Fast and fair mutual exclusion for shared memory systems *

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

Two fast mutual exclusion algorithms using read-modify-write and atomic read/write registers are presented. The first one uses both compare&swap and fetch&store; the second uses only fetch&store. Fetch&store are more commonly available than compare&swap. It is impossible to obtain better algorithms if "time" is measured by counting remote memory references. We were able to maintain the same level of performance with or without the support of compare&swap. However, fairness is degraded from 1-bounded bypass to lockout freedom without the support.

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, for large shared-memory multiprocessors, that more or less follows their ideas of busy waiting on local memory locations only. Zhang et al. [6] has similar algorithms. Herlihy et al. [11] used the MCS algorithm as the backbone for a lock-based concurrent counting primitive. Recently Fu and Tzeng[10] presented a circular list-based mutual exclusion scheme (referred to as **CL algorithms** in this article) also for large shared-memory multiprocessors. All of these studies did make an attempt to provide algorithms using only fetch&store since compare&swap is not commonly available in production-quality multiprocessors. Bershad [12] indicated that only two out of eight production-quality shared memory multiprocessors have a processor which implements compare&swap. We should note that the memory system does not necessarily support the primitive even when the processor does. That may explain why almost all studies of fast mutual exclusion algorithms using read-modify-write primitives include an alternative version that uses only fetch&store, and conduct performance evaluation based on that version.

The success of MCS algorithms and CL algorithms is largely due to the elimination of busy waiting (with unpredictable number of memory references) on remote memory locations. Unfortunately, the fetch&store version of MCS algorithm is not fair at all: it suffers from starvation. The major merit of CL algorithm is the elimination of remote memory references needed in MCS algorithm to re-direct an address link for each privilege passing during resource busy period. While it provides considerable performance improvements over MCS, the CL algorithms (with or without the support of a powerful primitive similarly to compare&swap) suffer from the following drawbacks:

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

In this article, two algorithms that follow the line of CL algorithms but suffer from neither of the drawbacks are presented. The first one uses compare&swap and fetch&store; the second one uses only fetch&store. Furthermore, each one is proved optimal in minimizing the number of remote memory references required for a self-scheduling unit of the requesting processes, and in guaranteeing the best possible fairness under the constraints. We show that any further improvement beyond what is achieved by the algorithms is impossible.

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

The rest of the paper is organized as follows. Section 2 provides definitions and models. Section 3 presents the

*This work was supported by National Science Council, Republic of China, under Grant NSC88-2213-E-009-014

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

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

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

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

three fast algorithms. Section 4 gives some lower bounds. Section 5 is the conclusion.

2 The mutual exclusion problem

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 on 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 on variables that satisfy the following condition:

(1) All references are to private variables;
(2) At most one reference is to a shared variable; or
(3) Multiple references to a shared variable via exactly one execution of a RMW primitive.

If the set of references 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 since it reduces a large number of possibilities of interleaving among the processes that would have to be dealt with otherwise. Several state transitions in a rule-abiding node can be lumped together as one transition in our correctness arguments. Note that full advantages of such lumping may not have been taken in all flowcharts of this article. But the rule is always observed; no node needs to be split.

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

2.3 Flowcharts as mutual exclusion algorithms

Formal definition of mutual exclusion problem can be found in [8]. Lynch [5] gave a more succinct definition and introduced several impossibility results in the mutual exclusion problem that uses 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 that 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 an exclusive set. We found it easier to argue for the entire exclusive set than to do so for the critical region directly.

For mutual exclusion algorithm to meet *deadlock freedom* requirement, both progress for the trying region and progress for the exit region must hold. That is, at any point in a *low-level fair execution*, (1) if at least one process is in T region and no other process is in C region, then at some later point some process enters C region; and (2) if at least one process is in E region, then at some later point some process enters R region.

The abovementioned requirements are necessary for a mutual exclusion algorithm to be correct. It is often desirable to have some confidence in the level of fairness in accessing critical region for each individual process. The **first-in-first-out (FIFO)** order is the most stringent requirement. It is not clear what kind of applications would demand such strong fairness in accessing critical regions.

