A Prioritized Multiprocessor Spin Lock

Theodore Johnson Krishna Harathi
Dept. of Computer and Information Science
University of Florida

Abstract

In this paper, we present the PR-lock, a prioritized spin-lock mutual exclusion algorithm for real time systems. The PR-lock is a contention-free spin lock, in which blocked processes spin on locally stored or cached variables. In contrast to previous work on prioritized spin locks, our algorithm maintains a pointer to the lock holder. As a result, our spin lock can support priority inheritance protocols. In addition, all work to maintain a priority queue is done while a process acquires a lock, when it is blocked anyway. Releasing a lock is a constant time operation. We present simulation results that demonstrate the prioritized acquisition of locks, and compare the performance of the PR-lock against that of an alternative prioritized spin lock.

Keywords: mutual exclusion, parallel processing, real time system, spin-lock, priority queue.

1 Introduction

Mutual exclusion is a fundamental synchronization primitive for exclusive access to critical sections or shared resources on multiprocessors [17]. The spin-lock is one of the mechanisms that can be used to provide mutual exclusion on shared memory multiprocessors [2]. A spin-lock usually is implemented using atomic read-modify-write instructions such as `Test&Set` or `Compare&Swap`, which are available on most shared-memory multiprocessors [16]. Busy waiting is effective when the critical section is small and the processor resources are not needed by other processes in the interim. However, a spin-lock is usually not fair, and a naive implementation can severely limit performance due to network and memory contention [1, 11]. A careful design can avoid contention by requiring processes to spin on locally stored or cached variables [19].

In real time systems, each process has timing constraints and is associated with a priority indicating the urgency of that process [26]. This priority is used by the operating system to order the rendering of services among competing processes. Normally, the higher the priority of a process, the faster it's request for services gets honored. When the synchronization primitives disregard the priorities, lower priority processes may block the execution of a process with a higher priority and a stricter timing constraint [24, 23]. This *priority inversion* may cause the higher priority process to miss its deadline, leading to a failure of the real time system. Most of the work done in synchronization is not based on priorities, and thus is not suitable for real time systems. Furthermore, general purpose parallel processing systems often have processes that are "more important" than others (kernel processes, processes that hold many locks, etc.). The performance of such systems will benefit from prioritized access to critical sections.

In this paper, we present a prioritized spin-lock algorithm, the *PR-lock*. The PR-lock algorithm is suitable for use in systems which either use static-priority schedulers, or use dynamic-priority schedulers in which the relative priorities of existing tasks do not change while blocked (such as Earliest Deadline First [26] or Minimum Laxity [15]). The PR-lock is a contention-free lock [19], so its use will not create excessive network or memory contention. The PR-lock maintains a queue of records, with one record for each process that has requested but not yet released the lock. The queue is maintained in sorted order (except for the head record) by the acquire lock operations, and the release lock operation is performed in constant time. As a result, the queue order is maintained by processes that are blocked anyway, and a high priority task does not perform work for a low priority task when it releases the lock. The lock keeps a pointer to the record of the lock holder, which aids in the implementation of priority inheritance protocols [24, 23]. A task's lock request and release are performed at well-defined points in time, which makes the lock predictable. We present a correctness proof, and simulation results which demonstrate the prioritized lock access, the locality of the references, and the improvement over a previously proposed prioritized spin lock.

We organize this paper as follows. In Section 1.1 we describe previous work in this area and in Section 2, we present our algorithm. In Section 3 we argue the correctness of our algorithm. In Section 4 we discuss an extension to the algorithm presented in Section 2. In Section 5 we show the simulation results which compare the performance of the PR-lock against that of other similar algorithms. In Section 6 we conclude this paper by suggesting some applications and future extensions to the PR-lock algorithm.

1.1 Previous Work

Our PR-lock algorithm is based on the MCS-lock algorithm, which is a spin-lock mutual exclusion algorithm for shared-memory multiprocessors [19]. The MCS-lock grants lock requests in FIFO order, and blocked processes spin on locally accessible flag variables only, avoiding the contention usually associated with busy-waiting in multiprocessors [1, 11]. Each process has a record that represents its place in the lock queue. The MCS-lock algorithm maintains a pointer to the tail of the lock queue. A process adds itself to the queue by swapping the current contents of the tail pointer for the address of its record. If the previous tail was `nil`, the process acquired the lock. Otherwise, the process inserts a pointer to its record in the record of the previous tail, and spins on a flag in its record. The head of the queue is the record of the lock holder. The lock holder releases the lock by reseting the flag of its successor record. If no successor exists, the lock holder sets the tail pointer to `nil` using a Compare&Swap instruction.

Molesky, Shen, and Zlokapa [20] describe a prioritized spin lock that uses the test-and-set instruction. Their algorithm is based on Burn's fair test-and-set mutual exclusion algorithm [5]. However, this lock is not contention-free.

Markatos and LeBlanc [18] presents a prioritized spin-lock algorithm based on the MCS-lock algorithm. Their acquire lock algorithm is almost the same as the MCS acquire lock algorithm, with the exception that Markatos' algorithm maintains a doubly linked list. When the lock holder releases the lock, it searches for the highest priority process in the queue. This process' record is moved to the head of the queue, and its flag is reset. However, the point at which a task requests or releases a lock is not well defined, and the lock holder might release the lock to a low priority task even though a higher priority task has entered the queue. In addition, the work of maintaining the priority queue is performed when a lock is released. This choice makes the time to release a lock unpredictable, and significantly increases the time to acquire or release a lock (as is shown in section 5). Craig [10] proposes a modification to the MCS lock and to Markatos' lock that substitutes an atomic Swap for the Compare&Swap instruction, and permits nested locks using only one lock record per process.

Goscinski [12] develops two algorithms for mutual exclusion for real time distributed systems. The algorithms are based on token passing. A process requests the critical section by broadcasting its intention to all other processes in the system. One algorithm grants the token based on the priorities of the processes, whereas the other algorithm grants the token to processes based on the remaining time to run the processes. The holder of the token enters the critical section.

The utility of prioritized locks is demonstrated by rate monotonic scheduling theory [9, 24]. Suppose there are N periodic processes $T_1, T_2, T_3, \ldots, T_N$ on a uniprocessor. Let E_i and C_i represent the execution time and the cycle time (periodicity) of the process T_i. We assume that $C_1 \leq C_2 \leq \ldots \leq C_N$. Under the assumption that there is no blocking, [9] show that if for each j

$$\sum_{i=1}^{j} E_i/C_i \leq \quad j\,(2^{1/j}-1)$$

Then all processes can meet their deadlines.

Suppose that B_j is the worst case blocking time that process T_j will incur. Then [24] show that all tasks can meet their deadlines if

$$\sum_{i=1}^{j} E_i/C_i + B_j/C_j \leq \quad j\,(2^{1/j}-1)$$

Thus, the blocking of a high priority process by a lower priority process has a significant impact on the ability of tasks to meet their deadlines. Much work has been done to bound the blocking due to lower priority processes. For example, the Priority Ceiling protocol [24] guarantees that a high priority process is blocked by a lower priority process for the duration of at most one critical section. The Priority Ceiling protocol has been extended to handle dynamic-priority schedulers [7] and multiprocessors [23, 8].

Our contribution over previous work in developing prioritized contention-free spin locks ([18] and [10]) is to more directly implement the desired priority queue. Our algorithm maintains a pointer to the head of the lock queue, which is the record of the lock holder. As a result, the PR-lock can be used to implement priority inheritance [24, 23]. The work of maintaining priority ordering is performed in the acquire lock operation, when a task is blocked anyway. The time required to release a lock is small and predictable, which reduces the length and the variance of the time spent in the critical section. The PR-lock has well-defined points in time in which a task joins the lock queue and releases its lock. As a result, we can guarantee that the highest priority waiting task always receives the lock. Finally, we provide a proof of correctness.

