A Bounded Spin Lock Algorithm with Preemption

Hiroaki Takada and Ken Sakamura

Department of Information Science,
Faculty of Science, University of Tokyo

Abstract

Predictable interprocessor synchronization and fast interrupt response are important for real-time systems constructed using asymmetric shared-memory multiprocessors. This paper points out the problem that existing spin lock algorithms cannot satisfy both requirements at the same time, and proposes a new algorithm to solve this problem. The algorithm, an extension of queueing spin locks modified to be preemptable for servicing interrupts, can give an upper bound on the time to acquire and release an interprocessor lock while achieving fast response to interrupt requests. The effectiveness of the algorithm is demonstrated through performance evaluation.

1 Introduction

Requirements for large-scale and high performance real-time systems are increasing with the expansion of the application areas of embedded real-time systems. In particular, these requirements are rapidly increasing in the areas of large-scale control systems (industrial-plant control and aircraft control systems), and communication servers (packet switchers and network routers). In these applications, many external devices such as sensors, actuators, and network controllers are connected to a system and not only massive computational power but also fast and predictable response to external events from the devices are required. Adopting a multiprocessor architecture is a promising approach to make a system responsive to growing numbers of external events.

Since the required processing time for each external device can be estimated beforehand in most of these applications, it is preferable that each device be handled by a fixed processor (or a fixed set of processors) and that the interface with the device be connected to the local bus of the processor. A distributed shared-memory architecture is also adopted in which memory modules are connected to the local bus of processors. In this kind of *asymmetric multiprocessor system*, because the code and data areas of the program handling an external device are placed in the local memory of the processor for the device, the number of shared-bus (or interconnection network) transactions can be reduced compared to a symmetric architecture. This is profitable not only because the high-performance shared bus and expensive cache mechanisms can be omitted, but also because the predictability of the system can be improved through the reduction of access conflicts on the shared bus.

We are designing a real-time kernel specification called ITRON-MP[1] mainly for this kind of asymmetric shared-memory multiprocessor, and implementing it experimentally [2].

In order to realize predictable real-time systems using shared-memory multiprocessors, a predictable interprocessor synchronization mechanism is of primary importance. In addition to adopting a real-time scheduling algorithm with resource constraints (e.g. the algorithm in [3]) or a real-time synchronization protocol (e.g. [4]), the execution time of the underlying mutual exclusion

[1]ITRON-MP is a shared-memory multiprocessor extension of ITRON, a real-time kernel specification for embedded systems [1].

mechanism must be bounded. In this paper, we focus on *bounded spin lock algorithms*, with which the time to acquire and release an interprocessor lock is bounded[2].

Fast response to external events is also important for high performance real-time systems. Because external events are notified to each processor in the form of interrupts in asymmetric shared-memory multiprocessors, the major reason for response degradation is to inhibit interrupt services for realizing mutual exclusion among tasks and interrupt handlers on the processor. Particularly, the maximum interrupt inhibition time should be given independently of the number of processors in order to make the system scalable. Fast interrupt response is also important in making blocking-based interprocessor synchronization fast, because a synchronization condition is usually notified using an interprocessor interrupt mechanism.

We first point out the problem that these two requirements, bounded spin lock and fast interrupt response, are not compatible using existing spin lock algorithms in Section 2. In Section 3, a new spin lock algorithm is proposed to solve this problem. The algorithm can give an upper bound on the time to acquire and release an interprocessor lock while realizing fast response to interrupt requests. In Section 4, the effectiveness of the proposed algorithm is demonstrated through performance evaluation.

2 Spin Locks in Multiprocessor Real-Time Systems

2.1 Existing Spin Lock Algorithms

Spin lock algorithms for shared-memory multiprocessors have been intensively studied under various conditions. In this paper, we assume that atomic read-modify-write operations on a single word of shared memory are supported in hardware. Typical examples of the operations are test_and_set, fetch_and_store (swap), fetch_and_add, and compare_and_swap.

On the same assumption, J. M. Mellor-Crummey and M. L. Scott have classified major spin lock algorithms into following four categories [5].

Test-and-set Locks

Each processor trying to acquire a lock repeatedly executes a test_and_set operation on a shared Boolean variable indicating the lock status. It releases the lock by clearing the variable. There are many variations of this algorithm in how each processor retries to execute a test_and_set operation [6].

Ticket Locks

Two shared counters are used in ticket locks: a request counter and a release counter. Each processor increments the request counter using a fetch_and_add operation and obtains the old value of the counter, which indicates its turn to acquire the lock. Then, it waits until the release counter is equal to the value. To release the lock, the processor increments the release counter. There are some variations in how each processor retries to read the release counter.

Array-Based Queueing Locks

In this class of algorithms, each processor is linked to an array-based queue. An algorithm using a fetch_and_add operation [6] and another using a fetch_and_store operation [7] have been proposed. With these algorithms, the number of shared-bus transactions is bounded on cache-coherent multiprocessors independently of the number of processors, and the bus contention problem is resolved.

List-Based Queueing Locks

[2] When we say that there is an upper bound on the acquisition or release time in spin locks, we assume that the access time of the shared bus is bounded.

```
        type qnode = record
            next: pointer to qnode;
            locked: (Released, Locked)
        end;
        type lock = pointer to qnode;

        shared var L: lock;
        // L is initialized to NIL.

        var I: qnode;
        var pred: pointer to qnode;

        I.next := NIL;
        pred := fetch_and_store(&L, &I);
        if pred ≠ NIL then
            I.locked := Locked;
            pred→next := &I;
            repeat until I.locked = Released
        end;
        //
        // critical section.
        //
        if I.next = NIL then
            if compare_and_swap(&L, &I, NIL) then
                goto exit
            end;
            repeat until I.next ≠ NIL
        end;
        I.next→locked := Released;
    exit:
```

Figure 1: The MCS lock

In this class of algorithms, each processor trying to acquire a lock is linked to an list-based queue. The MCS lock algorithm using a fetch_and_store operation and a compare_and_swap operation has been proposed [5]. There is a variation which uses fetch_and_store operations only.

Pseudo-code for the MCS lock appears in Figure 1. In this figure, the keyword `shared` indicates that only one instance of the variable is allocated and shared in the system. Other variables are allocated for each processor. `Fetch_and_store` reads the memory addressed by the first parameter (which must be a pointer), returns the contents of the memory as its value, and atomically writes the second parameter to the memory. `Compare_and_swap` is a Boolean function with three parameters. It 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[3].

With the MCS lock, when the queue node area of each processor (variable `I` in Figure 1) is located on its local memory[4], the number of shared-bus transactions is bounded even on multiprocessors without a coherent cache.

In these algorithms, test-and-set locks are not appropriate for real-time systems because the time until a processor can acquire a lock cannot be bounded.

[3] The compare_and_swap instructions of many existing processors store the contents of the memory to the third parameter in this case. This mechanism is not used in this paper.

[4] The local memory of a processor is memory which can be accessed from the processor without using the shared bus and can be accessed from others through the shared bus.

```
acquire_lock;
disable_interrupts;
//
// critical section.
//
enable_interrupts;
release_lock;
```

Figure 2: Acquiring a lock precedes disabling interrupts

```
disable_interrupts;
acquire_lock;
//
// critical section.
//
release_lock;
enable_interrupts;
```

Figure 3: Disabling interrupts precedes acquiring a lock

2.2 Bounded Spin Lock and Interrupt Latency

In order to bound the time until a processor acquires a lock for accessing shared data, the duration that each processor holds the lock as well as the number of contending processors that the processor waits for must be bounded. The latter condition can be met with ticket locks or queueing locks described in the previous section. To satisfy the former condition, the relationship with interrupt services must be considered.

In asymmetric multiprocessor systems, interrupt services for external devices are requested for each processor. When multiple devices are connected to a processor, interrupt requests from them are usually raised independently and the maximum time to service all of the requests becomes long. Consequently, in order to give a practical bound on the duration that a processor holds a lock, interrupt services should be inhibited for that duration.

