1998 International Computer Symposium
Workshop on Algorithms
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

FAST AND FAIR MUTUAL EXCLUSION ALGORITHMS USING FETCH&STORE PRIMITIVE *

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

Although the compare&swap primitive and the fetch&store primitive are important for fast mutual exclusion algorithms, a large portion of production-quality multiprocessors does not support the former. A sequence of fast mutual exclusion algorithms using only fetch&store is shown to have qualitative improvements from one to the next. The last algorithm is optimal in minimizing the number of remote memory accesses for a self-scheduling unit and in guaranteeing the best possible fairness under such circumstances. We prove that no better algorithms exist. Minimizing the number of remote accesses serves a good purpose since remote access contributes to memory contention problem in large shared-memory multiprocessors. The results show that we were able to maintain the same level of performance with or without the support of compare&swap. However, fairness is degraded from 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. [5] has similar algorithms. Herlihy et al. [9] used the MCS algorithm as the backbone for a lock-based concurrent counting primitive. Recently Fu and Tzeng[8] presented a circular list-based mutual exclusion scheme (referred to as CL algorithms in this paper) 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 [10] 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 accesses) 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 access 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, a sequence of algorithms using only fetch&store that follows the line of CL algorithms but suffer from neither of the drawbacks is provided. Furthermore, each one in the sequence has some improvements over the previous one, and the last one is proved optimal in minimizing the number of remote memory accesses required for a self-scheduling unit of the requesting processes, and in guaranteeing the best possible fairness under such circumstances. We show that any further improvements beyond what is achieved by the algorithm is impossible.

In addition to the read-modify-write shared variable that is accessed by the fetch&store primitive, the algorithms require N atomic read/write shared variables

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

```
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: Compare&swap and Fetch&store primitives.

(called q-nodes later in this paper), 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 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 of this article. But the rule is always observed; none of the nodes 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 are thick. If the condition is not satisfied, either the algorithm itself is wrong or the selection is wrong.

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 user 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. 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 is nevertheless **lockout free**: it 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, of course, **no fairness** at all. That is, an individual process may be kept waiting indefinitely for either C region or for R region.

3 The fast mutual exclusion algorithms using fetch&store only

Most optimal mutual exclusion algorithms aim at minimizing the size of the shared variables or minimizing the number of shared variables. Burns and Lynch [11] showed that n binary shared variables are necessary and sufficient to solve n-process mutual exclusion using only atomic read/write shared variables. Peterson [12] provided a nearly optimal algorithm, allowing processes to fail, uses four values of shared memory per process, which is within one value of the known lower bound. Lycklama and Hadzilacos [13] presented an algorithm that satisfies the "first-come-first-served" property and requires only five shared bits per process. Styer and Peterson [14] established some tight bounds on the number of variables required for symmetric mutual exclusion problems. Few studies aim at 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 contention, 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 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.

```
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.

Figure 2 shows the data structure of the memory space that are 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.

3.1 A deadlock-free CL algorithm using fetch&store only

Figure 3 is an improved version of the circular list-based mutual exclusion algorithm using only fetch&store with the original deadlock error removed and with the original schematic diagram expanded to the full algorithm in flow chart.

For readers who are not familiar with the circular list-based algorithms and for those who will read all the algorithms in this paper 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 it *seems* that no other processes are interested in entering critical sections, which will be explained in more details. If T3 answers "yes", P is entitled to enter critical section and all competing processes are now waiting at T4 node.

E1, E2 and E3 together take care of the followings. If L is pointing at some other process's q-node, P should be prepared to wake up one of the requesting processes, which is *tail1* as returned by the *fetch&store* of E1. E3 is the preparation that is needed. E4 is to wake up that process. E5 is to wait until the chain of privilege passing come back to P. Then P should go back to

