A Fair and Space-efficient Mutual Exclusion

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 shared memory systems with time and resource constraints such as embedded real-time systems, mutual exclusion mechanism that is both fair and space-efficient can be very useful. In this paper, we present a bounded-bypass algorithm using only two shared variables, regardless of the number of contending processes, by operation $fetch\&store$ as well as atomic $read/write$. To achieve the same level of fairness, we show that, by the same set of operations, two shared variables are necessary, and therefore our algorithm is space-optimal.

1. Introduction

The mutual exclusion problem [4] is fundamental in multiprocessing systems for managing accesses to a single indivisible resource. In this problem, a process accesses the resource within a distinct part of code called its *critical region*. Before and after executing the critical region, a process executes trying and exit regions, respectively, in order to guarantee the following requirements.

Mutual Exclusion: At most one process at a time is permitted to enter its critical region.

Progress: If some process is in the trying region and no one is in the critical region, then at some later point some process enters the critical region. In addition, a process in the exit region will eventually enter the rest of code, called the remainder region.

Within the last few years there has been a surge of interest in embedded real-time systems such as automotive control systems, mobile computing devices and home electronics. In general, an algorithm for such systems should take time and resource constraints into consideration. As shown below, a mutual exclusion algorithm, in particular, should take fairness and space-efficiency into consideration.

A mutual exclusion algorithm may not guarantee that the critical region is granted "fairly" to each individual process; that is, starvation may occur. A fair mutual exclusion algorithm means that it has the ability to control the order of granting requests in a fair manner. In systems with time constraint, a process has a deadline in executing a particular job. The goal of a fair mutual exclusion is to reduce the worst-case waiting time and thereby make the algorithm more feasible for such systems.

Besides, the major goal of a space-efficient mutual exclusion algorithm is to reduce the memory consumption. It is crucial for systems with resource constraint. For instance, embedded systems often have small memory (about 32–64 kBytes [13]) since making low production costs is one of the primary concerns in their design. Thus, an algorithm for such systems must be space-efficient.

In terms of space complexity, most of n-process mutual exclusion algorithms in the literature use at least n shared variables; see surveys of Raynal [11] and Anderson et al. [1] However, the number n may be very large and even unknown because a process, in practice, can usually be created and destroyed dynamically. Hence, these algorithms using at least n variables are not suitable for space-limited systems.

The primary contribution of this paper is a fair and space-efficient mutual exclusion algorithm. Our algorithm has the following advantages.

Fair: The algorithm is 2-bounded-bypass.

Space-efficient: Only constant two shared variables are needed, regardless of the number of contending processes.

We say that a mutual exclusion 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. An algorithm is bounded-bypass if it is b-bounded-bypass for some constant b. In fact, 2-bounded bypass is very close to the first-in-first-out (FIFO) order, the most stringent fairness requirement, which satisfies 1-

Proceedings of the 2005 11th International Conference on Parallel and Distributed Systems (ICPADS'05)
0-7695-2281-5/05 $20.00 © 2005 IEEE

bounded bypass. (More precisely, a FIFO algorithm is also 1-bounded-bypass, but the reverse is not true.) For real-time systems, we suggest that a fair mutual exclusion algorithm should satisfy at least bounded bypass so that a process can roughly estimate the waiting time. For instance, in our 2-bounded-bypass algorithm, after requesting the critical region, a process will not be bypassed more than $2(n-1)$ times totally by all other processes. In contrast, if an algorithm only satisfies starvation freedom (i.e., no process will starve), it is possible that a process might be bypassed unbounded times.

To implement our algorithm, we use operation $fetch\&store$ in addition to atomic $read$ and $write$. Burns and Lynch [3] have shown n shared variables are necessary to solve the n-process mutual exclusion problem if only $read$ and $write$ are available. Thus, we need certain more powerful operations to reduce the space requirement. Fortunately, modern microprocessors often provide some read-modify-write (RMW) operations such as $test\&set$, $fetch\&store$, $compare\&swap$, etc. In one instantaneous step, an RMW operation can read a shared variable and write back a new value according to the current value and the submitted function. Operation $fetch\&store$ is adopted to implement our algorithm since it is a commonly supported instruction in modern microprocessors such as a series of processors of Intel and AMD, Motorola 88000, and SPARC [12]. Especially, it is also available in the ARM processor family[1] which is arguably the most popular embedded architecture today. Thus, using $fetch\&store$ makes the algorithm more portable.

