A Fair and Space-efficient Mutual Exclusion Using *read/write* and *fetch&store* Primitives[*]

Sheng-Hsiung Chen[†] and Ting-Lu Huang
Dept. Comp. Sci. & Info. Engr.
National Chiao Tung University
Hsinchu, Taiwan, R.O.C.
{chenss,tlhuang}@csie.nctu.edu.tw

Abstract

For an asynchronous shared memory model with only *read/write* primitives, Burns et al. show that at least n shared variables are required to solve n-process mutual exclusion problem. With a commonly available hardware primitive *fetch&store* in addition to *read/write*, we present a bounded-bypass mutual exclusion algorithm with constant 2 shared variables. Our algorithm achieves the limit of these primitives to reduce the space requirement since we prove that it is impossible to obtain bounded-bypass algorithms with less than 2 shared variables using the same primitives. Although there are several algorithms with only 1 shared variable in literature, they use hypothetical read-modify-write primitives which are not supported by any modern multiprocessor architectures. This paper first reveals the data requirement of bounded-bypass mutual exclusion using *read/write* and *fetch&store* primitives.

Keywords: mutual exclusion, atomic instructions, shared-memory systems, fairness

1 Introduction

The mutual exclusion problem is fundamental to resource allocation in asynchronous shared memory systems. A mutual exclusion algorithm provides exclusive accesses to a common nonshareable resource among processes. Using basic *read/write* operations, many researchers have devised mutual exclusion algorithms, such as Dijksta [5], Burns [2], Peterson [17], Lamport [11], etc. Commonly, all of these algorithms use at least n shared variables, where n is the number of processes. This is not an accident. Burns and Lynch [4] have shown that n-process mutual exclusion problem cannot be solved at all with fewer than n shared variables if only *read/write* operations are available.

The lower bound n exhibits some inherent drawbacks. If processes can be created and destroyed dynamically, the number n may be unknown. Moreover, the number n may be very large and therefore these algorithms are not suitable for space-limited systems such as embedded systems.

To relieve the above drawbacks by reducing the number of needed shared variables, some previous work assumed certain read-modify-write primitive, abbreviated as *RMW*, is available besides *read/write*. In one instantaneous step, a *RMW* primitive can access a shared variable and write back a new value according to the current variable value and the submitted function. Trivially, we can devise a simple mutual exclusion algorithm with only one shared variable, using a *RMW* primitive *test&set* additionally. The shared variable has value *false* initially. Any process tests the shared variable until it discovers the shared variable equals to *false*, at which time it immediately sets the variable as *true*. After accessing the resource, the process resets the shared variable as *false*. Unfortunately, this trivial algorithm does not guarantee any fairness among processes, that is, some process may not gain the resource always.

In literature, there are several algorithms using only one shared variable and guaranteeing certain level of fairness. Fischer et al. [7] devised a first-in-first-out algorithm. Burns et al. [3] devised a bounded-bypass algorithm and a starvation-free algorithm[1]. However, all of these algorithms used hypothetical *RMW* primitives which have never been implemented in any system. Modern multiprocessors generally include *RMW* primitives with simple submitted functions, such as *fetch&store*, *fetch&add*, *fetch&increment*, *fetch&decrement*, *compare&swap*, etc.

The main focus of this paper is to implement a fair mutual exclusion algorithm directly by commonly available primitives supported by modern multiprocessor architectures. We select *fetch&store* to implement our algorithm since it commonly appears at modern processors[2] and widely be used to design al-

[*]This work is supported by National Science Council, Republic of China, under Grant NSC 92-2213-E-009-064.

[†]Corresponding Author: Sheng-Hsiung Chen

[1]Indeed, their work aimed at theoretical discussion between data requirements and different fairness conditions.

[2]For instance, the "XCHG" instruction in a series of processors of Intel and AMD. The instruction set can be got at http://www.intel.com and http://www.amd.com.

gorithms for multiprocessor systems [16, 8, 9]. We present a 2-bounded-bypass algorithm with 2 shared variables, using a hardware primitive $fetch\&store$ as well as *read/write* in our asynchronous shared memory model. Our algorithm has the following advantages.