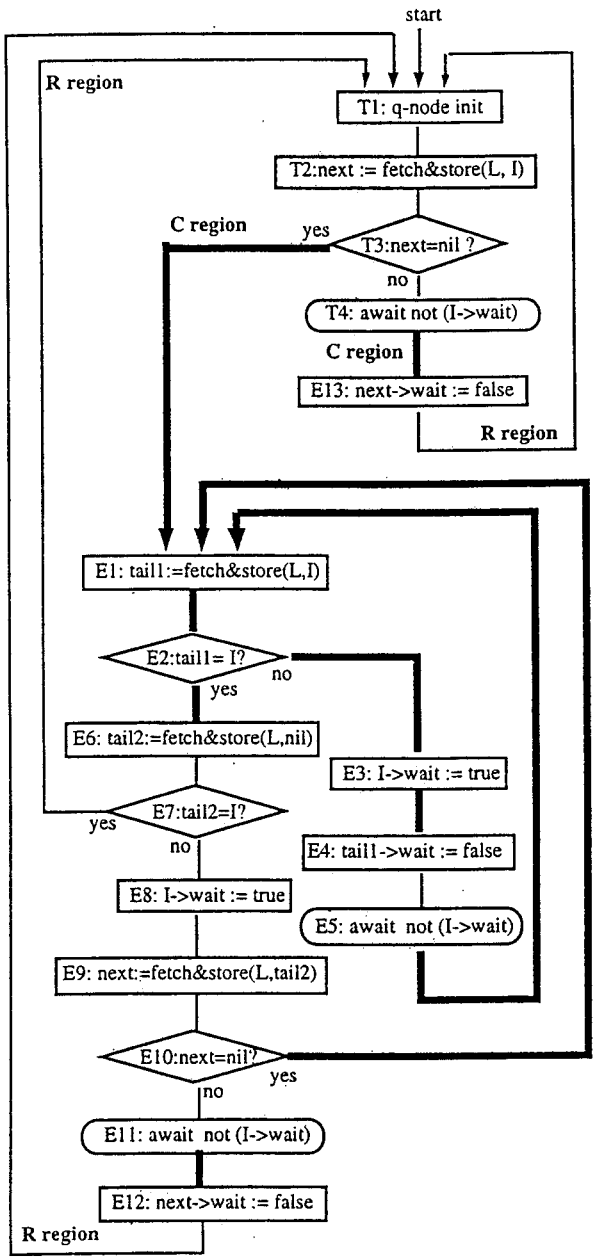

Figure 3: The CL mutual exclusion algorithm using only fetch&store with the deadlock error removed.

E1 and repeats the role of playing controller for other processes.

E1, E2 and E6 together take care of the followings. If L is still pointing to P 's q-node, it *seems* that no other processes are interested in entering critical sections. E6 is to make such an assumption and go ahead to assign L as *nil*. The assumption may be correct, in which case the result of E7 will be "yes," and then P has nothing else to do in the exit region. The assumption may be wrong, in which case the result of E7 will be "no." The processes that are requesting should not be left forgotten by the controlling process, or starvation will arise. Since there is no powerful primitive like *compare&swap* available, there is no way to avoid such kind of guess work. Fortunately, with the help of the *fetch&store* of E9, the list of the requesting processes will be inserted back to the mainstream that possesses the control privilege. If the control privilege has come back to P immediately at E9, the test at E10 will be "yes," and P should act as the controller again. If the control privilege is still at some other process, P has already done the right thing for the requesting processes: *tail2* has already been pointed to by L as the result of E9. The control privilege will eventually come to *tail2* and will wake up in sequence each of the requesting processes in the list. E11 will then terminate and P should do nothing but to pass the privilege to the next (as in E12) in the mainstream.

Note that E8 should precede E9 and E3 should precede E4 in execution, or deadlock may occur. Details of the deadlock error in the original CL algorithm can be found in [3].

Note also that P will be kept indefinitely in the cycle of playing controller as long as either there are processes interested in entering critical section (with the "no" result at E2,) or the control privilege has been reclaimed successfully (with the "yes" result at E10.) Later, we will show two other algorithms that suffer from no such severe unfairness.

3.2 A lockout-free CL algorithm using fetch&store only

The algorithm, see Figure 4, aims to avoid the starvation unfairness imposed on the controller process. The main idea is to use two more bits in each q-node to transfer the controller role from the current process 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 appointed next controller at E15 until the current controller finishes a complete run and executes E9. The initialization at T1 is to set *direct* false and *hold* true. The algorithm is largely the same as the previous one, except the necessary changes involving the role transfer using the two bits. It is easy to observe from the flowchart that no process will be kept in E region indefinitely provided that the *await* statements at E11 and E8 terminate. A process cannot be kept indefinitely at the *await* statement at E8 because the *fetch&store* primitive regulates the q-node address sharing in such a way that the wake-up signal sent by P at E7 is bound to come back in finite steps as a signal to release P at E8. Similarly, a process cannot be kept indefinitely at the *await* statement at E11 because the process had

