Fast mutual exclusion algorithms using read-modify-write and atomic read/write registers *

Ting-Lu Huang

Dept. of Computer Science
and Information Engineering
National Chiao Tung University
1001 Ta-Hsueh Road
Hsin-Chu, Taiwan 30050
Republic of China
E-mail: tlhuang@csie.nctu.edu.tw

Abstract: Three fast mutual exclusion algorithms using read-modify-write and atomic read/write registers are presented in a sequence, with an improvement from one to the next. The last algorithm is shown to be optimal in minimizing the number of remote memory accesses required in a resource busy period. Remote memory access is the key factor of memory access bottleneck in large shared-memory multiprocessors. The algorithm is particularly suitable in such systems for applications with small critical sections and frequent resource requests.

Keywords: mutual exclusion, multiprocessors, memory access bottleneck, multiprocessing, fairness.

1 Introduction

Critical section facilities must be provided for user programs to share resources in multiprocessing systems. A large number of mutual exclusion algorithms have been proposed during the last thirty some years. Nevertheless, designing mutual exclusion algorithms that are both *practical* and *correct* has always been a very tricky task. Even when powerful primitives are available, mistakes in designing mutual exclusion algorithms [1, 3] are not uncommon.

Mellor-Crummey and Scott [2] (referred to as **MCS algorithms** in literature) initiates a series of studies that more or less follow their ideas of busy waiting on local memory locations only. Zhang et al. [5] has similar algorithms. Recently Fu and Tzeng [8] presented a circular list-based mutual exclusion scheme (referred to as **CL algorithms** in this paper) for large-scaled multiprocessor systems. While it provides considerable performance improvements, the CL algorithms suffer from the following drawbacks:

1) Deadlock error in the trying protocol, and
2) Starvation unfairness in the exit protocol.

In this article, several algorithms that follows the line of CL algorithms but suffer from neither of the drawbacks is provided. Furthermore, one of the algorithms is proved optimal in minimizing the number of remote memory accesses executed in the algorithm during resource busy period. Remote memory access is probably the most important factor that contributes to memory access bottleneck in large-scaled shared-memory multiprocessors. The success of MCS algorithm and CL algorithm is largely due to the elimination of busy waiting (with unpredictable number of accesses) on remote memory locations. The major merit of CL algorithm is the elimination of remote memory access needed in MCS algorithm to re-direct an address link for each privilege passing during resource busy periods. Our algorithm is shown to be optimal in reducing the number of remote memory accesses: we show that any further reduction beyond what is achieved by our algorithm is impossible.

Like MCS algorithms, our algorithms assume that *fetch&store* and *compare&swap* primitives are available. Such primitives have been widely implemented in current multiprocessor systems. The CL algorithms assume the same *fetch&store* and a fictitious *swap&compare*. The latter one has never been im-

*This work was supported by National Science Council, Republic of China, under Grant NSC87-2213-E-009-009

```
compare&swap
(r: public register, old, new: value) returns(value)
    previous := r
    if previous = old
        then r := new
    fi
    return previous

swap&compare
(r: public register, old: private register, new: value)
    previous := r
    r := old
    old := previous
    if r = old
        then r := new
    fi

fetch&store
(r: public register, my: value) returns(value)
    previous := r
    r := my
    return previous
```

Figure 1: Compare&swap, Swap&compare and Fetch&store primitives.

plemented in any system.

In addition to the read-modify-write shared variable that is accessed by the primitives, the algorithms require N atomic read/write shared variables (called q-nodes later in this paper), one for each participating process. A small number of private variables for each process is also required.

Rest of the paper is organized as follows. Section 2 provides definitions and models. Section 3 presents the three fast algorithms. Section 4 is the conclusion.

2 Definitions and models

2.1 The RMW primitives

Definitions of the read-modify-write (RMW) primitives used in this article are given in Figure 1. To follow the convention in literature of RMW primitives, the definitions use "register" to refer to *variable* in common usage.