1. The algorithm is 2-bounded-bypass. We say that an algorithm satisfies b-bounded bypass if a requesting process cannot be bypassed by any certain process in accessing the resource for more than b times.

2. 2 shared variables are used. Without any hypothetical primitives, our algorithm requires only one more shared variable than those algorithms needing hypothetical primitives.

Our algorithm is inspired by the circular list-based mutual exclusion algorithm proposed by Fu and Tzeng [8, 10]. Informally, a mutual exclusion algorithm is to arrange competing processes in order for accessing the resource. We organized the order to enter the critical region among processes via a shared variable and a $fetch\&store$ primitive.

In addition, we show that it is impossible to obtain bounded-bypass algorithms with less than 2 shared variables using $fetch\&store$ as well as *read/write* primitives. Due to this impossibility result, our algorithm achieves the limit of these primitives to reduce the space requirement. We use the proving strategies proposed by Burns and Lynch [4] in our impossibility result. Their method is for *read/write* primitive only. We extend their method for $fetch\&store$ to gain our impossibility.

Other recent work on mutual exclusion has focused on designing local-spin algorithms by hardware primitives for distributed shared memory systems in which shared memory physically distributed to all processors [16, 8, 10, 9, 1]. In local-spin algorithms, all busy waiting is by repeatedly testing a locally-accessible spin variable without a processor-to-memory interconnection traversal. These algorithms aim at minimizing the number of required remote memory accesses. Since each process must have a shared variable which is locally-accessible by itself, at least n shared variables are needed, where n is the number of processors. These algorithms are suitable for distributed shared memory systems because each process has its own local shared memory. However, if we concern centralized shared memory systems in which all memory references must traverse the processor-to-memory interconnection, counting remote memory accesses doesn't make sense since each memory access is remote, that is, local-spin algorithms take no advantage and use at least n shared variables. In such centralized shared memory systems, our constant space algorithm is better in terms of space complexity.

The rest of the paper is organized as follows. Section 2 provides the system model and definitions about the problem. In section 3, we present our algorithm. Section 4 gives a impossibility result. Finally, section 5 is the conclusion.

2 System model and Definitions

2.1 Asynchronous Shared Memory Model

We adopt the asynchronous shared memory model [14] using I/O automata [15] since it is easy to specify which portions of the code comprise indivisible steps under this model. We slightly modify the model such that processes communicate by means of instantaneous primitives to shared variables, but not by means of external events in the original model.

The system is modelled as a triple (P, V, δ), where P is a nonempty finite set of processes, V is a nonempty finite set of shared variables, and δ is a transition relation for the entire system. Each process i is a kind of state machine with the following elements.

- $state_i$: a set of state;
- $start_i$: a subset of $state_i$, indicating the start states;
- $step_i$: a set of steps, describing the activities in which it participates.

These steps are classified as either *input*, *output*, or *internal* steps. An internal step may involve the shared memory. If it does, we assume that it only involves one shared variable.

A system state is a combination of states for all processes and values for all shared variables. The transition relation δ is a set of (s, π, s'), where s and s' are system states, and where $\pi \in step_i$ for some process i.

The transition relation δ has some locality restrictions. That is, if step π associated with process i does not involve any shared variable, only the state of process i can be involved. In contrast, if π involves a shared variable x, only the state of process i and the value of x can be involved. We assume that whether a shared memory action is enabled depends only on the process state and not on the value of the shared variable.

A step π is enabled for system state s if there exists s' such that $(s, \pi, s') \in \delta$. The system is input-enabled, that is, for every system state s and input step π, there exists s' such that $(s, \pi, s') \in \delta$. The above assumption is made since the input steps are controlled by arbitrary external users. In comparison, the internal and output steps are locally controlled by the system itself.

An *execution fragment* is defined as an alternating finite or infinite sequence, $s_0, \pi_1, s_1, \ldots$, consisting of system states alternated with steps, where successive (state, step, state) triples satisfy the transition relation. An *execution* is an execution fragment whose s_0 is a starting system state in which each process i's state is in $start_i$. A system state s' is reachable from system state s if there exists a finite execution fragment $s, \ldots, s'$. We want to exclude a situation that a process has no chance to take a step, when some locally controlled step is enabled. Thus, we define the following *low-level fairness* condition.