2 PR-lock Algorithm

Our PR-lock algorithm is similar to the MCS-lock algorithm in that both maintain queues of blocked processes using the Compare&Swap instruction. However, while the MCS-lock and Markatos' lock maintain a global pointer to the tail of the queue, the PR-lock algorithm maintains a global pointer to the head of the queue. In both the MCS-lock and the Markatos' lock, the processes are queued in FIFO order, whereas in the PR-lock, the queue is maintained in priority order of the processes.

2.1 Assumptions

We make the following assumptions about the computing environment:

1. The underlying multiprocessor architecture supports an atomic Compare&Swap instruction. We note that many parallel architectures support this instruction, or a related instruction [13, 21, 3, 28].

2. The multiprocessor has shared memory with coherent caches, or has locally-stored but globally-accessible shared memory.

3. Each processor has a record to place in the queue for each lock. In a NUMA architecture, this record is allocated in the local, but globally accessible, memory. This record is not used for any other purpose for the lifetime of the queue. In Section 4, we allow the record to be used among many lock queues.

4. The higher the actual number assigned for priority, the higher the priority of a process (we can also assume the opposite).

5. The relative priorities of blocked processes do not change. Acceptable priority assignment algorithms include Earliest Deadline First and Minimum Laxity.

It should be noted that each process p_i participating in the synchronization can be associated with an unique processor P_i. We expect that the queued processes will not be preempted, though this is not a requirement for correctness.

2.2 Implementation

The PR-lock algorithm consists of two operations. The **acquire_lock** operation acquires a designated lock and the **release_lock** operation releases the lock. Each process uses the **acquire_lock** and **release_lock** operations to synchronize access to a resource:

```
acquire_lock(L, r)
        critical section
release_lock(L)
```

The following sub-sections present the required version of Compare&Swap, the needed data structures, and the **acquire_lock** and **release_lock** procedures.

2.2.1 The Compare&Swap

The PR-lock algorithms make use of the Compare&Swap instruction, the code for which is shown in Figure 1. Compare&Swap is often used on pointers to object records, where a *record* refers to the physical memory space and an *object* refers to the data within a record. `Current` is a pointer to a record, `Old` is a previously sampled value of `Current`, and `New` is a pointer to a record that we would like to substitute for `*Old` (the record pointed to by `Old`). We compute the record `*New` based on the object in `*Old` (or decide to perform the swap based on the object in `*Old`), so we want to set `Current` equal to `New` only if `Current` still points to the record `*Old`. However, even if `Current` points to `*Old`, it might point to a different object than the one originally read. This will occur if `*Old` is removed from the data structure, then re-inserted as `Current` with a new object. This sequence of events cannot be detected by the Compare&Swap and is known as the A-B-A problem.

Following the work of Prakash et al. [22] and Turek et al. [27], we make use of a double-word Compare&Swap instruction [21] to avoid this problem. A counter is appended to `Current` which is treated as a part of `Current`. Thus `Current` consists of two parts: the *value* part of `Current` and the *counter* part of `Current`. This counter is incremented every time a modification is made to `*Current`. Now all the variables

```
Procedure CAS(structure pointer *Current, *Old, *New)
/* Assume CAS operates on double words */
atomic{
        if( *Current == *Old ) {
                *Current = *New;
                return(TRUE);
        }
        else {
                *Old = *Current;
                return(FALSE);
        }
}
```

Figure 1: CAS used in the PR-lock Algorithm

`Current`, `Old` , and `New` are twice their original size. This approach reduces the probability of occurrence of the A-B-A problem to acceptable levels for practical applications. If a double-word Compare&Swap is not available, the address and counter can be packed into 32 bits by restricting the possible address range of the lock records.

We use a version of the Compare&Swap operation in which the current value of the target location is returned in *old*, if the Compare&Swap fails. The semantics of the Compare&Swap used is given in Figure 1. A version of the Compare&Swap instruction that returns only `TRUE` or `FALSE` can be used by performing an additional read.

2.2.2 Data Structures

The basic data structure used in the PR-lock algorithm is a priority queue. The lock L contains a pointer to the first record of the queue. The first record of the queue belongs to the process currently using the lock. If there is no such process, then L contains *nil*.

Each process has a locally-stored but globally-accessible record to insert into the lock queue. If process p inserts record q into the queue, we say that q is p's record and p is q's process. The record contains the process priority, the next-record pointer, a boolean flag `Locked` on which the process owning the element busy-waits if the lock is not free, and an additional field `Data` that can be used to store application-dependent information about the lock holder.

The next-record pointer is a double sized variable: one half is the actual pointer and the other half is a counter to avoid the A-B-A problem. The counter portion of the pointer itself has into two parts: one bit of the counter called the `Dq` bit is used to indicate whether the queuing element is in the queue. The rest of the bits are used as the actual counter. This technique is similar to the one used by Prakash et al. [22] and Turek et al. [27]. Their counter refers to the record referenced by the pointer. In our algorithm, the counter refers to the record that contains the pointer, not the record that is pointed to.

If the `Dq` bit of a record q is FALSE, then the record is in the queue for a lock L. If the Dq bit is TRUE, then the record is probably not in the queue (for a short period of time, the record might be in the queue with its `Dq` bit set TRUE). The `Dq` bit lets the PR-lock avoid garbage accesses.

Each process keeps the address of its record in a local variable (`Self`). In addition, each process requires two local pointer variables to hold the previous and the next queue element for navigating the queue during the enqueue operation (`Prev_Node` and `Next_Node`).

The data structures used are shown in Figure 2. The `Dq` bit of the `Pointer` field is initialized to TRUE, and the `Ctr` field is initialized to 0 before the record is first used.

A typical queue formed by the PR-lock algorithm is shown in Figure 3 below. Here L points to the record q_0 of the current process holding the lock. The record q_0 has a pointer to the record q_1 of the next process having the highest priority among the processes waiting to acquire the lock L. Record q_1 points to record q_2 of the next higher priority waiting process and so on. The record q_n belongs to the process with the least priority among waiting processes.

2.2.3 Acquire_Lock Operation

The **acquire_lock** operation is called by a process $\hat{p}$ before using the critical section or resource guarded by lock L. The parameters of the **acquire_lock** operation are the lock pointer L and the record $\hat{q}$ of the process (passed to local variable `Self`).

An **acquire_lock** operation searches for the correct position to insert $\hat{q}$ into the queue using `Prev_Node` and `Next_Node` to keep track of the current position. In Figure 4, `Prev_Node` and `Next_Node` are abbreviated to P and N. The records pointed by P and N are q_i and q_{i+1}, belonging to processes p_i and p_{i+1}. Process $\hat{p}$ positions itself so that $Pr(p_i) \geq Pr(\hat{p}) > Pr(p_{i+1})$, where Pr is a function which maps a process to its priority. Once such a position is found, $\hat{q}$ is prepared for insertion by making $\hat{q}$ point to q_{i+1}. Then, the insertion is committed by making q_i to point to $\hat{q}$ by using the Compare&Swap instruction. The various stages and final result are shown in Figure 4.

The **acquire_lock** algorithm is given in Figure 5. Before the **acquire_lock** procedure is called, the `Data` and the `Priority` fields of the process' record are initialized appropriately. In addition, the `Dq` bit of the `Next` pointer is implicitly TRUE.

