A simple correctness proof of the MCS contention-free lock

Theodore Johnson
Krishna Harathi
Computer and Information Sciences Department
University of Florida

Abstract

Mellor-Crummey and Scott present a spin-lock that avoids network contention by having processors spin on local memory locations. Their algorithm is equivalent to a lock-free queue with a special access pattern. The authors provide a complex and unintuitive proof of the correctness of their algorithm. In this paper, we provide a simple proof that the MCS lock is a correct critical section solution, which provides insight into why the algorithm is correct.

Keywords: mutual exclusion, serializability, parallel processing, spin lock

1 Introduction

Mellor-Crummey and Scott present a simple spin-lock for mutual exclusion [5], which they call the *MCS lock*. Their algorithm has several desirable properties: it is is fair, because it is equivalent to a FIFO queue ADT with special access pattern, and it is is contention free, because each processor spins on a local variable. In contrast, test-and-set spin waiting is not fair, and will severely limit performance [1, 2].

The authors of the MCS locks provide a complex proof of the correctness of their algorithm. In [4], they show correctness through a series of algorithm transformations. Since this proof provides little insight, they also provide intuitive correctness arguments.

In this paper, we present a simple correctness proof for the MCS lock. We show that the underlying queue is decisive-instruction serializable, and that no operation access a garbage address. We conclude that the MCS lock is a correct solution to the critical section problem.

2 MCS-Lock Algorithm

The code for the MCS lock, which is shown below, relies on two atomic read-modify-write operations. The `swap` instruction takes two parameters, `L`, a pointer to a memory location, and `I`, a value. The execution of `swap(L,I)` puts `I` into `L` and returns the old value of `L`. The `compare_and_swap` instruction takes a third parameter `X`, a value. The execution of `compare_and_swap(L,I,X)` puts `I` into `L` only if `L` currently contains the value `X`. In either case, `compare_and_swap` returns the old value in `L`.

Each processor that sets a lock first allocates a locally-accessible record. If process p inserts record r into the queue, then we say that r is p's record, and p is r's process. This record contains a boolean flag for spin waiting and a link for forming a queue. The tail of the queue is pointed to by `L`, and the head of the queue is implicitly pointed to by the lock holder. To set a lock, a processor executes the `acquire_lock` procedure and adds its record, pointed to by `I`, to the end

of the queue. The enqueuing is performed by executing the `fetch_and_store` instruction, which atomically stores `I` in `L` and returns `L`'s old value. If the processor has a predecessor in the queue, the processor completes the link; otherwise the processor is the lock holder.

To release a lock, the processor must reset the busy-waiting bit of its successor. If the processor has no successor, it sets `L` to nil atomically with the `compare_and_swap`.

```
type qnode = record
   next :  *qnode                    // ptr to successor in queue
   locked :  Boolean                 // busy-waiting necessary
type lock = *qnode                   // ptr to tail of queue

// I points to a queue link record allocated (in an enclosing scope)
// in shared memory locally accessible to the invoking processor

procedure acquire_lock( L: *lock, I: *qnode )
   var pred:  *qnode
   I->next := nil                    // Initially, no successor
   pred := swap(L, I)                // Queue for lock
   if pred ≠ nil                     // Lock was not free
      i->locked := true              // Prepare to spin
      pred->next := I                // Link behind predecessor
      repeat while I->locked         // Busy wait for lock

procedure release_lock( L: *lock, I: *qnode )
   if I->next = nil                  // No known successor
      if compare_and_swap(L, I, nil)
         return // No successor, lock free
      repeat while I->next = nil     // Wait for successor
   I->next->locked := false          // Pass lock
```

MCS-Lock Algorithm (from [5]).

3 Correctness

We show that the MCS lock is correct by showing that it maintains a queue, and the head of the queue is the process that holds the lock. The MCS lock is *decisive-instruction serializable* [6]. Each operation has a single *decisive instruction*, and corresponding to a concurrent execution $\mathcal{C}$ of the queue operations, there is an equivalent serial execution $\mathcal{S}_d$ such that if operation O_1 executes its decisive instruction before operation O_2 does in $\mathcal{C}$, then $O_1 < O_2$ in $\mathcal{S}_d$. The decisive-instruction serializability of the MCS lock greatly simplifies its correctness proof, because the equivalent queue is in a single state at any instant. In contrast, a concurrent data structure that is linearizable but not serializable might be in several states simultaneously [3].