On the other hand, in order to realize a system with fast response to external events, each processor must be able to service external interrupts with short latency time. Therefore, interrupt mask times should be minimized. Particularly, when the extensibility of the system is an important issue, the maximum interrupt mask time should be given independently of the number of processors in the system.

Here, a problem arises in deciding whether interrupts should be disabled or an interprocessor lock should be acquired first. Figure 2 presents a method in which acquiring an interprocessor lock precedes disabling interrupts. With this method, interrupts are serviced while the processor holds the lock, and the condition that interrupt services should be inhibited while a processor holds a lock is not satisfied. Figure 3 presents another method where acquiring a lock follows disabling interrupts. In this method, the interrupt mask time includes the time to acquire an interprocessor lock and its bound depends on the number of processors.

To satisfy both of the requirements, bounded interprocessor mutual exclusion and fast interrupt response, interrupt services should not be inhibited while a processor waits for an interprocessor lock and should be kept inhibited after the processor acquires the lock. One method to realize this principle is the following. The processor first disables interrupts and tries to acquire the lock. If it fails to acquire the lock, the processor probes interrupt requests before it retries to acquire the lock. When interrupt requests are detected, it suspends trying to acquire the lock, enables interrupts, and services them. Pseudo-code for the test-and-set lock with preemption, an extension of the simple test-and-set lock algorithm with this method, appears in Figure 4 [8].

In ticket locks and queueing locks, on the other hand, a processor modifies some shared data and reserves its turn to acquire a lock when it begins to wait for the lock, and the lock is passed to the processor by another when its turn comes. Therefore, if the processor simply branches to the interrupt handler in detecting requests and if its turn comes during the interrupt service,

```
type lock = (Released, Locked);

shared var L: lock;
// L is initialized to Released.

disable_interrupts;
while test_and_set(L) = Locked do
    if interrupt_requested then
        enable_interrupts;
        // interrupt service.
        disable_interrupts
    else
        delay
    end
end;
//
// critical section.
//
L := Released;
enable_interrupts;
```

Figure 4: The test-and-set lock with preemption

the remaining interrupt service time is included in the time that the processor holds the lock. Consequently, simple extensions of ticket locks and queueing locks with the above method do not satisfy the above principle. In the following section, we present a new algorithm, an extension of a queueing lock, with which a processor can service interrupts with short latency while satisfying the principle.

3 A Queueing Lock with Preemption

In this section, we present a new algorithm that can give an upper bound on the time until a processor acquires a lock and that enables interrupt services while the processor waits for the lock.

In all spin lock algorithms which can give a bound on the time until a processor acquires a lock, a processor modifies some shared data and determines its turn to acquire a lock when it begins to wait for the lock. If the processor simply branches to an interrupt handler while waiting for the lock, it cannot begin executing the critical section immediately after the lock is passed to the processor by another, and makes the contending processors wait wastefully until the interrupt service is finished. Therefore, when a processor begins to service interrupts while waiting, it must inform other processors that it is servicing interrupt requests and should not be passed the lock. The processor trying to release the lock checks if the succeeding processor is servicing interrupts. If the succeeding processor is found to be servicing interrupts, its turn to acquire the lock is canceled or deferred, and the lock is passed to the next in line.

Pseudo-code for an extended algorithm of the MCS lock with the above method appears in Figure 5. In this figure, the right hand side of the operator `and` is assumed to be evaluated if its left hand side is true. `CAS` is an abbreviation of `compare_and_swap`.

In this algorithm, a processor informs others that it is servicing interrupts by writing the value `Preempted` to the `state` field of its queue node record (i.e. `I.state`).

If the processor releasing the lock (P_0) finds that the succeeding processor (P_1) is servicing interrupts, P_0 dequeues P_1 from the waiting queue for the lock and passes the lock to the successor of P_1. When only P_1 is waiting for the lock, P_0 makes the waiting queue empty. P_0 informs P_1 that P_1 is dequeued by changing the value of the `state` field of P_1's queue node to `Released`. During this process, a transient status occurs that P_1's queue node has been dequeued but that the node area must not be reused because the value of its `next` field is necessary. P_0 informs P_1 of this transient status by writing the value `Canceled` to the `state` field of P_1's queue node.

```
    type qnode = record
        next: pointer to qnode;
        state: (Released, Locked, Preempted, Canceled)
    end;
    type lock = pointer to qnode;

    shared var L: lock;
    // L is initialized to NIL.

    var I: qnode;
    var pred, succ, sn: pointer to qnode;

retry:
    I.next := NIL;
    disable_interrupts;
    pred := fetch_and_store(&L, &I);
    if pred ≠ NIL then
        I.state := Locked;
        pred→next := &I;
        while (I.state ≠ Released) do
            if interrupt_requested and
                    CAS(&(I.state), Locked, Preempted) then
                enable_interrupts;
                // interrupt service.
                disable_interrupts;
                if ¬CAS(&(I.state), Preempted, Locked) then
                    enable_interrupts;
                    repeat while I.state ≠ Released;
                    goto retry
                end
            end
        end
    end;
    //
    // critical section.
    //
    succ := I.next;
    if succ = NIL then
        if CAS(&L, &I, NIL) then goto exit end;
        repeat succ := I.next until succ ≠ NIL
    end;
    while ¬CAS(&(succ→state), Locked, Released) do
        if CAS(&(succ→state), Preempted, Canceled) then
            sn := succ→next;
            if sn = NIL then
                if CAS(&L, succ, NIL) then
                    succ→state := Released;
                    goto exit
                end;
                repeat sn := succ→next until sn ≠ NIL
            end;
            succ→state := Released;
            succ := sn;
        end
    end;
exit:
    enable_interrupts;
```

Figure 5: The queueing lock with preemption

When the processor that has branched to an interrupt handler while waiting for a lock finishes the handler, it reads the `state` field of its queue node and checks whether it has been dequeued during the interrupt service or not. If it has been dequeued, it re-executes the lock acquiring routine from the beginning after waiting until its queue node area becomes reusable. Otherwise, it recovers its `state` field to the value `Locked` and resumes waiting for the lock.

In this algorithm, a processor waiting for a lock can acquire the lock in the order of the waiting queue and the time until it acquires the lock can be bounded if no interrupt request is raised on the processor. In releasing a lock, the algorithm also gives an upper bound on the number of search loops for identifying to which processor the releasing processor should pass the lock, unless interrupt services start and finish repeatedly on the waiting processors[5]. As interrupt services are inhibited while the processor holds a lock, no interrupt service time is included in the lock holding time. Consequently, both the time until a processor acquires a lock and the time until it releases the lock can be bounded in this algorithm under the above conditions.

When a processor services interrupts while waiting for a lock and is dequeued from the waiting queue, the processor must re-execute the lock acquiring routine and link itself to the end of the waiting queue. Therefore, the waiting time after it first links itself to the queue until it branches to the interrupt handler is wasted. When the schedulability of the system is analyzed, this re-execution overhead should be added to the interrupt service time.

When the execution time of the code inside the critical section is bounded, the interrupt mask time is also bounded under the same condition that the releasing time is bounded. Because a processor observes interrupt requests while it is waiting for a lock, the upper bound of the interrupt mask time in the lock acquiring routine does not depend on the number of processors. On the other hand, the interrupt mask time in the lock releasing routine depends on the number of processors. However, it can be considered to be bounded in practice, because the number of search loops follows an exponential distribution and because the processing time of one loop is short.

The proofs of the important features of this algorithm, mutual exclusion and deadlock freedom when a certain condition is laid on interrupt occurrence, is presented in Appendix A.

The queueing lock with preemption proposed in this section is based on the MCS lock. Array-based queueing locks can be extended similarly. On the other hand, ticket locks cannot be extended in this method, since the algorithms do not have a shared data area with which a processor informs others of its status.