The **acquire_lock** operation begins by assuming that the lock is currently free (the lock pointer `L` is

```
structure Pointer {
        structure Object *Ptr;
        int31 Ctr;
        boolean Dq;
}

structure Record {
        structure structure_of_data Data;
        boolean Locked;
        integer Priority;
        structure Pointer Next;
}
```

Shared Variable

```
        structure Pointer L;
```

Private Variables

```
        structure Pointer Self, Prev_Node, Next_node;
        boolean Success, Failure;
        constant TRUE, FALSE, NULL, MAX_PRIORITY;
```

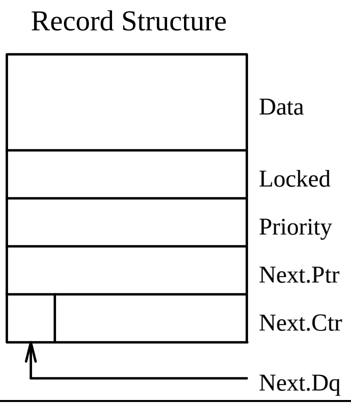

Figure 2: Data Structures used in the PR-lock Algorithm

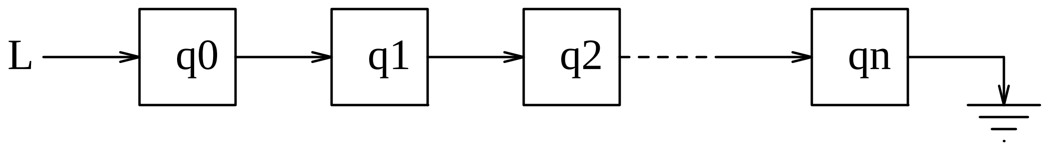

Figure 3: Queue data structure used in PR-lock algorithm

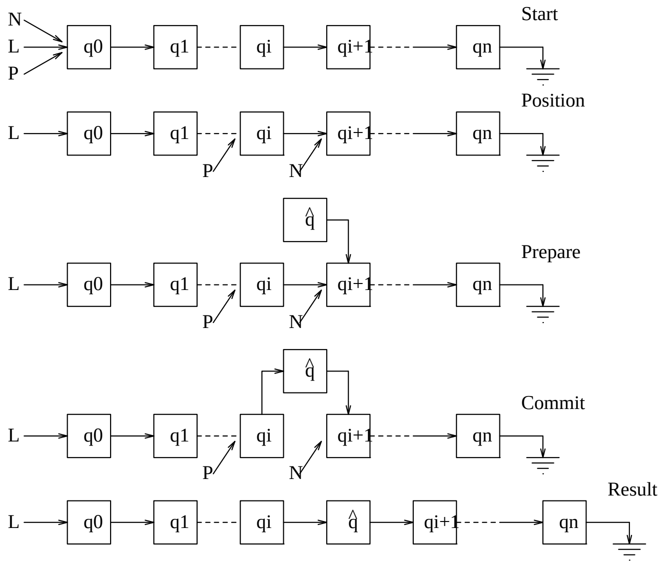

Figure 4: Stages in the acquire_lock operation

`null`). It attempts to change `L` to point to its own record with the Compare&Swap instruction. If the Compare&Swap is successful, the lock is indeed free, so the process acquires the lock without busy-waiting. In the context of the composite pointer structures that the algorithm uses, a `NULL` pointer is all zeros.

If the swap is unsuccessful, then the acquiring process traverses the queue to position itself between a higher or equal priority process record and a lower priority process record. Once such a junction is found, `Prev_Node` will point to the record of the higher priority process and `Next_Node` will point to the record of the lower priority process. The process first sets its link to `Next_Node`. Then, it attempts to change the previous record's link to its own record by the atomic Compare&Swap.

If successful, the process sets the `Dq` flag in its record to FALSE indicating its presence in the queue. The process then busy-waits until its `Locked` bit is set to FALSE, indicating that it has been admitted to the critical section.

There are three cases for an unsuccessful attempt at entering the queue. Problems are detected by examining the returned value of the failed Compare&Swap marked as **F** in the algorithm. Note that the returned value is in the `Next_Node`. In addition, a process might detect that it has misnavigated while searching the queue. When we read `Next_Node`, the contents of the record pointed to by `Prev_Node` are fixed because the record's counter is read into `Next_Node`.

1. A concurrent **acquire_lock** operation may overtake the **acquire_lock** operation and insert its own

```
Procedure acquire_lock(L, Self) {
   Success=FALSE;
   do {
      Prev_Node=NULL; Next_Node=NULL
      if(CAS(&L, &Next_Node, &Self)) { /* No Lock */
         Success=TRUE; Failure=FALSE; /* Use Lock */
         Self.Ptr->Next.Dq=FALSE;
      }
      else { /* Lock in Use */
         Failure=FALSE; Self.Ptr->Locked=TRUE;
         do {
            Prev_Node=Next_Node;
            Next_Node=Prev_Node.Ptr->Next;
            if((Next_Node.Dq==TRUE) /* Deque, Try Again */                          ii
                  or (Prev_Node.Ptr->Priority<Self.Ptr->Priority)) {                iii
               Failure=TRUE;
            }
            else {
               if(Next_Node.Ptr==NULL or (Next_Node.Ptr!=NULL and
                        Next_Node.Ptr->Priority<Self.Ptr->Priority)){
                  Self.Ptr->Next.Ptr=Next_Node.Ptr
                  if(CAS(&(Prev_Node.Ptr->Next), &Next_Node, &Self)) {              F
                     Self.Ptr->Next.Dq=FALSE;
                     while(Self.Ptr->Locked); /* Busy Wait */
                     Success=TRUE; /* Then, use lock */
                  }
                  else {
                     if((Next_Node.Dq==TRUE) /* Deque, Try Again */                 ii
                           or Prev_Node.Ptr->Priority < Self.Ptr->Priority)) {      iii
                        Failure=TRUE;
                     }
                     else
                        Next_Node=Prev_Node;                                        i
                  }
               }
            }
         }while(!Success and !Failure);
      }
   }while(!Success);
}
```

Figure 5: The **acquire_lock** operation procedure

record immediately after `Prev_Node`, as shown in Figure 10. In this case the Compare&Swap will fail at the position marked **F** in Figure 5. The correctness of this operation's position is not affected, so the operation continues from its current position (line marked by **i** in Figure 5).

2. A concurrent **release_lock** operation may overtake the **acquire_lock** operation and removes the record pointed to by `Prev_Node`, as shown in Figure 11. In this case, the `Dq` bit in the link pointer of this record will be TRUE. The algorithm checks for this condition when it scans through the queue and when it tries to commit its modifications. The algorithm detects the situation in the two places marked by **ii** in the Figure 5. Every time a new record is accessed (by `Prev_Node`), its link pointer is read into `Next_Node` and the `Dq` bit is checked. In addition, if the Compare&Swap fails, the link pointer is saved in `Next_Node` and the `Dq` bit is tested. If the `Dq` bit is TRUE, the algorithm starts from the beginning.

3. A concurrent **release_lock** operation may overtake the **acquire_lock** operation and remove the record pointed to by `Prev_Node`, and then the record is put back into the queue, as shown in Figure 12. If the record returns with a priority higher than or equal to Self's priority, then the position is still correct and the operation can continue. Otherwise, the operation cannot find the correct insertion point, so it has to start from the beginning. This condition is tested at the lines marked **iii** in Figure 5.

The spin-lock busy waiting of a process is broken by the eventual release of the lock by the process which is immediately ahead of the waiting process.

2.2.4 Release_Lock Operation

The **release_lock** operation is straight forward and the algorithm is given in Figure 6. The process p releasing the lock sets the `Dq` bit in its record's `Link` pointer to TRUE, indicating that the record is no longer in the queue. Setting the `Dq` bit prevents any **acquire_lock** operation from modifying the link. The releasing process copies the address of the successor record, if any, to `L`. The process then releases the lock by setting the `Locked` boolean variable in the record of the next process waiting to be FALSE. To avoid testing special cases in the **acquire_lock** operation, the priority of the head record is set to the highest possible priority.