2.2 Flowcharts as algorithms

The algorithms are represented by flowcharts. When an algorithm involves only a few actions but is nevertheless very subtle, a flowchart provides a clear picture of the control flow and leads to an easier correctness argument, at least for the algorithms in this article.

A rectangular node contains a sequence of actions that satisfy one of the following conditions:

(1) All memory accesses are to private variables;
(2) No more than one memory access is to the shared variables;
(3) Multiple memory accesses occur for the shared variables via exactly one execution of a RWM primitive.

If the set of accesses in a rectangular node does not satisfy the above condition, the sequence of actions should be split into two or more nodes. The rule is helpful in simplifying correctness reasoning when we need to consider all possibilities of interleaving among the processes: State transitions in a rule-abiding node can be lumped together as one transition. Note that full advantages of such lumping may not have been taken in all flowcharts given in this article. But the rule is followed and there is no node that needs to be split.

A diamond node contains a test of condition that involves at most one access to shared variables.

An oval node represents a sequence of test operations that will block the process until the awaited condition becomes true. Each test operation involves at most one access to shared variables.

While there may be more than one incoming edge to any node, there is exactly one outgoing edge for a rectangular or an oval node. The two outgoing edges for a diamond node are labeled "yes" and "no", respectively.

2.3 Flowcharts as mutual exclusion algorithms

For a non-terminating algorithm such as mutual exclusion, the flowchart has an incoming edge (labeled start) but no escaping edges.

Formal definition of mutual exclusion problem can be found in [6]. Lynch [4] introduced several impossibility results in mutual exclusion algorithms using only one RMW register. Here we extract from various sources and re-define the problem in terms of our model. For mutual exclusion algorithms to meet *well-formedness* requirement, a flowchart prescribes an endless loop of life cycles for each process: trying (T) region, critical (C) region, exit (E) region and remainder (R) region. The label in each node starts with a T for trying regions, an E for exit regions. No

path exists for a process to bypass any region in the life cycles.

For mutual exclusion algorithms to meet *mutual exclusion* requirement, the set of edges (as a whole) that are labeled "critical region" cannot be visited by more than one process at any time. If there is one process visiting one such edge, no other processes visit such edges at the same time. Instead of proving such "exclusiveness" for the critical region set, we may want to prove there exists a set of edges for a flowchart, called the **exclusive set**, that enjoys exclusiveness, and that it includes the critical region set. We use thick lines to mark the edges of the exclusive sets. We found it easier to argue for the entire exclusive set than to do so for the critical region directly. Of course, an in-depth understanding is required in selecting the exclusive set for a given algorithm. One necessary condition for correct selection is that the incoming edges to any particular node must all be thick or none is thick. If the condition is not satisfied, either the algorithm itself is wrong or the selection is wrong.

3 The fast mutual exclusion algorithms

Most optimal mutual exclusion algorithms aim at minimizing the size of the shared variables or minimizing the number of shared variables. Few are minimizing the number of memory accesses. Lamport [7] tried to minimize the number of accesses in a period of no competing requests. We try to minimize the number of accesses in a period of frequent competing requests. We also take into account the difference between local memory and remote memory. Access to local memory does not incur memory access bottleneck, while remote memory access does. Therefore, we count only the number of remote memory accesses. Minimizing remote accesses in a period of frequent competing requests serves a good purpose since memory access bottleneck in large shared-memory multiprocessor in such periods can lead to bad performance, and remote memory access is a key factor of memory access bottleneck.

Figure 2 shows the data structure of the memory space that is allocated for each process in the mutual exclusion algorithm. The first algorithm actually uses only one bit (the boolean *wait*) for the q-node and the second algorithm uses three bits (adding *direct* and *hold*) for the q-node. The permission-word is used by the third algorithm.

```
type q-node = record
     wait :   Boolean
     direct : Boolean
     hold :   Boolean
type permission-word = fullword
     head :            halfword
     tail :            halfword
```

Figure 2: Per process data structures for the algorithms.