4 Performance Evaluation

In this section, the effectiveness of the queueing lock with preemption presented in Figure 5 (called QL/P, in this section) is examined through performance evaluation. The performance of the algorithm is compared with the MCS lock without inhibiting interrupts (QL/ei), the algorithm in which interrupts are disabled before an interprocessor lock is acquired as appeared in Figure 3 using the MCS lock (QL/di), and test-and-set locks with preemption presented in Figure 4. We have adopted two versions of test-and-set locks with preemption: one with constant delay (T&S/P/const) and another with exponential backoff in which the delay between successive test_and_set operations is exponentially increased up to a predetermined bound (T&S/P/exp). Past studies show that a test-and-set lock has good scalability with exponential backoff [5, 6]. However, because the lock acquisition time varies widely with exponential backoff, it is expected to be inappropriate for real-time systems. This conjecture is also confirmed through our experiments.

[5]A processor can be visited twice in the search loops in the following case. Immediately after the processor is dequeued from the waiting queue, it finishes the interrupt service and links itself to the end of the queue. If this case repeatedly occurs until the processor to which to pass the lock is identified, the number of the loops is not bounded. This case rarely occurs. But, when this problem cannot be ignored (when the number of processors is large and when interrupts are requested frequently, in general), the algorithm should be modified so that writing `Released` to the `state` field of a dequeued processor should be delayed until the processor to which to pass the lock is identified.

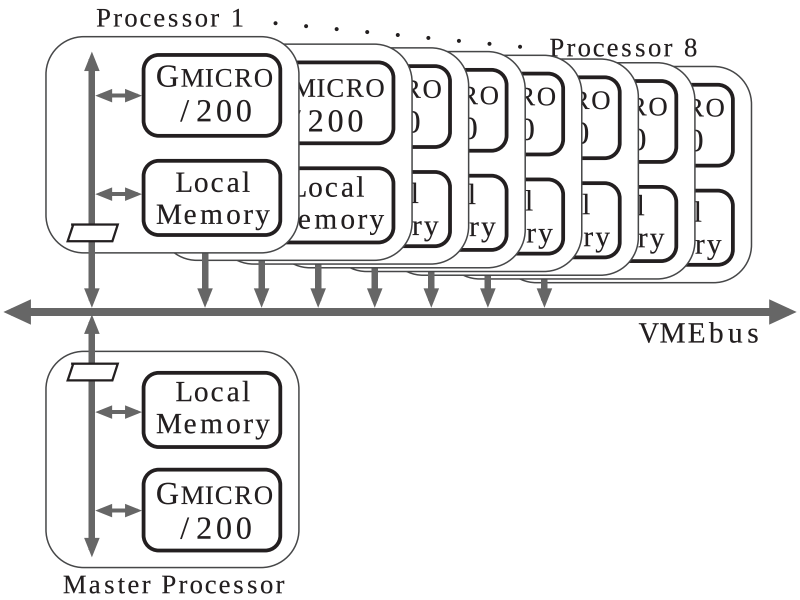

Figure 6: Evaluation Environment

4.1 Evaluation Environment

We have used a shared-bus multiprocessor system for the evaluation. The shared bus is based on the VMEbus specification, and each processor node consists of a 20 MHz GMICRO/200 processor, 1 MB of local memory, and some I/O interfaces (Figure 6). The GMICRO/200 is a TRON-specification CPU rated at approximately 10 MIPS with a 20 MHz clock [9]. The local memory can be accessed from other processor nodes through the shared bus. No cache memory is equipped on the processor node. In our experiments, the data area necessary for each processor and the program code area are placed in the local memory of the processor. Data requiring only one instance in the system is placed in the local memory of the master processor, which does not execute spin locks.

A TRON-specification CPU supports three read-modify-write instructions: bit_test_and_set (BSETI), bit_test_and_clear (BCLRI), and compare_and_swap (CSI). Since the fetch_and_store operation necessary for the MCS lock and our algorithm is not supported, it was emulated using the compare_and_swap instruction and a retry loop. Therefore, the feature of the MCS lock bounding the number of shared-bus transactions was not realized in this experiments. The evaluation programs were written in C with inline assembler code for the read-modify-write instructions. There is some overhead in passing data between code written in C and code in assembler.

The round-robin arbitration of the VMEbus was adopted in our experiments. As the VMEbus has only four pairs of bus request/grant lines, the round-robin scheme can be applied to at most four bus masters. Therefore, processors are classified into four classes by the bus request line they use, and the static priority scheme is applied among processors belonging to a same class. Accessing local memory on other processor nodes takes nearly 1 μs and is a relatively slow operation compared with the performance of the processor.

4.2 Measurement Method

We have adopted the following method in measuring the performance of the algorithms. Each processor executes the code presented in Figure 7 while periodic interrupt requests are raised on the processor by a cyclic timer. The execution time of a critical region (the region between ① and ② in Figure 7) is measured for each execution of the region, and the distributions when no interrupt is serviced during the region and when an interrupt is serviced are obtained. The

```
for i := 1 to NoLoop do
 ① acquire_lock_and_disable_interrupts;
    //
    // critical section.
    //
    release_lock;
 ② enable_interrupts;
    random_delay
end;
```

Figure 7: Measurement Program Skeleton

interrupt latency is also measured for each interrupt service and its distribution is obtained.

Inside the critical section, a processor accesses memory through the shared bus some number of times (for making the effect of bus traffic explicit) and waits for a while using empty loops. When `acquire_lock` and `release_lock` are omitted, the execution time of the critical region is about 40 μs including some overhead for obtaining the start and termination time of the region. In order to change timing conditions, each processor waits for a random time following an exponential distribution before it re-enters the critical region (*random_delay* in Figure 7). Here, the processor also records the execution time of the critical region. The average time of the random delay plus this recording time is about 40 μs.

Empty loops are also included in the interrupt handler in addition to the processing for recording interrupt latency. The total execution time of an interrupt handler is about 80 μs. The period of interrupt requests is about 2 ms. The exact length of this period is varied in 0–3% for each processor in order that the timing of interrupt requests for each processor should not be synchronized. Other interrupt requests are inhibited during the measurement.

4.3 Performance Metrics

Figure 8 presents the distributions of the execution time of the critical region with QL/P and T&S/P/const, when four processors execute spin locks[6]. The fluctuations in short cycle appearing in T&S/P/const is an effect of constant delay between test_and_set operations (*delay* in Figure 4). Figure 9 presents the distributions of the interrupt latency under the same conditions.

Figure 9 shows that there are practical upper bounds on the interrupt latency with both algorithms. On the contrary, Figure 8 indicates that it is difficult to determine the upper bound of the execution time of the critical region with T&S/P/const. This is because which processor acquires the lock is randomly determined with test-and-set locks, and illustrates that test-and-set locks are not appropriate for real-time systems. As presented later in Figure 12, the difference of the average execution times of the critical region with these two algorithms is only 10% or so.

In real-time systems, the effectiveness of algorithms should not be evaluated with their average performance but with their worst-case execution (or response) times. However, in the case of spin lock algorithms, worst-case times cannot be obtained through experiments because of unavoidable non-determinism in multiprocessor systems. Worst-case times are also inadequate as a metric in our evaluation because the execution time of a critical region cannot be bounded in test-and-set locks. Therefore, in place of worst-case times we have adopted p-reliable times, the probability p with which a processor finishes to execute a critical region (or responds to an interrupt request), as a performance metric. In the following section, we show the evaluation results when p is 0.999 (i.e. 99.9%)[7].

4.4 Evaluation Results

Figures 10 and 11 present the 99.9%-reliable execution time of the critical region (when no interrupts are serviced during the region) and the 99.9%-reliable interrupt latency time, respectively,

[6] Note that the vertical axis of Figure 8 and 9 (probability density) is in logarithmic scale.

[7] In our experiments, similar results were obtained when appeared worst-case times are used.