3 Correctness of PR-lock Algorithm

In this section, we present an informal argument for the correctness properties of our PR-lock algorithm. We prove that the PR-lock algorithm is correct by showing that it maintains a priority queue, and the head

```
Procedure release_lock(L, Self){
    Self.Dq=TRUE;
    L=Self.Ptr->Next; /* Release Lock */
    if(Self.Ptr->Next!=NULL){
        L.Ptr->Priority=MAX_PRIORITY;
        L.Ptr->Locked=FALSE;
    }
}
```

Figure 6: The **release_lock** operation procedure

of the priority queue is the process that holds the lock. The PR-lock is *decisive-instruction serializable* [25]. Both operations of the PR-lock algorithm have a single *decisive instruction.* The decisive instruction for the **acquire_lock** operation is the successful Compare&Swap and the decisive instruction for the **release_lock** operation is setting the Dq bit. Corresponding to a concurrent execution C of the queue operations, there is an equivalent (with respect to return values and final states) serial execution S_d such that if operation O_1 executes its decisive instruction before operation O_2 does in C, then $O_1 < O_2$ in S_d. Thus, the equivalent priority queue of a PR-lock is in a single state at any instant, simplifying the correctness proof (a concurrent data structure that is linearizable but not decisive-instruction serializable might be in several states simultaneously [14]).

We use the following notation in our discussion. PR-lock $\mathcal{L}$ has lock pointer L, which points to the first record in the lock queue (and the record of the process that holds the lock). Let there be N processes p_1, p_2, ..., p_N that participate in the lock synchronization for a priority lock $\mathcal{L}$, using the PR-lock algorithm. As mentioned earlier, each process p_i allocates a record q_i to enqueue and dequeue. Thus, each process p_i participating in the lock access is associated with a queue record q_i. Let $Pr(p_i)$ be a function which maps a process to its priority, a number between 1 and N. We also define another function $Pr(q_i)$ which maps a record belonging to a process p_i to its priority.

A *priority queue* is an abstract data type that consists of:

- A finite set Q of elements q_i, i = 1...N
- A function $Pr : q_i \rightarrow n_i$,where $n_i \in \mathcal{N}$. For simplicity, we assume that every n_i is unique. This assumption is not required for correctness, and in fact processes of the same priority will obtain the lock in FCFS order.
- Two operations *enqueue* and *dequeue*

At any instant, the state of the queue can be defined as

$$Q = (q_0, q_1, q_2, \ldots, q_n)$$

where $q_1 <_Q q_2 <_Q \ldots <_Q q_n$, and $q_i <_Q q_j$ iff $Pr(q_i) > Pr(q_j)$.

We call q_0 the *head record* of priority queue Q. The head record's process is the current lock holder. Note that the non-head records are totally ordered.

The *enqueue* operation is defined as

$$enqueue((q_0, q_1, q_2, \ldots, q_n), \hat{q}) \rightarrow (q_0, q_1, q_2, \ldots, q_i, \hat{q}, q_{i+1}, \ldots, q_n)$$

where $Pr(q_i) > Pr(\hat{q}) > Pr(q_{i+1})$.

The *dequeue* operation on a non-empty queue is defined as

$$dequeue((q_0, q_1, q_2, \ldots, q_n)) \rightarrow (q_1, q_2, \ldots, q_n)$$

where the return value is q_0. A dequeue operation on an empty queue is undefined.

For every PR-lock $\mathcal{L}$, there is an abstract priority queue Q. Initially, both $\mathcal{L}$ and Q are empty. When a process $\hat{p}$ with a record $\hat{q}$ performs the decisive instruction for the **acquire_lock** operation, Q changes state to $enqueue(Q, \hat{q})$. Similarly, when a process executes the decisive instruction for a **release_lock** operation, Q changes state to $dequeue(Q)$.

We show that when we observe $\mathcal{L}$, we find a structure that is equivalent to Q. To observe $\mathcal{L}$, we take a *consistent snapshot* [6] of the current state of the system memory. Next, we start at the lock pointer L and observe the records following the linked list. If the head record has its Dq bit set and its process has exited the **acquire_lock** operation, then we discard it from our observation. If we observe the same records in the same sequence in both $\mathcal{L}$ and Q, then we say that L and Q are *equivalent*, and we write $L \Leftrightarrow Q$.

Theorem 1 *The representative priority queue Q is equivalent to the observed queue of the PR-lock $\mathcal{L}$.*

Proof: We prove the theorem by induction on the decisive instructions, using the following two lemmas.

Lemma 1 *If $Q \Leftrightarrow L$ before a* **release_lock** *decisive instruction, then $Q \Leftrightarrow \mathcal{L}$ after the* **release_lock** *decisive instruction.*

Proof: Let $Q = (q_0, q_1, q_2, \ldots, q_n)$ before a **release_lock** decisive instruction. A **release_lock** operation is equivalent to a *dequeue* operation on the abstract queue. By definition,

$$dequeue((q_0, q_1, q_2, \ldots, q_n)) \rightarrow (q_1, q_2, \ldots, q_n)$$

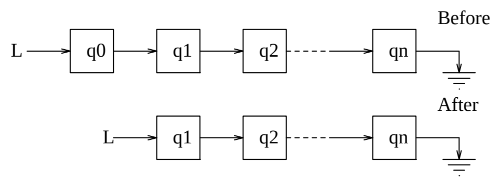

Figure 7: Observed queue $\mathcal{L}$ before and after a **release_lock**

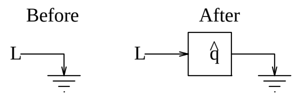

Figure 8: Observed queue $\mathcal{L}$ before and after an **acquire_lock**

The before and after states of $\mathcal{L}$ are shown in Figure 7. If L points to the record q_0 before the **release_lock** decisive instruction, the **release_lock** decisive instruction sets the Dq bit in q_0 to TRUE, removing q_0 from the observable queue. Thus, $Q \Leftrightarrow \mathcal{L}$ after the **release_lock** operation. Note that L will point to q_1 before the next **release_lock** decisive instruction. $\square$

Lemma 2 *If* $Q \Leftrightarrow \mathcal{L}$ *before an* **acquire_lock** *decisive instruction, then* $Q \Leftrightarrow \mathcal{L}$ *after the* **acquire_lock** *decisive instruction.*

Proof: There are two different cases to consider:

Case 1: $Q = ()$ before the **acquire_lock** decisive instruction. The equivalent operation on the abstract queue Q is the enqueue operation. Thus,

$$enqueue((), \hat{q}) \rightarrow (\hat{q})$$

If the lock $\mathcal{L}$ is empty, $\hat{q}$'s process executes a successful decisive Compare&Swap instruction to make L to point to $\hat{q}$ and acquires the lock (Figure 8).

Clearly, $Q \Leftrightarrow$ after the **acquire_lock** decisive instruction.