For most applications, **bounded bypass** is good enough. If the algorithm guarantees that a requesting

process cannot be bypassed by any certain process in entering critical region for more than b times, we say b-bounded bypass for C region is assured. Likewise, if the algorithm guarantees that an exiting process cannot be bypassed by any certain process in entering remainder region for more than b times, we say b-bounded bypass for R region is assured. For many mutual exclusion algorithms, the logical structures in exit regions are quite trivial, and therefore only bounded bypass for C region is discussed in some literature. This article particularly defines bounded bypass for R region since we found that almost all fast algorithms using RMW primitives have non-trivial logical structures in exit regions.

There are algorithms that cannot guarantee any bounded value on the number of bypasses, but are nevertheless **lockout free**: they guarantee that, assuming a low-level fair execution, no process can be kept waiting indefinitely either for C region or for R region.

The worst kind of fairness is **no fairness** at all (better known as starvation.) That is, an individual process may be kept waiting indefinitely for either C region or for R region.

3 The algorithms

Most optimal mutual exclusion algorithms aim at minimizing the size of the shared variables or minimizing the number of shared variables. Burns and Lynch [13] showed that n binary shared variables are necessary and sufficient to solve n-process mutual exclusion using only atomic read/write shared variables. Lycklama and Hadzilacos [14] presented an algorithm that satisfies the "first-come-first-served" property and requires only five shared bits per process. Styer and Peterson [15] established some tight bounds on the number of variables required for symmetric mutual exclusion problems. Few studies aim at minimizing the number of memory references. Yang and Anderson [16] gave an algorithm that assumes no read-modify-write primitives and therefore requires much more remote memory references. Lamport [9] tried to minimize the number of references in a period of no competing requests. We try to minimize the number of references in a period of frequent competing requests. We also take into account the difference between local memory and remote memory. Local memory access does not incur memory contention, while remote memory access does. Therefore, we count only the number of remote memory references. Minimizing remote references in a period of frequent competing requests serves a good purpose since memory contention in large shared-memory multiprocessor in such periods can lead to bad performance and remote memory access is a key factor of memory contention.

Figure 2 shows the data structure of the memory space that are allocated for each process in the mutual exclusion algorithm. The CL algorithm actually uses only one bit (the boolean *wait*) for the q-node. *Permission-word* is used by the permission word algorithms.

3.1 The CL algorithm without deadlock

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

Figure 2: Per process data structures for the algorithms.

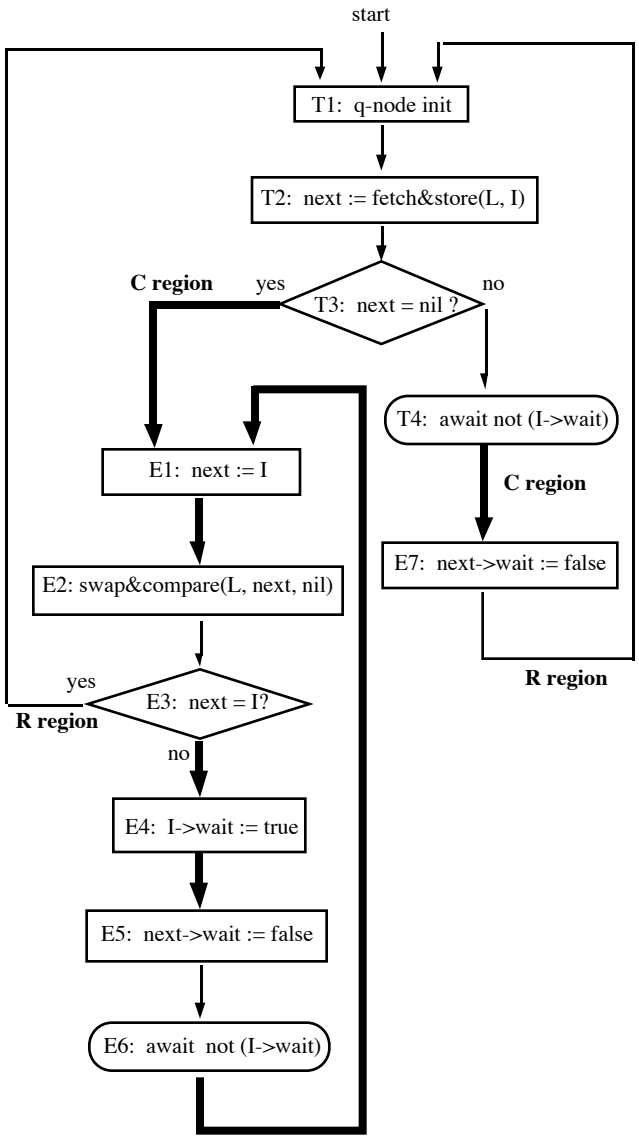

Figure 3: The CL mutual exclusion algorithm with deadlock error removed.