3.1 The Queue ADT

A *queue* is an Abstract Data Type that consists of finite set Q and two operations: *enqueue* and *dequeue*. The elements of Q are totally ordered. We write the state a queue as $Q = (q_1, q_2, \ldots, q_n)$, where $q_1 <_Q q_2 <_Q \ldots <_Q q_n$.

The enqueue operation is specified by

$$enqueue((q_1, q_2, \ldots, q_n), q') \rightarrow (q_1, q_2, \ldots, q_n, q')$$

The dequeue operation on a non-empty queue is specified by

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

where the return value is q_1. A dequeue operation on an empty queue is undefined.

We define two functions on a queue Q: $head(Q)$ and $tail(Q)$. The *head* function returns the least element of the queue and the *tail* function returns the greatest element of the queue.

Corresponding to an actual MCS lock M, there is an abstract queue Q. Initially, both M and Q are empty. When process p with record r performs the decisive instruction for an `acquire_lock` operation, Q changes state to $enqueue(Q, r)$. When a process performs the decisive instruction for a `release_lock` operation, Q changes state to $dequeue(Q)$. We will show that the actual MCS lock corresponds to the abstract queue.

3.2 Execution Sequences

The MCS lock executes correctly because it is presented with a special sequence of concurrent operations. We assume that each processor uses the `acquire_lock` and `release_lock` operations to synchronize access to a resource:

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

If the MCS lock is correct, then it is a *multiple-enqueue/single-dequeue* queue. Any number of `acquire_lock` operations might execute concurrently, but we are guaranteed that

1. At most one `release_lock` operation executes at any given time. Let D_1 and D_2 be two different release lock executions, executing in the time intervals I_1 and I_2, respectively. Then I_1 and I_2 do not overlap.

2. No `release_lock` operation executes between the time that a `release_lock` sets `L` to `Nil` and the time that the first subsequent `acquire_lock` operation terminates. Let D be the execution of a `release_lock` operation that sets `L` to `NIL`, and let D complete its execution at time t_d. Let E be the execution of the `acquire_lock` operation that performs the first decisive instruction after time t_d. If E completes its execution at time t_e, then no `release_lock` operation executes in the time interval (t_d, t_e).

3.3 Queue Invariants

The MCS lock maintains three invariant conditions.

Tail Invariant: If the abstract queue is non-empty, then `L` points to the record at the tail of the abstract queue. If the queue is empty, `L` is Nil.

We define the *head process* to be the process whose record is at the head of the abstract queue. A process that is waiting for `I->locked` to become false is *blocked*.

Head Invariant: If the head process is blocked, it will be unblocked by the time that the preceding `release_lock` operation terminates.

Blocking Invariant: A non-head process will not exit the `acquire_lock` procedure.

3.4 Decisive Operations

The decisive instruction in the `acquire_lock` procedure is the fetch_and_store instruction. The decisive instruction in the `release_lock` procedure is the reading of `I->next` if the fetch returns a non-nil value, and is the compare_and_swap instruction otherwise.

Theorem 1 *The MCS lock is decisive-instruction serializable.*

Proof: To show that the MCS lock is decisive-instruction serializable, we first show that the three invariants are always maintained. To show that the invariants are maintained, we assume that only the head process executes a `release_lock` operation. We then show that this assumption holds by induction.

Lemma 1 *The tail invariant is always maintained.*

Proof: When a process performs the decisive instruction for an `acquire_lock` operation, it sets the tail pointer (`L`) to its record. Therefore, `acquire_lock` operations maintain the tail invariant. A `release_lock` operation modifies the tail pointer by the compare_and_swap instruction only. The compare_and_swap will succeed only if the queue contains exactly one element. Therefore, the `release_lock` operation also maintains the queue invariant.□

Lemma 2 *If only the head process executes a* `release_lock` *operation, the blocking invariant is always maintained.*

