can be used in multiprocessors with and without coherent caches.

7 Acknowledgments

I would like to thank Tom LeBlanc, Michael Scott and John Mellor-Crummey for valuable discussions and suggestions in earlier drafts of this paper. The material in this paper is based upon work supported by the National Science Foundation under Grant number CDA-8822724.

References

[1] T. E. Anderson. "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors". *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[2] S.R. Biyabani, J.A. Stankovic, and K. Ramamritham. "The Integration of Deadline and Criticalness in Hard Real-Time Scheduling". In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 152–160, 1988.

[3] J. Blazewicz. "Deadline Scheduling of Tasks with Ready Times and Resource Constraints". *Information Processing Letters*, 8(2):60–63, 1979.

[4] J. E. Burns. "Mutual Exclusion with Linear Waiting using Binary Shared Variables". *SIGACT News*, 10(2), Summer 1978.

[5] S.C. Cheng, J.A. Stankovic, and K. Ramamritham. "Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey". *Tutorial: Hard Real-Time Systems*, pages 150–173, 1988.

[6] Motorola Inc. "MVME135, MVME135-1, MVME135A, MVME136 and MVME136A 32-bit Microcomputers User's Manual", 1989.

[7] C.L. Liu and J.W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment". *Journal of the ACM*, 20(1):46–61, 1973.

[8] J. M. Mellor-Crummey and M. L. Scott. "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors". *ACM Transactions on Computer Systems*, 9(1), 1991. (to appear), Earlier version published as TR 342, University of Rochester, Computer Science Department, April 1990.

[9] L.D. Molesky, C. Shen, and G. Zlokapa. "Predictable Synchronization Mechanisms for Multiprocessor Real-Time Systems". *Real-Time Systems*, 3(2), 1990.

[10] R. Rajkumar, L. Sha, and J.P. Lehocsky. "An Experimental Investigation of Synchronization Protocols". In *Proceedings 6th IEEE Workshop on Real-time Operating Systems and Software*, pages 11–17, Pittsburgh, PA, May 1989.

[11] R. Rajkumar, L. Sha, and J. P. Lehoczky. "Real-Time Synchronization Protocols for multiprocessors". In *IEEE Real-Time Systems Symposium*, pages 259–169, 1988.

[12] L. Sha, J. Lehoczky, and R. Rajkumar. "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling". In *Proceedings of the 7th Real-Time Systems Symposium*, pages 181–191, New Orleans, LA, 1986.

[13] J.A. Stankovic. "Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems". *IEEE Computer*, 21(10):10–19, Oct 1988.

```
} // else there are spinning processors
```

- *Processor h is just after processor I in the queue:* In this case there is no need to move h to the end of the queue, because once I is removed from the queue, then h will be the last processor in the queue, and the holder of the lock. The code to do this is:

```
h = highest_priority_spinning_processor() ;
if (I->next == h){
    //the queue is: I -> h -> ...  <- L
    h->previous = nil ; // dequeue I
    // give the lock to processor h
    h->locked = FALSE ;
    return ;
}
```

- *Processor h is neither penultimate, nor first in the queue:* In this case we can move h to the tail of the queue, without any possibility of race condition, because processors join the queue through its head, and alter only the pointers of the processor that is the head of the queue. The code in this case is:

```
// the queue is : I->... ->h->...<-L
p = h->previous ;
if (h->next != nil) {
    // the queue: I->...p->h->...<-L
    // remove h from the Q
    h->previous->next = h->next ;
    h->next->previous = h->previous ;
    // put h at the end of the Q
    h->next = I->next ; h->locked = FALSE ;
    h->previous = nil ;
    return ;
}
```