Figure 3 is the circular list-based mutual exclusion algorithm with the original deadlock error removed. Explanation of the algorithm follows.

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*.

Suppose process *P* starts to execute the algorithm. 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 *fetch&store* primitive is defined in Figure 1. T3 checks whether *P* is the first process that references L, either since system start-up or since the last event that the value *nil* was written back to L. If T3 answers "yes", *P* is entitled to enter critical section and all other competing processes are now waiting at T4 node.

The *swap&compare* primitive is defined in Figure 1. E1, E2 and E3 together take care of the followings. Two possibilities exist. If L is still pointing at *P*'s q-node ("yes" after E3), 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 ("no" after E3), there are some other processes that are waiting. *P* stores the value of L to the private variable *next* (as a result of E2) 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 had been granted permission to enter critical section 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 two new 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 The permission word algorithms

The main idea of the algorithms, see Figure 4 and Figure 6, 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 permission word. 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 5 is an example to help explain how it works.

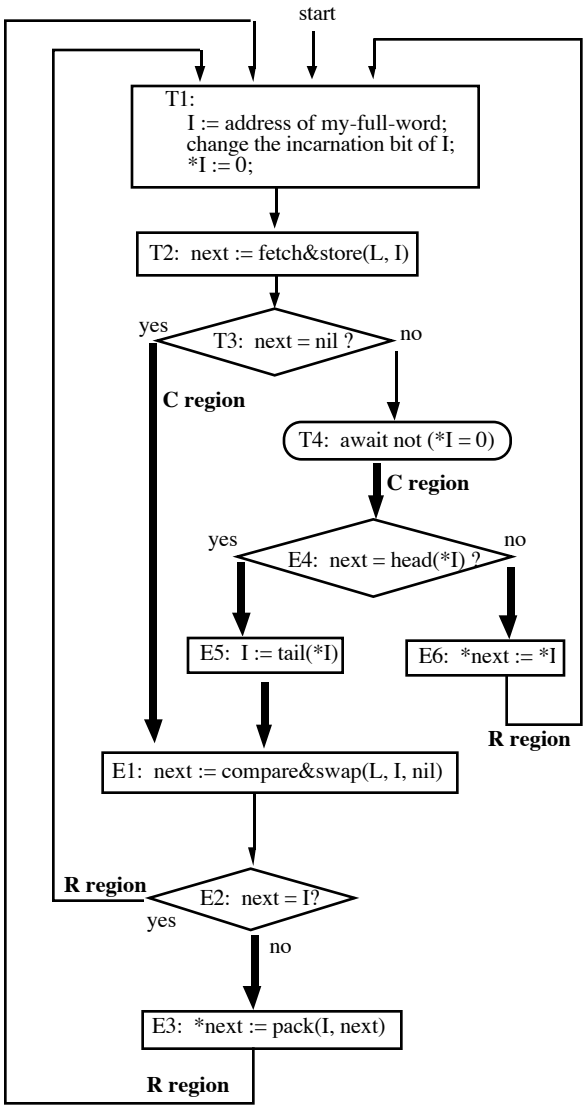

Figure 4: The permission word mutual exclusion algorithm using fetch&store and compare&swap.

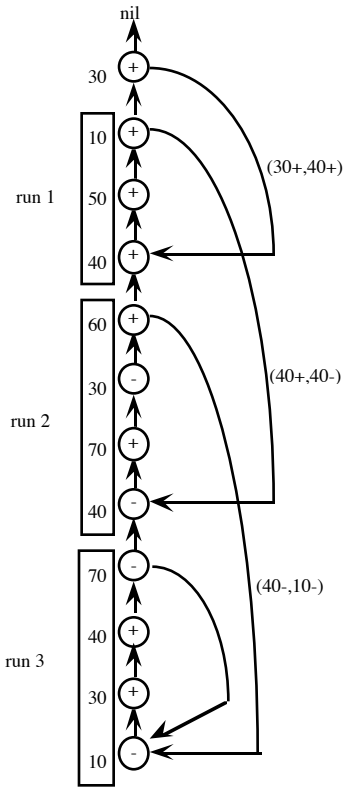

Figure 5: An example of a busy period consisting of 3 runs.