3.1 A deadlock-free CL algorithm

Figure 3 is the circular list-based mutual exclusion algorithm with the original deadlock error removed. The original CL algorithm was given in a C-like language. Here the flowchart version has some advantages for illustration purposes. The exit region is apparently divided into two separate paths: one for those processes that may stay in the exit region indefinitely, and one for those that move to the remainder region immediately after one step. The C-like version, in contrast, represents the exit region using one procedure with single entry point and therefore requires an extraneous decision statement to branch to the two separate paths. Correctness arguments on the flowchart version are also much easier than those on the C-like version.

For those who are not familiar with the circular list-based algorithms and for those who will read all three algorithms carefully, the rest of this section explains the interaction among processes. Initial state is such that (1) the RMW variable L has the *nil* value; (2) each process is allocated a data structure (called q-node) the address of which is stored in the private variable *I*; and (3) the value of the *wait* variable in each q-node is *true*.

For brevity, the process that is the focus of our discussion is referred to as *P*. T1 is to set the *wait* bit *true*, as is required for each new life cycle. T2 is to make public the address of *P*'s q-node via the shared RMW variable L, and to obtain the address of the q-node which will be needed for *P* to wake up the next process when *P* is through with its critical section. The RMW primitive *fetch&store* is defined in Figure 1. T3 checks whether *P* is the first process that accesses the RMW variable L either since system start-up or since the last event that the value *nil* was written back to L. The *nil* is written back to L by a process when no other processes are interested in entering critical sections. If T3 answers "yes", *P*

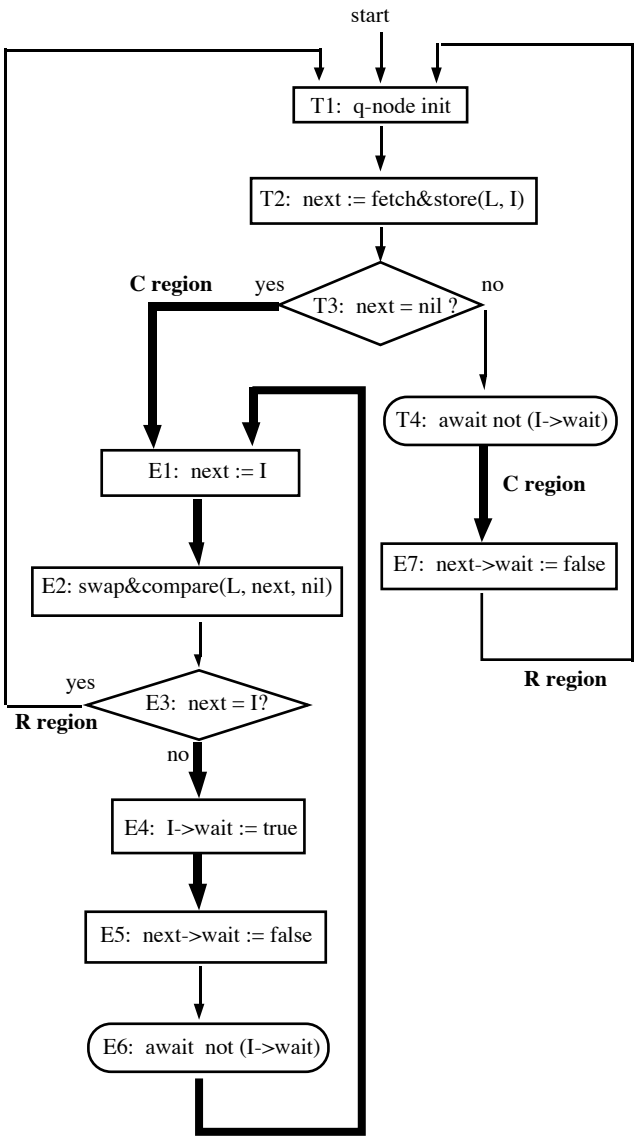

Figure 3: The CL mutual exclusion algorithm with deadlock error removed.

is entitled to enter critical section and all competing processes are now waiting at T4 node.