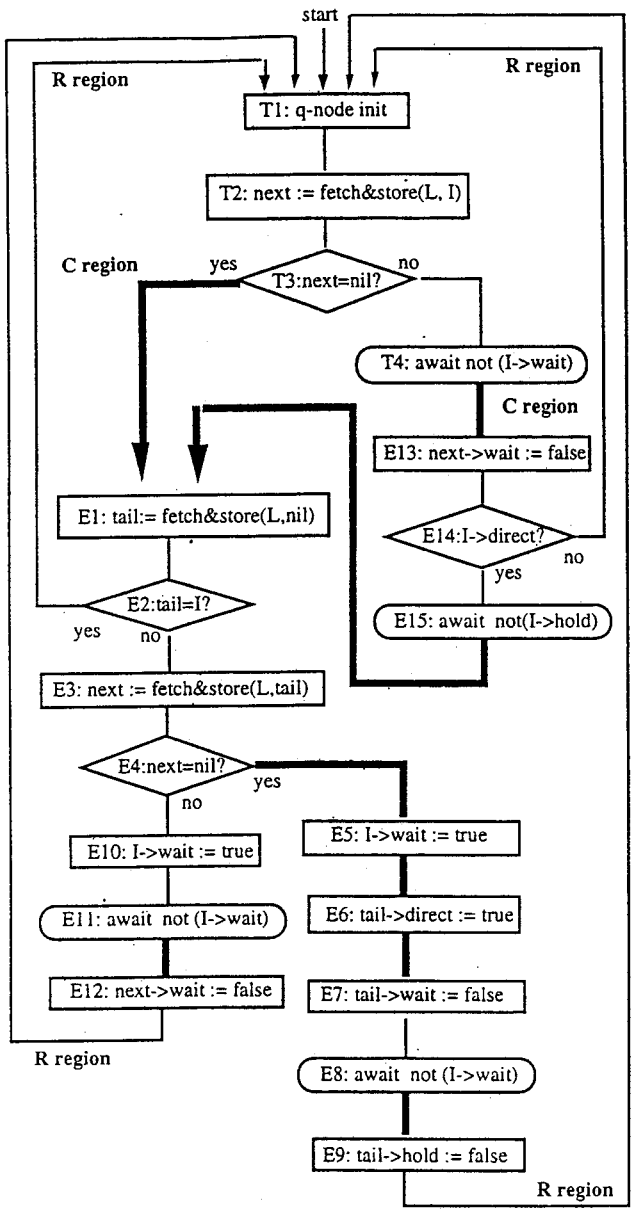

Figure 4: The lockout-free CL mutual exclusion algorithm using fetch&store and bit messages.

just inserted the list of requesting processes into the mainstream which must have the control privilege going. Eventually, the privilege will come to release the busy waiting at E11. Therefore, lockout freedom of the controller process in E region is assured. However, the number of bypasses observed by the controller process in E region cannot be bounded mainly because fairness in accessing L only guarantees eventual access. Later, we will prove that there in no *decisive* mutual exclusion algorithm (the definition of which is given in section 3.4) using only *fetch&store* that guarantees bounded bypass.

3.3 The permission word algorithm

We establish a **tight bound** on the number of remote accesses required for the mutual exclusion problem allowing only the *fetch&store* primitive. A lower bound is shown by an impossibility theorem in the next section. In addition, we establish a **tight bound** on the best possible fairness one can expect in algorithms under the circumstances that only the fetch&store primitive is available and that the algorithm must be decisive. Again, the impossibility theorem for fairness is shown in the next section. This section provides an algorithm that delivers both of the lower bounds.

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 half words, *(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. Reading the previous two algorithms also help.

A **busy period** is an execution sequence that starts with the first process making a request, and ends with some process deciding correctly that there is no request pending. We are interested in busy periods in which two or more requests have been made and served. A **run** in a busy period is an execution sequence that starts with a process executing E9 and ends with some process executing E9, 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 E9 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 E9, 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.

Since the least significant bit of an address is not used in most computer architecture, we can use that