Case 2: $Q = (q_0, q_1, q_2, \ldots, q_n)$ before the **acquire_lock** decisive instruction. The state of the queue Q after the **acquire_lock** is given by

$$enqueue = ((q_0, q_1, q_2, \ldots, q_n), \hat{q}) \rightarrow (q_0, q_1, q_2, \ldots, q_i, \hat{q}, q_{i+1}, \ldots, q_n)$$

The corresponding $\mathcal{L}$ before and after the **acquire_lock** is shown in Figure 9. The pointers P and N are the *Prev_Node* and *Next_Node* pointers by which $\hat{q}$'s **acquire_lock** operation positions its record such that the process observes $Pr(q_i) > Pr(\hat{q}) > Pr(q_{i+1})$. Then, the `Next` pointer in $\hat{q}$ is is set to the address of q_{i+1}. The

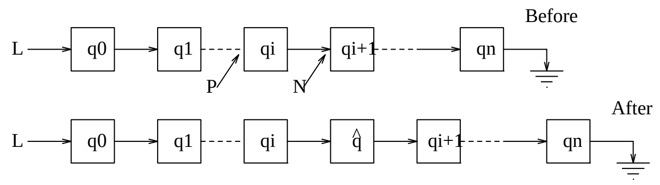

Figure 9: Observed queue $\mathcal{L}$ before and after an **acquire_lock**

Compare&Swap instruction, marked **F** in Figure 5, attempts to make the `Next` pointer in q_i point to $\hat{q}$. If the Compare&Swap instruction succeeds, then it is the decisive instruction of $\hat{q}$'s process and the resulting queue $\mathcal{L}$ is illustrated in the Figure 9. This is equivalent to Q after the enqueue operation. If the Compare&Swap succeeds only when q_i is in the queue, q_{i+1} is the successor record, and $Pr(q_i) < Pr(\hat{q}) < Pr(q_{i+1})$, then $Q \Leftrightarrow \mathcal{L}$.

If there are no concurrent operations on the queue, we can observe that the P and N are positioned correctly and the Compare&Swap succeeds. If there are other concurrent operations, they can interfere with the execution of an **acquire_lock** operation, A. There are three possibilities:

Case a: Another **acquire_lock** *A'* enqueued its record q' between q_i and q_{i+1}, but q_i has not yet been dequeued. If $Pr(q_i) > Pr(\hat{q}) > Pr(q_{i+1})$, then $\hat{q}$'s process will attempt to insert $\hat{q}$ between q_i and q_{i+1}. Process A' has modified q_i's `next` pointer, so that $\hat{q}$'s Compare&Swap will fail. Since q_i has not been dequeued, $Pr(q_i) > Pr(\hat{q})$, and $\hat{q}$'s process should continue its search from q_i, which is what happens. If $Pr(q_{i+1}) > Pr(\hat{q})$ then $\hat{q}$'s process can skip over q' and continue searching from q_{i+1}, which is what happens. This scenario is illustrated in Figure 10.

Case b: A **release_lock** operation R overtakes A and removes q_i from the queue (i.e., R has set q_i's Dq bit), and q_i has not yet been returned to the queue (its Dq bit is still `false`). Since q_i is not in the lock queue, A is lost and must start searching again. Based on its observations of q_i and q_{i+1}, A may have decided to continue searching the queue or to commit its operation. In either case A sees the Dq bit set and fails, so A starts again from the beginning of the queue. This scenario is illustrated in Figure 11

Case c: A **release_lock** operation R overtakes A and removes q_i from the queue, and then q_i is put back in the queue by another **acquire_lock** *A'*. If A tries to commit its operation, then the pointer in q_i is changed, so the Compare&Swap fails. Note that even if q_i is pointing to q_{i+1}, the version numbers prevent the decisive instruction from succeeding. If A continues searching, then there are two possibilities based on the new value of $Pr(q_i)$. If $Pr(\hat{q}) > Pr(q_i)$, A is lost and cannot find the correct place to insert $\hat{q}$. This condition is detected when the priority of q_i is examined (the lines marked **iii** in Figure 5), and operation A restarts from the head of the queue. If $Pr(\hat{q}) < Pr(q_i)$, then A can still find a correct place to insert $\hat{q}$ past

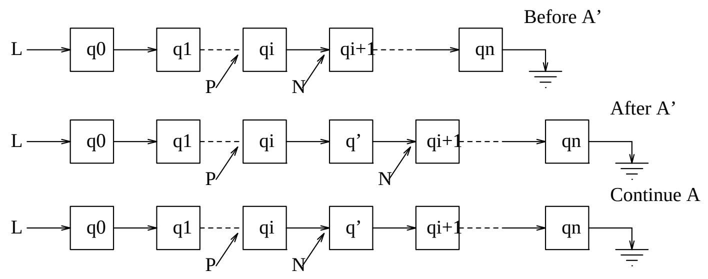

Figure 10: A concurrent **acquire_lock** A' succeeds before A

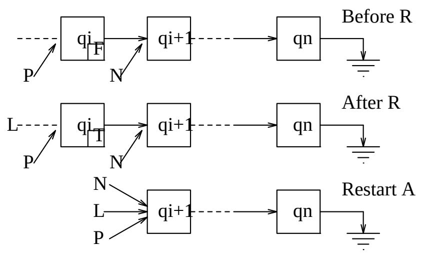

Figure 11: A concurrent **release_lock** R succeeds before A

q_i, and A continues searching. This scenario is illustrated in Figure 12.

No matter what interference occurs, A always takes the right action. Therefore, $Q \Leftrightarrow \mathcal{L}$ after the **acquire_lock** decisive instruction. $\Box$

To prove the theorem we use induction. Initially, $Q = ()$ and L points to *nil*. So, $Q \Leftrightarrow \mathcal{L}$ is trivially true. Suppose that the theorem is true before the i^{th} decisive instruction. If the i^{th} decisive instruction is for an **acquire_lock** operation, Lemma 2 $\Rightarrow Q \Leftrightarrow \mathcal{L}$ after the i^{th} decisive instruction. If the i^{th} decisive instruction is for a **release_lock** operation, Lemma 1 $\Rightarrow Q \Leftrightarrow \mathcal{L}$ after the i^{th} decisive instruction. Therefore, the inductive step holds, and hence, $Q \Leftrightarrow \mathcal{L}$. $\Box$

4 Extensions

In this section we discuss a couple of simple extensions that increase the utility of the PR-lock algorithm.

4.1 Multiple Locks

As described, a record for a PR-lock can be used for one lock queue only (otherwise, a process might obtain a lock other than the one it desired). If the real-time system has several critical sections, each with their own locks (which is likely), each process must have a lock record for each lock queue, which wastes space.

Fortunately, a simple extension of the PR-lock algorithm allows a lock record to be used in many different lock queues. We replace the Dq bit by a Dq string of l bits. If the Dq string evaluates to $i > 0$ when interpreted

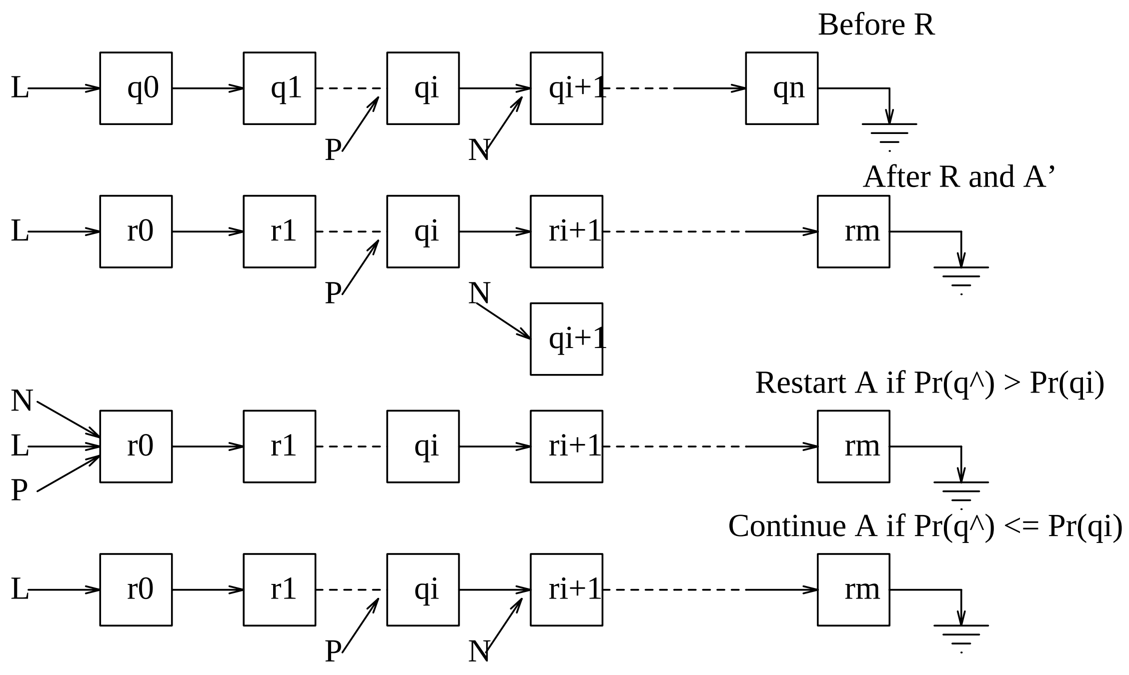

Figure 12: **Release_lock** R and **acquire_lock** A' succeed before A

as a binary number, then the record in in the queue for lock i. If the Dq string evaluates to 0, then the record is (probably) not in any queue. The `acquire_lock` and `release_lock` algorithms carry through by modifying the test for being or not being in queue i appropriately.