The RMW primitive *swap&compare* is defined in Figure 1. E1, E2 and E3 together take care of the followings. If L is still pointing at P's q-node, no other processes are interested in entering critical section. P writes *nil* to L and moves to remainder region. If L is pointing to some other q-node, there are some other processes that are waiting. P stores the address of its q-node at the private variable *next* and will use a remote write to wake up that process at E5.

E4 is to make sure that P cannot pass E6 until some other process writes to P's q-node. E4 should precede E5 in execution, or deadlock may occur. Details of the deadlock error can be found in [3].

After passing E6, several processes have been granted permission to enter critical sections but more processes may have arrived and have been kept waiting. P is the sole controller among the competing processes and therefore should go back to E1 to prepare for the next run of playing controller. P will be kept in this potentially unbounded number of runs of playing controller as long as there are processes interested in entering critical section. Later, we will show that the other two algorithms suffer from no such severe unfairness.

E7 is to wake up the next process that either is waiting at T4 for permission to enter critical section or is waiting at E6 for the role of playing controller.

3.2 A bounded-waiting CL algorithm

The main idea of this algorithm, see Figure 4, is to use two more bits in each q-node for better cooperation among processes and to prevent the unfair burden on the controller. The popular *compare&swap* primitive, see Figure 1, is used by current controller to decide whether there are other processes interested in entering critical section. If not, the *nil* value is assigned to L and the controller has nothing else to do. If so, the value of L is returned and assigned to the private variable *next*. That value is the address of the next controller's q-node. The two extra bits of each q-node are used to transfer the role of controller from the current one to the next. The *direct* bit is to inform the next controller that it is chosen as such. The *hold* bit is to hold the next controller at E10 until the current controller finishes a complete run and executes E7. It is easy to observe from the flowchart that no process will be kept in E region for an unbounded number of runs since a process will be able to leave E region if it passes E6. A process cannot be kept indefinitely at E6 because the

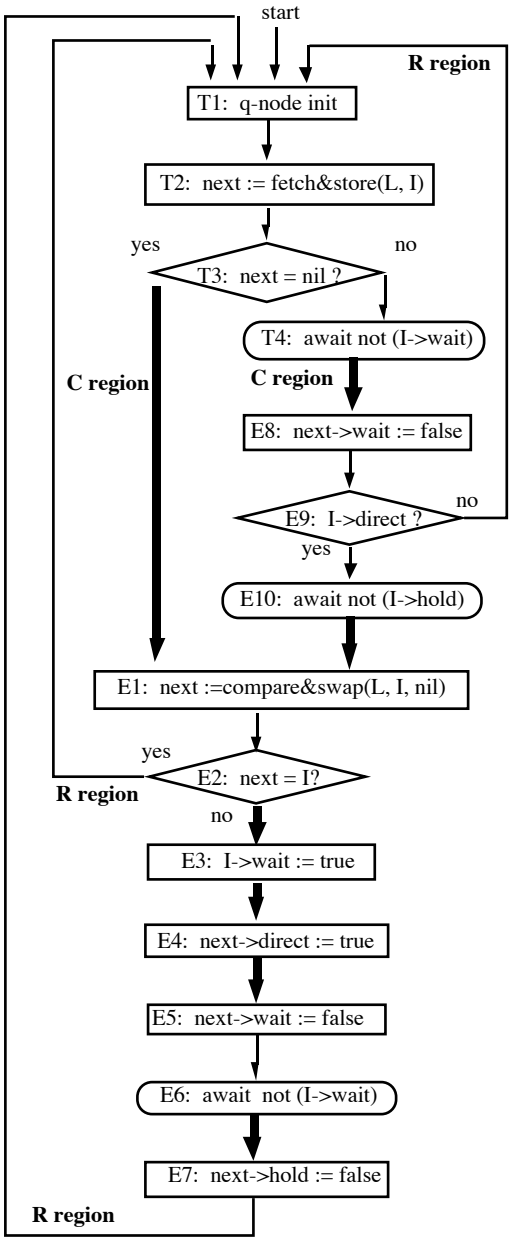

Figure 4: The bounded-waiting CL mutual exclusion algorithm using 1-bit messages.

fetch&store primitive regulates the q-node addresses in such a way that the wake-up signal sent by P at E5 is bound to come back in finite steps as a signal to release P at E6. In fact, after P enters E region, no process can bypass P in passing E region for more than twice. Such level of fairness is good enough for almost all applications.