- *Processor h is the head of the queue:* This case is more difficult than the previous ones, because we have to move h to the end of the queue, while at the same time other processors may join the queue, and change the pointers of the record of h. Again we use the help of the `compare_and_swap` atomic instruction. Define p to be the predecessor of h. We will use the `compare_and_swap` to move p to the head of the queue, and move h to the tail of the queue without any race conditions. We atomically do the following:

  if h is the head of the queue then set p to be the head of the queue

  Now, if the above atomic operation succeeded, then p is the head of the queue, and h can be safely moved without any race conditions. Otherwise, it means that one or more processors have just join the queue, and in this case we can move h to the tail of the queue safely. The pseudocode to do this is:

```
p->next = nil ;
if (compare_and_swap(L, h, p)) {
  // the swap was successful
  // after the compare_and_swap the
  // queue becomes  I->...->p<- L
  // p is now the head of the queue
  //  move h to the tail
  h->next = I->next ; h->previous = nil ;
  h->locked = FALSE ; return ;
} else {
  // some other processor just joined
  // compare and swap did not return true
  repeat while h->next = nil ;  // spin
  h->previous->next = h->next ;
  h->next->previous = h->previous ;
  h->next = I->next ; h->previous = nil ;
  h->locked = FALSE ;
}
```

5 Complexity

We describe the performance of our priority spinlock algorithms in terms of running time, memory requirements, and remote memory references. Both the algorithms have the same asymptotic complexity in terms of running time and remote memory references (or cache invalidations). Suppose that at most P processors want to lock the spinlock i. The time to acquire a lock may be unbounded, because there may always be high priority processors in the systems that request the lock, and some low priority processor may never get the chance to acquire it. The time to release the lock is $O(P)$. The number of remote memory references (or cache invalidations), to acquire a lock is constant, and $O(P)$ references to release it. The memory requirements of the two algorithms are different. If there are P processors in the system and M locks, then the total memory required by the prioritized MCS lock is $O(P+M)$, while for the prioritized Burns's lock is $O(P \cdot M)$.

6 Conclusions

In this paper we defined a low level multiprocessor synchronization mechanism called priority spinlock, and we proposed two algorithms that implement priority spinlocks. Priority spinlocks take into account the timing constraints of the processes that want to acquire the lock, and favor high priority processes. Priority spinlocks can be used in many places in a real-time application. They can be used as described to provide prioritized access to a shared resource, or as a building block for priority binary semaphores, priority counting semaphores, and priority monitors. Our algorithms have predictable implementations because they are based on local spinning. The first can be used in multiprocessors with coherent caches, and the second, albeit more complicated,

```
struct queue_link {
    struct lock_struct * next ;
    Boolean locked ;
}
typedef struct queue_link * qlink ;

local qlink I  ;
    // initialized to point to a queue link record
    // in the local portion of shared memory

procedure acquire_lock (L)
    qlink  predecessor  ;
    I->next = nil ;
    predecessor = fetch_and_store (L, I)
    if (predecessor != nil){   // queue was non-empty
        I->locked = true ;
        predecessor->next = I ;
        repeat while (I->locked == TRUE) ;// spin
    }

procedure release_lock (L)
    if (I->next == nil) { // no known successor
        if compare_and_swap (L, I, nil)
            return ;
            // assuming compare_and_swap
            // returns true iff it swapped
        repeat while (I->next == nil) ; // spin
    }
    I->next->locked = FALSE ;
```

Fig. 3: The MCS list-based queuing lock

```
struct queue_link{
    struct queue_link * next ;
    struct queue_link * previous  ;
    int priority ;
    Boolean  locked ;
}
typedef struct queue_link * qlink  ;

qlink I  ;
  // initialized to point to a queue link record in
  // the local portion of shared memory

procedure acquire_lock (L)
{
    qlink  predecessor  ;
    I->next = nil ;    // initialize the fields
    I->previous = nil ;// of the record
    I->priority = my_priority ;
    predecessor = fetch_and_store (L, I) ; // join
    if (predecessor != nil) { //queue was non-empty
        I->locked = TRUE ;   //the lock is locked
        I->previous = predecessor ;//join the queue
        predecessor->next = I ;
        repeat while (I->locked == TRUE)  ;  // spin
    }
}
```

Fig. 4: Priority Spinlock with local memory spinning

Data structures: We use a global queue of processors that spin for the lock. The queue is implemented as a doubly linked list. Each processor has a record in the queue that contains some information about the processor. The lock is a pointer to the head of the queue. If the queue is empty, then the lock is free. If the queue is not empty, then the holder of the lock is the tail of the queue. The global definitions for the extended MCS lock are in Fig. 4.