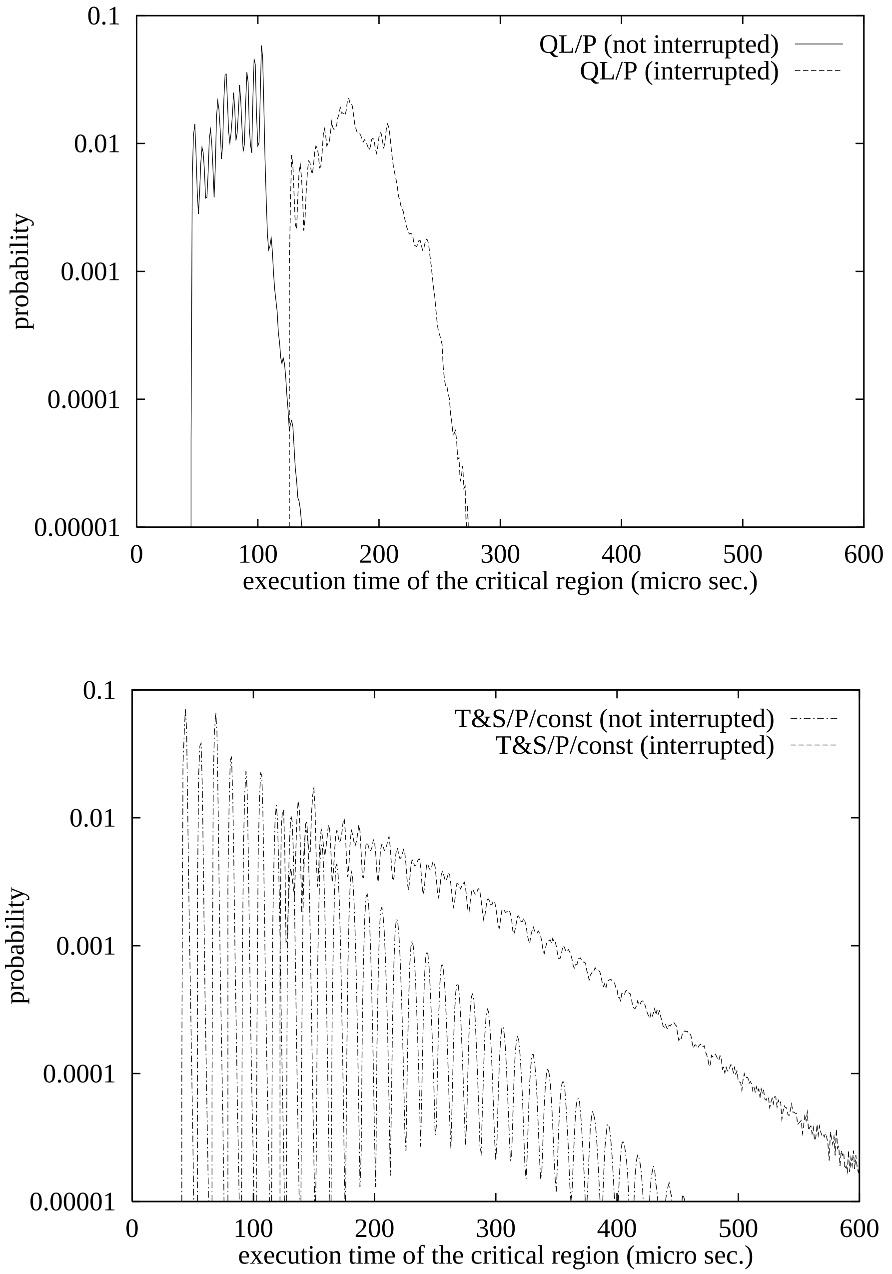

Figure 8: Distributions of the execution times of a critical region

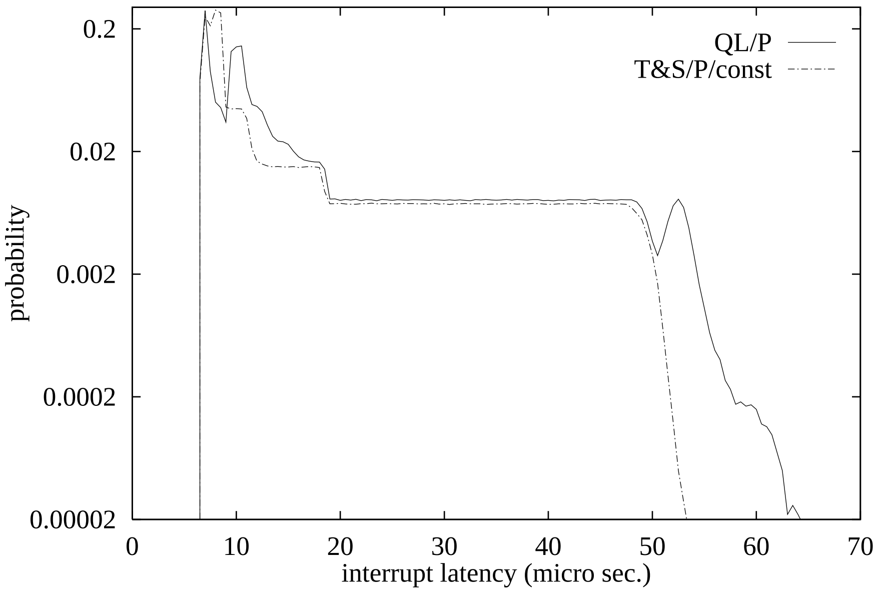

Figure 9: Distribution of interrupt latency times

as the number of processors is increased from one to eight.

In Figure 10, the execution time of the critical region increases linearly with the number of processors with QL/P, and the algorithm is found to be scalable. QL/ei exhibits poorer performance because processors service interrupt requests during the critical region. With T&S/P/const, the execution time increases rapidly when the number of processor becomes large, and the algorithm does not scale well. T&S/P/exp, the test-and-set lock with exponential backoff, has the worst scalability.

In Figure 11, the interrupt latency time is nearly independent of the number of processors with QL/P. With QL/di, on the contrary, the interrupt latency becomes long as the number of processors increases. With T&S/P/const, the interrupt latency slowly increases because the execution time of the code inside the critical section becomes longer due to the effect of shared-bus contention.

From these observations, it is demonstrated that QL/P can give a practical upper bound on the time to acquire and release an interprocessor lock while achieving fast response to interrupt requests. The other algorithms cannot satisfy these two requirements at the same time.

Finally, in order to examine the average performance of the algorithms, we present the average execution time of the critical region (when no interrupts are serviced during the region) in Figure 12. When the number of processors is small, QL/P is slower than T&S/P/const by about 10%. As the number of processors becomes larger, the average performance of T&S/P/const becomes worse. This is an effect of the bus contention problem and is not observed with T&S/P/exp, which adopts the exponential backoff scheme. Consequently, the exponential backoff scheme is appropriate when average performance is the major concern but inadequate for real-time systems where worst-case behavior is important.

5 Conclusion

Existing spin lock algorithms cannot satisfy two important requirements for real-time systems using asymmetric shared-memory multiprocessors, bounded spin lock and fast interrupt response, at the same time. In this paper, we propose a new spin lock algorithm that can give an upper