3.3 The permission word algorithm

We establish a **tight bound** on the number of remote accesses required for the mutual exclusion problem allowing the use of any RMW primitives. A lower bound is shown by an impossibility proof in the next section. This section provides an algorithm that requires exactly the lower bound.

The main idea of the algorithm, see Figure 5, is to write a fullword in each remote write, instead of writing a single bit. The fullword, called **permission word**, consists of a pair of non-zero halfwords, *(head,tail)*, each being the address of a q-node. The permission word not only serves as permission to enter critical section, but also carries enough information for processes to maintain proper control of role playing, without using any other control message. The scheme is simple, but the encoding of the permission word may be confusing at first glance. Figure 6 is an example to help explain how it works.

A **busy period** is an execution sequence that starts with a state in which the RMW variable L has the *nil* value, and ends with a later state in which L has the *nil* value, with at least one process enters and leaves critical section and no states with *nil* in L in between. A **run** in a busy period is an execution sequence that starts with a process executing E1 and ends with some process executing E1, with no such events and at least one process enters and leaves critical section in between. The set of processes permitted to enter critical section (passing T4) in a run is called the **relay** of that run. The process that executes E1 defining a new run is called the **controller** of the new run. One process in the relay of the old run is to be selected as the controller for the new run. Exactly how the controller is selected is explained later. A controller cannot also be in the relay of the new run because when it executes E1, all members of the relay must be somewhere between T2 and T4, waiting for permission. In Figure 6, process 30 is the controller for relay of run 1; process 10 for relay of run 2; process 60 for relay of run 3; and process 70 defines the end of the whole busy period since L does not change (still is 10-) during the whole run. Process 70 puts *nil* in L when it executes E1.

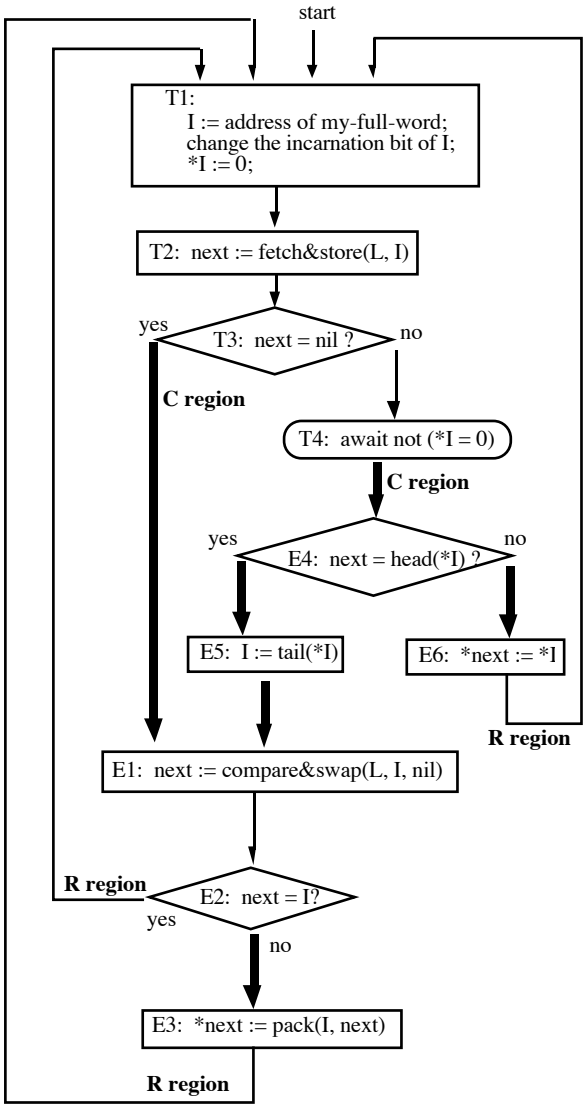

Figure 5: The permission word mutual exclusion algorithm.