1. If the execution is finite, then no locally controlled action for each process is enabled.

2. If the execution is infinite, then for each process i there are either infinitely many occurrences of locally controlled steps of i, or else infinitely many places where no such step is enabled.

2.2 The primitives

Under this model, shared variables will be accessed by processes through atomic primitives. In this paper, *fetch&store* and *read/write* primitives are involved. To prove a impossibility result later, we formally define these primitives as follows.

A *fetch&store* primitive is defined below.

```
fetch&store(constant function $u$, variable $v$)
        $previous := v$
        $v := u$
        return $previous$
```

The second parameter v denotes the variable it involves, equipped with an arbitrary set V_v of values. The first parameter is the submitted function of this *RMW* primitive. The submitted functions of a *fetch&store* primitive are restricted to constant functions. Let value u also denotes a constant function, where $u : V_v \to \{u\}$. In one instantaneous *fetch&store* operation on v, it returns the prior value of v and assigns value u into v. Note that once a *fetch&store* primitive is enabled, the value to update variable v is determined no matter what the current value of v is.

Two other basic primitives are *read* and *write*. A *read* primitive, *read*(variable v), returns the value of v atomically. A *write* primitive, *write*(value u, variable v), updates the value of variable v as u atomically. A *write* primitive can also be treated below.

```
write(constant function $u$, variable $v$)
        $v := u$
```

That is, once a *write* primitive is enabled, the value to update variable v is also fixed.

2.3 The Problem

A formal definition of mutual exclusion problem can be found in [12, 13]. In order to consist with the asynchronous shared memory model using I/O automata, we still adopt the definition of the problem given by Lynch [14].

The mutual exclusion problem is to devise protocols for n users, $U_1, \ldots, U_n$, to control accesses to a designated region of code called the *critical region.* The users can be thought of as application programs and such code might manipulate a common nonshareable resource that requires exclusive accesses.

In order to gain the permission to its critical region, a user executes a *trying protocol*, and when the user leaves its critical region, it executes an *exit protocol* and returns to the remainder of its code, called the *remainder region.* Each user obeys an endless loop of life cycle: remainder (R) region (initially), trying (T) region, critical (C) region and exit (E) region.

Considering the mutual exclusion problem within the asynchronous shared memory model, we assume that the system contains n processes, numbered $1, \ldots, n$, each corresponding to one user U_i. Each process i acts on behalf of user U_i. The input to process i are the try_i step, which models a request by user U_i for admission to enter its C region, and the $exit_i$ step, which models a notification U_i has finished its C region. The outputs of process i are $crit_i$, which models the granting to U_i for entering its C region, and rem_i, which informs U_i that it can enter its R region.

We classify each process i in an execution into the following regions, according which events i is in between.

- *remainder region*: initially and in between any rem_i event and the following try_i event.

- *trying region*: in between any try_i event and the following $crit_i$ event.

- *critical region*: in between any $crit_i$ event and the following $exit_i$ event.

- *exit region*: in between any $exit_i$ event and the following rem_i event.

Finally, an algorithm which specifies the actions of each process solves the mutual exclusion problem must meet the conditions below.

Mutual exclusion: In any execution, there is no reachable system state in which more than one process is in its critical region.

Progress: 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.
2. If at least one process is in E region, then at some later point some process enters R region.

The above-mentioned requirements are necessary for a mutual exclusion algorithm to be correct. However, there is no guarantee that the critical region is granted fairly to different individual process, i.e. lockout (also known as starvation) may occur. Thus, it is often desirable to have some level of fairness of granting the critical region.

A mutual exclusion algorithm is **lockout-free** provided that it guarantees, assuming a low-level fair execution, no process can be kept waiting indefinitely either for C region or for R region if all processes always return the C region.

A mutual exclusion algorithm is **bounded-bypass** if it guarantees b-bounded bypass for some b. b-bounded bypass is defined as follows.

b-bounded bypass: After a process i has performed a locally controlled step in T region, process i cannot be bypassed by any certain process in entering critical region for more than b times.