We note that if a process sets nested locks, a new lock record must be used for each level of nesting. Craig [10] presents a method for reusing the same record for nested locks.

4.2 Backing Out

If a process does not obain the lock after a certain deadline, it might wish to stop waiting and continue processing. The process must first remove its record from the lock queue. To do so, the process follows these steps:

1. Find the preceding record in the lock queue, using the method from the algorithm for the `acquire_lock` operation. If the process determines that its record is at the head of the lock queue, return with a "lock obtained" value.
2. Set the Dq bit (Dq string) of the process' record to "Dequeued".
3. Perform a compare and swap of the predecessor record's `next` pointer with the process' `next` pointer. If the Compare&swap fails, go to 1. If the Compare&swap succeeds, return with a "lock released" value.

Step 2 fixes the value of the process's successor. If the process removes itself from the queue without obtaining the lock, the Compare&swap is the decisive instruction. If the Compare&swap fails, the predecessor might have released the lock, or third process has enqueued itself as the predecessor. The process can't distinguish between these possibilities, so it must re-search the lock queue.

5 Simulation Results

We simulated the execution of the PR-lock algorithm in PROTEUS, which is a configurable multiprocessor simulator [4]. We also implemented the MCS-lock and Markatos' lock to demonstrate the difference in the acquisition and release time characteristics.

In the simulation, we use a multiprocessor model with eight processors and a global shared memory. Each processor has a local cache memory of 2048 bytes size. In PROTEUS, the units of execution time are *cycles*. Each process executes for a uniformly randomly distributed time, in the range 1 to 35 cycles, before it issues an acquire-lock request. After acquiring the lock, the process stays in the critical section for a fixed number of cycles (150) plus another uniformly randomly distributed number (1 to 400) of cycles before releasing the lock. This procedure is repeated fifty times. The average number of cycles taken to acquire a lock by a process is then computed. PROTEUS simulates parallelism by repeatedly executing a processor's program for a time quanta, Q. In our simulations, $Q = 10$. The priority of a process is set equal to the process/processor number and the lower the number, the higher the priority of a process.

Figures 13 and 14 show the average time taken for a process to acquire a lock using the MCS-lock algorithm and the PR-lock algorithm, respectively. A process using MCS-lock algorithm has to wait in the FIFO queue for all other processes in every round. However, a process using the PR-lock algorithm will wait for a time that is proportional to the number of higher priority processes. As an example, the highest and second highest priority process on the average waits for about one critical section period. We note that the two highest priority processes have about the same acquire lock execution time because they alternate in acquiring the lock. Only after both of these processes have completed their execution can the third and fourth highest priority processes obtain the lock. Figure 14 clearly demonstrates that the average acquisition time for a lock using PR-lock is proportional to the process priorities, whereas the average acquisition time is proportional to the number of processes in case of the MCS-lock algorithm. This feature makes the PR-lock algorithm attractive for use in real time systems.

In Figure 15, we show the average time taken for a process to acquire the lock using Markatos' algorithm. The same prioritized lock-acquisition behavior is shown, but the average time to acquire a lock is 50% greater than when the PR-lock is used. At first this result is puzzling, because Markatos' lock performs the majority of its work when the lock is released and the PR-lock performs its work when the lock is acquired. However, the time to release a lock is part of the time spent in the critical section, and the time to acquire a lock depends primarily on time spent in the critical section by the preceding lock holders. Thus, the PR-lock allows much faster access to the critical section. As we will see, the PR-lock also allows more predictable access to the critical section.

Figure 16 shows the cache hit ratio at each instance of time on all the processors. Most of the time the cache-hit ratio is 95% or higher on each of the processors, and we found an average cache hit rage of 99.72% to 99.87%. Thus, the PR-lock generates very little network or memory contention in spite of the processes using busy-waiting.

Finally, we compared the time required to release a lock using both the PR-lock and Markatos' lock. These results are shown in Figure 17 and for Markatos' lock in Figure 18. The time to release a lock using PR-lock is small, and is consistent for all of the processes. Releasing a lock using Markatos' lock requires significantly more time. Furthermore, in our experiments a high priority process is required to spend significantly more

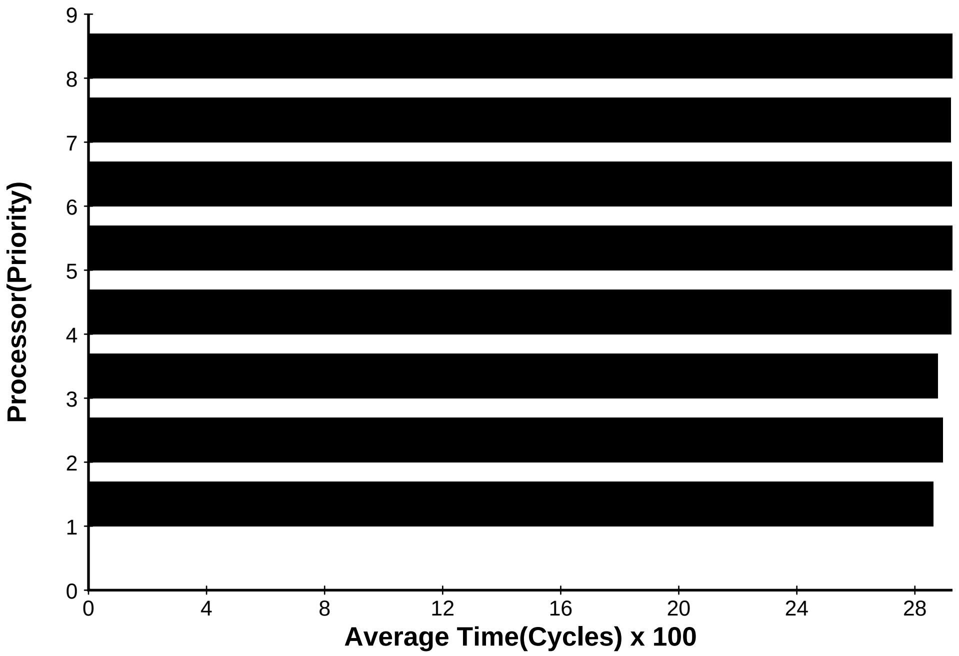

Figure 13: Lock acquisition time for the MCS-lock Algorithm

time releasing a lock than is required for a low priority process. This behavior is a result of the way that the simulation was run. When high priority processes are executing, all low priority processes are blocked in the queue. As a result, many records must be searched when a high priority process releases a lock. Thus, a high priority process does work on behalf of low priority processes. The time required for a high priority process to release its lock depends on the number of blocked processes in the queue. The result is a long and unpredictable amount of time required to release a lock. Since the lock must be released before the next process can acquire the lock, the time required to acquire a lock is also made long and unpredictable.