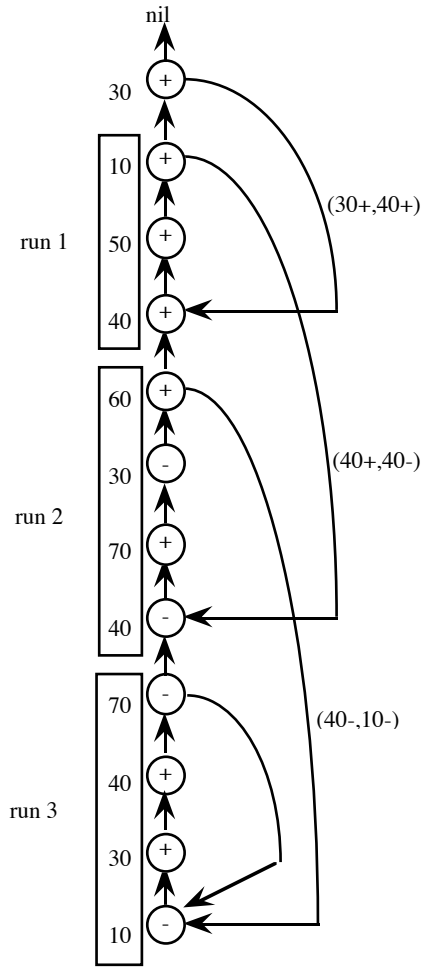

Figure 6: An example of a busy period consisting of 3 runs.

Since the least significant bit of an address is not used in most computer architectures, we can use that bit as the *incarnation* bit to avoid a subtle situation. Although a process cannot appear more than once in a relay, it may appear in both relays of two neighboring runs. A process's incarnation bit (indicated as "+" or "-" in the circles) from one incarnation to the next must be different since a process always flips its incarnation bit at T1. Therefore, no two incarnation bits of the same process in two consecutive runs are the same. This is important for a process to determine whether it should act as the controller for the next run. A process executing E4, which is to check whether the value of *next* equals the *head* halfword in the permission message it receives, will identify itself as the controller if the result (taking into account the incarnation bit) is "yes". For example, process 30 in run 3 would not be able to tell the difference between 40+ in run 3 and 40- in run 2 without the incarnation bit. With the difference of the bit, process 30 should pass the permission message to process 40, rather than taking up the role of controller. Process 70 in run 3 should be the controller since it receives (40-,10-) as the (head,tail) pair and its *next* has value 40-. (It will get "yes" result at E4.) Therefore, it should go to E5 to take up the role of controller. The subtlety occurs whenever the *next* process of E3 after having been given permission to enter critical section, quickly makes a new request (at T2) in the next run. Fortunately, the subtlety needs to be resolved only between two neighboring runs, thus a single bit suffices.

This algorithm enjoys an even better fairness in E region than the previous one does. There exists no blocking statement in E region. After P enters E region, no other process can bypass P in passing E region more than once. Such level of fairness is probably the best we can get in asynchronous systems.

3.4 An impossibility result

To prove that the permission word algorithm is optimal, we must prove that there is no algorithm that requires less remote accesses than our algorithm does. The proof is informal in nature. More rigorous proof is possible, but should convey similar ideas as what we now provide.

Theorem 1 *There is no mutual exclusion algorithm using RMW registers and atomic read/write registers that requires less than 2K+1 remote accesses in any interval (within a busy period) with K life cycles.*

$< proof >$ The k life cycles within a busy period entails a chain of privilege passing from the first exit region to the K-th exit region. By way of contradiction, assumes that only $2K$ remote accesses are needed for the complete chain of K cycles. One life cycle requires at least 2 remote writes: one in the trying region and another in the exit region. Under the constraint of 2 remote writes for one life cycle, a process must have announced its q-node address when it access the RMW register in the trying region, and it must have used a remote write to wake up its successor in the chain. Let P be a process that is the first one to access the RMW register making public its address and trying to pick up an address of others. P is destined to fail in getting any address in that access since no one has put address in the RMW register, yet. For P to be able to wake up some one, it must use an extra (besides the $2K$ accesses aforementioned) remote access to the RMW register in order to obtain the address. If P wakes up no one, then the system will be deadlock since every process is held waiting.