Figure 4 is used to explain how the permission word algorithms work. The other algorithm in Figure 6 works similarly. 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 5, 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.

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.

4 Impossibility results

The permission word algorithm is considered the best among a set of possible solutions to an extended mutual exclusion problem that aims to eliminate the overhead of link re-directions. A link re-direction is the synchronization mechanism that establishes a privilege passing chain among the processes in an order that respects the actual first-in-first-out (FIFO) order of the process request. The CL algorithm specifically seeks to avoid such link re-direction, since the FIFO order is not only non-essential in fairness requirements but also the cause of too high a cost in terms of remote memory access when link re-direction is implemented. The permission word contains enough information for a controller to identify itself as such and to take on the role without using extra synchronization messages. Past experiences showed that avoiding link re-direction save significant amount of remote references while fairness can still be kept at an acceptable level. The following definition captures the salient feature of the set of possible solutions that avoid link re-directions.

Definition 1 (decisiveness) *A mutual exclusion algorithm is decisive if no more than one remote reference is required in the trying region to determine whether the process should enter immediately or it should wait.*

In the algorithms using fetch&store, decisiveness can be expressed as a state formula:

$$(L = nil) \Longrightarrow \text{(no process is in critical region.)}$$

Thus, a requesting process is able to decide, using one reference to L, whether it is allowed to enter critical region right away or it should spin on a local variable whose value clearly indicates whether permission has arrived.

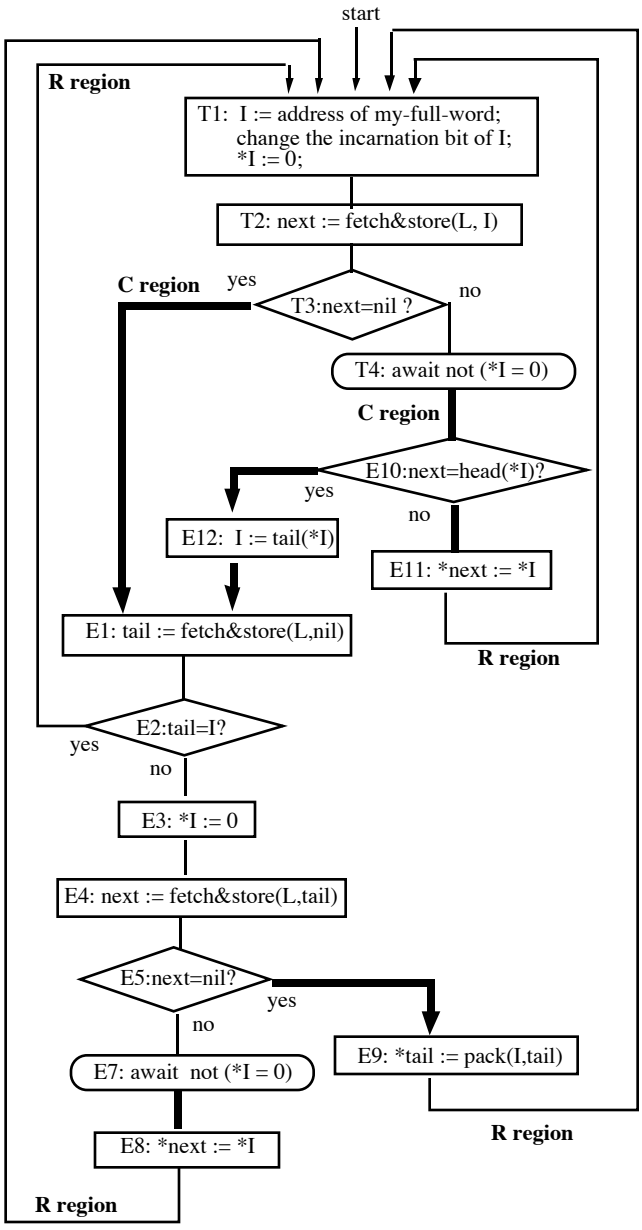

Figure 6: The permission word mutual exclusion algorithm using fetch&store only.