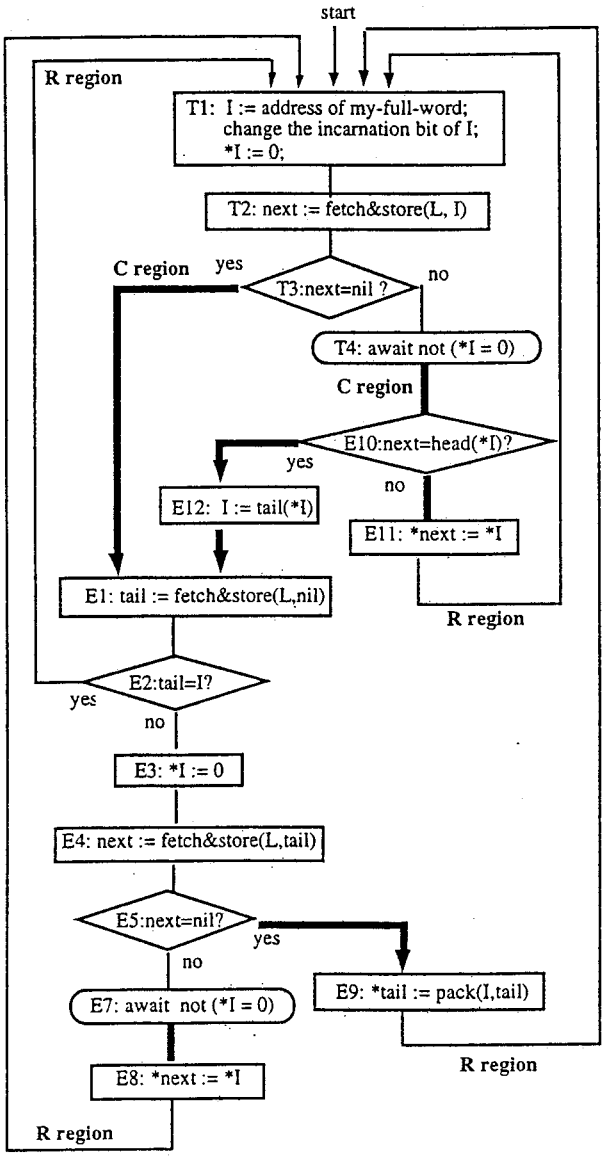

Figure 5: The permission word mutual exclusion algorithm using fetch&store.

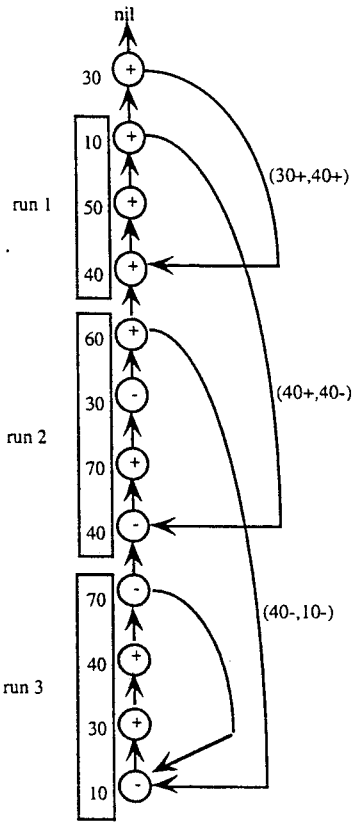

Figure 6: An example of a busy period consisting of 3 runs.

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 E10, which is to check whether the value of *next* equals the *head* half word 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 acting as the controller after receiving the permission word (40-, 10-). 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 E10.) Therefore, it should go to E12 to act as the controller. The subtlety occurs whenever the *tail* process of E9 having been given permission to enter critical section did so and then quickly made a new request (at T2) in the next run. Such subtlety arises since we don't require a controller to be appointed earlier and then be held waiting for the release signal which is sent only when the old controller actually transfers its privilege. Fortu-

nately, the subtlety needs to be resolved only between two neighboring runs, thus a single bit suffices.

3.4 Two 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 accesses 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 access 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 access 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.

Definition 2 (cluster) *A cluster is the set of processes that has made requests in an interval between two consecutive states with (L = nil).*

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 accesses required for a cluster controller to act properly.

Theorem 1 (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 accesses for any cluster of K processes.

< *proof* > By way of contradiction, assumes that only $2K$ remote accesses are needed for the complete chain of the 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 announce its q-node address when it access the RMW register in the trying region, and it must use a remote write in the exit region 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 rest of the cluster will be deadlock since each such process is held waiting. Thus, a contradiction is found.

By way of contradiction, assumes that only $2K + 1$ remote accesses are needed for the complete chain of the K cycles in the cluster. The extra remote access 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 accesses 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 accesses 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 two ordinary accesses since there may be requesting processes in the cluster that need to be schedule properly.

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

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

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 accesses. Since these two accesses are control overhead, one more remote access is still needed to actually wake up one process that is waiting. In total, we know at least three remote accesses are needed for the cluster controller in its exit region. Therefore, we need at least $2K + 2$ remote accesses for the complete chain of K cycles in the cluster. Q.E.D.

Theorem 2 (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 3 *The permission word algorithm requires $2K+2$ remote accesses 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 accesses 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.

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. It guarantees only lockout freedom, however. We show that a better fairness is impossible if only fetch&store is used. (A related study by the author showed that when we are given the additional support of compare&swap, fairness can be improved to be bounded bypass.) The third (and the last) algorithm reduces the number of remote accesses required in a self-scheduling unit to such extend that any further reduction of remote access is impossible. It also maintains the same fairness as 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.

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

[10] 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.

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

[12] Gary L. Peterson. A new solution to Lamport's concurrent programming problem using small shared variables. *ACM Transactions on Programming Languages and Systems*, 5(1): 56–65, January 1983.

[13] 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.

[14] 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.