6 Conclusion

In this paper, we present a priority spin-lock synchronization algorithm, the PR-lock, which is suitable for real-time shared-memory multiprocessors. The PR-lock algorithm is characterized by a prioritized lock acquisition, a low release overhead, very little bus-contention, and well-defined semantics. Simulation results show that the PR-lock algorithm performs well in practice. This priority lock algorithm can be used as presented for mutually exclusive access to a critical section or can be used to provide higher level synchronization constructs such as prioritized semaphores and monitors. The PR-lock maintains a pointer to the record of the lock holder, so the PR-lock can be used to implement priority inheritance protocols. Finally, the PR-lock algorithm can be adapted for use as a single-dequeuer, multiple-enqueuer parallel priority queue.

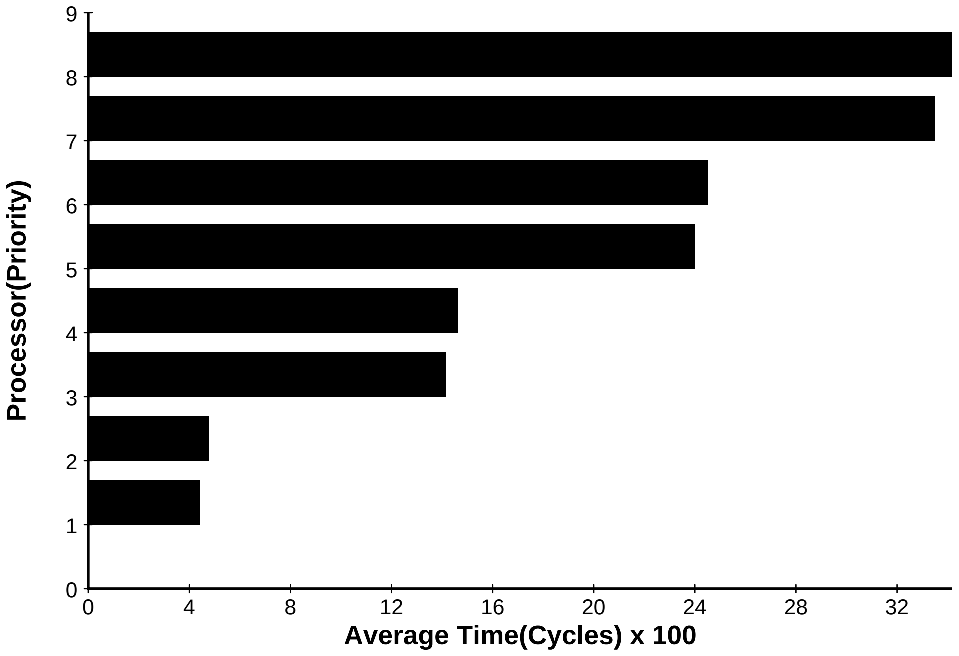

Figure 14: Lock acquisition time for the PR-lock Algorithm

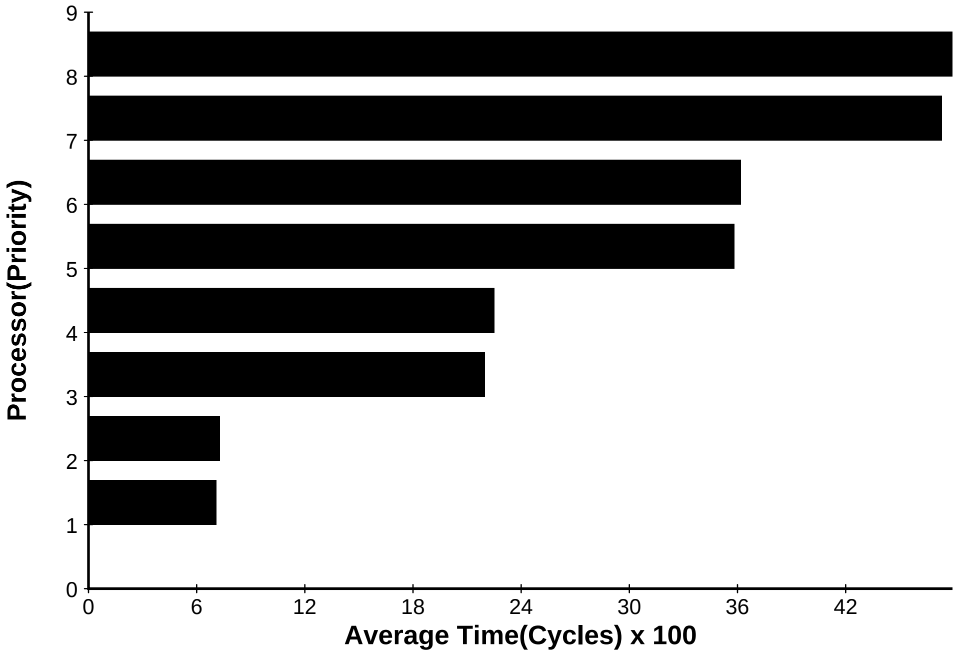

Figure 15: Lock acquisition time for the Markatos' Algorithm

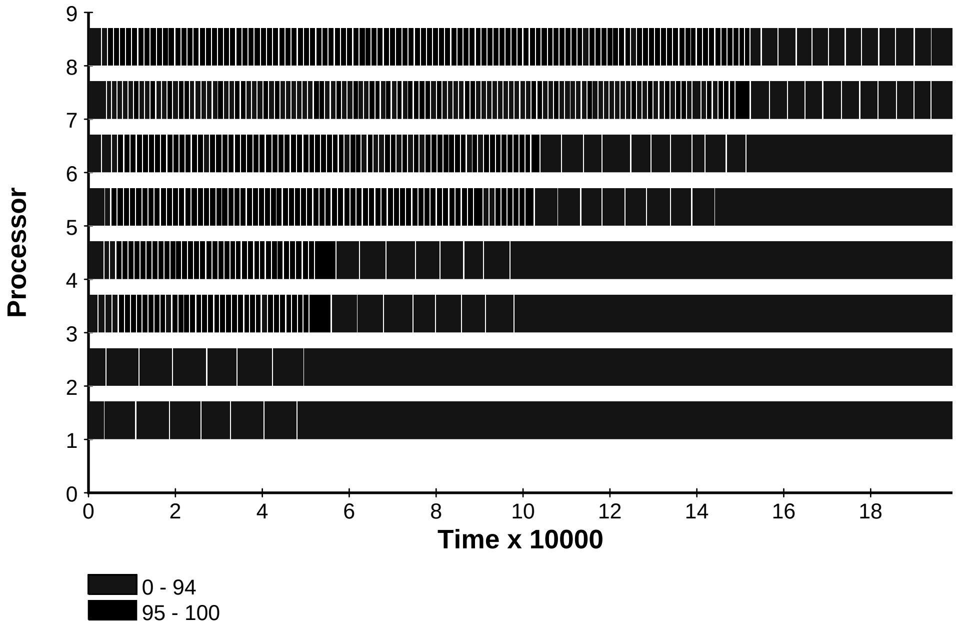

Figure 16: Cache hit ratio for the PR-lock Algorithm

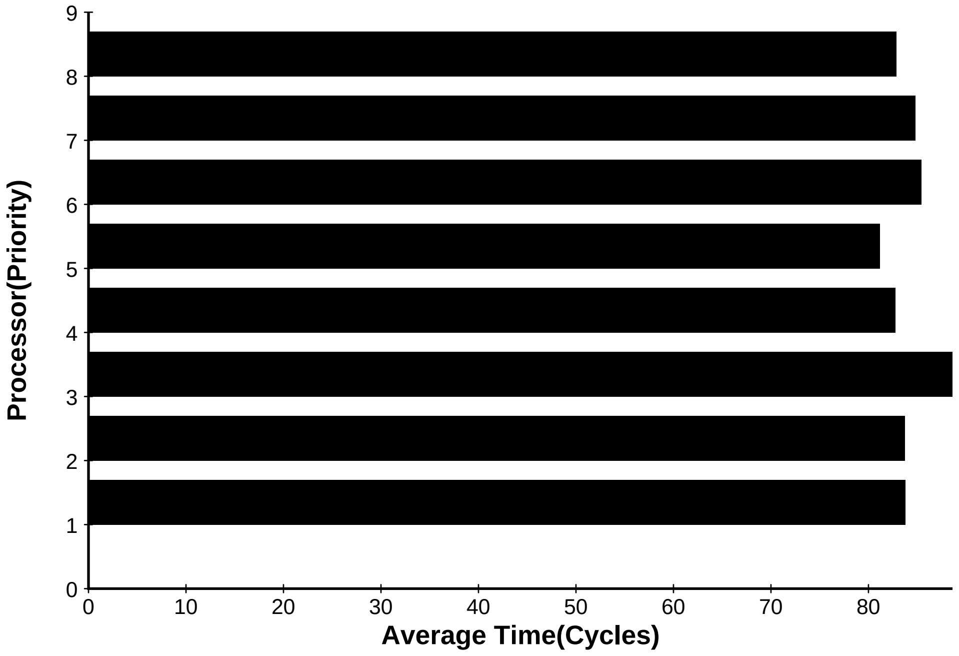

Figure 17: Lock release time for the PR-Lock Algorithm

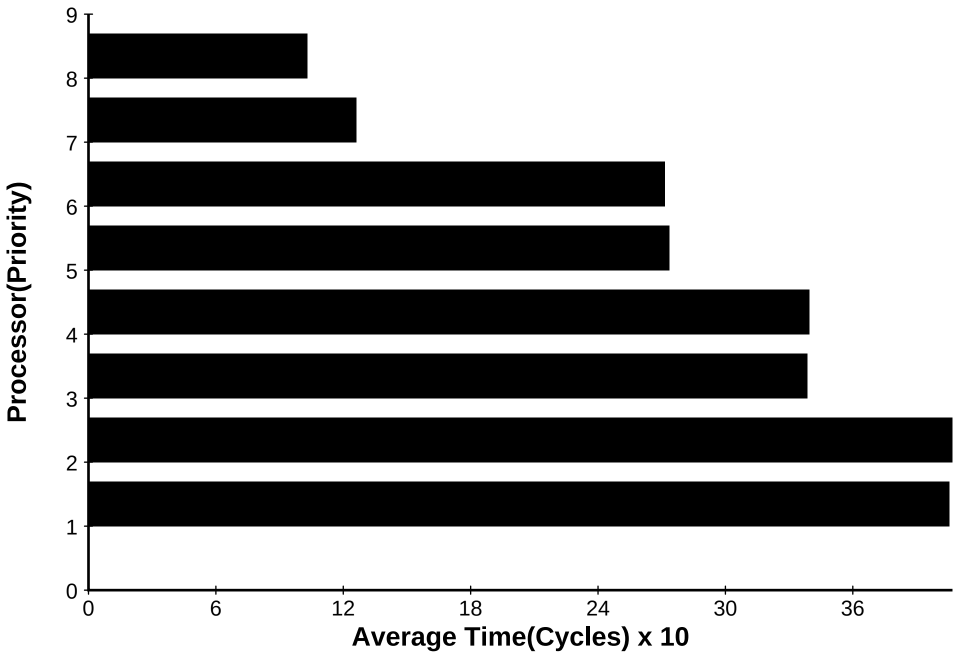

Figure 18: Lock release time for Markatos' Algorithm

While several prioritized spin locks have been proposed, the PR-lock has the following advantages:

- The algorithm is contention free.
- A higher priority process does not have to work for a lower priority process while releasing a lock. As a result, the time required to acquire and release a lock is fast and predictable.
- The PR-lock has a well-defined acquire-lock point.
- The PR-lock maintains a pointer to the process using the lock that facilitates implementing priority inheritance protocols.

For future work, we are interested in prioritizing access to other operating system structures to make them more appropriate for use in a real-time parallel operating system.

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, 1990.

[2] G.R. Andrews. *Concurrent Programming Principles and Practice.* Benjamin/Cummings, 1991.

[3] Inc. BBN Advanced Computers. Tc2000 programming handbook.

[4] E.A. Brewer, Chrysanthos, and N. Dellarocas. PROTEUS user documentation. Technical report, MIT Laboratory for Computer Science, 1991.

[5] J.E. Burns. Mutual exclusion with linear waiting using binary shared variables. *SIGACT News*, 10(2), 1978.

[6] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[7] M.I. Chen and K.J. Lin. Dynamic priority ceiling: A concurrency control protocol for real-time systems. *Real-Time Systems Journal*, 2(4):325–346, 1990.