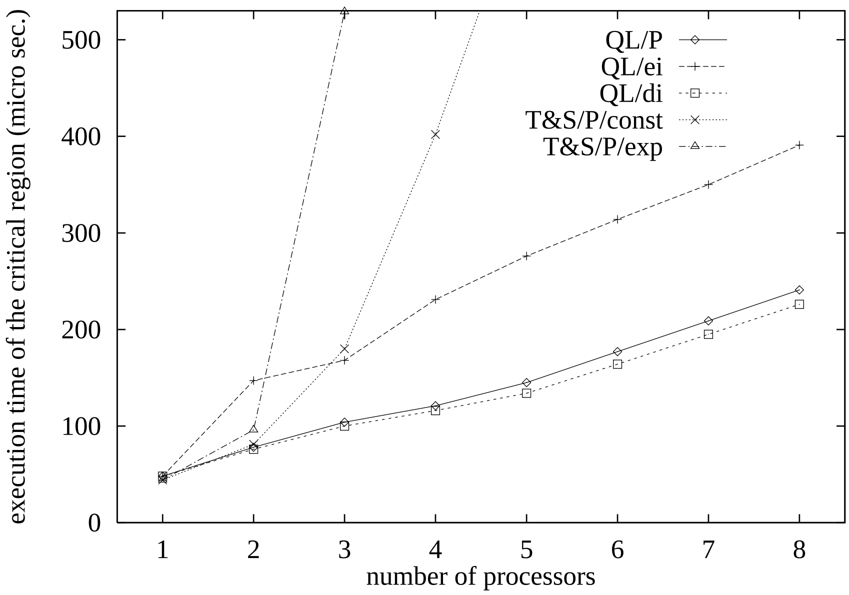

Figure 10: The 99.9%-reliable execution time of the critical region

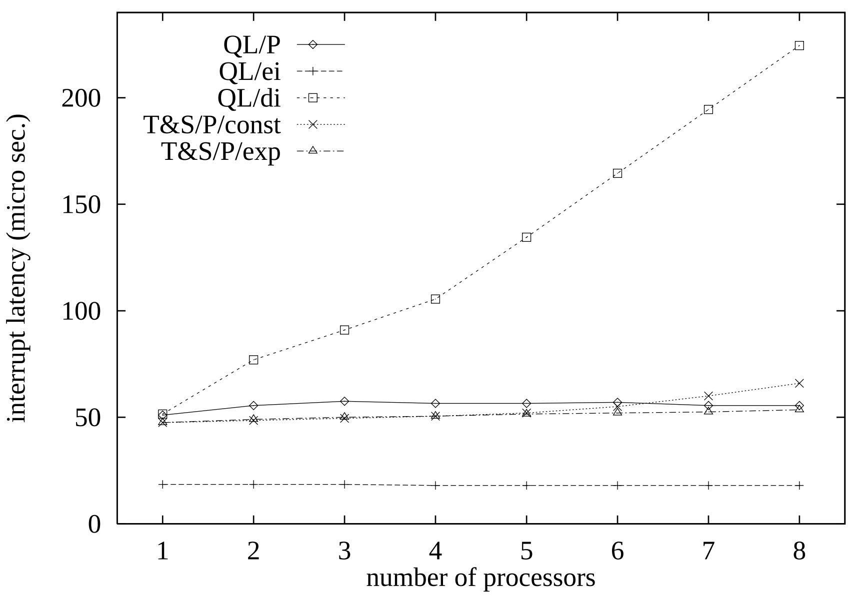

Figure 11: The 99.9%-reliable interrupt latency time

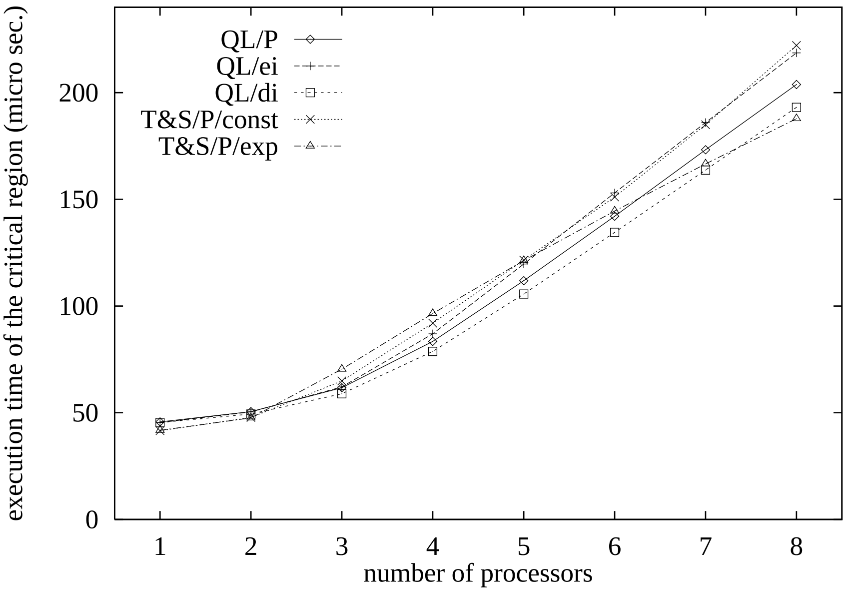

Figure 12: The average execution time of the critical region

bound on the time to acquire and release an interprocessor lock while realizing fast response to interrupt requests. To evaluate its effectiveness, we have measured its performance through experiments and confirmed that the algorithm has required properties.

Although the proposed algorithm is somewhat slower than the test-and-set lock with preemption in its average behavior, it is more appropriate for real-time systems in which the average performance can be degraded to improve worst-case behavior.

The combination of the proposed algorithm with prioritized spin locks [10] remains as future work. It is also important to adopt the algorithm in a real-time kernel based on the ITRON-MP specification and to evaluate the algorithm in real applications.

A Proofs on the Queueing Lock with Preemption

We first show that the algorithm in Figure 13 realizes mutual exclusion. The difference between the algorithm and the one in Figure 5 is: (1) the initial value of the `state` field is determined to be `Released` and (2) compare_and_swap operations are used in assigning `Released` to the `state` field of queue nodes (in the lines marked with ⑯ and ⑱). Then, we show that the algorithm is deadlock free. Once mutual exclusion and deadlock freedom are proved, the equivalence of these two algorithms is straightforward.

First, the state of a processor is classified into nineteen states by the execution point of the processor, which is presented in Figure 13 as ①–⑲. A state transition occurs when the processor accesses a shared data, with which the processor interacts with others. For example, the transition from ① to ② occurs when the processor reads `I.next`. Similarly, the transition from ② to ③ or ⑨ occurs when the processor executes the fetch_and_store operation. Whether the processor moves to ③ or ⑨ is fixed at this moment. The only exception is the transition from ⑲ to ⑫ which occurs when the processor modifies its private variable `succ`.

The state of a processor is also classified by the value of the `state` field of its queue node into released state (R state, in short), locked state (L state), preempted state (P state), and canceled state. Canceled state is further classified into two states: the state that the variable `L` is kept non-`NIL` all after `Canceled` is assigned to the `state` field (C state), and the state after `L` becomes

```
     type qnode = record
          next: pointer to qnode;
          state: (Released, Locked, Preempted, Canceled)
     end;
     type lock = pointer to qnode;

     shared var L: lock;
     // L is initialized to NIL.

     var I: qnode;
     // I.state is initialized to Released.
     var pred, succ, sn: pointer to qnode;

retry:
 ①  I.next := NIL;
     disable_interrupts;
 ②  pred := fetch_and_store(&L, &I);
     if pred ≠ NIL then
      ③ I.state := Locked;
      ④ pred→next := &I;
      ⑤ while (I.state ≠ Released) do
               if interrupt_requested and
                    ⑥ CAS(&(I.state), Locked, Preempted) then
                    enable_interrupts;
                    // interrupt service.
                    disable_interrupts;
                  ⑦ if ¬CAS(&(I.state), Preempted, Locked) then
                         enable_interrupts;
                       ⑧ repeat while I.state ≠ Released;
                         goto retry
                    end
               end
          end
     end;
     //
 ⑨  // critical section.
     //
     succ := I.next;
     if succ = NIL then
      ⑩ if CAS(&L, &I, NIL) then goto exit end;
      ⑪ repeat succ := I.next until succ ≠ NIL
     end;
 ⑫  while ¬CAS(&(succ→state), Locked, Released) do
      ⑬ if CAS(&(succ→state), Preempted, Canceled) then
           ⑭ sn := succ→next;
               if sn = NIL then
                ⑮ if CAS(&L, succ, NIL) then
                     ⑯ CAS(&(succ→state), Canceled, Released);
                         goto exit
                    end;
                ⑰ repeat sn := succ→next until sn ≠ NIL
               end;
           ⑱ CAS(&(succ→state), Canceled, Released);
           ⑲ succ := sn
          end
     end;
exit:
 ①  enable_interrupts;
```

Figure 13: The queueing lock with preemption (modified for the proofs)