Definition 2 (cluster) *A cluster is the set of distinct processes that have made requests at about the same time and are hence scheduled by the same controller.*

A cluster is a self-scheduling unit in the decisive mutual exclusion algorithms that allow no link re-directions. A leader among the cluster must be selected to act as the controller for the cluster. Such controller inevitably uses some remote access overhead in its controlling task. The following is to establish a lower bound on the number of remote references required for a cluster controller to act properly.

4.1 Impossibility results assuming read-modify-write primitives

To prove that the permission word algorithm is optimal, we must prove that there is no algorithm that requires less remote references 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 (0-overhead impossibility) *There is no decisive mutual exclusion algorithm using RMW registers and atomic read/write registers that requires less than 2K+1 remote references for any cluster of K processes.*

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 references 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 $2K$ references 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. $Q.E.D.$

Conjecture 1 (FIFO impossibility) *There is no decisive mutual exclusion algorithm using RMW registers and atomic read/write registers that requires no link re-direction and guarantees first-in-first-out fairness.*

> *Observation.* It seems that link re-direction is inevitable in order to maintain FIFO order and to be decisive in entering critical section at the same time. There are algorithms [4] that maintain FIFO order at the expense of high overhead either in deciding the winner in the trying region or in complicated scheme of link re-directions.

Theorem 2 *The permission word algorithm which uses both compare&swap and fetch&store requires 2K+1 remote references for a cluster of K processes and guarantees 1-bounded bypass.*

Proof. 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 reference at E1: the controller.

Note that 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; thus 1-bounded bypass is guaranteed. $Q.E.D.$

4.2 Impossibility results when only fetch&store is available

Theorem 3 (1-overhead impossibility) *There is no decisive mutual exclusion algorithm using fetch&store primitives and atomic read/write registers that requires less than 2K+2 remote references for any cluster of K processes.*

Proof. There is no algorithm that requires less than $2K + 1$ remote references due to the 0-overhead impossibility result which was proved.

By way of contradiction, assumes that only $2K + 1$ remote references are needed for the complete chain of the K cycles in the cluster. The extra remote reference to the RMW register in the previous argument should now be examined in more detail since now we don't have the full power of general read-modify-write primitive. Rather, what is available is fetch&store only. The smallest number of remote references overhead for a controller of a self-scheduling cluster in the exit region is two, explained in the followings.

Case (1): If no fetch&store is used, then at least two ordinary references are needed. One is to obtain the address of some q-node from L, the other is to set L as *nil*. The controller cannot finish its controlling task using only the two ordinary references since there may or may not be requesting processes in the cluster that need to be scheduled. To handle them correctly in two references requires read-modify-write primitives.

Case (2): If fetch&store is used, then the first one must be of the form

$$\text{tail} := \text{fetch\&store}(\text{L}, \textit{nil}),$$

and if the variable *tail* does not have the same value as I (meaning there are requesting processes to be handled,) one more fetch&store is required in order to schedule the requesting processes properly. Note that the controller cannot foretell whether there will be requesting processes or not. It should anticipate both outcomes properly. This cannot be accomplished by using less than two remote references. Since these two references are control overhead, one more remote reference is still needed to actually wake up one process that is waiting. In total, we know at least three remote references are needed for the cluster controller in its exit region. Therefore, we need at least $2K + 2$ remote references for the complete chain of K cycles in the cluster. $Q.E.D.$

Theorem 4 (Bounded bypass impossibility) *There is no decisive mutual exclusion algorithm using only the fetch&store primitives and atomic read/write registers that guarantees bounded bypass.*

Proof. Suppose there is one such algorithm that guarantees a bounded bypass value B. We are to construct a "bad" sequence of events that leads to a contradiction. Let *p1, p2, p3* be the three processes that are about to request. Process *p1* requests and enters first while no one is requesting. When it is in exit region, it must set L as *nil* since no one is requesting. Then it enters remainder region. The system stays idle for a while. Then *p2* requests and enters critical section, and then leaves. The system stays idle for a while, again. Then *p3* requests and enters while no one is requesting. Since the algorithm is deadlock free, *p3* should be able to repeatedly enter and leave critical section for an unbounded number of times. Certainly it can enter and leave critical section for $B + 1$ times.