Acquire lock: The `acquire_lock` operation is not changed. If a processor wants to acquire the lock it joins the lock queue atomically, using the help of the `fetch_and_store` atomic instruction. If it finds out that it is the tail of the queue, then it continues, knowing that it has the lock. Otherwise, it spins on a local variable, waiting its turn to become the tail of the queue and the holder of the lock. The code for the `acquire_lock` operation is in Fig. 4.

Release lock: The processor that releases the lock searches the queue to find the highest priority spinning processor and gives it the lock. The action of giving the lock from processor A to processor B is done by moving processor B to the end of the queue, and removing processor A from the queue. This is not a trivial task to do, because at the time processor B is being moved to the end of the queue, other processors may join the queue, and race conditions may happen. Note that we can not use a simple (not priority) lock to protect the queue, because this lock may introduce priority inversion that may violate the priority ordering property. During the `release_lock` operation there are many cases to be considered. Depending on where in the queue is the highest priority processor, we have to execute different code to move it to the end of the queue.

Define h to be the highest priority processor in the queue. Define I to be the processor that releases the lock. The different cases to be considered are:

- *I is the only processor in the queue:* In this case I will release the lock. However, it has to make sure that no processor joined the queue after I found that it is the only processor in the queue, and before releasing the lock. In the case where some processor did join the queue between the two events, then processor I should give the lock to this processor[7].

```
if (I->next == nil) { // no one in the queue
    if compare_and_swap (L, I, nil) return ;
    // compare_and_swap returns true iff swapped
    // some processor just joined the queue
    repeat while I->next = nil  ;  // spin
    // give the lock to the first processor
    I->next->locked = false ;
    return ;
```

[7]Note that more than one processors, may have joined the queue in the meantime. The lock is given to the first of them without considering priorities. The highest priority processor will acquire the lock the next time, which is consistent with our definition of priority spinlocks.

```
boolean array want[0..P-1] ;
integer array priority[0..P-1] ;
initially for all k: want[k] = FALSE;
                     priority[k] = 0 ;
local int my_priority ;
local int i ; // processor's id

procedure acquire_lock(L)
   // i is the identity  of the processor
   priority[i] = my_priority ;
   want[i] = TRUE ;
   while ((want[i] == TRUE) AND
          (L == locked OR
           test_and_set(L) == locked))  ;
        // while the spinlock is held spin

procedure release_spinlock(L)
   want[i] = FALSE ;
   j = find_max_priority() ;
   // find the highest priority spinning processor
   if (j == -1) {//if no processor spins
     L = free ;  // clear the spinlock
   }
   else { // if there is a spinning processor
     // allow it to continue, without releasing
     // the spinlock;   the spinlock is inherited
     // by  processor j from processor i
     want[j] = FALSE ;
   }
```

Fig. 2: Priority spinlock with local cache spinning

configurations with hundreds of processors. The algorithm we describe in this section works equally well on multiprocessors with coherent caches or with local memories. The priority spinlock algorithm we propose is a modified version of the MCS lock [8]. Our algorithm and the MCS lock spin on local memory only[4], require a small constant amount of space per lock, and work equally well on machines with and without coherent caches. Moreover our algorithm guarantees the *Priority ordering* property.

4.1 The MCS lock

The MCS lock uses a queue of processors for each lock. Processors that want to acquire the lock join the queue. The order in which processors join the queue is the order in which they acquire the lock. The lock is nothing more than a pointer to the head of the queue. The tail of the queue represents the processor that has the lock.

Acquire lock: If processor `proc` wants to acquire the lock, it joins the queue by executing a `fetch_and_store`[5] atomic instruction. If `proc` is the tail of the queue (no predecessor), then it continues (because the tail of the queue is the holder of the lock), else it busy-waits by spinning on a local variable. The pseudocode to acquire the lock is:

```
predecessor = fetch_and_store(lock, my_record) ;
// my_record now is a pointer to the head
// of the queue, and the lock pointer now
// points to my_record
if  (predecessor == nil)//I am the holder
    return ;
else
    // spin on some local variable
```

Release lock: When processor `proc` wants to release the lock then:

- If the lock queue is not empty, it gives the lock to its successor, by changing the value of its successor's spinning variable.
- If the queue is empty, then the code is more complicated. If processor `proc` releases the lock after it verifies that the queue is empty, then there is the possibility that in the meantime (after verifying, but before releasing) some other processor will join the queue and will stay there forever. In order to avoid this deadlock situation the releasing processor (`proc`) uses the atomic instruction `compare_and_swap`[6]: Processor `proc` checks if it is the head of the queue, and if yes it dequeues itself. The check and dequeuing are done atomically with the help of the atomic `compare_and_swap` instruction. If the atomic instruction doesn't succeed, it means that some other processor joined the queue. In this case processor `proc` gives the lock to the processor that just joined the queue. If more than one joined it, the first one gets the lock.