Theorem 2 *There is no mutual exclusion algorithm using RMW registers and atomic read/write registers that requires less than 2K+2 remote accesses in any interval (within a busy period) with K life cycles among which some process completes more than one life cycle.*

$< proof >$ The k life cycles within a busy period entails a chain of privilege passing from the first exit region to the K-th exit region. By way of contradiction, assumes that only $2K + 1$ remote accesses are needed for the complete chain of K cycles with at least one process completes two life cycles. One life cycle requires at least 2 remote writes: one in the trying region and another in the exit region. Under the constraint of 2 remote writes for one life cycle, a process must have announced its q-node address when it access the RMW register in the trying region, and it must have used a remote write to wake up its successor in the chain. Let P be a process that appears twice in the chain. The assumption that P is able to finish the first cycle entails that one extra remote write has been spent, as we understood in the proof of the previous theorem. Now, there are no more extra re-

mote writes besides the $2K$ accesses. For P to be allowed in the critical section for the second time, its predecessor must have used a remote write to wake up P. The predecessor's predecessor must have used a remote write to wake up P's predecessor. The argument goes on and we have that the first occurrence of P must have used a remote write to wake up its successor. For one process to know where to write a wake-up message, the successor must have written its q-node address in a RMW register, and the predecessor must have somehow read the address from a RMW register. Therefore, the address-write happened before the address-read. A contradiction is inevitable since then it entails that the address-write in the second occurrence of P happened before the address-read in the first occurrence of P. A process's current incarnation cannot happen before its previous incarnation.

Theorem 3 *The permission word algorithm requires 2K+2 remote accesses in any interval (within a busy period) with K life cycles among which some process completes two life cycles.*

$< proof >$ All remote accesses are either to the RMW registers or write operations to read/write registers. There are no remote reads from read/write registers in this algorithm.

In an interval within a busy period with K life cycles among which no process completes more than *one* life cycle, the algorithm requires $2K + 1$ remote writes: one at T2 for announcing the q-node address and one at either E3 or E6 to wake up some successor. Only one process needs to use an extra remote access at E1: the controller.

In an interval within a busy period with K life cycles among which some process completes *two* life cycles but none completes more than *two*, the algorithm uses $2K + 2$ remote accesses. Two executions of E1 are sufficient to allow some process to complete two life cycles.

4 Conclusions

Three algorithms have been presented in a sequence and each is shown to have better quality than the previous one. The first removes the deadlock error from its previous one. The second eliminates the starvation unfairness from its previous one. In fact, it guarantees bounded bypass in the exit protocol: when a process P wishes to exit, no process can exit more than two times before P is allowed to do so. The third reduces the number of remote accesses required in a busy period to such extend that any further reduction is impossible. It also enjoys better fairness in exit protocol than the second one does.

References

[1] T. L. Huang. Letter(Correction to the array-link-based distributed lock). *IEEE Parallel and Distributed Technology*, 2(3): 3–4, Fall 1994.

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

[3] T. L. Huang and C. H. Shann. A comment on a circular list-based mutual exclusion scheme for large shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 9(4): 414–415, April 1998.

[4] Nancy A. Lynch. *Distributed Algorithms*, Morgan Kaufmann Publishers, 1996.

[5] X. Zhang, R. Castaneda, and E. W. Chan. Spin-lock synchronization on the Butterfly and KSR1. *IEEE Parallel and Distributed Technology*, 2(1): 51–63, Spring 1994.

[6] Leslie Lamport. The mutual exclusion problem – Part I and II. *Journal of the ACM*, 33(2): 313–348, April 1986.

[7] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1): 1–11, Feb. 1987.

[8] S. S. Fu and N.-F. Tzeng, "A circular list-based mutual exclusion scheme for large shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 6, pp. 628-639, June 1997.