However, the sequence of events can be turned "bad" at the point just before *p1* sets L as *nil*. Since there is only fetch&store available, *p1* has no way of telling whether there is any process requesting at the point. It is perfectly legal to insert to this point a sequence of events that *p2* is requesting by accessing L. After setting L as *nil*, *p1* will be able to detect that there is some one requesting, and that it should try to reclaim the privilege in order to wake up some process that is waiting. However, once L has become *nil*, a **decisive** mutual exclusion algorithm should allow some other process to cut in immediately. And since the fair access to L only guarantees eventual access, there is no way to assure that *p1* will be able to access L before the $B + 1$ consecutive entering and leaving critical section has completed. A "bad" sequence that *p2* is bypassed by *p3* for more than B times can be constructed. $Q.E.D.$

Theorem 5 *The permission word algorithm which uses only fetch&store requires 2K+2 remote references for a cluster of K processes and guarantees lockout freedom for fairness.*

Proof. Observe that at most three remote writes are required for a controller to complete the exit region before it enters the next remainder region: path (E1), path (E1, E4, E8), or path (E1, E4, E9). The cost of E8 and E9 can be considered as the inherent cost to wake up some other process. Hence, only two remote references are control overhead for the cluster.

Observe that a process executing in the exit region is bound to enter remainder region since all the *await* statements are terminating. Hence, lockout freedom is guaranteed. $Q.E.D.$

5 Conclusions

5.1 Related work

Dwork et al. [7] gave a formal model of memory contention and established a tight bound $\Theta(1)$ on memory contention incurred by one life cycle for a sub-problem of mutual exclusion called one-shot mutual exclusion. Our algorithms, like MCS and CL algorithms, are in the same complexity class. Asymptotically, all these algorithms are optimal since a lower bound of a sub-problem must be a lower bound of the full problem. The measure of memory contention is very similar to the measure of counting remote memory references, but not exactly the same. For the same execution, the measure of memory contention is less than or equal to the measure of remote references. In this sense, our measure is more stringent than memory contention. Our lower bounds in the impossibility results are exact. Theirs are asymptotic.

Yang and Anderson [16] gave a $\mathcal{O}(log\ n)$ time mutual exclusion algorithm without using any read-modify-write primitive. If the bound can be proved tight, read-modify-write primitives are instrumental in cutting down memory access requirement.

5.2 Our contribution

The two algorithms presented, with or without compare&swap, are optimal in terms of remote references and of fairness. We show that lack of compare&swap causes the degradation from 1-bounded bypass to lockout freedom in fairness.

The algorithms did not include software combining trees but can be easily modified to do so, in a manner similar to what the original CL algorithm did. While it is believed that combining tree helps reduce hot spot contention, it nevertheless adds more steps in the trying region and therefore may introduce contention along the way. It remains unclear whether adding combining trees actually results in better performance in practice.

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] T. L. Huang and J. H. Lin. An assertional proof of a lock synchronization algorithm using fetch_and_store atomic instructions. In *Proceedings of the 1994 International Conference on Parallel and Distributed Systems*, Dec. 1994, pp. 759–768. IEEE Computer Society.

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

[6] 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.

[7] Cynthia Dwork, Maurice Herlihy and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6): 779–805, November 1997.

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

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

[10] 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.

[11] Maurice Herlihy, Beng-Hong Lim and Nir Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13(4): 343–364, Nov. 1995.

[12] Brian N. Bershad. Practical considerations for lock-free concurrent objects. *Technical report CMU-CS-91-183*, School of Computer Science, Carnegie-Mellon University, Pittsburg, PA 15213, U.S.A, Sep. 1991.

[13] James Burns and Nancy Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2): 171–184, December 1993.

[14] Edward Lycklama and Vassos Hadzilacos. A first-come-first-served mutual-exclusion algorithm with small communication variables. *ACM Transactions on Programming Languages and Systems*, 13(4): 558–576, October 1991.

[15] Eugene Styler and Gary L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proceedings of the Eight Annual ACM symposium on principles of distributed computing*, pp. 177–191, The ACM SIGACT, ACM Press, 1989.

[16] Jae-Heon Yang and James H. Anderson. Fast, scalable synchronization with minimal hardware support. In *Proceedings of the 12th Annual ACM symposium on principles of distributed computing*, pp. 171–182, The ACM SIGACT, ACM Press, 1993.