The pseudocode for the algorithm is in Fig. 3.

4.2 Priority lock

Our priority spinlock is very similar to the MCS lock. The basic difference is that we dequeue the processors from the lock queue in a priority order, because we want to implement priority spinlocks that have the priority ordering property. Dequeuing the processors from the queue in priority order is not trivial because the queue can be accessed by any processor that wants to join it, and we need to keep it in consistent state. Of course, we can not use locks for the enqueue and dequeue operations, because these locks can only be simple (not priority) spinlocks, and they may violate the priority ordering property. So, we should protect the queue using only atomic instructions.

[4] More precisely, on statically allocated processor specific memory, which will be local to any machine in which shared memory is distributed or coherently cached.

[5] `fetch_and_store(x,y)` atomically sets `x` to `y` and returns the old value of `x`

[6] `compare_and_swap(x,y,z)` atomically compares `x` and `y` and if equal, sets `x` to `z` and returns `TRUE`, else it returns `FALSE`.

3 First algorithm

This section presents an algorithm for acquiring and releasing priority spinlocks that can be efficiently implemented on cache-coherent multiprocessors.

3.1 Burns's algorithm

Our algorithm is a modification of Burns's algorithm [4], that guarantees mutual exclusion and *linear waiting*. Linear waiting means that if a processor tries to enter the critical region, it will enter it before any other processor has entered the critical region more than once.

Burns's algorithm uses a shared global array `want[0..P-1]`, where `P` is the number of processors in the system. If processor `i` wants to enter the critical region, it sets `want[i]` to `TRUE`. Then, it starts spinning executing the `test_and_set` atomic instruction and checking the value of `want[i]`.

A processor `i` that exits the critical region searches the array `want` sequentially, finds the first spinning processor with index larger than the index of processor `i` (modulo the number of processors), and allows it to enter the critical region. Burns's algorithm can be found in Fig. 1.

```
local int i ; // the id of this processor
boolean array want[0..P-1] ;
initially for all k: want[k] = FALSE;

acquire_lock(l)
{
   want[i] = TRUE;//register that I want the lock
   while ((want[i] == TRUE)  AND
     (test_and_set(l) == locked)) ;// spin
}
release_lock(l)
{
   int j ;
   want[i] = FALSE ; // release the lock
   // get the next processor in line
   j = (i+1) % P ;
   while ((want[j] == FALSE)  AND (j != i))
     // while no processor wants the lock
     //  go to the next one
     j = (j+1) % P ;
   if (i == j)  // if none wants the lock
     clear(l) ;  // release it
   else // hand  processor j  the lock
     want[j] = FALSE ;
}
```

Fig. 1: Burns's FIFO lock

3.2 Adding priorities to Burns's lock

Our first algorithm is a straightforward modification of Burns's algorithm and achieves the following properties:

1. **Priority ordering** : The processor that releases the lock sequentially scans a shared array that contains the identity of all the processors that currently spin for this lock, and finds the highest priority spinning processor. Then, it gives the lock to this processor.

2. **Local spinning** : In Burns's algorithm processors continually execute the `test_and_set` atomic instruction and poll a (potentially) remote memory location. We modify the algorithm in such a way that if implemented on a multiprocessor with *coherent caches*, then processors spin only on local cache copies. The spinning code in Burns's algorithm is:

   ```
   while((want[i] == TRUE)  AND
         (test_and_set(l) == locked));
   ```

   Our spinning code is:

   ```
   while ((want[i] == TRUE) AND
          (l == locked OR
           test_and_set(l) == locked));
   ```

   Note that if the value of lock `l` is coherently cached, then all the processors will spin on local copies for as long as the lock is held. Moreover, we free the lock in such a way, that when the lock is given from one processor to another, the spinning processors do not cache miss, and so they do not generate any bus or switch traffic.