Notice that, in the literature, there are several algorithms using only one shared variable and guaranteeing certain level of fairness. For instance, Fischer et al. [7] devised a FIFO algorithm. Burns et al. [2] devised a bounded-bypass algorithm and a starvation-free algorithm[2]. Unfortunately, all of these algorithms used hypothetical RMW operations which have never been implemented in any system shipping today. In contrast, our algorithm uses no hypothetical RMW operation and requires only one more shared variable than these algorithms.

Our algorithm is inspired by the circular list-based mutual exclusion algorithm proposed by Fu and Tzeng [8, 9]. As their method, our algorithm also let waiting processes form a list. But the way of conveying permission in a list and between two lists is very different from theirs. By their way, a process may be blocked in the exit region. In contrast, our algorithm eliminates this drawback. Actually, the problem they tackle is to reduce the number of remote memory accesses, whereas we try to reduce the space complexity and guarantee certain level of fairness.

In addition, we prove that it is impossible to obtain any bounded-bypass algorithm with fewer than two shared variables by $fetch\&store$ as well as $read$ and $write$. Our algorithm is therefore space-optimal by the same set of operations. To follow the convention in the literature on impossibility results, a shared variable associated with a set of operations is called an object. We prove this impossibility result by showing a more general result: using only *historyless* objects, two object instances are required to implement a bounded-bypass mutual exclusion algorithm. Since a shared variable associated with $read/write$ and $fetch\&store$ is a historyless object, our algorithm is space-optimal. The definition of a historyless object is given by Fich et al. [6] and will be restated in Section 4. Informally, an object is historyless if its value after a sequence of operations applied to it depends only on the last nontrivial operation in the sequence. A nontrivial operation is one that will write a value into the object. For example, a shared variable associated with any subset of $read$, $write$, $fetch\&store$ and $test\&set$ operations is a historyless object. This lower bound holds even if the objects have infinite size.

Our lower bound proof technique is related to the method introduced by Burns and Lynch in proving the lower bound of n on the number of $read/write$ objects required to solve the n-process mutual exclusion problem [3]. The difference is that our lower bound applies to all historyless objects rather than only $read/write$ objects. Moreover, our lower bound is for the bounded-bypass mutual exclusion problem, whereas Burns and Lynch consider the general mutual exclusion problem.

Although there are many fair mutual exclusion algorithms in the literature, few of these algorithms are also space-efficient [11, 1]. For those algorithms that are both fair and space-efficient, however, they used hypothetical operations [2, 7]. Without any hypothetical operations, this paper provide a fair and space-efficient algorithm. This algorithm may be useful for systems with time and resource constraints.

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 an impossibility result. Finally, Section 5 is the conclusion.

2. System model and Definitions

2.1. Asynchronous Shared Memory Model

An algorithm in an asynchronous shared memory model is modelled as a triple $(\mathcal{P}, \mathcal{V}, \delta)$, where $\mathcal{P}$ is a nonempty finite set of processes, $\mathcal{V}$ is a nonempty finite set of shared variables, and δ is a transition relation for the entire sys-

1 The ARM processor provides the SWP instruction performing the same functionality of $fetch\&store$. The instruction set can be found at http://www.arm.com/documentation/ARMProcessor_Cores/.

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

tem. Each variable $v \in \mathcal{V}$ has an associated set of values, among which some are designated as the initial values. Each process i is a kind of state machine with the following elements.

- Σ_i: a set of states;
- I_i: a subset of Σ_i, indicating the start states;
- Π_i: a set of steps, describing the activities in which it participates.

A step may involve the shared memory. If it does, we assume that it involves only one shared variable.

A system state is a tuple consisting of the state of each process in $\mathcal{P}$ and the value of each shared variable in $\mathcal{V}$. For a system state s, we write $s(i)$, $i \in \mathcal{P}$, to denote the state of process i in s, and $s(v)$, $v \in \mathcal{V}$, to denote the value of shared variable v. An initial system state is a system state s in which $s(i) \in I_i$ for each process $i \in \mathcal{P}$ and $s(v)$ is a value in the set of initial values for each shared variable $v \in \mathcal{V}$.

The transition relation δ is a set of (s, e, s'), where s and s' are system states, and where $e \in \Pi_i$ for some process i. The transition relation δ has a locality restriction as follows. If step e of process i does not involve any shared variable, only the state of process i can be involved. Otherwise—e involves a shared variable v, only the state of process i and the value of v can be involved.