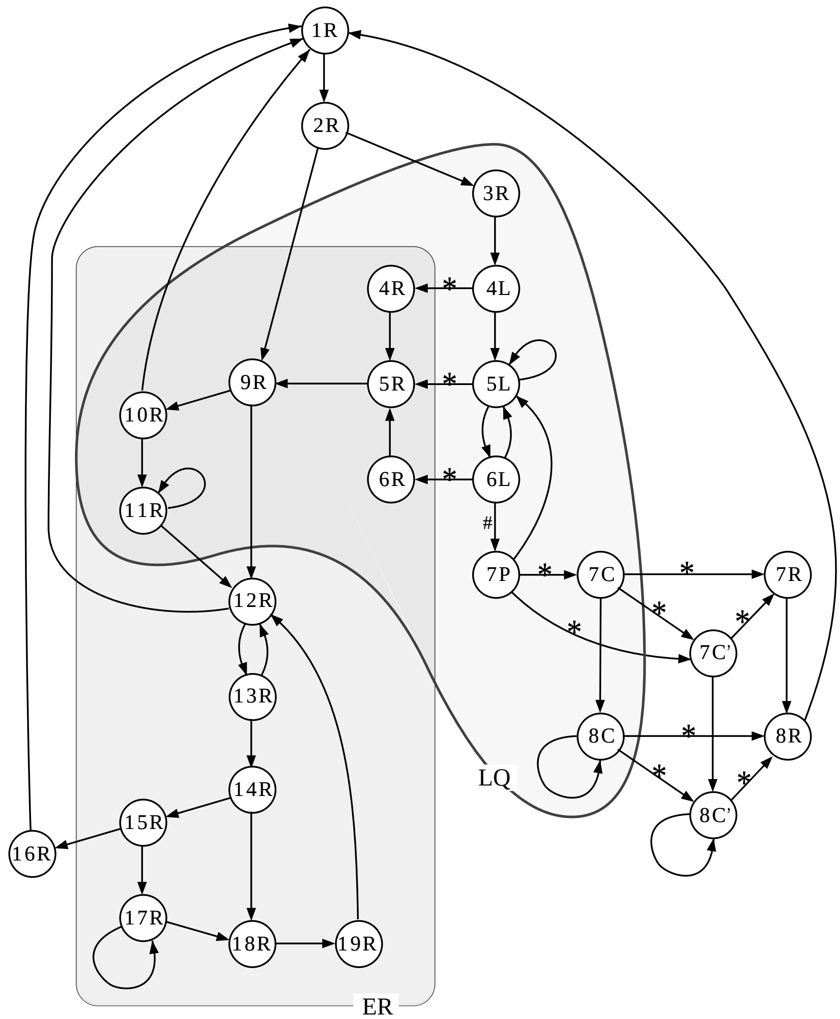

Figure 14: The state transition diagram of a processor

`NIL` (C' state).

The state transition diagram of a processor presented in Figure 14 can be obtained from these two classifications and some observations of the code in Figure 13 such as the fact that a processor assigns `Locked` to the `state` field of its queue node with the transition from ③ to ④, the fact that a processor changes the `state` field of another processor only from `Locked` to `Released`, from `Preempted` to `Canceled`, and from `Canceled` to `Released`, and the fact that the transition from C' state to C state does not exist by definition[8]. The transitions marked with "∗" in the diagram are caused by other processors, and the transition with "#" occurs only when an interrupt request is raised on the processor.

A processor is called to be in the *exclusive region* (ER, in short), when its state is included in ER in Figure 14. In the following, we call the `state` (and `next`) field of the queue node of a processor simply as the `state` (and `next` respectively) field of the processor.

Lemma 1 When `L` is `NIL`, no processor is in ER. When `L` is not `NIL`, there is one (and only one) processor that is in ER.

Proof: In the initial state, the condition is satisfied because `L` is initialized to `NIL` and the state of each processor is 1R. Then, the lemma can be proved by showing that for each transition, if

[8]Following discussions reveal two other facts that a processor never becomes 4R state and that the transition from 7P to 7C' does not occur.

the condition is satisfied before the transition, it is preserved with the transition. We may safely check only the transitions with which a processor enters/leaves ER or `L` is modified.

- 2R→9R (The processor enters ER and `L` is modified.)

  This transition occurs only when `L` is `NIL` and changes it to non-`NIL`. There are no processor in ER before the transition since `L` is `NIL`. Therefore, the condition is preserved.

- 4L→4R, 5L→5R, 6L→6R (The processor enters ER.)

  These transitions occur only when another processor changes the `state` field to `Locked`; in other words, it makes the transition from 12R to 1R. In this case, a processor enters ER while another leaves ER. As `L` is not modified in these transitions, the condition is preserved.

- 12R→1R (The processor leaves ER.)

  A processor making this transition changes the `state` field of another processor from `Locked` to `Released`; in other words, it causes a transition from 4L/5L/6L to 4R/5R/6R on another processor. This is the same situation with the above.

- 10R→1R, 15R→16R (The processor leaves ER and `L` is modified.)

  These transitions occur only when `L` is not `NIL` and change it to `NIL`. Therefore, the condition is preserved.

- 2R→3R (`L` is modified.)

  `L` is kept non-`NIL` with this transition. Therefore, the condition is preserved. □

Theorem 2 (Mutual Exclusion) There is at most one processor which is in 9R state.
Proof: This directly follows from Lemma 1. □

In the following, the processor in ER is called the *lock holder* (LH, in short), if any. A processor is called to be designated by a pointer variable when its queue node is pointed to by the pointer.

Next, we define the *lock queue* which is an ordered list of processors. The last processor of the lock queue is defined to be the one designated by `L`. When `L` is `NIL`, the lock queue is defined to be empty. The predecessor of a processor in the lock queue is the one designated by its `pred` variable. When `L` is not `NIL`, the first processor of the queue is defined according as the state of LH (which exists from Lemma 1) as follows.

(1) When LH is in 4R, 5R, 6R, 9R, 10R, or 11R, LH is the first processor of the lock queue.

(2) When LH is in 12R, 13R, 14R, 15R, 17R, or 18R, the processor designated by the `succ` variable of LH is the first one of the lock queue.

(3) When LH is in 19R, the processor designated by the `sn` variable of LH is the first one of the lock queue.

In the next lemma, we show that the lock queue is well-structured and handled focusing only on the lock queue operations. We need the following assumption for further discussion.

Assumption 3 Any processor has not been included in the lock queue when it is in 1R state. □

In the initial state, this assumption is satisfied because all processors are in 1R and because the lock queue is empty. To show that the assumption always holds, it is necessary to prove that a processor is not included in the lock queue when it returns to 1R state. The algorithm in Figure 13 realizes this property by introducing the transient status in which the `state` field is `Canceled`.

In the following, we suppose that this assumption alway holds. It is proved that a processor is not included in the lock queue when it returns to 1R state in Lemma 7 after the discussions which take the value of `state` fields into consideration. This result shows that the assumption is preserved if it is satisfied in the initial state. Therefore, the assumption is proved inductively using Lemma 7.

Lemma 4 Following two conditions hold under Assumption 3.

(1) A processor modifies the lock queue with only two kind of operations: (a) inserting itself at the end of the lock queue when it is not included in the queue and (b) removing the first processor of the lock queue from the queue.

(2) When the `next` field of a processor included in the lock queue is not `NIL`, it designates the successor of the processor in the lock queue.

Proof: In the initial state, the conditions are satisfied because no operation has been done on the lock queue and because the lock queue is empty. Then, the lemma can be proved by showing that for each transition, if the conditions are satisfied before the transition, they are preserved with the transition. We may safely check only the transitions with which the lock queue is changed or with which the `next` field of a processor included in the lock queue is modified. The lock queue is modified in the following four cases: (a) `L` is changed, (b) the `pred` variable of a processor in the lock queue is changed, (c) LH is changed, and (d) LH makes a transition beyond the boundaries with which the first processor of the lock queue is defined.

- 2R→3R, 2R→9R (`L` is changed and the `pred` variable is changed.)