The code of our algorithm is in Fig. 2.

3.2.1 Description

For each spinlock there is an array `want[0..P-1]` that represents the desire of each processor to use the spinlock.

Acquire lock: If processor `i` wants to acquire the lock then it sets `want[i]` to `TRUE`, and registers its priority (found in local variable `my_priority`) in the shared array `priority[0..P-1]`. Then, it starts spinning waiting for the lock.

Release lock: When the spinlock is released, the array `want` is sequentially scanned to find the highest priority processor currently spinning to give it the lock. The spinlock is given from processor `i` to processor `j` by negating the spinning variable `want[j]` on which processor `j` spins.

4 Second algorithm

The efficient implementation of the previous algorithm requires a multiprocessor with coherent caches. Multiprocessors with coherent caches do not scale to large numbers of processors due to bus bandwidth limitations. Multiprocessors without coherent caches are widespread and come in

For our proposed algorithms $k = 1$. This means that if more than one processor attempts to acquire the lock, then the highest priority processor acquires it, unless the highest priority processor attempts to acquire the lock *after* it has been decided that the lock should be given to some other lower priority processor.

Our assumptions about the execution model are:

- Processors communicate via a medium (e.g. bus, switch) that is either prioritized, or has bounded access time. This means that a high priority processor can not be prevented from using the medium indefinitely. If the communication medium does not have bounded access time, there is no way to built a predictable system on top of it [9]. Buses that provide bounded access time include synchronous buses, and asynchronous buses configured with Round-Robin access, like the VME bus [6].
- Processes that request, hold, or release locks are not preemptable during the time they request, hold or release the lock[1].

In this paper we describe two algorithms that implement priority spinlocks. The algorithms are modifications of previous algorithms for FIFO spinlocks by Burns [4] and Mellor-Crummey and Scott [8]. The major properties we add to Burns's algorithm are *priority ordering* of the spinlock and *local spinning*[2]. The major property we add to the MCS lock is priority ordering.

The next section summarizes previous work on hard real-time multiprocessor synchronization primitives, and how it differs from our work. Section three describes our first priority spinlock algorithm that can be efficiently implemented on UMA (Uniform Memory Access) multiprocessors. Section four describes our second priority spinlock algorithm that can be efficiently implemented both in UMA and NUMA (Non Uniform Memory Access) multiprocessors. Section five compares the complexity of the two algorithms in terms of running time, memory requirements, and remote memory references. Finally, section six presents our conclusions.

2 Previous work

Synchronization in multiprocessor hard real-time systems is a relatively new field. Molesky, Shen and Zlokapa [9], describe predictable algorithms for semaphores with linear waiting. Although their proposed algorithms are predictable, they do not take into account the priorities of the processes that want to acquire the semaphore.

Rajkumar, Sha and Lehoczky [11] presented a multiprocessor extension of the priority ceiling protocol [10]. The priority ceiling protocol is a protocol that minimizes priority inversion for a set of periodic real-time processes that access exclusively some shared data. The multiprocessor priority ceiling protocol generalizes the uniprocessor priority ceiling protocol by executing all the critical regions associated with a semaphore on a particular processor (called *synchronization processor*). So, the critical regions in the programs are substituted by an invocation to a remote server that deals with all the critical regions associated with a particular semaphore. Remote invocation is an attractive alternative to mutual exclusion algorithms, especially in multiprocessors where the cost of accessing non-local memory is very high. However, the existence of a remote centralized server limits the scalability of the solution, and increases the cost of executing fine grain sharing applications.

Anderson [1], and Mellor-Crummey and Scott[8] derived spinlock implementations that service lock requests in FIFO order and can be used in real-time systems that just want to bound (not minimize) the priority inversion.

Our work focuses on how to built prioritized low-level synchronization primitives. In this respect, it is mostly related to [9], where they describe how to build predictable FIFO low-level synchronization primitives for real-time multiprocessor systems.