A step e is enabled at system state s if there exists a system state s' such that $(s, e, s') \in \delta$. We assume that whether a step of a process is enabled at a system state depends only on the process state. Namely, if $e \in \Pi_i$ (i.e., e is a step of process i) is enabled at system state s, then e is also enabled at any system state s' that $s(i) = s'(i)$.

An *execution fragment* is defined as an alternating finite or infinite sequence, $s_0, e_1, s_1, \ldots$, 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 an initial system state. A system state s is a reachable one if there exists a finite execution that ends with s.

2.2. The operations

Under this model, shared variables will be accessed by processes through atomic operations. In addition to atomic $read$ and $write$, operation $fetch\&store$ is involved in this paper. A $fetch\&store$ operation is formally defined below.

$fetch\&store$(variable v, value u)
    $previous := v$
    $v := u$
    return $previous$

It atomically writes value u into variable v and returns the old value.

2.3. The Problem

So far, we have described an asynchronous shared memory model. We now give a formal definition of the mutual exclusion problem which is similar to the definition proposed by Burns et al. in [2].

Informally, the mutual exclusion problem is to devise algorithms for each process to access a designated region of code called the *critical region*. A process can only occupy its critical region while no other process is in its own. In order to gain the permission to enter its critical region, a process executes the *trying region* code, and when the process leaves its critical region, it executes the *exit region* code and then returns to the rest of its code, called the *remainder region*.

For each process i, Σ_i is partitioned into nonempty disjoint subsets R_i, T_i, C_i and E_i. We say that a process i is in the remainder (R) region, trying (T) region, critical (C) region and exit (E) region at system state s if $s(i)$ belongs to R_i, T_i, C_i and E_i, respectively. A system state is said to be *idle* if all processes are in R region. For each initial system state, we assume that it is idle. Besides, we assume that each process obeys an endless loop of life cycle: remainder region, trying region, critical region and exit region.

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 at which more than one process is in C region.

The next condition depends on a low-level fairness assumption for executions. We say that an execution is low-level fair if for each process i that contains only finite steps in this execution, the state of i belongs to R_i after i performs its last step. Namely, a process halts in a low-level fair execution only if it is in R 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 each individual process, i.e., starvation may occur. Thus, it is often desirable to have some level of fairness of granting the critical region.

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

Shared variables:
$L \in \{nil, 1, \ldots, n\}$, initially nil
$P \in \{(current, head) \mid current, head \in \{nil, 1, \ldots, n\}\}$,
initially $(nil, \text{arbitrary})$

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

Private variables of i**:**
$next, tail, c, h \in \{nil, 1, \ldots, n\}$

```
    while true do
R:      Remainder region
T1:     next := fetch&store(L, i);
T2:     (c, h) := P;
T3:     if next = nil then
T4:         while c ≠ nil do          ▷ await current = nil
T5:             (c, h) := P od
T6:         P := (i, h);
T7:     else
T8:         while c ≠ i do            ▷ await current = i
T9:             (c, h) := P od
T10:    fi
C:      Critical region
E1:     if next = nil then                ▷ as a controller
            ▷ close the waiting list
E2:         tail := fetch&store(L, nil);
E3:         if tail ≠ i then
                ▷ wake up the tail and set head as i
E4:             P := (tail, i);
E5:         else
E6:             P := (nil, h) fi          ▷ reset current
E7:     else                              ▷ as a list member
E8:         if next = h then
E9:             P := (nil, h);            ▷ reset current
E10:        else
                ▷ wake up the predecessor
E11:            P := (next, h) fi
E12:    fi
    od
```

Figure 1. The algorithm.