  A processor making one of these transitions becomes the last processor of the lock queue after the transition. In case of 2R→3R, the last processor before the transition is designated by the `pred` variable. The first processor of the lock queue remains unchanged. In case of 2R→9R, the lock queue is empty before the transition and includes only the processor making the transition after the transition. In both cases, the processor making the transition is inserted at the end of the lock queue.

  Because a processor in 1R is not included in the lock queue from Assumption 3 and because a processor is not inserted to the lock queue by another processor from Condition (1), a processor in 2R is not included in the lock queue.

  Since the `next` field of a processor is modified only when it is designated by the `pred` variable of another processor, the `next` field of the processor which is not included in the lock queue or is at the end of the lock queue is not modified by another processor. Because the processor making the transition 2R→3R/9R is not included in the lock queue before the transition and is at the end of the lock queue after the transition, the `next` field of the processor is not modified for the while. Therefore, the `next` field of the processor is `NIL` immediately after the transition.

  From the above discussions, if the conditions are satisfied before one of the transitions, they are preserved after the transition.

- 10R→1R, 15R→16R (`L` is changed.)

  Before these transitions, the lock queue includes only one processor (LH in case of 10R→1R, and the processor designated by the `succ` variable of LH in case of 15R→16R) because the first processor of the lock queue is designated by `L`. After the transitions, the lock queue becomes empty. Therefore, the transitions remove the unique processor (witch is the first processor obviously) in the lock queue from the queue, and the conditions are preserved with the transitions.

- 4L→4R, 5L→5R, 6L→6R (LH is changed.)

  These transitions occur only when another processor makes the transition from 12R to 1R. Before the transitions, the first processor of the lock queue is the one designated by the `succ` variable of the latter processor, which is the former processor obviously. After the transitions, the former processor is the first one. Consequently, the lock queue is not modified with these transitions and the conditions are preserved.

- 12R→1R (LH is changed.)

  A processor making this transition causes a transition from 4L/5L/6L to 4R/5R/6R on another processor. This is the same situation with the above.

- 9R→12R, 11R→12R (LH makes a transition beyond the boundaries.)

  The first processor of the lock queue is changed from LH to the one designated by the `succ` variable of LH with these transitions. The `succ` variable of LH equals to `I.next` and designates the successor of LH in the lock queue. Therefore, the transitions remove LH, which is the first processor of the lock queue, from the queue, and the conditions are preserved.

- 18R→19R (LH makes a transition beyond the boundaries.)

  The first processor of the lock queue is changed from the one designated by the `succ` variable of LH (P_0) to the one designated by the `sn` variable (P_1) with this transition. The `sn` variable of LH equals to `succ→next` and designates the successor of P_0 in the lock queue. Therefore, the transitions remove P_0, which is the first processor of the lock queue, from the queue, and the conditions are preserved.

- 19R→12R (LH makes a transition beyond the boundaries.)

  The first processor of the lock queue is changed from the one designated by the `sn` variable of LH to the one designated by the `succ` variable with this transition from the definition. Because the `succ` variable after the transition equals to the `sn` variable before the transition, the first processor is not changed in actual and the conditions are preserved.

- 4L→5L, 4R→5R (The `next` field is modified.)

  The processor making one of these transitions makes the `next` field of the processor designated by its `pred` variable designate itself. Therefore, the `next` field designates the successor in the lock queue, and Condition (2) is shown to be preserved with the transitions. Since the lock queue is not modified with the transitions, Condition (1) is preserved obviously. □

Lemma 5 Following conditions hold under Assumption 3.

(1) When LH is in 14R, 15R, 17R, or 18R, the processor designated by the `succ` variable of LH is in C state. Conversely, a processor in C state is designated by the `succ` variable of another processor in 14R, 15R, 17R, or 18R.

(2) When a processor is in 16R, the processor designated by its `succ` variable is in C' state. Conversely, a processor in C' state is designated by the `succ` variable of another processor in 16R.

Proof: First, we prove that the following condition is satisfied under Assumption 3.

(0) When a processor is in 14R, 15R, 16R, 17R, or 18R (we call the processor is in *SC* in the following), the `state` field of the processor designated by its `succ` variable is `Canceled`. Conversely, a processor whose `state` field is `Canceled` is designated by the `succ` variable of another processor in SC.

Since this condition obviously holds in the initial state, it is proved to be satisfied by showing that every transition preserves the condition. We may safely check only the transitions with which a processor enters/leaves SC and the ones with which the `state` field of a processor is changed from/to `Canceled` to/from another.

- 13R→14R

  With this transition, LH enters SC and `Canceled` is assigned to the `state` field of the processor designated by the `succ` variable of LH. Therefore, if Condition (0) is satisfied before the transition, it is also satisfied after the transition.

- 18R→19R

  With this transition, LH leaves SC and `Released` is assigned to the `state` field of the processor designated by the `succ` variable of LH. Therefore, Condition (0) is preserved.

- 16R→1R

  From the proof of Lemma 4, the processor designated by the `succ` variable (P_0) is not included in the lock queue immediately after the transition from 15R to 16R. Since the Condition (0) is assumed to be satisfied before the transition 16R→1R, the `state` field of P_0 is kept to be `Canceled`. Because a new processor is added to the lock queue only with the transition from 2R to 3R/9R (from the proof of Lemma 4), the processor P_0, whose `state` field is kept to be `Canceled`, is not inserted to the lock queue. Consequently, the processor designated by the `succ` variable of another processor in 16R is proved to be not included in the lock queue. Since the processor designated by the `succ` variable of another processor in 14R, 15R, 17R, or 18R is the first one in the lock queue by definition, it is never designated by the `succ` variable of any processor in 16R.

  Suppose the case that more than two processors are in 16R state. Because these processors have made the transition from 15R and because their `succ` variables are not modified for the while, the `succ` variables of each two of them never designate the same processor.

  From the above discussions, the transition 16R→1R does not change the states of the processors designated by the `succ` variables of other processors in SC and preserves Condition (0).

Since `L` does not become `NIL` while LH exists from Lemma 1, `L` is kept non-`NIL` while a processor is in 14R, 15R, 17R, or 18R. Therefore, the processor designated by the `succ` variable of LH is in C state for the while. As a processor assigns `NIL` to `L` with the transition from 15R to 16R, the processor designated by its `succ` variable becomes C' state after the transition. Condition (1) and (2) follow from the above discussion. □

Lemma 6 Following conditions hold under Assumption 3.

(1) The transition 13R→14R (and only the transition) causes the transition 7P→7C (not 7P→7C') on the processor designated by the `succ` variable.

(2) The transition 15R→16R (and only the transition) causes the transition 7C→7C' or 8C→8C' on the processor designated by the `succ` variable.

(3) The transition 16R→1R (and only the transition) causes the transition 7C'→7R or 8C'→8R (not 7C→7R or 8C→8R) on the processor designated by the `succ` variable.

(4) The transition 18R→19R causes (and only the transition) the transition 7C→7R or 8C→8R (not 7C'→7R or 8C'→8R) on the processor designated by the `succ` variable.

Proof: Because the processor designated by the `succ` variable of another processor in 14R is in C state from Lemma 6, the transition 13R→14R causes the transition 7P→7C (not 7P→7C') on the former processor. Since there are no other transitions which change the `state` field from `Preempted` to `Canceled`, Condition (1) is shown to be satisfied.

They are also shown from Lemma 6 that the transition 16R→1R causes a transition from C' state to R state on another processor and that 18R→19R causes a transition from C state to R state. Since there are no other transitions which change the `state` field from `Canceled` to `Released`, Condition (3) and (4) are shown to be satisfied.

Similarly, the transition 15R→16R causes a transition from C state to C' state on the processor designated by the `succ` variable from Lemma 6.

