        end
    end;

    procedure acquire_first_lock(lock: LockPtr, me: NodePtr);
        var pred: NodePtr;
    begin
        me→next := NULL;
        // enqueue myself.
        pred := fetch_and_store(&(lock→last), me);
        if pred ≠ NULL then
            // when the queue is not empty
            me→locked := Locked;
            me→prio := my_prio;
            pred→next := me;
*7          raise_priority_notify(lock, my_prio);
            repeat until me→locked = Released
        else
            // succeed to acquire the lock without waiting.
            raise_priority(lock, my_prio)
        end;
*8      lock→notifyp := &my_notify
    end;

    procedure acquire_second_lock(lock: LockPtr,
                                  me: NodePtr, lock1: LockPtr);
        var pred: NodePtr;
    begin
        me→next := NULL;
        pred := fetch_and_store(&(lock→last), me);
        if pred ≠ NULL then
            me→locked := Locked;
            me→prio := my_prio;
            pred→next := me;
*7          raise_priority_notify(lock, my_prio);
            my_notify := TRUE;
            repeat
                // check if a priority inheritance is notified.
*6              if my_notify then
                    my_notify := FALSE;
                    if lock1→maxprio > my_prio then
                        my_prio := lock1→maxprio;
                        me→prio := my_prio;
*7                      raise_priority_notify(lock, my_prio)
                    end
                end
            until me→locked = Released
        else
            raise_priority(lock, my_prio)
        end;
*8      lock→notifyp := &my_notify
    end;
```

Fig. 10: The second algorithm

```
    procedure release_lock(lock: LockPtr, me: NodePtr);
        var top, entry, pred: NodePtr;
        var hentry, hpred: NodePtr;
    begin
        lock→maxprio := MIN_PRIO;
*9      lock→notifyp := NULL;
        top := me→next;
        if top = NULL then
            if CAS(&(lock→last), me, NULL) then
                // the queue becomes empty.
                return
            end;
            repeat top := me→next until top ≠ NULL
        end;
        // search the higest priority processor
        hentry := top;
        hpred := NULL;
        pred := top;
        entry := pred→next;
        while entry ≠ NULL do
            if (entry→prio > hentry→prio) then
                hentry := entry;
                hpred := pred
            end;
            pred := entry;
            entry := pred→next
        end;
        // now, hentry is the higest priority processor.
        if hentry ≠ top then
            move_to_top(lock, hentry, hpred, top)
        end;
        raise_priority(lock, hentry→prio);
        hentry→locked := Released
    end;
```

Fig. 11: The second algorithm (cont.)

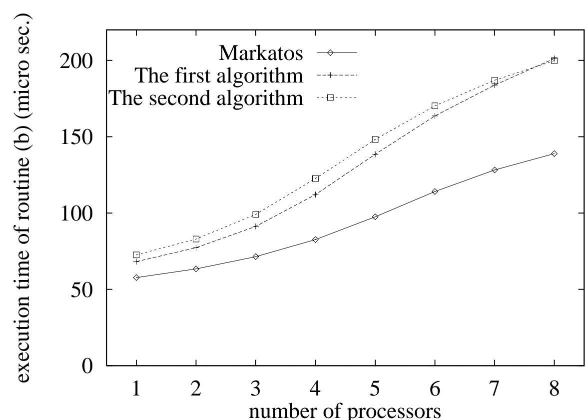

Fig. 12: Average execution times of routine (b)