According to the definitions, lockout freedom condition implies progress condition. Thus, if we prove an algorithm satisfies lockout freedom, this algorithm also satisfies progress condition. For granting the critical region, bounded bypass provides a stronger fairness than lockout freedom since bounded bypass guarantees that not only every requesting process will enter its critical region but also there exists a bound b such that no requesting process will be bypassed by any certain process for more than b times.

```
Shared variables:
$L \in \{nil, 1, \ldots, n\}$, initially $nil$
$P \in \{nil, 1, \ldots, n\}$, initially $nil$

Process $i$ :    $(1 \le i \le n)$

Private variables:
$next \in \{nil, 1, \ldots, n\}$
$tail \in \{nil, 1, \ldots, n\}$


   while $true$ do
R:        Remainder region
T1:       $next := fetch\&store(L, i)$;
T2:       if $next = nil$ then
T3:            await $P = nil$;
T4:            $P := i$;
T5:       else
T6:            await $P = i$;
T7:       fi
C:        Critical region
E1:       if $next = nil$ then
E2:            $tail := fetch\&store(L, nil)$;
E3:            if $tail \ne i$ then
E4:                 $P := tail$
E5:                 await $P = i$;
E6:            fi
E7:            $P := nil$;
E8:       else
E9:            $P := next$;
E10:      fi
   od
```

Figure 1: The algorithm

3 A Fair and Space-efficient Algorithm

In this section, we propose a bounded-bypass mutual exclusion algorithm using mere 2 variables by $fetch\&store$ as well as $read/write$ primitives. Due to space limitation, the correctness proof is omitted.

3.1 The Algorithm

We begin by presenting the main idea of the algorithm in an informal pseudocode style as shown in Figure 1. Exactly two shared variables are used in the algorithm: variable L is used to arrange processes' requests to critical regions; while variable P to indicate which process has permission to enter its critical section. Initially, variables L and P are set to nil, respectively.

Through variable L and $fetch\&store$ primitive, the order to enter the critical region is organized as a circular waiting list in which the first element has the identity of the last one, and each other element has the identity of its predecessor. A circular list is formed as follows. Each process i makes a request by a $fetch\&store$ onto L (T1), announcing its process identity and obtaining the predecessor's identity if has one. Any process which acquires a nil from L (i.e., $next = nil$) becomes the header; otherwise, it becomes a list member. (A header is also dubbed a *controller* and has extra duty at its exit region.) A waiting list is closed after the controller leaves its critical region and resets L as nil (E2). The controller stores the identity of the last element in the list into its private variable *tail*. This closed waiting list contains all processes making a request between the controller obtaining nil from the L (T1) and resetting L as nil (E2). Note that, only after the current controller closes its waiting list such that L will become nil again, a new list might start to form.

The value of shared variable P indicates which process has permission to enter its critical region now. After making a request, a controller repeatedly tests the value of P until P is equal to nil (T3), a specific permission for a controller. The controller takes the permission by assigning P as its identity (T4). (This action prevents another new controller to enter its critical region.) In contrast, a list member i—that is, if $next_i \ne nil$—checks the value of P until $P = i$ (T6) indicating i gains the permission to enter its critical region. Since P is nil initially, the first controller at all will gain the permission to enter its critical region.

After a process leaves its critical region, it should convey the permission to certain waiting process if has one. As a list member, the process simply transfers the permission to its predecessor by setting P as $next$ (E9) and then enters its remainder region. As a controller, after closing the waiting list, if the list contains any process other than the controller, it passes the permission to the last element in the list by setting P as *tail* (E4). The permission will be passed from the last element back to the controller, i.e., in the reverse order of processes making a request. The controller is blocked until the permission passes back to itself (E5).

Although resetting L as *nil* might introduce a new controller, this new controller and subsequent requesting processes will not obtain the permission and this new waiting list will not be closed unless all processes in the previous circular waiting list have finished their critical regions. (Hence, there are at most 2 waiting lists simultaneously, and at most one of these two lists contains the permission.) This contributes to the bounded bypass property of our algorithm. The new controller will get the permission after the permission passes back to the previous controller causing the previous controller to reset P as *nil* (E7).