There are two transitions 15R→16R and 10R→1R which make `L` to `NIL`. As a processor making the transition from 10R to 1R is LH before the transition, there are no other processor in 14R, 15R, 17R, or 18R. Therefore, if there are some processors whose `state` fields are `Canceled`, they are proved to be in C' state from Lemma 6. Consequently, the transition 10R→1R does not cause a transition from C state to C' state on another processor, and Condition (2) is proved to be satisfied. □

Lemma 7 The state of the processor linked to the lock queue is included in LQ in Figure 14. The processor whose state is included in LQ is linked to the lock queue.

Proof: In the initial state, the condition is satisfied because `L` is initialized to `NIL` and the state of each processor is 1R. Then, the lemma can be proved by showing that for each transition, if the condition is satisfied before the transition, it is preserved with the transition. We may safely check only the transitions with which a processor enters/leaves LQ or the lock queue is modified.

- 2R→3R, 2R→9R (The processor enters LQ and the lock queue is modified.)

  The processor making one of these transitions is added at the end of the lock queue (from the proof of Lemma 4). Therefore, the condition is preserved.

- 10R→1R (The processor leaves LQ and the lock queue is modified.)

  This transition occurs when only the processor making the transition is included in the lock queue, and the lock queue becomes empty after the transition. Therefore, the condition is preserved.

- 9R→12R, 11R→12R (The processor leaves LQ and the lock queue is modified.)

  The processor making one of these transitions is removed from the lock queue (from the proof of Lemma 4). Therefore, the condition is preserved.

- 7C→7C', 8C→8C' (The processor leaves LQ.)

  These transitions occur only when LH makes the transition from 15R to 16R from Lemma 6 (2). Since the processor making one of these transitions, which is designated by the `succ` variable of LH, is removed from the lock queue, the condition is satisfied after the transition.

- 15R→16R (The lock queue is modified.)

  This transition causes the transition from 7C/8C to 7C'/8C' on the processor designated by the `succ` variable from Lemma 6 (2). This is the same situation with the above.

- 7C→7R, 8C→8R (The processor leaves LQ.)

  These transitions occur only when LH makes the transition from 18R to 19R from Lemma 6 (4). Since the processor making one of these transitions, which is designated by the `succ` variable of LH, is removed from the lock queue, the condition is satisfied after the transitions.

- 18R→19R (The lock queue is modified.)

  This transition causes the transition from 7C/8C to 7R/8R on the processor designated by the `succ` variable from Lemma 6 (4). This is the same situation with the above.

- 7P→7C' (The processor leaves LQ.)

  The only transition which changes the state of another processor from P state to C/C' state is 13R→14R. Because it is shown that the transition 13R→14R changes the state of another processor from P state to C state from Lemma 6 (1), the transition from 7P to 7C' never occurs.

None of the transitions 4L→4R, 5L→5R, 6L→6R, 12R→1R, and 19R→12R actually changes the lock queue from the proof of Lemma 4. □

From this lemma, it is proved that a processor is not included in the lock queue when it returns to 1R, and Assumption 3 can be proved by induction.

To prove deadlock freedom of the algorithm, we assume that each processor makes the next transition in finite duration of time. First, we show that the `next` field is written non-`NIL` value in finite duration of time.

Lemma 8 If a processor included in the lock queue is not the last one in the queue, its `next` field becomes non-`NIL` in finite duration of time under the assumption that each processor makes the next transition in finite duration of time.
Proof: Suppose the case that a processor makes the transition from 2R to 3R and inserts itself at the end of the lock queue. From the assumption, the processor makes the `next` field of its

predecessor designate itself, makes the field non-`NIL` in other words, within finite duration of time after the transition. From the other point of view, the `next` field of the processor which is included in the lock queue but not the last one in the queue becomes non-`NIL` in finite duration of time. □

The deadlock freedom of the algorithm can be derived as the following theorems.

Theorem 9 (Deadlock Freedom (1)) When no processor holds a lock and some processors try to acquire the lock, one of them can acquire the lock within finite duration of time.
Proof: When no processor holds the lock (or is in ER), `L` is `NIL` from Lemma 1. Therefore, the lock queue is empty by definition and there is no processor whose state is in LQ from Lemma 7. Then, all of the processors trying to acquire the lock are in 7C', 8C', 7R, 8R, 1R, or 2R.

A processor in 8C' moves to 8R in finite duration of time because the state 8C' is a result of the transition 15R→16R on another processor and because the transition 16R→1R occurs in finite duration of time on the processor. Similarly, a processor in 7C' moves to 7R or 8C' in finite duration of time.

Therefore, every processor trying to acquire the lock reaches 2R in finite duration of time. The first processor trying the transition from 2R moves to 9R since `L` remains to be `NIL` and succeeds in acquiring the lock. □

Theorem 10 (Deadlock Freedom (2)) A processor trying to release a lock finishes to release the lock within finite duration of time, if the number of interrupt requests raised on other processors during the release operation is bounded.
Proof: There are four loops in the lock releasing routine: 11R→11R, 17R→17R, 12R→13R→12R, and 12R→ ⋯ →19R→12R. This theorem can be proved by showing that a processor trying to release a lock finishes these loops in finite duration of time under the condition that the number that other processors make the transition from 6L to 7P is bounded.

1. 11R→11R, 17R→17R

   A processor finishes these loops in finite duration of time from Lemma 8.

2. 12R→13R→12R

   When LH is in 12R or 13R, `succ→state` never becomes `Released` or `Canceled`. It never becomes `Released` because the processor designated by `succ` is included in the lock queue and is not LH. It never becomes `Canceled` from Lemma 5.

   Consequently, the transition 13R→12R occurs only when `succ→state` is modified from `Preempted` to `Locked` while LH is in 13R. From the assumption that the number of interrupt requests raised on other processors during the release operation is bounded, the number of the transition from 6L to 7P, which is the only transition changing the `state` field to `Preempted`, is bounded, and the execution of this loop is finished in finite duration of time.

3. 12R→ ⋯ →19R→12R

   When LH makes the transition from 18R to 19R, the first processor of the lock queue is removed from the queue. Therefore, the length of the lock queue becomes shorter as the processor executes this loop. From the assumption that the number of interrupt requests raised on other processors during the release operation is bounded, the maximum number of processors which are included in the lock queue when release operation is started and the processors which are inserted to the queue afterwards is bounded. Therefore, the maximum execution number of this loop is bounded. □

Finally, we show the equivalence of the algorithm in Figure 5 and the one in Figure 13. When a processor is in 16R or 18R, `succ->state` is fixed to be `Canceled` from Lemma 5. Therefore, the compare_and_swap operations in the lines marked with ⑯ and ⑱ in Figure 13 are equivalent to simple assignments.

A processor refers to the `state` field of another processor only when the latter processor is designated by a `next` field of LH or other processors in the lock queue. In other words, the `state` field of a processor is referred to only when the processor is included in the lock queue and is not LH and after it makes the `next` field of its predecessor designate itself. In short, it is referred only when the processor is in ⑤, ⑥, ⑦, or ⑧. Consequently, its initial value is never referred to.

References

[1] K. Sakamura, *ITRON Specification ITRON2.* TRON Association, Tokyo, 1990.

[2] H. Takada and K. Sakamura, "ITRON-MP: An adaptive real-time kernel specification for shared-memory multiprocessor systems," *IEEE Micro*, vol. 11, pp. 24–27,78–85, Aug. 1991.

[3] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Preemptive scheduling under time and resource constraints," *IEEE Trans. Computers*, vol. 36, pp. 949–960, Aug. 1987.

[4] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proc. Int'l Conf. Distributed Computing Systems*, pp. 116–123, May 1990.

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

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

[7] G. Graunke and S. Thakkar, "Synchronization algorithms for shared-memory multiprocessors," *IEEE Computer*, vol. 23, pp. 60–69, June 1990.

[8] K. Sato, H. Tsubota, O. Yamamoto, and K. Saitoh, "An experimental implementation of unified real-time operating system," in *Proc. of the Eighth TRON Project Symposium*, pp. 57–68, IEEE CS Press, 1991.

[9] O. Tomisawa, "Recent advances in TRON-spec. chips," in *Proc. of the Eighth TRON Project Symposium*, pp. 178–184, IEEE CS Press, 1991.

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