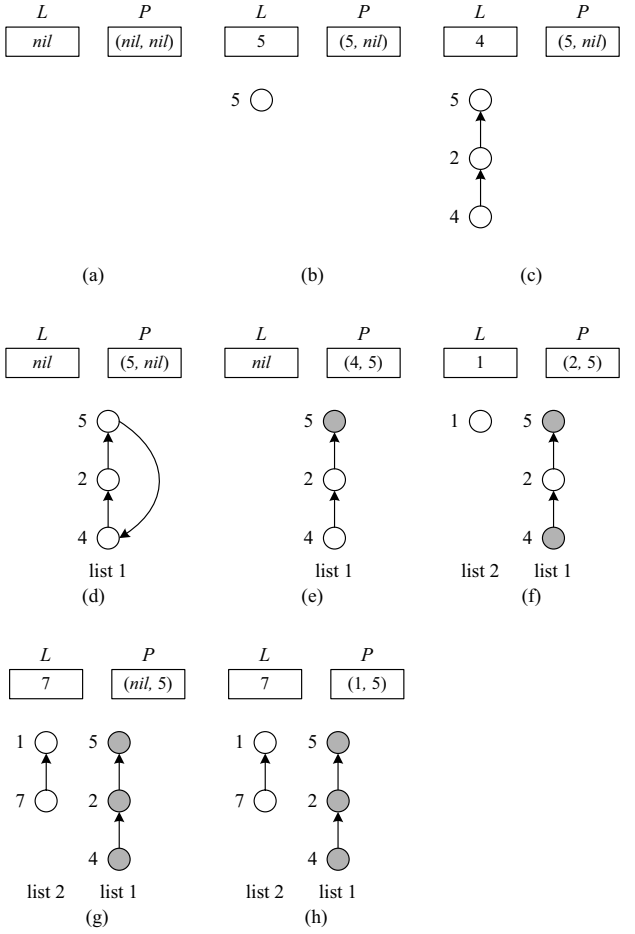

Figure 2. An execution of the algorithm. A gray node indicates a process that has finished one life cycle.

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

3. A Fair and Space-efficient Algorithm

In this section, we propose a bounded-bypass mutual exclusion algorithm using only 2 shared variables by $read/write$ and $fetch\&store$ operations.

The algorithm is shown in Figure 1. Figure 2 is an example to explain how it works. Exactly two shared variables are used in the algorithm: variable L is used to arrange processes' requests to C region; while variable P is used to indicate which process has permission to enter C region. Variable P consists of two parts $(current, head)$, each being the identity of a process or nil. Initially, variables L and P are set to nil and $(nil, \text{arbitrary})$, respectively. In addition, each process has several private variables. If v is a private variable of process i, we write v_i to denote v. Note that in shared memory systems, private variables of a process can be seen as part of the state of the process. Thus, the space consumption does not include private variables. In fact, it costs little to implement private variables by hardware since no memory consistency protocol is needed.

In T region, a process makes a request by executing $fetch\&store$ to L (T1), announcing its process identity and obtaining the predecessor's identity if there is one. As a result, a waiting list will be formed. If a process acquires nil from L (i.e., $next = nil$), it is selected as the controller of

the list. Otherwise (i.e., $next \neq nil$), it is just a list member. For example, in Figure 2(b–c), process 5 first makes a request and acquires nil from L. Next, processes 2 and 4 execute T1 in turn. The waiting list is shown in Figure 2(c).

The value of shared variable P indicates which process has permission to enter C region. After executing T1, the requesting process repeatedly tests P. As a controller, it repeatedly tests until $current$ is equal to nil (T4–T5) which is a specific permission for a controller. The controller takes the permission by setting $current$ as its identity (T6). (This action prevents another new controller to enter C region.) As a list member, it repeatedly tests until $current$ is equal to its identity (T8–T9) indicating it gains the permission to enter C region. Since $current$ is nil initially, the first controller at all will gain the permission to enter C region. For instance, in Figure 2(b), since $current = nil$ process 5 will enter C region after setting $current$ as 5. In Figure 2(c), because neither process 2 nor process 4 gets nil from L, processes 2 and 4 are waiting at T8–T9 until $current = 2$ and $current = 4$, respectively.

The waiting list will be closed after the controller of the list leaves C region. The controller closes the list by E2 which returns the identity of the tail of the list and meanwhile resets L as nil. The controller stores this identity into its private variable $tail$. This closed waiting list contains all processes making requests between the controller obtaining nil from L (T1) and resetting L as nil (E2). After the list is closed, the permission will be transmitted along the list. As shown in Figure 2(d), when process 5 leaves C region, it closes the list by E2. The edge from 5 to 4 indicates that $tail_5 = 4$.

We now show how the permission is transmitted. As a controller, two cases are discussed after it closes the waiting list (E2). Suppose process i is a controller. (i) If the list contains any process other than the controller (i.e., $tail \neq i$), the controller passes the permission to the tail of the list and sets $head$ as i by writing $(tail, i)$ into P (E4). The value of $head$ will be used to check whether all processes in the list have finished their C region. (ii) Otherwise, it just resets the $current$ of P (E6).