[8] M.I. Chen and K.J. Lin. A priority ceiling protocol for multiple-instance resources. In *Real Time Systems Symposium*, pages 140–149, 1990.

[9] C.L.Liu and W.J. Leyland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–63, 1973.

[10] T.S. Craig. Queuing spin lock alternatives to support timing predictability. Technical report, University of Washington, 1993.

[11] R.R. Glenn, D.V. Pryor, J.M. Conroy, and T. Johnson. Characterizing memory hotspots in a shared memory mimd machine. In *Supercomputing '91.* IEEE and ACM SIGARCH, 1991.

[12] A. Goscinski. Two algorithms for mutual exclusion in real-time distributed computer systems. *The Journal of Parallel and Distributed Computing*, 9:77–82, 1990.

[13] M. Herlihy. A methodology for implementing highly concurrent data objects. Technical Report CRL 91/10, DEC Cambridge Research Lab, 1991.

[14] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[15] J. Hong, X. Tan, and D. Towsley. A performance analysis of minimum laxity and earliest deadline in a real-time system. *IEEE Trans. on Computers*, 38(12):1736–1744, 1989.

[16] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. In *Proc. 5th ACM Symp on Principles of Distributed Computing*, pages 218–228, 1986.

[17] Maekawa, Oldehoeft, and Oldehoeft. *Operating Systems: Advanced Concepts.* Benjamin/Cummings, 1987.

[18] E.P. Markatos and T.J. LeBlanc. Multiprocessor synchronization primitives with priorities. Technical report, University of Rochester, 1991.

[19] J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Computer Systems*, 9(1):21–65, 1991.

[20] L.D. Molesky, C. Shen, and G. Zlokapa. Predictable synchronization mechanisms for real-time systems. *Real-Time Systems*, 2(3):163–180, 1990.

[21] Motorola. M68000 family programmer's reference manual.

[22] S. Prakash, Y.H. Lee, and T. Johnson. A non-blocking algorithm for shared queues using compare-and-swap. In *Proc. Int'l Conf. on Parallel Processing*, pages II68–II75, 1991.

[23] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real Time Systems Symposium*, 1988.

[24] R. Rajkumar, L. Sha, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on Computers*, 39(9):1175–1185, 1990.

[25] D. Shasha and N. Goodman. Concurrent search structure algorithms. *ACM Transactions on Database Systems*, 13(1):53–90, 1988.

[26] J.A. Stankovic and K. Ramamritham. *Tutorial Hard Real-Time Systems.* EEE Computer Society Press, 1986.

[27] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *ACM Symp. on Principles of Database Systems*, pages 212–222, 1992.

[28] C.-Q Zhu and P.-C. Yew. A synchronization scheme and its applications for large multiprocessor systems. In *Proceedings of the 4th International Conference on Distributed Computing Systems*, pages 486–493, 1984.