2.1 Our approach

Mutual exclusion algorithms for multiprocessors exist for quite a while now. All these algorithms focus on providing mutual exclusion, along with one or more *desired* properties that a multiprocessor mutual exclusion mechanism should have, such as *starvation* avoidance, *fairness*, FIFO service of lock requests, *local spinning*, etc. However, none of the previous multiprocessor mutual exclusion algorithms has considered properties such as *timeliness* or *priority ordering*, which are vital for real-time systems. So, previous mutual exclusion mechanisms service requests for the lock independently of the timing constraints of the processes that requests the lock. In this report we focus on mutual exclusion algorithms that consider the timing constraints of the processes that request it. The timing constraints should be expressed in the form of priorities[3], so that the mutual exclusion implementation, will be able to service the requests for the lock in priority order.

[1] This assumption is not as restrictive as it sounds, because spinlocks are usually used by the kernel to build high level synchronization primitives. So, the time that a processor holds a spinlock is very small and known beforehand. In addition, non-preemptability of processes that execute spinlock synchronization code enables us to focus on the aspects related to synchronization only and not to preemptability

[2] A spinlock implementation has the property of local spinning if each processor spins on a local memory location, or on a local cache copy, without generating any remote memory references or bus accesses. Remote memory accesses can easily saturate the bus (or switch) and slow down the whole system. Local spinning is essential for the efficient and predictable behavior of real-time systems.

[3] The priorities may represent deadlines, or periods, or some other form of timing constraints.

Multiprocessor Synchronization Primitives with Priorities

Evangelos P. Markatos
markatos@cs.rochester.edu
University of Rochester
Computer Science Department
Rochester, New York 14627

Abstract

Low-level multiprocessor synchronization primitives, such as spinlocks, are usually designed with little or no consideration about timing constraints, which makes them inappropriate for real-time systems. In this paper, we propose a new synchronization mechanism, the *priority spinlock*, that takes into account the priorities of the processes that want to acquire it, and favors high priority processes. We define what a priority spinlock is, and propose two algorithms to implement priority spinlocks with local spinning. Priority spinlocks can be used to provide prioritized *mutually exclusive* access to shared resources in real-time multiprocessor systems. They can also be used as building blocks for higher level priority synchronization primitives, such as priority semaphores.

1 Introduction

Real-time systems have timing constraints that must be met, otherwise catastrophic effects may happen [13]. Timing constraints usually take the form of deadlines, earliest starting times, or value functions. Schedulers translate these timing constraints along with recourse requirements and criticalities into priorities [2, 3, 5, 7, 12], and assign the resources of the system to higher priority processes. These priorities must be preserved throughout the system, otherwise the correctness of a real-time system can not be demonstrated. If there is a server where priorities are not preserved, then a high priority process may be delayed (possibly indefinitely) by a low priority process. This event is called priority inversion, and must be avoided, bounded, or minimized, otherwise the feasibility of the schedule and the correctness of the scheduler implementation are questionable. Our work tries to minimize the priority inversion that can happen in the implementation of the most fundamental synchronization mechanism of a multiprocessor real-time system: the spinlock. Spinlocks are low-level synchronization mechanisms for multiprocessor systems that are used to provide mutual exclusion. They are usually implemented with the help of atomic instructions like `test_and_set` that most multiprocessors provide. However, spinlock implementations are usually proposed with little or no consideration of timing constraints. So, current implementations service requests for the lock randomly, or at best FIFO. Random service of requests, are inappropriate for real-time systems; FIFO service is appropriate in real-time applications, because it bounds the worst case time where a processor has to wait to acquire the spinlock. However, FIFO implementations may force a high priority process that wants to acquire a spinlock to wait for all the low priority processes that happen to have requested the spinlock before it. This potentially long wait leads to very conservative scheduling, because although the worst case priority inversion is bounded, the bound is very large.

In this paper we propose a new version of spinlocks called *priority spinlocks*. Priority spinlocks take into account the priority of the process that wants to acquire the lock, and assign the lock to the highest priority process that has requested the lock at the time the decision is being made.

1.1 Priority spinlocks

Priority spinlocks are defined as follows:

1. No more than one processor may possess a lock at one time.

2. Each processor that competes for a priority spinlock has a *dynamic priority* that reflects the importance of the process it runs. This priority may change over time. All processors have different priorities.

3. Priority spinlocks have the **priority ordering** property.

   > A spinlock implementation has the priority ordering property iff there is a constant k, such that if a high priority processor requests a lock, then lower priority processors will acquire the lock at most k times before the high priority processor acquires it.