As a list member, the process simply transfers the permission to its predecessor by setting $current$ of P as $next$ (E11) and then enters its remainder region. However, if the predecessor is the head of this list, the process should not pass the permission to the head which has finished C region. Some information is needed for checking this situation. Part $head$ of P is used to provide this information. If the value of $next$ of the process is equal to $head$, the process will set $current$ of P as nil instead of $next$ (E9) to indicate that all processes in the list have finished C region.

For example, in Figure 2(e), after process 5 closes the waiting list, process 5 passes the permission to the tail, process 4, and sets $head$ as 5 by writing $(4, 5)$ into P. Process 4 will gain the permission and pass it to process 2 by setting $current$ as 2 (E11) after finishing C region. (See Figure 2(f).) Since $next_2 = head$, process 2 will set $current$ as nil to indicate that all processes in the list have finished C region.

Although resetting L as nil might introduce a new controller as a controller closes a waiting list, this new controller and subsequent requesting processes will not obtain the permission (since $current \neq nill$) until all processes in the previous waiting list have finished their critical regions. This contributes to the bounded bypass property of our algorithm. As shown in Figure 2(f–h), a new list called list 2 forms after L is reset as nil by process 5. The head of list 2, process 1, will not obtain the permission until all processes in list 1 have finished C region. When process 2 in list 1 resets $current$ as nil, process 1 in list 2 will gain the permission to enter C region.

4. Impossibility Result

In this section, we show that the bounded-bypass mutual exclusion problem can not be solved at all with fewer than two shared variables by only $read/write$ and $fetch\&store$ operations. In the proof of this impossibility result, a shared variable associated with a set of operations is defined as an object. To prove this result, we show a more general one: using only historyless objects, two object instances are necessary to solve the bounded-bypass mutual exclusion problem. In our model, each shared variable can be manipulated by operations $read/write$ and $fetch\&store$. That is, the model provides the object associated with $read/write$ and $fetch\&store$. As shown later, this object is a kind of historyless objects. Thus, the lower bound about historyless objects can be applied to our model and thereby implies that our algorithm is space-optimal. We first give the definition of historyless objects proposed by Fich et al. [6] and then present the proof.

An object is a variable shared by processes. Each object has a *type* which consists of a set of possible values and a set of operations that provide the only means to manipulate the object. An operation of an object type is regarded as *trivial* if it leaves the value of the object unchanged after performing this operation. We say that an operation e overwrites an operation e' on an object, if, starting from any value, applying e' and then e yields the same value in the object as applying just e. An object is historyless if all its nontrivial operations overwrite one another. For example, $read$ is a trivial operation. Operations $write$, $fetch\&store$ and $test\&set$ overwrite one another. Thus, an object associated with any subset of $read$, $write$, $fetch\&store$ and $test\&set$ is historyless. (This implies that the object provided in our model is historyless.) For a historyless object, its value depends only on the last nontrivial operation ap-

plied to it because the last nontrivial operation will overwrite the value that might have been written into the object.

Next, we start to present the proof. We follow the proving strategies proposed by Burns and Lynch [3]. Two more definitions are needed. The first one is borrowed from [10].

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 object instances are the same in s and s'.*

The second definition generalized the one defined by Burns and Lynch [3]. According to their original definition, a process *covers* shared variable x if a $write$ operation of the process is enabled to write x. An enabled $write$ operation will overwrite the variable it involves. Similarly, a nontrivial operation of a historyless object will also overwrite the object. Thus, we generalize the concept of "covering" to historyless objects.

Definition 2 *Process i covers a historyless object instance x at system state s provided that i enables a nontrivial operation of x.*

Once process i covers a historyless object instance x, i will overwrite the value of x by performing this nontrivial operation.

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

Before proving the lower bound, we present a basic lemma showing that a process in E region must take a nontrivial operation on some object instance.

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

Proof. Let α_1 be any finite execution fragment that starts from s (at which i is in C region), involves steps of i only, and ends with process i in R. By way of contradiction, suppose that α_1 does not include any nontrivial operation to any object instance. Let s' be the system state at the end of α_1. Since the values of all object instances remain unchanged, we have $s \overset{j}{\sim} s'$, for all $j \neq i$.