Suppose that the record of a process is in the queue but isn't the head record. By the tail invariant, the process must have read a non-nil tail pointer when it performed the decisive instruction for the `acquire_lock` operation. The process will therefore wait until the `locked` field of its record, which is initialized to `true`, is set to `false`. This bit will only be reset when the process of the predecessor record executes a `release_lock` operation. But, after the `release_lock` decisive instruction, the process becomes the head process. □

Lemma 3 *The head invariant is always maintained.*

Suppose that when a head process P performs its decisive instruction on a `release_lock` operation, $dequeue(Q)$ is non-empty. In this case, the decisive instruction will report that tail pointer is not the process' record. Then, P will wait until the `next` field of its record, R is non-nil. By the tail invariant, this field will eventually point to R's successor in Q, r (because of the execution of r's process, p, in the `acquire_lock` procedure). After P's decisive instruction, r is the head record and p is the head process. Process P will reset the `locked` bit of r, unblocking p before P completes.

Suppose that when P performs its decisive instruction, Q becomes empty. If p is the process that executes the next decisive instruction (which must be for a `acquire_lock` operation), p will become the head process and will find that L is `nil`. Since p finds L is `NIL`, p never blocks. □

Lemma 4 *A process executes a* `release_lock` *operation only when it is the head process.*

Proof: We proceed by induction on the i^{th} decisive instruction. For the base case, consider the operation that executes a decisive instruction. Since the queue is empty, the operation must be an `acquire_lock` and the lemma holds.

Suppose that the lemma holds for the i^{th} decisive instruction. If the $i+1^{th}$ decisive instruction is due to an `acquire_lock`, the lemma holds. Suppose that the $i+1^{th}$ decisive instruction is due to

an `release_lock` operation. By the execution sequence assumption, the process that executes this operation must have previously executed a `acquire_lock` operation. By the blocking invariant, the process must be the head process, so the lemma holds. □

Since the head invariant is always maintained, locks are granted in the order requested, where the order of requests is the order of the decisive instructions □

3.5 Garbage

We next show that the queue does not access garbage locations. We assume when a process executes the `acquire_lock` procedure, it first allocates a record. When a process exits the `release_lock` procedure, it discards the record that was at the head of the queue (pointed to by `I`). For simplicity, we assume that a record is never reused. A record is *valid* during the time between its creation and its deletion.

Theorem 2 *No process accesses an invalid queue record.*

Proof: By the blocking invariant, only the head process performs a `release_lock` operation, and the process dequeues its own record. Therefore, whenever a process access its record, it accesses a valid record.

When a process executes the `acquire_lock` operation, the process inserts a pointer to its record's predecessor in Q (if one exists). The process of the predecessor record does not exit the `release_lock` procedure until the `next` field of its record is modified. Therefore, no process accesses an invalid record when executing the `acquire_lock` operation.

When a process executes the `release_lock` operation, it modifies its record's successor in Q. By the execution sequence assumption, this record remains in the queue until the process completes the operation. Therefore, no process access an invalid record when executing the `release_lock` operation. □

4 Critical Section Solution

A critical section solution is correct if it satisfies the following three criteria [7]:

1. At most one process executes in the critical section at any given time.
2. If no process is executing in the critical section, and at least one process wishes to enter the critical section, then a process enters the critical section in a finite amount of time.
3. If a process wishes to enter the critical section, then a finite number of processes enter first.

We can see that the MCS lock is a correct solution to the critical section problem. We define the set of processes that are in or wish to enter the critical section as the set of processes whose records are in Q. By the blocking invariant, criteria 1 holds. By the head invariant, criteria 2 holds. The MCS lock is a decisive-instruction serializable FIFO queue, so criteria 3 holds.

5 Conclusion

The MCS lock is method of implementing mutual exclusion that avoids network contention. Further, the MCS lock is fair, unlike the simple test-and-set solution. We present a simple correctness

proof for the MCS lock. We show that the MCS lock is equivalent to a decisive-instruction serializable queue, and show that several queue invariants always hold. Whereas the correctness proof provided by Mellor-Crummey and Scott is indirect and provides little insight, our correctness proof is direct and provides insight into the algorithm's execution.

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] 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.

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

[4] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. Technical Report TR90-114, Rice University Dept. of CS, 1990.

[5] 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.

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

[7] A. Silberschatz, J. Peterson, and P. Galvin. *Operating System Concepts*. Addison Wesley, 1991.