4 Impossibility Result

In this section, we show that there is no mutual exclusion algorithm guaranteeing bounded bypass with fewer than 2 shared variables, using $fetch\&store$ as well as $read/write$ primitives. We follow the proving strategies proposed by Burns and Lynch [4]. Their model contains only $read/write$ primitive. We extend the model to include $fetch\&store$ and prove our result. The following definitions will be used in the proof. The first two are directly borrowed from [14].

Definition 1 *System states s and s' are indistinguishable to process i, written as $s \overset{i}{\sim} s'$, if the state of process i and the values of all the shared variables are the same in s and s'.*

Definition 2 *A system state s is idle if all processes are in their remainder regions in s.*

Following from the progress condition, a process starting from an idle state and involving its steps only will reach the critical region. Furthermore, a process starting from a system state which is indistinguishable to an idle state for this process and involving its steps only will also reach its critical region, since the state of this process and the values of shared variables are the same in this two system states.

The last definition generalized the one defined by Burns and Lynch [4]. According to their original definition, a process *covers* shared variable x if a *write* primitive of the process is enabled to write x. An enabled *write* primitive will overwrite the variable it involves. Inspecting $fetch\&store$ primitive, once it is enabled, it will write a pre-specified value into the variable and overwrite other processes might have written to x. Thus, we generalize the concept of "covering" to $fetch\&store$ primitive. The difference between *write* and $fetch\&store$ primitives is that a $fetch\&store$ primitive will return the value of the shared variable it overwritten.

Definition 3 *Process i covers shared variable x in system state s provided that in state s, a primitive of process i is enabled to assign a value v into x, the value v depending on a constant function.*

Once process i covers shared variable x with constant value v, i will assign v into x in its next step.

The main idea of the lower bound is that when a process covers shared variable x, it will overwrite other processes might have written to x. If a request of some process is overwritten, we may let another process enter its critical region so many times that violate the bounded bypass condition.

Before proving the lower bound, a basic lemma is needed, showing that a process in its exit region must write something into a shared variable.

Lemma 1 *Suppose A is a mutual exclusion algorithm for $n \geq 2$ processes. Suppose that s is a reachable system state in which process i is in the critical region. If process i reaches R in an execution fragment starting from s that involves steps of i only, then it must write some shared variable along the way.*

Proof. Let α_1 be any finite execution fragment that starts from s (i in C), involves steps of i only, and ends with process i in R. By way of contradiction, suppose that α_1 does not include any write to a shared variable. Let s' be the state at the end of α_1. $s \overset{j}{\sim} s'$, for all $j \neq i$, since all the shared variables remain unchanged.

According to the progress condition, there is an execution fragment starting from s' and not including any steps of process i, in which some other process reaches C. Because $s \overset{j}{\sim} s'$, for all $j \neq i$, there is also such an execution fragment starting from s.

An execution α violating the mutual exclusion is easily constructed as follows. Execution α begins with a finite execution fragment leading to reachable state s, then let another process go to C without any steps of i. Since there are two processes in C at the end of α, this violates the mutual exclusion condition. $\square$

Theorem 2 *If algorithm A solves the mutual exclusion problem for $n > 2$ processes and guarantees bounded bypass, using $fetch\&store$ as well as $read/write$ primitives, then A must use at least 2 shared variables.*

Proof. Suppose for the sake of contradiction that there is such an algorithm, A, using a single shared variable x and guaranteeing b-bounded bypass. We construct an execution of A that violates bounded bypass.

There is an execution involving process 1 only, starting from an initial state s which is idle, that causes process 1 to enter C once and back to an idle state s'. Lemma 1 implies that when process 1 is in the exit region, it must write the shared variable x. Since $read/write$ and $fetch\&store$ primitives are concerned, to write a shared variable will use *write* or *fetch&store* primitives which will assign a constant value into variable x when enabled.

First, we construct α_1 by running process 1 alone from s until it **last** covers the shared variable x. Then we extend α_1 to α_2 by causing process 2 to perform a locally controlled step in the try region and continuing to run process 1 one step, which writes a value into x. Let the final states of α_1 and α_2 be s_1 and s_2, respectively. In states s' and s_2, the shared variable x has the same value and therefore $s' \overset{i}{\sim} s_2$, for all $i \neq 1$ and 2. Only process 1 might know that process 2 has preformed a locally controlled step by the return value when process 1 overwritten x.