According to the progress condition, there is an execution fragment starting from s' and not including any step of process i such that some other process reaches C region. 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 region without any step of i. Since there are two processes in C region at the end of α, this violates the mutual exclusion condition. □

Theorem 2 *If algorithm A solves the bounded-bypass mutual exclusion problem for $n > 2$ processes, using only historyless objects, then A must use at least 2 object instances.*

Proof. Suppose for the sake of contradiction that there is such an algorithm, say A, using only one historyless object instance, say x, and guaranteeing b-bounded bypass. We construct an execution of A that violates bounded bypass. This construction is depicted in Figure 3.

Starting from an initial system state s, which is idle, the progress condition implies that there is an execution involving only process i that causes process i to enter C region once and back to an idle system state s'. Lemma 1 implies that process i must take a nontrivial operation on some object instance in E region. Since only one object instance is involved, process i must take a nontrivial operation to historyless object instance x in E region. That is, i must ever cover x in E region in this execution.

Next, let α_1 be the prefix of this execution up to the point where process i **last** covers x. Then we extend α_1 to α_2 by letting process j perform a step in T region and continuing to run process i one step which overwrites the value of x. Let the final system states of α_1 and α_2 be s_1 and s_2, respectively. In s' and s_2, the object instance x has the same value and therefore $s' \overset{k}{\sim} s_2$, for all $k \neq i$ and j. Only process i might know that process j has preformed a step by the return value when process i overwrote x.

Since $s' \overset{k}{\sim} s_2$, for all $k \neq i$ and j, and s' is an idle system state, we can run process k alone from s_2 (k exists since $n > 2$), and let process k enter the critical region $b+1$ times. This is the needed contradiction because process k bypasses j more than b times. □

5. Conclusion

For systems with time and space constraints, we have provided a fair and space-efficient algorithm. The algorithm is 2-bounded-bypass and requires only 2 shared variables by commonly available operation $fetch\&store$ as well as $read/write$.

By the same set of operations, we have proved that it is impossible to obtain any bounded-bypass algorithm better than ours in terms of space requirement. We prove this impossibility result by showing that using historyless objects including shared variables associated with $read/write$ and $fetch\&store$, two object instances are necessary to solve the bounded-bypass mutual exclusion problem. The proof

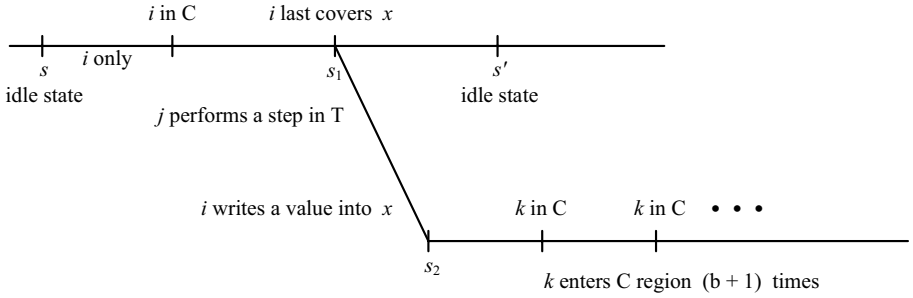

Figure 3. The execution for the proof of Theorem 2.

technique is related to the method, called a *covering argument*, introduced by Burns and Lynch [3].

The hot spot contention [5], the maximal number of pending operations for any individual variable in any execution, of our algorithm is n, where n is the number of processes. Since only constant number of variables are used in the algorithm, $\Omega(n)$ hot spot contention is unavoidable. To alleviate this problem, the number of variables might increase. In his paper, we mainly focus on reducing space consumption to meet the resource constraint.

Actually, there are several fair algorithms using only one shared variable [2, 7]. However, all of these algorithms are implemented by hypothetical read-modify-write operations. In contrast, our algorithm uses only operations commonly provided by hardware, and requires only one more variable than these algorithms.

References

[1] J. H. Anderson, Y.-J. Kim, and T. Herman. Shared-memomry mutual exclusion: major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, Sept. 2003.

[2] 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. *J. ACM*, 29(1):183–205, Jan. 1982.

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

[4] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, Sept. 1965.

[5] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *J. ACM*, 44(6):779–805, Nov. 1997.

[6] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, Sept. 1998.

[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, Jan. 1989.

[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*, 8(6):628–639, June 1997.

[9] 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, Apr. 1998.

[10] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[11] M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986.

[12] I. Rhee. Optimizing a FIFO, scalable spin lock using consistent memory. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 106–114, Dec. 1996.

[13] K. M. Zuberi and K. G. Shin. An efficient semaphore implementation scheme for small-memory embedded systems. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, pages 25–34, June 1997.