Since $s' \overset{i}{\sim} s_2$, for all $i \neq 1$ and 2, and s' is an idle state, we run process 3 alone, starting from s_2, and let process 3 to enter the critical region $b+1$ times, which causes process 3 to bypass process 2 more than b times. This is the needed contradiction. □

5 Conclusion

We have presented an n-process bounded-bypass mutual exclusion with 2 shared variables which can contain $n+1$ distinct values, respectively, using *fetch&store* besides *read/write* primitives. We proved that it is impossible to achieve the same level of fairness with fewer than 2 shared variable by the same primitives.

The *hot spot contention* [6], the maximal number of pending operations for any individual variable in any execution, of our algorithm is n. To alleviate hot spot contention, the number of shared variables might increase. In this paper, we mainly focus on the data requirement for the implementation of mutual exclusion using hardware primitives. It would be interesting to investigate whether there exists a trade-off between hot spot contention and space requirement.

The *remote-memory-reference* complexity of our algorithm is unbounded in distributed shared memory systems. This is unavoidable since only constant 2 shared variables are used among n processes. Consequently, our algorithm is suitable for centralized shared memory systems, benefiting from constant space consumption.

References

[1] J. Anderson and Y.-J. Kim. "Local-spin mutual exclusion using fetch-and-phi primitives." In *Proceeding of the 23rd IEEE International Conference on Distributed Computing Systems*, pp. 538-547, May 2003.

[2] J.E. Burns. "Mutual exclusion with linear waiting using binary shared variables." *ACM SIGACT News*, 10(2): 42-47, summer 1978.

[3] J.E. Burns, P. Jackson, N.A. Lynch, M.J. Fischer and G.L. Peterson. "Data requirements for implementation of N-process mutual exclusion using a single shared variable." *Journal of the ACM*, 29(1): 183-205, January 1982.

[4] J.E. Burns and N.A. Lynch. "Bounds on shared memory for mutual exclusion," *Information and Computation.* 107(2): 171-184, December 1993.

[5] E.W. Dijkstra. "Solution of a problem in concurrent programming control," *Communiations of the ACM.* 8(9): 569, September 1965.

[6] C. Dwork, M. Herlihy and O. Waarts. "Contention in shared memory algorithms." *Journal of the ACM*, 44(6): 779-805, November 1997.

[7] M.J. Fischer, N.A. Lynch, J.E. Burns and A. Borodin. "Distributed FIFO allocation of identical resources using small shared space." *ACM Transactions on Programming Languages and Systems*, 11(1): 90-114, January 1989.

[8] S.S. Fu and N.-F. Tzeng. "A circular list-based mutual exclusion scheme for large shared-memory multiprocessors." *IEEE Tansactions on Parallel and Distributed Systems*, 6(6): 343-364, June 1997.

[9] T.-L. Huang. "Fast and fair mutual exclusion for shared memory systems." In *Proceeding of the 19th IEEE International Conference on Distributed Computing Systems*, pp. 224-231, June 1999.

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

[11] L. Lamport. "A new solution of Dijkstra's concurrent programming problem." *Communiations of the ACM*, 17(8): 453-455, August 1974.

[12] L. Lamport. "The mutual exclusion problem part I: a theory of interprocess communication." *Journal of the ACM*, 33(2): 313-326, April 1986.

[13] L. Lamport. "The mutual exclusion problem part II: statement and solutions." *Journal of the ACM*, 33(2): 327-348, April 1986.

[14] N.A. Lynch. *Distributed Algorithm.* Morgan Kaufmann Publisher, 1996.

[15] N.A. Lynch and M.R. Tuttle. "An introduction to input/output auaomata." *CWI-Quarterly*, 2(3): 219-246, September, 1989.

[16] 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.

[17] G.L. Peterson. "Myths about the mutual exclusion problem." *Information Processing Letters*, 12(3): 115-116, June 1981.